SIEMENS EDA

Austemper™ KaleidoScope™ User Guide

Safe Verification

Release 2023.3
Document Revision 26

SIEMENS

About Siemens Digital Industries Software

Siemens Digital Industries Software is a leading global provider of product life cycle management (PLM) software and services with 7 million licensed seats and 71,000 customers worldwide. Headquartered in Plano, Texas, Siemens Digital Industries Software works collaboratively with companies to deliver open solutions that help them turn more ideas into successful products. For more information on Siemens Digital Industries Software products and services, visit www.siemens.com/plm.

Support Center: support.sw.siemens.com
Send Feedback on Documentation: support.sw.siemens.com/doc_feedback_form

Revision History

Revision	Changes	Status/Date
26 (2023.3)	Modifications to improve the readability and comprehension of the content. Approved by Matt Laney. All technical enhancements, changes, and fixes listed in the Austemper Release Highlights for this product are reflected in this document. Approved by Tim Peeke.	Released July 2023
25 (2023.2)	Modifications to improve the readability and comprehension of the content. Approved by Matt Laney. All technical enhancements, changes, and fixes listed in the Austemper Release Highlights for this product are reflected in this document. Approved by Tim Peeke.	Released April 2023
24 (2023.1)	Modifications to improve the readability and comprehension of the content. Approved by Matt Laney. All technical enhancements, changes, and fixes listed in the Austemper Release Highlights for this product are reflected in this document. Approved by Tim Peeke.	Released February 2023
23 (2022.4_2)	Modifications to improve the readability and comprehension of the content. Approved by Matt Laney. All technical enhancements, changes, and fixes listed in the Austemper Release Highlights for this product are reflected in this document. Approved by Tim Peeke.	Released December 2022

Author: In-house procedures and working practices require multiple authors for documents. All associated authors for each topic within this document are tracked within the Siemens documentation source. For specific topic authors, contact the Siemens Digital Industries Software documentation department.

Revision History: Released documents include a revision history of up to four revisions. For earlier revision history, refer to earlier releases of documentation on Support Center.

Table of Contents

List of Figures

List of Tables

Chapter 1
Safety Validation: Fault Campaign Overview

Safety validation tests and verifies the resilience of mission-critical electronic systems for systematic and random faults. Systematic faults arising from design or manufacturing flaws typically are resolved with functional and manufacturing tests prior to deployment. Random faults require mechanisms that operate throughout the lifetime of a part. Hardware portions of the system, specifically integrated circuits, insert special safety mechanisms (SMs) into the design. The validation process of these hardware mechanisms prior to deployment is called a *fault campaign*, which validates systems by injecting faults into safety-critical nodes in the design to see if the SMs detect them.

The KaleidoScope™ tool injects faults and manages the fault campaigns that test the circuits of mission-critical systems. The Kaleidoscope tool injects faults into safety-critical nodes in the design, which are then tested to determine if the SMs detect the faults.

Given the size of chips and the number of transistors involved, fault campaigns quickly become a bottleneck to chip development for safety-critical applications. The KaleidoScope tool solves this bottleneck. Although KaleidoScope can analyze any mission-critical application, this document focuses on the *ISO-26262 Road Vehicles—Functional Safety* Standard, which defines functional safety for automotive equipment throughout the lifecycle of automotive electronic and electrical safety systems.

Restriction:
KaleidoScope does not support:

- Hierarchical references in design files.
- Edge-sensitive user-defined primitives (UDPs).

KaleidoScope Functional Safety
Kaleidoscope Design for Testability
Limitations and Restrictions

KaleidoScope Functional Safety

The following sections summarize how KaleidoScope handles fault injection while running fault campaigns as part of the functional safety workflow.

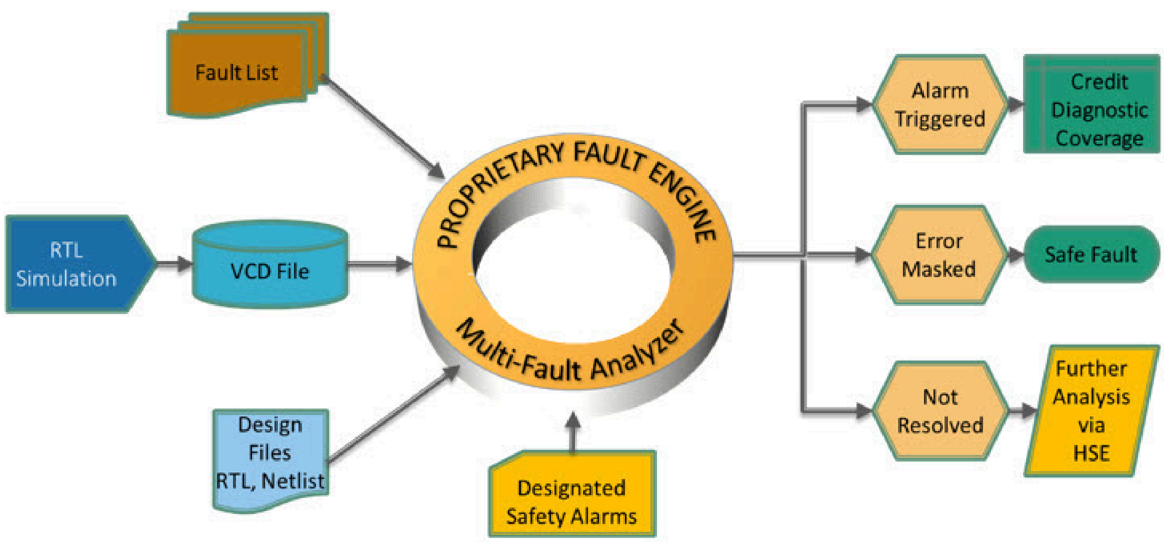
KaleidoScope injects and propagates faults using the safety context in the VCD from RTL simulation.

Fault Injection Flow

During the fault injection flow, KaleidoScope injects and propagates faults by using the safety context in the VCD from RTL simulation.

KaleidoScope contains a proprietary parallel fault propagation engine that can inject multiple faults concurrently without any overlap dependency. Think of each fault as its own small simulation; KaleidoScope runs multiple simulations concurrently to run a fault campaign.

Figure 1. Fault Process and Outcomes



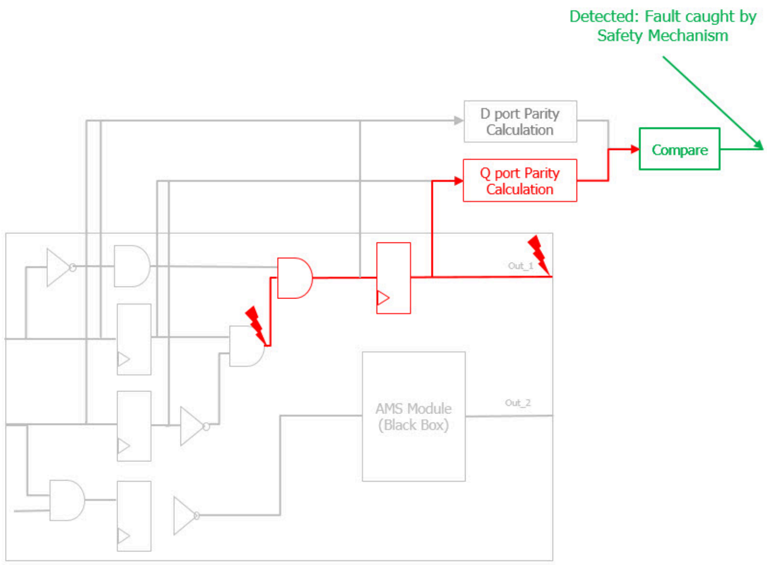
Fault Outcomes

Each fault can trigger one of these fault outcomes: detected, safe, unsafe, or unresolved.

Alarm Trigger: Detected Fault

Machine state is different from the *golden* safety context; the fault is propagated to an alarm. Credit is given to diagnostic coverage (DC).

Figure 2. Alarm Triggered: Detected Fault

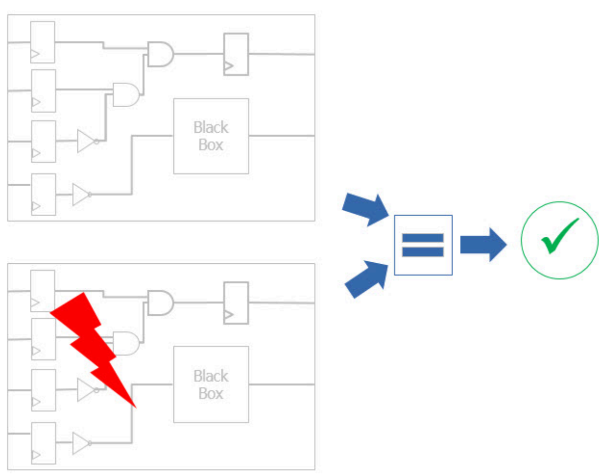


Safe Fault

Machine state is not different from the *golden* safety context.

The following figure shows both types of safe faults. The top fault does not affect safety critical logic. The fault at the bottom it is a safe but detected fault.

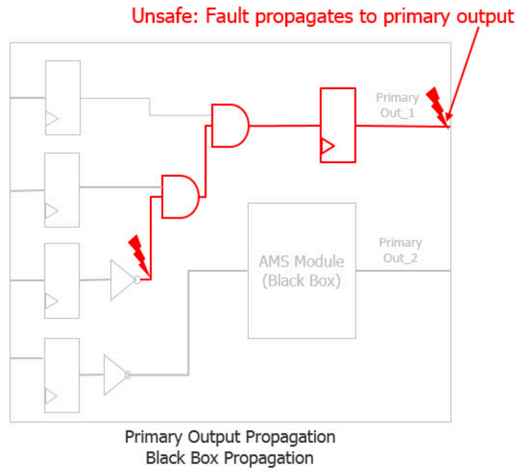
Figure 3. Safe Fault



Unsafe Fault

Machine state is different than the *golden*, and the alarm did not fire.

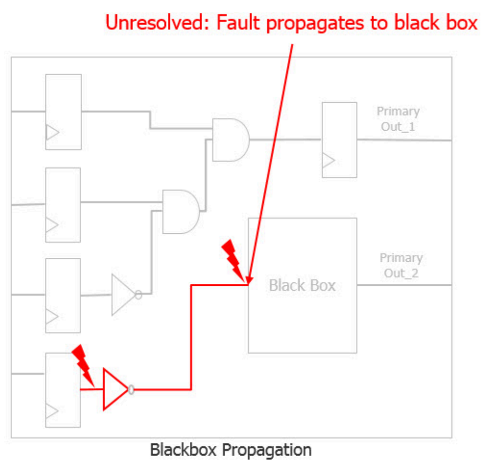
Figure 4. Unsafe Fault: No Alarm Triggered



Unresolved Fault

Machine state is different from the *golden*. Fault is propagated to a black box (analog or user-defined).

Figure 5. Unresolved Fault



Memory Fault Support

You can specify the injection of faults at the memory inputs. The tool propagates the faults through the memory to the outputs or to an alarm

Memory Fault Use Model

Injecting faults into memories follows a specific KaleidoScope process that includes specific input files and tool options. There are limitations on the types of memories the tool supports.

Note:
Fault propagation through a memory is not default behavior and can affect runtime.

Usage, Assumptions, and Limitations

- Non-synthesizable memories are automatically black boxed.
- The memory must be synthesizable. KaleidoScope infers the memory. Fault propagation does not apply to any fault through memory that is not synthesizable, and no fault inside the memory is specified. If the memory fault propagation is disabled, then all memories need a memory information file and KaleidoScope does not infer the memory.
- KaleidoScope does not infer memory from RTL code.
- Unless you provide a memory initialization file, KaleidoScope initializes memories to 0 (zero).
- You must specify the faults using either *--fault_list* or *--fault_file_list* with the KaleidoScope Command.
- You must provide the memory information. See Initializing Memory Arrays for details.

Requirements for Memory Fault Injection

You must specify specific requirements to inject and propagate faults in a memory. Fault injection and propagation in memories is disabled by default. To enable both options, specify the memory_fault_propagation argument in the Fusaini as follows:

memory_fault_propagation = true

See Fusaini option memory_fault_propagation = true | false for more details.

Memory Fault Example

This example provides a simple illustration of using KaleidoScope to perform memory fault injection, memory fault propagation, and simulation of the fault.

Note:
Refer to the example and associated design and input files from the *$FUSA_HOME/share/examples/SafeVerification/FaultCampaignMEMORY* directory in your Austemper software tree.

The following is an example fault list KaleidoScope takes as input. Specify this file with --fault_list or fault_file_list in the KaleidoScope Command.

```
simple_dual_one_clock.dia[0]
simple_dual_one_clock.dia[1]
simple_dual_one_clock.dia[2]
simple_dual_one_clock.dia[3]
simple_dual_one_clock.dia[4]
...
simple_dual_one_clock.dia[15]
```

To enable memory fault injection and propagation, you must specify the memory_fault_propagation Fusaini option.

```
KaleidoScope
 --filelist inputs/synthesized_files.f
 --clkdef inputs/top.clks
 --sim_vcd inputs/vcd.f
 --fault_list inputs/fault.list
 --alarm inputs/alarm.list
 --mode kmanager_single
 --error_inject_inst tb_RAM.simple_dual_one_clock
 --top simple_dual_one_clock
 --ini memory_fault_propagation=true
```

Upon invocation, KaleidoScope loads the input files, injects and propagates the faults, and reports the results of the simulation as shown by the log file:

```
KaleidoScope
 --filelist inputs/synthesized_files.f
 --clkdef inputs/top.clks
 --sim_vcd inputs/vcd.f
 --fault_list inputs/fault.list
 --alarm inputs/alarm.list
 --mode kmanager_single
 --error_inject_inst tb_RAM.simple_dual_one_clock
 --top simple_dual_one_clock
 --ini memory_fault_propagation=true
------------------------------------------------------------------------
OPTION                                  VALUES
------------------------------------------------------------------------
memory_fault_propagation                true
------------------------------------------------------------------------
Started reading from file inputs/synthesized_files.f
Done reading from file inputs/synthesized_files.f
Started reading from file inputs/top.clks
```

```
Done reading from file inputs/top.clks
Started reading from file inputs/vcd.f
...
```

Upon completion, KaleidoScope reports a summary of the simulation run. See "Fault and Alarm Output" on page 38.

```
End of simulation, Start printing final results

Faults Summary
------------------------------------------------------------------------
Total Number of Faults        :  16
Number of Faults Resolved     :  4
          Alarm Detected      :  4
          Alarm NotDetected   :  0
          High Fan Out        :  0
          Reached BBox        :  0
          Reached Observe     :  0
          Missing Sim_Data    :  0
Fault Threshold Number        :  16
Fault Threshold Percentage    :  0
------------------------------------------------------------------------

End of simulation, Done printing final results
RunFaultInjection finished
```

Initializing Memory Arrays

You can initialize memory arrays to specific values with the mem_init_file and ks_memory_module_format Fusaini options.

The mem_init_file = filename fusaini option specifies a file that associates an instance or module and an array name with another file (the memory initialization file that contains memory initialization values) according to the following syntax:

```
<instance/module name> <array name> <initialization file name>
```

The initialization file you specify in the third column can contain either hexadecimal address-value pairs or only values, depending on the value of ks_memory_module_format = false | true.

The default file format for pre-loading a memory via mem_init_file is Verilog LRM $readmemh. The specified file may contain hierarchical instance names or simple module names. The ks_memory_module_format option is not necessary, since KaleidoScope automatically detects simple module names.

You can still load the file format with an address preceding every data word using the memory_data_file_has_explict_addresses = true | false option, but Verilog LRM $readmemh is the recommended format. You can modify the data file to conform to it by stripping the address and adding $readmemh-style addresses where needed.

KaleidoScope supports memory arrays of more than two dimensions, with the following limitations:

1. All address dimensions in the array declaration must have 0 for the right range limit. Example: [7:0], but not [7:5] or [0:7].

2. Memory initialization with the mem_init_file Fusaini option is not supported.

The ks_memory_module_format option also controls whether the file you specify with mem_init_file contains instance names or module names.

Note:
When you specify ks_memory_module_format=true:

- The file you specify with mem_init_file must contain module names instead of instance names.
- The memory initialization file does not require addresses to pair with values because KaleidoScope assumes the list of values begins at address zero.

The following example initializes memories with ks_memory_module_format=true:

```
KaleidoScope
  --filelist                  inputs/synthesized_files.f
  --clkdef                    inputs/top.clks
  --sim_vcd                   inputs/vcd.f
  --fault_list                inputs/fault_init.list
  --alarm                     inputs/alarm.list
  --mode                      kmanager_single
  --error_inject_inst         tb_RAM.top
  --top                       top
  --ini                       memory_fault_propagation=1
  --ini                       mem_init_file=inputs/Memory_Init.txt
  --ini                       ks_memory_module_format=true
  --output_dir                Outputs_mem_propagation
```

Where the contents of *Memory_Init.txt* are the following:

```
#ModuleName           MemoryName       MemoryFile
ram_mod               RAM              inputs/Mem.init.file.txt
```

Where the contents of *Mem.init.file.txt* are:

```
#Content
0000
1234
FFFF
4444
2222
```

Fault List Generation

Using KaleidoScope, you can generate a fault list on the safety mechanisms and diagnostic coverage.

Fault List Generation Use Model

KaleidoScope generates a port fault list containing all the port faults in a design when run in generate_fault_list mode by default, but you can also exclude hierarchies from fault list generation.

Fault List Generation

To generate a fault list, you specify generate_fault_list mode with the KaleidoScope mode option as follows:

```
KaleidoScope \
--mode generate_fault_list  \
--filelist ./Inputs_KS/axi_cross_safety.f \
--clkdef ./Inputs_KS/axi_cross.clk  \
--top axi_cross_wrapper_ATD_wrapper_ATD_wrapper_ATD \
--alarm ./Inputs_KS/AlarmList.txt \
--max_fanout 30000 \
--log_file KS_sa1.log
```

Excluding Hierarchies From Fault List Generation

You can exclude faults when generating a fault list with the fault_exclude_inst and fault_exclude_sig Fusaini options. Both options allow you to specify a file with a list of instance names or signals to exclude. Consider a case where the overall fault list contains the following faults:

```
top.a1.b_reg1
top.a1.b_reg2
top.a2.b_reg1
top.a2.b_reg2
top.a3.b_reg1
top.a3.b_reg2
```

In this case, specifying top.a1 and top.a2 in the file with the fault_exclude_inst option results in a fault list containing only the following faults:

```
top.a3.b_reg1
top.a3.b_reg2
```

These Fusaini options also support wildcarding:

- fault_exclude_inst: each string can contain the wildcard (*).
- fault_exclude_sig: Each string can contain the wildcard (*) in the instance path.

KaleidoScope issues a warning if you specify any unmatched entries for exclusion.

Fault List Reporting

By default, KaleidoScope writes fault lists to a file named *<top_module>.KS.all.faults* in the *Output* directory.

The generated fault list name differs when you perform fault collapsing, as described by Fault Collapsing and Fault List Generation.

Related Topics

Fault Collapsing and Fault List Generation

Fault Collapsing and Fault List Generation

You can instruct KaleidoScope to collapse faults from one fault list to generate a new fault list with only primary faults.

Collapsed Fault List Use Model

Enable fault collapsing by specifying the ks_fault_collapse Fusaini option as shown by the example:

```
KaleidoScope \
  --mode generate_fault_list  \
  --filelist ./Inputs_KS/axi_cross_safety.f \
  --clkdef ./Inputs_KS/axi_cross.clk  \
  --top axi_cross_wrapper_ATD_wrapper_ATD_wrapper_ATD \
  --alarm ./Inputs_KS/AlarmList.txt \
  --max_fanout 30000 \
  --ini ks_fault_collapse=true \
  --log_file KS_sa1.log
```

See “ks_fault_collapse = true | false” on page 109.

Fault Collapsing Limitations

KaleidoScope fault collapsing support applies only to the following faults:

- Stuck At faults
- Port Faults

Collapsed Fault List Reporting

KaleidoScope generates two fault lists as a result of running in generate_fault_list mode with fault collapsing:

- *<design>.KS.primary.faults* – This file contains a list of all the primary faults resulting from fault collapsing. For example:

```
# Port   Fault_Type
RDS_process.mw_U_10reg_cval_reg[4].i4.s0  SA0
RDS_process.mw_U_10reg_cval_reg[4].i4.s0  SA1
RDS_process.mw_U_10reg_cval_reg[4].i4.i0  SA1
RDS_process.mw_U_10reg_cval_reg[4].i4.q  SA1
RDS_process.mw_U_10reg_cval_reg[4].i4.i1  SA0
RDS_process.mw_U_10reg_cval_reg[4].i4.i1  SA1
RDS_process.mw_U_10reg_cval_reg[4].i4.q  SA0
```

- *<design name>.KS.faults* – This file contains a list of all faults from the original fault list that KaleidoScope collapsed. This file contains both the resulting primary faults and the equivalent faults for each primary fault.

```
# Port  Fault_Type
# Collapsed Fault Set 1 Primary Fault -
 RDS_process.mw_U_10reg_cval_reg[4].i4.s0  SA0
RDS_process.mw_U_10reg_cval_reg[4].i4.s0  SA0
# Collapsed Fault Set 2 Primary Fault -
 RDS_process.mw_U_10reg_cval_reg[4].i4.s0  SA1
RDS_process.mw_U_10reg_cval_reg[4].i4.s0  SA1
RDS_process.mw_U_10reg_cval_reg[4].i4.i0  EQ_SA0
# Collapsed Fault Set 3 Primary Fault -
 RDS_process.mw_U_10reg_cval_reg[4].i4.i0  SA1
RDS_process.mw_U_10reg_cval_reg[4].i4.i0  SA1
```

Kaleidoscope Design for Testability

Kaleidoscope supports several Design for Testability (DFT) features including support for Time Delay Faults, Path Delay Faults, and User-Defined Fault Models (UDFM). You can enable these DFT features by setting --ini dft=true.

Tessent UDFM Fault Flow

A Tessent™ tool-produced User-Defined Fault Model (UDFM) defines a custom fault model. You can load a post-ATPG fault list produced by Tessent and perform fault simulation for permanent, transient, path delay, time delay, and UDFM.

Note:
For complete information on creating Tessent UDFMs, refer to the *Tessent Scan and ATPG User's Manual* on Support Center or, if applicable, your Tessent software tree.

Tessent UDFM Flow

KaleidoScope supports loading Tessent tool-generated UDFM files and simulating the faults.

The basic use model of the Tessent UDFM flow involves the following steps:

Note:
Set --ini dft=true to enable DFT fault simulation.

1. Create Tessent UDFM models using the Tessent CellModelGen and running ATPG. You use these UDFM files as input to KaleidoScope.

2. Create the UDFM fault list.

3. Use KaleidoScope to read the Tessent UDFM faults and perform the fault campaign.

   This includes loading the UDFM file, performing the fault injection, and creating the Tessent UDFM-specific report.

Supported Fault Types

In the Tessent UDFM flow, KaleidoScope supports the following faults:

- SA0, SA1 (static)

- TDF01, TDF10

Related Topics

Tessent UDFM Spec File and Fault List

Tessent UDFM Spec File and Fault List

KaleidoScope requires a Tessent UDFM spec file and a Tessent fault list as input for the Tessent UDMF Fault Flow.

Tessent UDFM Spec File

You create the UDFM spec file with the Tessent tool.

For complete details, refer to the *Tessent CellModelGen Tool Reference* and *Tessent Scan and ATPG User's Manual* available on Support Center at the following URL:

`https://support.sw.siemens.com/`

Tessent UDFM Fault List

The Tessent UDFM fault list is a file written by Tessent that contains the Tessent faults. At this time, a script is required to convert the Tessent UDFM fault list into a KaleidoScope compatible format. Contact a Siemens EDA Support Representative for access to this script.

Running the script to convert a Tessent UDFM Fault List results in a *<tessent_file>.ks.faults* file that you specify with the --tessent_udfm_fault_list option when running KaleidoScope.

KaleidoScope Fault Campaigns With Tessent UDFMs

To simulate UDFM faults with KaleidoScope, you must provide specific inputs to the tool. When the fault campaign completes, the tool produces a report of the results.

KaleidoScope Inputs

Provide the following inputs to KaleidoScope at runtime:

- Tessent UDFM Spec Files
- Tessent UDFM Fault List
- Design

This is in addition to the other KaleidoScope options for your fault campaign. See "Fault Campaign Input Requirements" on page 34.

KaleidoScope Invocation for Tessent UDFM Fault Campaign

During KaleidoScope invocation, you use the KaleidoScope command in conjunction with options you specify in the Fusaini file.

The following table identifies the Fusaini options you use in the Tessent UDFM flow.

Table 1. Tessent UDFM Flow Fusaini Options

Fusaini Option	Refer to...
default_tessent_udfm_type	"default_tessent_udfm_type = string" on page 106

Table 1. Tessent UDFM Flow Fusaini Options (continued)

Fusaini Option	Refer to...
dft = true	"dft = true | false" on page 106
tessent_udfm_fault_list	"tessent_udfm_fault_list = udfm_instances_fault_list" on page 116
tessent_udfm_fault_type	"tessent_udfm_fault_type = all | delay | static" on page 116
tessent_udfm_spec_files	"tessent_udfm_spec_files = cell_udfm_definition_list" on page 116

The following example serves as a guide for creating the KaleidoScope Fusaini file. Tessent UDFM flow-specific options are highlighted in red.

```
top = design_top
clkdef = clk
max_concurrent_fault = 1000
kman_parallel = 16
multiple_drivers = true
dft = true
dft_observe_points = inputs/dft_observe_points.txt
error_inject_inst = inputs/dut_instance_path
fsdb_file = patterns/test_setup.fsdb
tessent_udfm_spec_files = udfm_fm.ks
default_tessent_udfm_type = intra_cell_defects
tessent_udfm_fault_list = inputs/ks_short_udfm.faults
tessent_udfm_fault_type = static
```

Use the following KaleidoScope invocation.

```
KaleidoScope \
  --mode kmanager_distributed \
  --filelist  ./inputs/filelist.f \
  --fusaini   ./inputs/fusa_udfm.ini \
  --output_dir Outputs_test_setup \
  --log_file  KS_run_udfm_test_setup.log
```

Usage Notes

- Use the sim_start_time Fusaini option to specify the fault injection time for UDFM faults. If not specified, KaleidoScope uses 0 for the time. See "sim_start_time = integer" on page 115.
- Use the default_TDF_duration Fusaini option to specify the duration for UDFM faults inferred by KaleidoScope. See "default_TDF_duration = integer" on page 106.

Equivalent Faults

KaleidoScope reports equivalent as "EQ".

Fault Reporting

For Tessent UDFM faults, KaleidoScope generates a *<design>.TessentUDFM.rpt* file that reports the DFT classification of the faults and the equivalent faults.

The format of this report consists of five columns:

```
<instance> <Module> <fault_name> <DFT_class> <udfm_type>
```

For example:

```
#Instance  Module           Fault                        DFTClass  UDFMType
top.I1     leaf_and_gates   Bridge_D16:1.0_D_A_1.0_Ohm   TI        Defects
top.I0     leaf_and_always  SAa                          TI        Defects
top.I0     leaf_and_always  TIED                         EQ        Defects
```

KaleidoScope also generates a *<design>.TessentUDFM.Krpt* file as shown by the following example:

```
#ErrorNet      ErrorVal Trigger FailTime Alarm FaultResolution   DFTFault Rational SimData
                        Time    Time     Time                    Class             File
#TessentFault: Fault1 inputs/tess_udfm_faults.list:1
top.I_CP_2     1        5       5        20    Detected_Observed DS       -        rtl/o.vcd
top.H_COMP_2   0        0       0        0     Not_Injected UC            -        rtl/o.vcd
top.J_COMP_2   1        0       0        0     Not_Injected UC            -        rtl/o.vcd
#TessentFault: Fault1 inputs/tess_udfm_faults.list:2
top.I_COMP_a   1        5       5        20    Detected_Observed DS    top.err1    rtl/o.vcd
top.I_COMP_b   0        0       0        0     Not_Injected      UC       -        rtl/o.vcd
top.I_COMP_c   1        0       0        0     Not_Injected      UC       -        rtl/o.vcd
```

Related Topic

Fault Campaign Input Requirements

Run an Observation Off Fault Campaign

You can run an Observation Off Fault Campaign in which KaleidoScope injects faults but does not propagate them.

Prerequisites

- This task requires HDL files that describe your design, a clock definition file, fault list, alarm list, and simulation data files.

Procedure

Run KaleidoScope and set the dft and observation_off Fusaini options both to true as shown by the example:

```
KaleidoScope
   --top top
   --clkdef inputs/clk.s
   --vhdf inputs/files.f
```

```
    --error_inject_inst tb.UUT
    --mode kmanager_distributed
    --fault_list inputs/fault.list
    --sim_vcd inputs/vcd.f
    --alarm inputs/Alarm.list
    --ini dft=true
    --ini observation_off=true
```

Results

KaleidoScope reports the results of the Observation Off fault campaign in the DFT Faults Summary as follows:

```
-------------------------------------------------------------
DFT Faults Summary (Observability OFF)  Primary Faults Total Faults
----------------------------------------------------------------------
Total Number of Faults                 :    258              424
Detected                               :    138              237   (55.90%)
Not Detected                           :    120              187   (44.10%)
  **  Missing Simulation Data          :    0                  0
  **  Uncontrolled Unobserved          :    83               142
  **  Constant Fault                   :    17                25
  **  Unused Fault                     :    20                20
---------------------------------------------------------------------
```

KaleidoScope assigns the injected faults the DS.Off DFT fault class as shown by the example *Kman.DFT.rpt*:

```
#FaultType  FaultClass  FaultNode
0           TI          top.FIFO_1.i4.o
EQ_0        EQ          top.FIFO_1.add_15.i8.a
1           DS.Off      top.FIFO_1.i4.o
EQ_1        EQ          top.FIFO_1.add_15.i8.a
0           TI          top.FIFO_1.i5.o
EQ_0        EQ          top.FIFO_1.add_15.i7.a
1           DS.Off      top.FIFO_1.i5.o
```

You can read the *Kman.DFT.rpt* directly into Tessent.

Path Delay Fault Support

You can use path delay fault support for DFT fault simulation that requires a fault path delay.

To perform a fault simulation with path delays, you must provide a fault list file and a path delay file as inputs to KaleidoScope during invocation.

Note:
Set --ini dft=true to enable DFT fault simulation.

Only one fault per node is supported.

Fault List File

A fault list file is a text file you create that lists the faults that incorporate the path delay faults; specify either *--fault_list file* or *--fault_file_list file* with the KaleidoScope command. See "KaleidoScope Command" on page 95.

The fault list file is a list of hierarchical names to the signals you want to inject with faults, with one signal per line. Port faults are injected by default.

Fault List File Example

```
ripple_fn_safty_wrapper_ATD.u_ripple.dff0.q
ripple_fn_safty_wrapper_ATD.u_ripple.dff1.q
ripple_fn_safty_wrapper_ATD.u_ripple.dff2.q
ripple_fn_safty_wrapper_ATD.u_ripple.dff3.q
```

Path Delay File

A path delay file is an ASCII file you create. At runtime, use the *--pdf_file_list* argument with the KaleidoScope command to specify the path delay file. See "--pdf_file_list pdf file" on page 101.

In the file, describe each path delay using the following format:

```
Begin <path1>
<instance/port_of_the_path> <new_port_value>
End <path1>
```

The tool allows multiple path specifications in the same file as follows:

```
Begin <path1>
...
End <path1>
Begin <pathN>
...
End <pathN>
```

Path Delay File Example 1

```
# Instance/Port of the path     The New Value for the port
Begin P0
ripple_fn_safty_wrapper_ATD.u_ripple.dff0.d 1
ripple_fn_safty_wrapper_ATD.u_ripple.dff0.q 1
End P0
Begin P1
ripple_fn_safty_wrapper_ATD.u_ripple.dff1.d 0
ripple_fn_safty_wrapper_ATD.u_ripple.dff1.q 0
End P1
Begin P2
ripple_fn_safty_wrapper_ATD.u_ripple.dff2.d 0
ripple_fn_safty_wrapper_ATD.u_ripple.dff2.q 1
End P2
```

```
Begin P3
ripple_fn_safty_wrapper_ATD.u_ripple.dff3.d 1
ripple_fn_safty_wrapper_ATD.u_ripple.dff3.q 0
End P3
```

Path Delay File Example 2

```
# Instance/Port of the path       The New Value for the port
Begin Path1
top.COM_1.DFF_A.q 1
top.COM_1.AND_I.y 1
top.COM_1.DFF.d   1
top.COM_1.DFF.q   1
End Path1
```

KaleidoScope Invocation for Path Delay Faults

The following example illustrates invoking KaleidoScope to perform path delay fault simulation by specifying both the fault list file (*fault.list*) and the path delay file (*pdf_files.f*) in the *inputs* directory:

```
KaleidoScope
 --mode kmanager_single
 --filelist ./inputs/files.f
 --clkdef inputs/top.clks
 --simvcd inputs/vcd.f
 --errorinjectinst tb_top.top
 --top top
 --alarm inputs/Alarmlist.txt
 --fault_list inputs/fault.list
 --fusaini inputs/fusa.ini
 --pdf_file_list inputs/pdf_files.f
```

Time Delay Faults

You can use time delay faults for DFT fault simulation that requires time delay.

To perform fault simulation taking time delay into consideration, you must create a time delay fault list file for subsequent input to KaleidoScope during invocation.

Note:
Set --ini dft=true to enable DFT fault simulation.

Only one fault per node is supported.

Time Delay Fault List File

A time delay fault list file is an ASCII file you create as input into KaleidoScope. For example:

```
# begin time delay fault list
top.DFF_1.q TDF01 0 10
top.DFF_2.q TDF10 0 10
# end time delay fault list
```

Specify this fault list file at runtime to KaleidoScope using the --fault_file_list file argument.

For each fault you need to simulate, you must list the fault in the file using a specific format as follows:

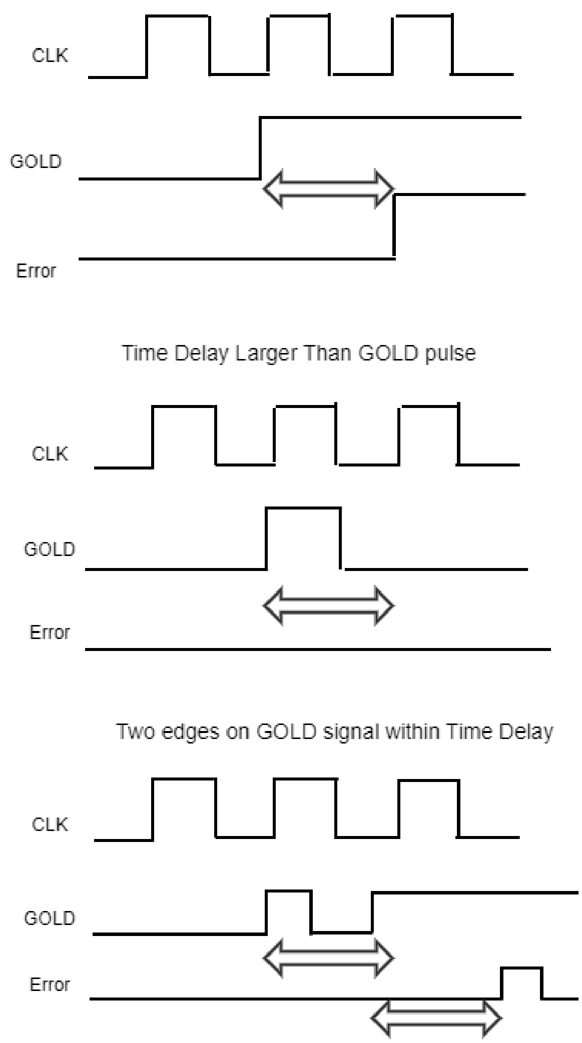
```
<fault_node>  <inject_value>  <inject_time>  <delay_value>
```

where:

- *fault_node* — Specifies the node where the tool injects the fault.
- *inject_value* — Specifies the time delay transition value. Choose one from the following:
    - TD01 — Transition from 0 to 1 (slow to rise).
    - TD10 — Transition from 1 to 0 (slow to fall).
- *inject_time* — An integer.
- *delay_value* — An integer that specifies the edge delay as follows:
    - If you specify an integer, KaleidoScope delays the edge by this value.
    - If GOLD signal has a pulse that is smaller than *delay_value*, no edge is seen on the Error signal.
    - If GOLD signal has multiple edges within time *delay_value*, the error edge value is delayed based on last edge in time delay window.

Figure 6: How delay_value Works illustrates how KaleidoScope uses the *delay_value*.

Figure 6. How delay_value Works



KaleidoScope Invocation for Time Delay Faults

The following example illustrates invoking KaleidoScope to perform time delay fault simulation by specifying the fault list file (*fault.list*) in the *inputs* directory:

```
KaleidoScope
  --mode kmanager_single
  --filelist ./inputs/files.f
  --clkdef inputs/top.clks
  --simvcd inputs/vcd.f
  --errorinjectinst tb_top.top
```

```
  --top top
  --alarm inputs/Alarmlist.txt
  --fault_file_list inputs/fault.list
  --fusaini inputs/fusa.ini
```

Usage Notes

- Reset time using the --sim_start_time integer argument to the KaleidoScope command.
- You can have different *inject_time*s per fault node.

Potential Faults

If KaleidoScope propagates an X value to an observe point during Design For Testability simulation, the fault resolution is known as a potential fault.

You can customize the credit assigned to potential faults with the potential_credit Fusaini option. The default potential credit is 50%, but you can specify any integer value between zero and 100. Specifying this option creates a new line in the faults summary report for potential faults, and adds two new lines in the sub-category.

If you specify the potential_faults Fusaini option as pessimistic, KaleidoScope sets the credit value to zero. If you specify it as optimistic, KaleidoScope sets the credit value to 50%, the same as default. KaleidoScope prints these values in the DFT Faults Summary as follows:

```
-------------------------------------------------------------------------
DFT Faults Summary
-------------------------------------------------------------------------
Total Number of Faults : 256
Detected : 0 (0.00%)
Potential Detected : 205 (40.04%, PT credit = 0.5; default)
Not Detected : 51 (19.92%)
** Controlled Observed Internal : 1
** High Fan Out : 0
** Reached BlackBox : 0
```

If you also specify dft_report = true, the *DFT.rpt* file lists potential faults with the abbreviation PT as follows:

```
#FaultType FaultClass FaultNode
1 PT top.Adder1.Reg1.Q[30]
0 U0 top.Adder1.Reg1.Q[30]
1 PT top.Adder1.Reg1.Q[29]
0 U0 top.Adder1.Reg1.Q[29]
```

Limitations and Restrictions

This topic summarizes high level KaleidoScope limitations and restrictions.

Hierarchical References

- Hierarchical references in design files are not supported.
- Edge sensitive sequential UDPs are not supported.

Chapter 2
Set Up Fault Campaign

The KaleidoScope tool supports two fault campaign modes: kmanager_distributed and kmanager_single. In most cases you will run KaleidoScope in kmanager_distributed mode, especially for large designs or those that require longer run times. The kmanager_distributed mode automatically divides fault campaigns into smaller parallel fault campaigns and, based on the options specified, distributes the fault injection process across multiple CPUs. Typically, you run KaleidoScope in kmanager_single mode to run a fault campaign on a short list of faults or while debugging. Before running KaleidoScope in either of these modes, you must set up the input for the fault campaign.

Feed the inputs listed in Fault Campaign Input Requirements into the KaleidoScope tool when run in either kmanager_single or kmanager_distributed modes, which distributes the list of VCDs across the faults.

Fault Campaign Input Requirements

A fault campaign requires design files, VCD files as stimulus, the fault list, the alarm list, and the safety context.

Table 2. Fault Campaign Input Requirements

Input	Description
Design and VCD files	All design and relevant VCD files.
fault list	A prioritized fault list of the safety-critical nodes in the design is required to start the campaign. Multiple fault modes can be modeled per node (stuck-at-1, stuck-at-0, and so on). **Note:** Time Delay Fault is same as Transition Fault.
alarm list	The alarm signal generated when the safety mechanisms detect the faults successfully. Most designs route this signal to a safety micro controller on-chip for further action. Alarms are mapped to faulted nodes or can be generic.
safety context	Testing the safety mechanisms (SMs) of a safety-critical IC is typically an in-context activity. The stimulus under which the fault is injected represents the system operating condition while performing a safety-critical function. Although *Safety Element out of Context* (SEooC) validation exists, this still requires test stimulus under a generic assumption of a standard operating condition. The KaleidoScope tool accepts fault contexts from VCD files, which can result from gate-level simulations. However, KaleidoScope technology only requires activity of state elements. This activity can be an RTL-based VCD with LVS mappings that indicate any name changes during synthesis.

Fault Lists
Alarm Lists
Specifying Observe Points

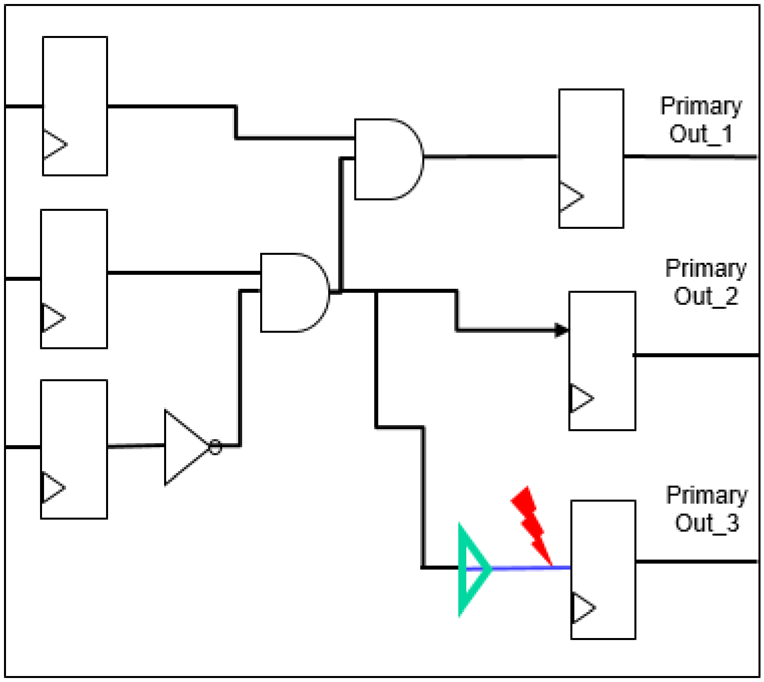
Fault Lists

This topic explains some of the details of fault lists, such as the difference between port and net faults, why it is recommended to simulate using port faults, and how to specify KaleidoScope options to ensure that the correct type of faults are simulated during a fault campaign.

Port Faults and Net Faults

The figure that follows illustrates port fault injection during KaleidoScope simulation:

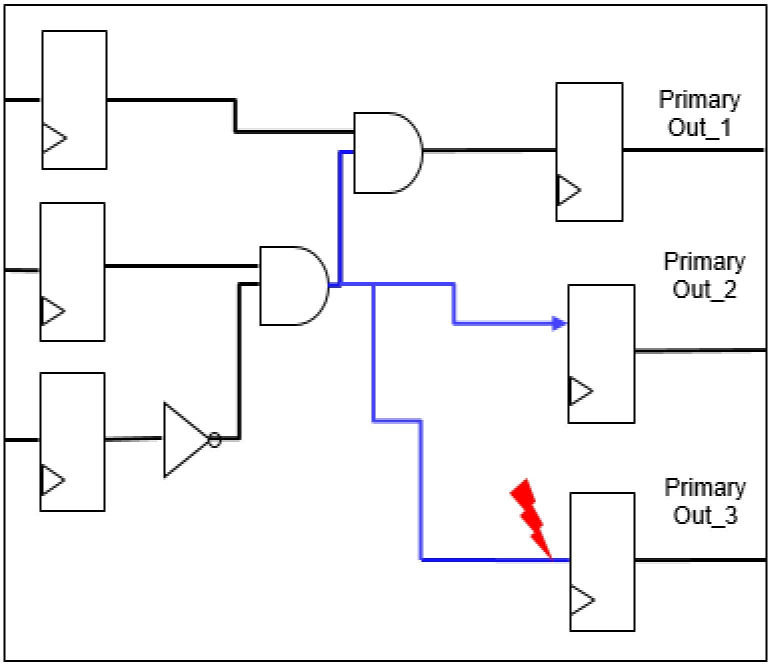
Figure 7. Port Fault



KaleidoScope inserts isolation buffers before injecting port faults, which prevents driving the net to a value.

However, KaleidoScope does not insert a buffer when simulating a net fault. Injecting a net fault drives the entire net to a value, which involves unnecessary use of computing resources. The figure that follows illustrates net fault injection:

Figure 8. Net Fault



KaleidoScope simulates the faults in your fault list as either port faults or net faults, depending on the options you use to specify the fault list.

- When you specify a fault list with the --fault_list option of the KaleidoScope command, KaleidoScope simulates port faults.
- When you specify a fault list with the --error_inject_nodes option, KaleidoScope handles every fault in the list as a net fault.

Example Fault Lists

The fault list file format supports four columns.

1. The first column lists the fault node.
2. The second column lists the type of fault or error value.
   - A value of 0 or SA0 corresponds to stuck at zero, whereas a value of 1 or SA1 corresponds to stuck at 1. A value of 0:1 corresponds to both SA0 and SA1.
3. The third column lists the fault injection time, specified in simulation time units.
4. The fourth column represents the time window, or duration, for which to simulate a transient fault. A value of -1 indicates a permanent fault.

Note:
Fault lists you generate with SafetyScope only contain values in the first column, whereas fault lists you generate with KaleidoScope, with fault collapsing enabled, only contain values in the first and second columns.

You must populate columns three and four to specify injection time, whether the fault is permanent or transient, and the simulation time window in the case of transient faults.

This example illustrates a fault list that contains permanent faults:

```
axi_cross_wrapper_ATD.u_axi_cross.u_axi_cross_wr_ch.state[2]  0 100  -1
axi_cross_wrapper_ATD.u_axi_cross.u_axi_cross_wr_ch.state[1]  0 100  -1
axi_cross_wrapper_ATD.u_axi_cross.u_axi_cross_wr_ch.state[0]  0 100  -1
axi_cross_wrapper_ATD_u_axi_cross.u_axi_cross_rd_ch.state[2]  0 100  -1
```

The next example illustrates a fault list that contains transient faults:

```
u_axi_cross.u_axi_cross_wr_ch.state[2]  1:0  250  2
u_axi_cross.u_axi_cross_wr_ch.state[1]  1:0  250  2
u_axi_cross.u_axi_cross_wr_ch.state[0]  1:0  250  2
```

Alarm Lists

An alarm list is a file you create that contains a list of output signals from safety mechanisms in a design or module.

Each line of an alarm list contains one signal as follows:

```
axi_cross_wrapper_ATD.ATD_safety_err_out_axi_cross[0]
axi_cross_wrapper_ATD.ATD_safety_err_out_axi_cross_wrapper_ATD[0]
axi_cross_wrapper_ATD.ATD_safety_err_out_axi_cross_wrapper_ATD[1]
```

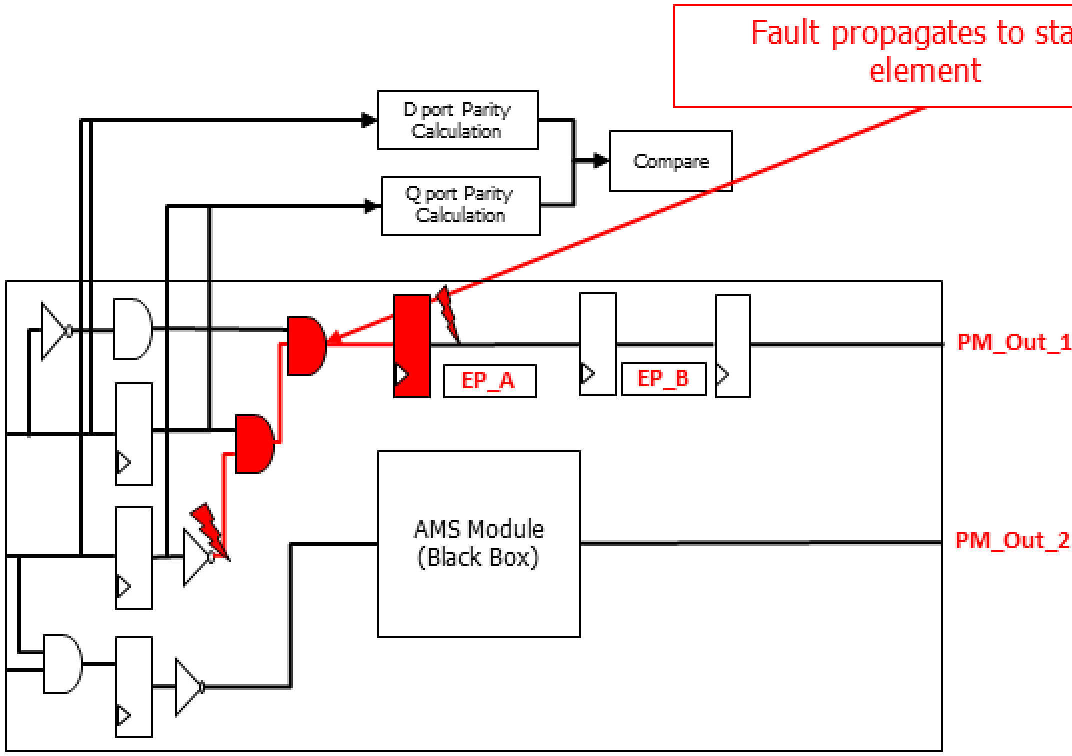
- Specify the full hierarchical path of each signal.
- List each bit of an alarm signal when necessary because bus or vector notation is not supported.

Specifying Observe Points

By default, KaleidoScope classifies a fault that propagates to a state element as observed. However, you can specify a file containing a list of observe points with the --observe_points KaleidoScope command option; in this case KaleidoScope does not classify the fault resolution as observed, unless the fault propagates to one of the observe points listed in the file.

The figure that follows depicts a fault injected by KaleidoScope that propagates to state element EP_A:

Figure 9. Propagation of Fault to State Element



By default, this fault is classified as Undetected_Observed or Detected_Observed, depending on whether or not the compare alarm fired or not, because it reached a state element.

However, if you specify a file with the --observe_points KaleidoScope command option that contains the following lines:

```
top.A.B.PM_Out_1
Top.A.B.PM_Out_2
```

Then the fault is classified as Undetected_Unobserved or Detected_Unobserved, again depending on whether the alarm fired or not, because although it reached state element EP_A, the fault did not propagate to the specified observe point, PM_Out_1.

Fault and Alarm Output

KaleidoScope generates Fault Reports (file extension *.Krpt*) and the Alarm Reports (file extension *.Arpt*) when run in kmanager_single mode. When you run KaleidoScope in kmanager_distributed mode, KaleidoScope generates many more output files fault and alarm reports with the file extension *.rpt*, including the *Kman.SafetyScope.rpt* file you can pass to SafetyScope during final metric validation. All reports are output to the *Outputs* directory.

The node list is fault campaign output covered by the relevant safety mechanisms (SMs). The eventual goal is to compute the *diagnostic coverage (DC)* for the IC or sub function.

Although RTL-based fault campaigns are popular, the ISO 26262 Functional Safety Standard specifically requires that faults be injected into the nodes. Nodes may not exist pre-synthesis. The standard recommends that the fault campaign be run at the gate-level that accurately reflects the IC after synthesis optimizations. KaleidoScope technology inputs RTL files and injects the faults into the gate-level nodes without loss of accuracy.

Fault Propagation Simulation Output

Table 3: Fault Categories outlines possible fault propagation simulation outcomes and the categories in which the faults are defined. The KaleidoScope Log File Faults Summary reports the percentage of detected and undetected faults, with more detailed fault information contained in the KaleidoScope Fault Report.

Table 3. Fault Categories

Outcome	Description
Detected	The fault was accurately detected by one of the designated SMs. The time the fault was detected and the time the alarm triggered are recorded.
Safe Fault	The fault was masked by the machine, based on its input vectors. No further output is provided.
Unsafe Fault	Fault in which the machine state is different than *golden*, and the alarm did not fire.
Unresolved	Faults that *do not* fit in any of the above categories are called *Unresolved* faults. These pose a particular problem for the design verification engineer because resolving these faults depends on the state of the fault at the end of simulation. The causes for unresolved faults are many. Table 4: Unresolved Faults and Suggested Resolution Methods list some causes and recommends a method to resolve the unresolved fault.

Table 4: Unresolved Faults and Suggested Resolution Methods outlines types of unresolved faults and suggested methods for resolving them.

Table 4. Unresolved Faults and Suggested Resolution Methods

Fault	Suggested Resolution Method
Fault not injectable	Additional Functional Safety Context needed (VCD).
Fault still propagating	Debug using the Fault_VCD dump.

KaleidoScope DFT Fault Resolution

The KaleidoScope Fault Report (*Krpt*) reports faults with either DFT or KaleidoScope fault resolutions:

DFT Resolution

- Controlled
    - Fault is injectable or controllable
- Observed == Detected
    - Fault propagated to alarm/observe point

- Observed_internal
    - Fault propagated to register or latch but did not reach an observe point
- Controlled_Observed_PO
    - Fault propagated to primary output but did not trigger alarm

KaleidoScope Resolution

- Detected
    - Fault propagated to alarm signal
- Observed
    - Fault propagated to state element or observe point

KaleidoScope Fault Report

KaleidoScope fault report (*Krpt*) filenames have the following format:

```
Modulename_<KrptString>.Krpt
```

The report's header row contains the following columns that list the status of all the faults for the current fault simulation:

#ErrorNet ErrorVal TriggerTime FailTime AlarmTime FaultResolution FaultClass Rational SimDataFile

- ErrorNet — Specifies the name of node in which the fault was injected.
- ErrorVal — The injected value, *stuck-at-1* or *stuck-at-0*.
- TriggerTime — The time the fault was injected.
- FailTime — The time the fault propagated to an observe point. Indicates when the error occurred.
- AlarmTime — Time fault propagated to Safety Mechanism. Denotes when the alarm was triggered.
- FaultResolution — Displays the result of the fault verification. This is raw data that KaleidoScope observes. You can optionally rename the default FaultResolution value to a custom name. See "map_fault_class = fault_resolution new_fault_class_name" on page 110.
    - Combo_Loop — The fault maps to the Unresolved ISO fault classification.
    - Dangerous_Fault — The fault was undetected. Run with additional stimulus. The fault maps to the Unresolved ISO classification.

One example of a dangerous fault is a fault reaches a primary output port without triggering an alarm.

- KaleidoScope simulates this output port for a defined number of clock cycles (default=2560). Reference the fusaini option “ks_op_to_alarm = clock_cycles” on page 110 to reset the default clock cycle value.

- The fault is marked as an Observe Point Fault

◦ Detected_Observed — The fault maps to the Multi_Point_Fault ISO fault classification.

◦ Detected_Unobserved — The fault maps to the Multi_Point_Fault ISO fault classification.

◦ High_Fanout_Cone — Occurs if the fanout is too large, so the tool drops it. Increase the maximum fanout to accommodate the high cost node with the *--max_fanout* option, which sets a value for the high cost node in fault simulation.

Refer to the “--max_fanout cone_fanout” on page 101 for details. If setting *--max_fanout* does not resolve faults, develop a workflow using an emulator. This situation rarely occurs.

◦ Not_Injected — Occurs if KaleidoScope could not find a toggle point to inject.

◦ Reached_BBox — Propagation ended in a black box. If the back box was assigned to the module for performance reasons, run again without the black box. The simulation may run slower.

◦ Undetected_Observed — The fault maps to the Residual ISO fault classification.

◦ Undetected_Unobserved — The fault maps to the Safe ISO fault classification.

◦ Missing_Sim_Data — Fault propagated to a state element, however simulation data did not exist for comparison at that state element.

- FaultClass — User configurable classification of faults.
- Rational — Lists the node where propagation ended.
- SimDataFileName — The Sim data used when Fault was resolved.

Example Reports

Example Fault Report

The example Figure 10: KaleidoScope Fault Report (Krpt) shows an example of a KaleidoScope Fault Report. The image of the report is split into two halves because the fault report format is too wide for a normal page.

Note:
The full paths in Figure 10: KaleidoScope Fault Report (Krpt) through Example 5: KMAN SafetyScope Report for elements in the *ErrorNet* and the *VCDFileName* columns are preceded with an ellipsis (...) indicating the full path is not visible. This convention allows the overall content of the report to be shown. For complete path information for elements in these reports, reference the specific report in the *Outputs* directory, generated after running the *FaultCampaignManager* example in the directory *$FUSA_HOME/share/examples/SafeVerification/FaultCampaignManager*. The *FUSA_HOME* environment variable points to the installation directory.

Figure 10. KaleidoScope Fault Report (Krpt)

#ErrorNet	ErrorVal	TriggerTime	FailTime	AlarmTime	FaultResolution
...s_arb.localgrant[2]	1	155	0	0	Undetected_Unobser
...s_arb.localgrant[1]	1	0	0	0	High_FanOut_Cone
...s_arb.localgrant[0]	1	0	0	0	High_FanOut_Cone
...s_arb.localgrant[3]	1	155	0	0	Undetected_Unobse
...s_arb.localgrant[2]	1	155	0	0	Undetected_Unobse
...s_arb.localgrant[1]	1	0	0	0	High_FanOut_Cone
...s_arb.localgrant[0]	1	0	0	0	High_FanOut_Cone
...s_arb.localgrant[3]	1	155	0	0	Undetected_Unobserv
...m_bvalid_st	1	155	160	185	Detected_Observed
...m_bresp_st[1]	1	155	160	185	Detected_Observed

nIndex	FaultClass	Rational	SimDataFile
1	Safe	-	...vcd
	Unresolved	FanOut_Cone_of_Error_is_higher_than_MaxFanOut_setting.	...vcd
	Unresolved	FanOut_Cone_of_Error_is_higher_than_MaxFanOut_setting.	...vcd
7	Safe	-	...vcd
	Safe	-	...vcd
9	Unresolved	FanOut_Cone_of_Error_is_higher_than_MaxFanOut_setting.	...vcd
0	Unresolved	FanOut_Cone_of_Error_is_higher_than_MaxFanOut_setting.	...vcd
1	Safe	-	...vcd
3	Multi_Point_Fault	[...]ATD.ATD_safety_err_out_axi_cross_wrapper_ATD[0]	...vcd
	Multi_Point_Fault	[...]ATD.ATD_safety_err_out_axi_cross_wrapper_ATD[0]	...vcd

Alarm Report (Arpt) Files

Alarm report (*Arpt*) filenames have the format:

```
Modulename_<KrptString>.Arpt
```

Alarm reports list the status of all the faults for the current fault campaign and have the same format as the Krpt reports. Example 1: KMAN Alarm Fired Report shows an alarm report.

KaleidoScope Manager Fault and Alarm Reports

When the KaleidoScope tool is run in kmanager_distributed mode, fault and alarm reports are output to the *Outputs* directory. Reports include triggered and nontriggered alarms as well as nontriggered faults for stuck-at-1 and stuck-at-0, and SafetyScope alarms. Example 1: KMAN Alarm Fired Report through Example 5: KMAN SafetyScope Report show excerpts of alarm and fault reports.

Example 1. KMAN Alarm Fired Report

```
 #ErrorNet           ErrorVal        FailTime  AlarmTime   FaultResolution       VCDFileName
...axi_m_bvalid_st       1            160        185           0             ...wavedump.vcd
...axi_m_bresp_st[1]     1            160        185           0             ...wavedump.vcd
...axi_m_bresp_st[0]     1            160        185           0             ...wavedump.vcd
...axi_m_bid_st[7        1             25        355           0             ...wavedump.vcd
...axi_m_bid_st[7]       0            160         18           0             ...wavedump.vcd
...axi_m_bid_st[6]       1            325        355           0             ...wavedump.vcd
...axi_m_bid_st[6]       0            160        185           0             ...wavedump.vcd
```

Example 2. KMAN Alarm Not Fired Report

```
#ErrorNet                    ErrorVal    FailTime  AlarmTime  FaultResolution
 VCDFileName
...u_axi_cross_wr_ch.state[2]    0         0         0            0
  ...wavedump.vcd
...u_axi_cross_rd_ch.state[2]    0         0         0            0
  ...wavedump.vcd
...u_axi_cros_rd_ch.state[1]0    0         0         0            0
  ...wavedump.vcd
...r_b_counter.countval[2]       0         0         0            0
  ...wavedump.vcd
...r_b_counter.countval[3        0         0         0            0
  ...wavedump.vcd
...r_b_counter.countval[2]       0         0         0            0
  ...wavedump.vcd
.
```

Example 3. KMAN Nontriggered Fault 1

```
#ErrorNet
...u_axi_cross_wrapper_ATD.u_axi_cross.u_axi_cross_wr_ch.axi_s_bready[0]
...u_axi_cross_wrapper_ATD.u_axi_cross.u_axi_cross_wr_ch.axi_s_bready[1]
...u_axi_cross_wrapper_ATD.u_axi_cross.u_axi_cross_wr_ch.axi_s_bready[2]
...u_axi_cross_wrapper_ATD.u_axi_cross.u_axi_cross_wr_ch.axi_s_wr_ack[0]
...u_axi_cross_wrapper_ATD.u_axi_cross.u_axi_cross_rd_ch.select_slave[1]
```

Example 4. KMAN Nontriggered Fault 0

```
#ErrorNet
...ss_wrapper_ATD.u_axi_cross.u_axi_cross_wr_ch.axi_m_bresp_st[1]
...u_axi_cross_wrapper_ATD.u_axi_cross.u_axi_cross_wr_ch.axi_m_bresp_st[0]
...u_axi_cross_wrapper_ATD.u_axi_cross.u_axi_cross_wr_ch.axi_s_awaddr[31]
...u_axi_cross_wrapper_ATD.u_axi_cross.u_axi_cross_wr_ch.axi_s_awaddr[30]
...u_axi_cross_wrapper_ATD.u_axi_cross.u_axi_cross_wr_ch.axi_s_awaddr[29]
```

```
...u_axi_cross_wrapper_ATD.u_axi_cross.u_axi_cross_wr_ch.axi_s_awaddr[28]
...u_axi_cross_wrapper_ATD.u_axi_cross.u_axi_cross_wr_ch.axi_s_awaddr[27]
...u_axi_cross_wrapper_ATD.u_axi_cross.u_axi_cross_wr_ch.axi_s_awaddr[26]
```

Example 5. KMAN SafetyScope Report

```
 #Fault                                          Coverage        AlarmName
StopAtMod  FaultType
 top_dup.top1.full_adder_clocked_test[7].Cout   ATD-KS-SA1-SA0  NULL
1          net
 top_dup.top1.full_adder_clocked_test[7].S      ATD-KS-SA1-SA0  NULL
1          net
 top_dup.top1.full_adder_clocked_test[6].Cout   ATD-KS-SA1-SA0  NULL
1          net
 top_dup.top1.full_adder_clocked_test[6].S      ATD-KS-SA1-SA0  NULL
1          net
 top_dup.top1.full_adder_clocked_test[5].Cout   ATD-KS-SA1-SA0  NULL
1          net
```

KaleidoScope Log File Faults Summary

KaleidoScope writes out a summary of the percentage of faults that were detected and not detected during the fault campaign to the logfile. The format of this section of the log file differs between runs that run with fusaini option --ini dft = true, and those that do not.

The Faults Summary also reports the categories of the resolution of the faults not detected. The fault reports in the output directory contain additional details about each fault.

- The following example illustrates the format of the Faults Summary section of the KaleidoScope log file:

```
-------------------------------------------------------
Faults Summary
-------------------------------------------------------------------
Total Number of Faults         :  91
Alarm Detected                 :  73  (80.22%)
Alarm Not Detected             :  18  (19.78%)
  **  Residual                 :  7
  **  High Fan Out             :  0
  **  Reached BlackBox         :  8
  **  Reached Async Port       :  0
  **  Missing Simulation Data  :  0
  **  Combinational Loop       :  0
  **  No Deviation             :  2
  **  Not Injected             :  1
-------------------------------------------------------------------
```

- The following example illustrates the format of the DFT Faults Summary section of the KaleidoScope log file:

```
-------------------------------------------------------------------
DFT Faults Summary
-------------------------------------------------------------------
Total Number of Faults                  :  346
Detected                                :  214  (61.85%)
Not Detected                            :  132  (38.15%)
  **  Controlled Observed Internal      :  0
  **  High Fan Out                      :  0
  **  Reached BlackBox                  :  2
  **  Reached Async Port                :  0
  **  Missing Simulation Data           :  0
  **  Combinational Loop                :  0
  **  Controlled Unobserved             :  4
  **  Uncontrolled Unobserved           :  126
  **  Constant Fault                    :  0
  **  Unused Fault                      :  0
-------------------------------------------------------------------
```

Specify Fusaini Options

Fusaini options are a superset of the command line options. Specify a fusaini option either in an initialization file using the *--fusaini* option or on the command line using the *--ini* option.

Create a Fusaini Initialization File

Specify the --fusaini option followed by the initialization filename as the following example illustrates.

Example 6. --fusaini Option Specifies Initialization File

```
KaleidoScope --mode kmanager_single
              --fusaini ks_single.txt
              --koutname Fault_SA1
              --log_file KS_sa1.log
```

A fusaini initialization file lists one or more fusaini options in the file. Specify each option in the file on a separate line. Fusaini options in an initialization file always delimit the option name from the argument with an equals sign (=) that includes at least one or more spaces on each side of the equals sign as the next example illustrates.

Example 7. Fusaini Initialization File *ks_single.ini*

```
uniquify           = 1
filelist           = Inputs_KS/axi_cross_safety.f
clkdef             = Inputs_KS/axi_cross.clk
syn_to_flop        = 1
black_box          = Inputs_KS/axi_cross.bbox
error_inject_inst  = tb_axi_cross_top.u_dut.u_axi_cross
sim_vcd            = Inputs_KS/vcd_filter.f
```

```
         top                = axi_cross_wrapper
         alarm              = Inputs_KS/AlarmList.txt
         fault_list         = Inputs_KS/Fault_SA1.list
```

Using the *-ini* option, you can specifiy a fusaini option on the command line to override an option that has been specified in a referenced fusaini file as Example 8: Fusaini Option on a Command Line illustrates. This can be helpful to retain an original fusaini initialization file but modify options to test specific parameters.

Specify a Fusaini Option on the Command Line

Generally, specify a fusaini option on the command line to override an option that was specified in the initialization file. You may want to do this to test a different option setting for a specific session while retaining the original initialization settings.

Use the *--ini* option to specify a fusaini option on a command line. The *--ini* option must precede the fusaini option, as shown in the following example. An equals sign (=) without spaces on each side of the equals sign delimits the option name from the option argument.

Example 8. Fusaini Option on a Command Line

In this example the *uniquify* option specified at the command line with --ini option disables uniquify that was enabled in the *ks_single.ini* file.

```
KaleidoScope  --mode kmanager_single
              --ini uniquify=0
              --fusaini ks_single.txt
              --koutname Fault_SA1
              --log_file KS_sa1.log
```

Note:
Notice the syntax difference when specifying fusaini options on the command line compared to in a file. No spaces exist on either side of the equals sign (=); either between the option name and the equals sign (=), or the equals sign and the option argument.

Using the *--ini* option to modify an existing option in a fusaini file returns a Warning message as this example illustrates.

Example 9. Warning Message When Fusaini Option Overridden

```
Warning   fusa-76   Option is specified multiple times in setup file.
```

Fault Campaign Setup

Running the fault campaign requires that the KaleidoScope tool is in either one of the kmanager modes and that you use the VCD filtered output.

1. Filter the VCD.

    a. Run the tool in filter mode using the *vcd_filter* option and specify the inputs described in Running a Parallel Fault Campaign.

    b. Pass the simulation waveform using the --*sim _vcd* option. This filters out unnecessary data in the VCD and leaves only the data needed for the fault campaign.

    Note:
    Validation is only as good as the richness of the VCD files. The safety context of the stimulus under which the fault is injected must represent the operating condition of the system while performing a safety-critical function.

2. Run the fault campaign.

    a. Run KaleidoScope with either fault campaign mode: --mode *kmanager_single* or --mode *kmanager_distributed*. Then specify the inputs described in Running a Parallel Fault Campaign or other input appropriate to your fault campaign objectives and safety goals.

    b. Use the output of the VCD filter mode (*FilterOut.vcd*) as simulation VCD input.

Instance Based Black Box Support

You can use KaleidoScope to list instances in a blackbox file with the --black_box_inst KaleidoScope command option.

Prerequisites

- HDL files that describe your design
- Simulation data files
- Clock definition file that contains clock port names for the design

Note:
Although KaleidoScope black-boxes instances you specify in the black box list, you must still provide at least module declarations, including correct ports, for the modules corresponding to these instances. If you do not provide any module definitions for instances in the black-box list, KaleidoScope issues a fusa-315 error message.

Procedure

1. Add an instance name to the black box list.

    The following code demonstrates a black box file, *bbox.list*, that contains two instances:

```
axi_cross_wrapper.u_axi_cross.u_axi_cross_rd_ch.u_rd_fifo.i_sram NULL
axi_cross_wrapper.u_axi_cross.u_axi_cross_wr_ch.u_wr_fifo.i_sram NULL
```

The second field of a row in a black box list is the port name of the clock port on the black box. This value can be NULL.

2. Run KaleidoScope and specify the following options, in addition to the other options required for your fault simulation:

    - --black_box_inst - Specify your black box list containing instance names with this option. This is different than the option for specifying a list of black box modules, which is --black_box.

    - --ini bbox_wildcard - Specify this fusaini option to enable KaleidoScope to locate the modules associated with each black box instance.

    - --ini print_blackbox_rpt - Specify this option to generate a report containing a list of the black box modules corresponding to the black box instances.

    The example code demonstrates how to run KaleidoScope with the previous options:

    Note:
    The design files shown in this example correspond to the *$FUSA_HOME/share/examples/SafeVerification/FaultCampaignManager/* although the install example does not contain the instance-based black box specific files and options.

```
KaleidoScope \
        --mode                  kmanager_distributed          \
        --filelist              ./Inputs_KS/axi_cross_safety.f \
        --clkdef                ./Inputs_KS/axi_cross.clk      \
        --black_box_inst        ./Inputs_KS/inst_bbox.bbox     \
        --ini                   bbox_wildcard=true             \
        --ini                   print_blackbox_rpt=true        \
        --error_inject_inst     tb_top.u_dut.u_axi_cross       \
        --sim_vcd               ./Inputs_KS/vcd_filter.f       \
        --top axi_cross_wrapper_ATD_wrapper_ATD_wrapper_ATD    \
        --alarm                 ./Inputs_KS/AlarmList.txt      \
        --sim_start_time        150                            \
        --max_concurrent_fault  500                            \
        --fault_list            ./Inputs_KS/Fault_KMan.list    \
        --log_file Kman_run.log
```

Results

KaleidoScope generates a <*design_name*>*_blackbox.rpt* file in the *outputs* directory. For example:

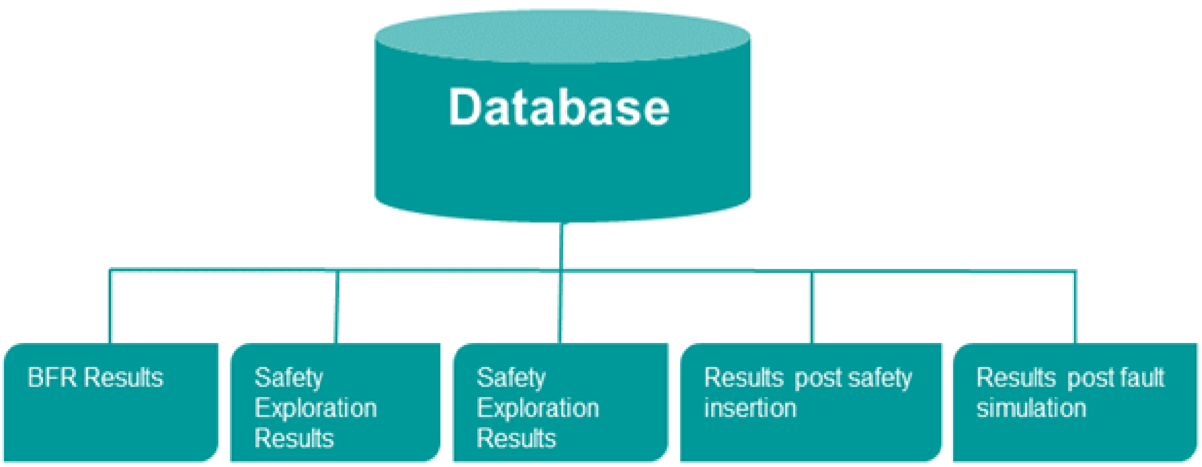
```
Black Box Report
-------------------------------------------- ---------------
Module Instance
------------ -----------------------------------------------
Black Box Report
```

```
-----------------------------------------------------------
Module                   Instance
-----------------------------------------------------------
sram_dw_33_fd_16_pw_4___ATD__2_uniq_1
axi_cross_wrapper.u_axi_cross_wr_ch.u_wr_fifo.i_sramsram_dw_33_fd_16_pw_4__
_ATD__1_uniq_1
axi_cross_wrapper.u_axi_cross_wrapper_ATD.u_axi_cross.u_axi_cross_rd_ch.u_r
d_fifo.i_sram
```

Reading and Writing From a Shared Database With KaleidoScope

KaleidoScope supports reading and writing from a FuSa Common Database (file extension *.fdb)*, which is a data management system that can store all data involved in the functional safety workflow. Different tools, such as SafetyScope can read and write from one database, wherein the data is stored in different partitions known as sessions.

Figure 11. Data Stored in Multiple Sessions of FuSa Common Database



The types of data you can store in a FuSa Common Database session include the following:

- Coverage metrics - FIT rate and Diagnostic Coverage values
- Fault lists generated by SafetyScope
- Fault simulation results from KaleidoScope
- Part and sub-part mapping information

All data within a database is stored in a specific session, and you must specify the session name, in addition to the database name, when reading or writing to a FuSa shared database.

FuSa Common Database Use Cases

You should use the FuSa Common Database as a complete solution to store, read, and write results from different stages of a functional safety workflow. You can also mix its use with a file-based workflow. For example:

- You can read fault list information from a fault list file as input to a KaleidoScope run, but still save the results to a database session.
- You can read all faults from a database session for KaleidoScope run and write results to another database session.
- You can read only specific fault types from a database session for a KaleidoScope run and write the results to another database session.

The following sections describe the Fusaini options involved in these use cases, as well as examples. Although these options are separated into reading and writing categories, it is common to both read and write to a database in the same run.

KaleidoScope Fusaini Options for Writing to a FuSa Common Database

- write_fusa_db = true | false
    - Enables or disables writing to a database.
- fusa_db_name = <DB_name.fdb::session_name>
    - Specifies the name of the *.fdb* database for writing, as well as the session name within the database.
- overwrite_session = true | false
    - Enables or disables overwriting a session within a database.
- overwrite_fusa_db = true | false
    - Specify this option to overwrite the entire database. If you only want to add another session to a database, specify a new session name with the fusa_db_name Fusaini option.

KaleidoScope Fusaini Options for Reading From a FuSa Common Database

This section describes the Fusaini options required for reading from a FuSa common database:

- fault_db_name = <DB_name.fdb::session_name>
    - Specifies the database and session name that contains the fault list information to read.

- read_faults_with_resolution = <fault resolution>
    - Specifies to read only faults with a specific resolution from the database. If you do not specify this Fusaini option, KaleidoScope reads all fault resolution types by default.

Reading Specific Fault List From the Database

You must specify the fusadb_extract_faults Fusaini option as follows:

```
--ini fusadb_extract_faults=<key>:<value>
```

You can use the following keys and values:

- name – It finds all faults with the given value at the beginning.
- value – {0 | 1 | SA0 | SA1 | TDF01 | TDF10}
- type – {perm | trans | all}. Default: perm
- kind – {node | port | all}. Default: port

You also have the option to specify faults with the following keys:

- iso_class
- dft_class
- rational
- stimulus
- alarm
- observe
- resolution

There are two cases for the previous group of keys:

- If you do not specify faults, KaleidoScope includes faults matching the given value.
- If you specify faults, KaleidoScope filters them from the simulated faults tree.

By default, KaleidoScope filters faults from the generated tree. However, if one of the results columns is specified, KaleidoScope filters the faults from the simulated faults tree based on the specified keys as follows:

```
(name AND value AND type AND kind AND fm AND platform AND attribute AND
 rational AND stimulus AND alarm AND observe) AND (iso_class OR dft_class
 OR resolution)
```

You can pass the fusadb_extract_faults fusaini option multiple times, but you should specify each key only once. If you specify a key multiple times, the last value is used. As an example, if you want to filter faults under instance top.A with stuck-at-0 faults and resolution Detected_Observed, you can pass the option multiple times:

```
--ini fusadb_extract_faults=name:top.A
--ini fusadb_extract_faults=value:SA0
--ini fusadb_extract_faults=resolution:Detected_Observed
```

When you use option fusadb_extract_faults=equiv:true, KaleidoScope reports or uses all the equivalent faults of the matched primary faults without any other filtering.

The fusadb_update_db_fault_schema fusaini option updates the database fault schema. KaleidoScope can use it to update the database session passed to the option fault_db_name.

Reading Fault Lists and Writing Fault Campaign Results

```
KaleidoScope    \
  --mode         kmanager_distributed                              \
  --fusaini      ./fusa.ini                                        \
  --output_dir   Outputs                                           \
  --ini          fault_db_name=Outputs/example.fdb::ss_results     \
  --ini          read_faults_with_resolution=Undetected_Unobserved \
  --ini          write_fusa_db=true                                \
  --ini          fusa_db_name=Outputs/example.fdb::ks_results      \
  --ini          overwrite_session=true                            \
```

KaleidoScope reads from and writes to two different sessions in the same database, *example.fdb*. It reads fault list information from the ss_results session and writes fault campaign results to the ks_results session as specified by the fusa_db_name Fusaini option.

Example of a Complete Workflow With SafetyScope and KaleidoScope

1. Run SafetyScope to generate and save a fault list to a database session.

2. Run KaleidoScope to read the fault list from Step 1 from its database session, conduct a fault campaign, and save the fault campaign results to a database session.

3. Run SafeyScope for a second time and read in the fault campaign results from Step 2 to generate final metrics and save the results to a new database session.

The following subsections provide examples and further detail for each of these steps.

Note:
This example section is not intended to provide the full detail of each example run, only to outline the general flow when working with a FuSa Common database.

Step 1

Run SafetyScope in analysis mode to generate a fault list and save it to a database session. The following example command references a Fusaini file that contains the database-specific Fusaini options:

```
SafetyScope \
        --fusaini         ./Inputs/common.ini \
        --safety_mech_ep  SM1.txt \
        --filelist        ./dut.f \
        --output_dir      ./SS_Outputs/
```

The *common.ini* file contains the following Fusaini options:

```
top                         = demoalu
mode                        = analysis
clkdef                      = ./Inputs/clocks.clk
ss_save_fault_list_to_db = true
write_fusa_db               = true
overwrite_session           = true
overwrite_fusa_db           = true
fusa_db_name                = ../db_home/demoalu.fdb::fault_list
```

Step 2

Run a fault campaign with KaleidoScope while reading in the results from the database session in step 1 and writing the results to a new database session. The example command that follows references a Fusaini file that contains the database specific Fusaini options:

```
KaleidoScope \
        --fusaini           ./Inputs/fusa.ini \
        --filelist          ./dut.f \
        --sim_vcd           ./Inputs/vcds.f \
        --error_inject_inst xrtl_top.DUT \
        --alarm             ./Inputs/alarms.list \
        --output_dir        Outputs
```

The *common.ini* file contains the following Fusaini options:

```
mode                 = kmanager_distributed
top                  = demoalu
clkdef               = ./Inputs/clocks.clk
syn_to_flop          = 1
sim_start_time       = 0
max_concurrent_fault = 150
max_fanout           = 99
write_fusa_db        = true
fault_db_name        = ../db_home/demoalu.fdb::fault_list
fusa_db_name         = ../db_home/demoalu.fdb::KS_results
overwrite_session    = true
```

Step 3

Run SafetyScope in analysis mode to read in the fault campaign results from KaleidoScope generated in step 2. The next example command references a Fusaini file that contains the database specific Fusaini options:

```
SafetyScope  \
--fusaini         ./Inputs/common.ini \
--filelist        ./dut.f
--output_dir      ./SS_Outputs/
```

The example *common.ini* file for this step contains the following Fusaini options:

```
top                         = demoalu
mode                        = analysis
clkdef                      = ./Inputs/clocks.clk
ss_save_fault_list_to_db = true
write_fusa_db               = true
overwrite_session           = true
fusa_db_name                = ../db_home/demoalu.fdb::final_metric
ss_read_kman_db             = ../db_home/demoalu.fdb::KS_results
```

Merging Databases

Merging fault campaign results contained in multiple databases into a single database is an important step in certain workflows. For example in a Co-Simulation workflow, you merge Co-Simulation results corresponding to non-synthesizable blocks in a design with other results from a standard KaleidoScope fault campaign.

Use the fusautils utility shipped with KaleidoScope to merge results from more than one fault campaign into a single database as shown by this example:

```
fusautils --mode merge --output_dir DB_Results --ini fusa_db_list=db.list
 --ini fusa_db_name=merged.fdb::KS_merged_results
```

- Specify the --mode merge option to the fusautils command.
- Specify the output_dir option to the fusautils command to control the output directory for the merged database.
- Specify a list of shared databases to merge together with the --fusa_db_list Fusaini option. The database list file must contain a list of databases and sessions in the following format:

  ```
  <DB_path>/<DB_name.fdb::session_name_0>
  <DB_path>/<DB_name.fdb::session_name_1>
  ```

  For example:

  ```
  ./Outputs_kman_db/cosim.fdb::KS_results
  ./Outputs_kman_db/cosim.fdb::KS_cosim_results
  ```

  You can also merge sessions from different databases, for example:

```
./Outputs_kman_db/db_a.fdb::KS_results_0
./Outputs_kman_db/db_b.fdb::KS_results_1
```

- Specify the name for the database resulting from the merge with the --fusa_db_name Fusaini option. For additional information on the fusautils utility, including how to generate reports from a database session, refer to Working With the fusautils Database Utility.

Working With Simulation Data

This section describes creating, troubleshooting, and working with simulation data files at various points of the safety verification process.

Create VCD File With Questa SIM

You can create a VCD file that the KaleidoScope tool can use to run a fault campaign. The steps outlined in this task are the only supported workflow to obtain a valid KaleidoScope VCD file from a Questa waveform.

Restrictions and Limitations

- VCD must be at leaf level
- You must use qwave2vcd version 10.6c or later

Prerequisites

- The KaleidoScope tool requires that the VCD file stores each node in the design, which includes these elements:
    - Flip-flops and gates at all hierarchical levels
    - Ports
    - Memories and 2D arrays

Procedure

1. Compile the design in Questa:

    ```
    vlib work
    vlog -l sim_vlog.log -sv -f axi_cross_top.inc \
    -f axi_cross_safety.f -f tb.f  +incdir+../rtl.cb
    ```

2. With Questa vopt, output a *design.bin* using the option *+design=design.bin*:

    ```
    vopt tb_axi_cross_top +acc -o tb_opt +designfile=design.bin -debug
    ```

3. Direct the Questa SIM tool to record signals, cells, and memories.

   The KaleidoScope tool requires that nodes are recorded so that it can propagate faults successfully.

```
vsim tb_opt -c -do cmd.do -l sim_vsim.log \
  -qwavedb=+signal+cells+memory=4096,2
```

   Restriction:
   Memory can be 4096 bytes or less.

   The output of the previous command creates a *qwave.db* file that is used in the next step.

4. Use Questa qwave2vcd to create a full VCD dump that the KaleidoScope tool can use:

```
qwave2vcd -outputfile kscope.vcd -wavefile qwave.db \
          -designfile design.bin
```

   Restriction:
   The VCD file must be created at leaf level.

5. Using the KaleidoScope tool, filter the VCD to remove unnecessary information:

```
KaleidoScope
   --top axi_cross
   --mode vcd_filter
   --fusaini fusa_safe.ini
   --error_inject_inst axi_cross
   --sim_vcd vcd_prefiltered.f
   --vcd_out_name VCD_Filter_Out
   --log_file VCD_Filter.log
```

6. Use the KaleidoScope *--ks_validate_vcd* options when validating the VCD:

```
KaleidoScope
  --top axi_cross
  --mode kmanager_single
  --ks_validate_vcd 1
  --fusaini fusa_safe.ini
  --error_inject_inst axi_cross_tb.axi_cross
  --fault_list axi_cross.PermFault.nodes
  --sim_vcd vcd_filtered.f
  --log_file KaleidoScope.log
```

7. Review the *<top>_missing_in_vcd.list* file for signals required by the KaleidoScope tool but that are not present.

Note:
Reference "Resolve Missing Signal Issues in the VCD" on page 61 to resolve missing signal issues.

Working With qwave.db Files

The *qwave.db* waveform file format is the standard output format for simulation results from Questa SIM. The topics in this section explain how to read and write *qwave.db* files with KaleidoScope, and to compare *qwave.db* files to *.vcd* files.

Reading Qwave.db files as Simulation Data

You can specify *qwave.db* files as stimulus for a KaleidoScope fault campaign.

Prerequisites

- A *qwave.db* file containing simulation data. You can only specify one type of simulation data per fault campaign. For example, you cannot mix *.vcd* and *qwave.db* stimulus in the same fault campaign.
- Design Files – The HDL files that describe your design.
- Clock definition file – A file containing clock port names for the design.

Procedure

Specify a *qwave.db* file with the --sim_qwave KaleidoScope command option or the Fusaini option qwave_file when you run KaleidoScope.

This example specifies the qwave_file Fusaini option:

```
KaleidoScope
  --mode kmanager_single
  --filelist ./inputs/files.f
  --clkdef inputs/top.clks
  --ini qwave_file=rtl/sample_qwave.db
  --error_inject_inst tb_top.top
  --top top --alarm inputs/Alarmlist.txt
  --fault_list inputs/fault.list
  --fusaini inputs/fusa.ini
```

You can specify multiple *qwave.db* files in a single file with the sim_qwave option, as well as mark a line as a comment with the (#) character. You can also specify simulation parameters following the name of each file as follows:

```
# qwave FILE    SIM_START_TIME    MAX_INJECT    SIM_END_TIME      FTTI
 DROP_NO_DEVIATION
one.db          100               500           10000             200
 700
two.db          150               -1            10000             -1
  600
```

```
three.db        50                  -1             5000                500
  -1
four.db
```

If you do not specify these parameters, KaleidoScope uses the parameters specified in your Fusaini file. You do not need to specify a value for every column, but you cannot skip a value and specify another after it. For example, you cannot skip SIM_START_TIME and MAX_INJECT, and then specify a value for SIM_END_TIME.

Note:
Specify the Fusaini option debug_qwave_stim to control the number of debug messages related to the *qwave.db* stimulus.

Results

KaleidoScope reads simulation data from the *qwave.db* files and writes the simulation timescale, number of signals and value changes extracted, and the last simulation time to the logfile.

Writing Qwave.db Files

You can output *qwave.db* files with KaleidoScope for debugging with Visualizer. KaleidoScope generates *qwave.db* files based on the simulation of a single fault because the Qwave engine can only record one sequence of values for each signal in the design.

Prerequisites

- The HDL files that describe your design.
- A file defining the clock port names for the design.
- Fault campaign inputs including simulation data, alarm files, and fault lists.

Procedure

Specify the following options when you run KaleidoScope:

- qwave_dump – Specify a value of 1 for this option to write a *qwave.db* file (default 0).
- qwave_dump_file – (Optional) Enter this option to specify a filename for the output *qwave.db* (the default filename is *qwave.db*)

```
KaleidoScope
  --mode kmanager_single
  --filelist ./inputs/files.f
  --clkdef inputs/top.clks
  --sim_vcd inputs/vcd.f
  --error_inject_inst tb_top.top
  --top top
  --alarm inputs/Alarmlist.txt
```

```
  --enable_node_faults 1
  --fault_list inputs/fault.list
  --qwave_dump 1
  --qwave_dump_file myqw.db
```

You can also specify qwave_dump_file as a Fusaini option.

Note:
If you intend to run Visualizer with a *design.bin* file you must also specify the qwave_sim_path KaleidoScope command option to ensure matching hierarchies between *qwave.db* files and *design.bin* files.

Results

You can now run Visualizer and pass in the *qwave.db* or *qwave.db*/*design.bin* pair as shown by the example:

```
Vis -designfile <design.bin filename> -wavefile <qwave.db filename>
```

Resolve Missing Signal Issues in the VCD

If KaleidoScope reports missing signals in the VCD list while correlating the input VCD with the fault list to ensure the VCD contains the required design nodes, you must review the VCD and resolve any missing signals in the VCD list. You can run KaleidoScope in vcd_mapping mode to create a mapping file to resolve missing signal issues related to 2D memories and arrays.

Prerequisites

- Obtain or create a complete VCD dump that has each node saved including flip-flops. ports, and 2D memories and arrays.
- Validate the VCD to ensure it contains all the necessary nodes.

Procedure

1. When a missing signal is reported, verify the signal exists in the VCD.

   Assume the following signal was reported missing:

   ```
   axi_cross.u_axi_cross_wr_ch.resp_error_Austemp_D
   ```

2. When a missing signal is reported, follow the hierarchy to the lowest instance level.

   Review the following code snippet:

   ```
   $scope module u_axi_cross_wr_ch $end
   $var wire 1 %o clk $end
   $var wire 1 %p rst_an $end
   $var wire 1 %q axi_i0_awready $end
   ...
   $var wire 1 &j ATD1_0_select_slave_0_ $end
   ```

```
$var wire 1 &k ATD1_0_select_slave_1_ $end
$var wire 1 &l ATD_Async_Out1 $end
$var wire 1 &m ATD_Clk_Out1 $end
$var reg 1 &n resp_error_Austemp_D $end  <-- Example: missing signal
$var reg 1 &o axi_s_bready_Austemp_D $end
$var reg 33 &p axi_s_wdata_Austemp_D [32:0] $end
```

If the signal does not exist in the VCD, the VCD is incomplete. The regression waveform dump did not log the required information. This scenario most often occurs for 2D arrays and memories, which typically are not logged by default. Additional debugging is required when the signal does not exist in the VCD.

3. Create a vcd mapping file with the vcd_mapping mode of the KaleidoScope command. The example that follows demonstrates how to run the KaleidoScope command in vcd_mapping mode.

```
KaleidoScope \
--mode vcd_mapping
--top counter
--filelist ./inputs/files.f
--clkdef ./inputs/clktxt
--sim_vcd ./inputs/vcd.f
--error_inject_inst tb.c0
```

KaleidoScope writes a *<design>_EP_Mapping.txt* file to the outputs directory.

4. Specify the *<design>_EP_Mapping.txt* file with the --eq_mapping_file option of the KaleidoScope command for subsequent runs.

The example following example demonstrates how to specify a *<design>_EP_Mapping.txt* file with the --eq_mapping_file option:

```
KaleidoScope \
--mode kmanager_single
--top counter
--filelist ./inputs/files.f
--clkdef ./inputs/clk.txt
--simvcd ./filtered_vcd.f
--error_inect_inst tb.c0
--alarm inpunts/Alarm.list
--eq_mapping_file ./KS_gen_mapping/counter_EP_Mapping.txt
```

Unsupported Questa to KaleidoScope VCD Workflows

The KaleidoScope tool requires the VCD file to have stored each node in the design. The WLF2VCD and Questa VCD dumps are unsupported because they do not generate the complete VCD file required by KaleidoScope.

WLF2VCD

WLF2VCD can dump 2D memories and arrays, but the WLF2VCD tool does not preserve this information during translation, which is required by the KaleidoScope tool.

Questa Dump VCD

Although Questa supports the direct dumping of VCD files using *vcd file + vcd add -r /**, it does not dump 2D memories and arrays the KaleidoScope tool requires.

Note:
To obtain the full VCD dump with all elements required by the KaleidoScope tool, you must use the Questa qwave2vcd utility as outlined in "Create VCD File With Questa SIM" on page 56.

Grading Simulation Data for Fault Injection

Use the sim_grade mode option to measure and rank VCD and FSDB files by fault injection capacity. The sim_grade mode option filters simulation data, processes fault injection lists for multiple files, and ranks simulation data files based on the faults injected.

Prerequisites

- Create VCD or FSDB files at the leaf level.
- Validate the simulation data files.

KaleidoScope sim_grade mode evaluates both SA0 and SA1 fault values for the faults provided with --fault_list for the purpose of ranking the simulation data files. Sim_grade mode honors all other values in the fault injection file.

KaleidoScope sim_grade mode honors sim_start_time, sim_end_time, and other KaleidoScope simulation options.

Procedure

1. Run the KaleidoScope command with the *sim_grade* mode option with either --sim_vcd or --sim_fsdb, depending on whether you are grading VCD or FSDB input.

```
KaleidoScope
  --mode sim_grade
  --filelist ./Inputs_KS/axi_cross_safety.f
  --clkdef ./Inputs_KS/axi_cross.clk
  --syn_to_flop 1
  --black_box ./Inputs_KS/axi_cross.bbox
  --error_inject_inst  tb_axi_cross_top.u_dut.u_axi_cross
  --sim_fsdb ./Inputs_KS/fsdb_files.f
  --ini verdi_install_path=$VERDI_HOME
  --top axi_cross_wrapper_ATD_wrapper_ATD_wrapper_ATD
  --alarm ./Inputs_KS/AlarmList.txt
  --fault_list ./Inputs_KS/fault.list
  --log_file KS_FSDB_grade.log
```

Note:
Access this example and associated design and input files from the *$FUSA_HOME/share/examples/SafeVerification/GradeFSDB* directory distributed in the software release.

2. Review the *KS_FSDB_grade.log* file and associated reports.

Results

KaleidoScope writes out two reports after grading the simulation data:

- *KS_Sim_Grade_rank.rpt* - This file ranks the simulation data files by the total number of injectable faults, such as the combined number of SA0 and SA1 faults. You can use this report to help decide the order to run tests during a fault campaign.

- *KS_Sim_Grade.rpt* - The KaleidoScope Sim Grade Report, summarizes the SA0 AND SA1 faults for each simulation data file.

To generate additional reports with information about triggered and non-triggered SA0 and SA1 faults, set --ini verbose_grade_reports=1 before running KaleidoScope.

Support for Force Signals During Fault Simulation

KaleidoScope allows you to specify a list of signals to force during fault simulation with the sim_force_list Fusaini option.

By default, KaleidoScope calculates (or simulates) logic until reaching an endpoint, and then compares the value of that endpoint to the simulation data provided as input.

If you specify a list of signals to force, however, then KaleidoScope does not simulate logic up until the specified endpoint, and instead picks up the value of the endpoint from simulation data without comparison.

The file you specify with sim_force_list must contain only signals of the following form:

```
<top_module>.<hier_path>.<signal_name>
```

Consider the following example:

```
KaleidoScope
  --mode kmanager_single
  --filelist ./inputs/files.f
  --clkdef inputs/top.clks
  --simvcd inputs/vcd.f
  --errorinjectinst tb_top.top
  --top top
  --alarm inputs/Alarmlist.txt
  --fault_list inputs/fault.list
  --ini sim_force_list=inputs/force.list
  --ini print_force_rpt=true
```

The example file, *force.list*, contains the list of signals:

```
top.M1.c
top.M1.d
```

If you set the value of the print_force_rpt Fusaini option to true, KaleidoScope writes a fusa-345 message to the transcript and logfile for each signal you specify in a force list:

```
** Info: (fusa-345) Simulation force handling added for node top.M1/c
```

Techniques to Improve Performance

KaleidoScope performance can be modeled as having two components that contribute to total performance: a fixed component and a variable component.

The fixed component factoring into KaleidoScope performance depends mainly on the following considerations:

- Design Size
- Simulation duration
- Toggle activity

The variable component is a function of:

- Number of faults
- Allowed fanout limit
- Simulation duration
- Number of alarms/observe points
- FTTI for functional safety simulation

These topics provide guidelines to optimize KaleidoScope performance with respect to your design.

Reducing Runtime
Distributed Runs and Multithreading

Reducing Runtime

This section describes methods for improving KaleidoScope performance by reducing the number of faults to simulate through fault collapsing, statistical fault sampling, and other methods that result in faster runtimes.

Performance Impact of Common KaleidoScope Options

While subsequent topics in this section cover methods of reducing the number of faults to simulate, the options covered in this section reduce runtimes in other ways, such as by decreasing the amount of time spent on each fault.

Common KaleidoScope command options with performance impacts include:

- Maximum Fanout (--max_fanout cone_fanout) – A fault with large fanout may take considerably longer to detect than a fault with small fanout. One approach to account for this is to run multiple fault simulations with successively larger fanouts.

  For example, you can run one simulation with a max_fanout value of 100. Some faults can be detected before fanning out past the value of 100, whereas KaleidoScope reports faults that cannot be detected below that threshold as High Fan Out. You can then run a subsequent fault campaign using only the High Fan Out faults from the first run with a higher max_fanout value such as 1000.

  Using this approach iteratively and merging the results together results in lower combined simulation times.

- FTTI (--FTTI integer) – This option specifies the simulation time window to detect a fault after injection. A shorter FTTI value results in shorter simulations. Specify this option according to design specifications. Overestimating FTTI can result in long fault simulation times (the default uses the full lengths of the simulation data file you input to the KaleidoScope command).

  Specifies window size in simulation data time units. This is the time for the alarm to detect the fault after a failure. If the fault propagates to a state element and does not reach the alarm within the FTTI, then KaleidoScope does not detect the fault, hence the fault is dangerous. Type: Integer. Default: -1, which indicates the full simulation data.

- Number of Alarms – KaleidoScope drops faults from simulation as they are detected, and a greater number of alarms increases the chance of detection, with the result of lower runtimes.

  Alarms are controlled by the safety mechanism used in the design.

- Simulation Start and End times – Setting appropriate simulating start and end times for fault injection can improve overall performance.

- Injecting faults before reset, or before activity on the chip as started, results in wasted simulation cycles with very low chances of detecting a fault.
- Similarly, injecting faults during idle time at the end of a simulation can also result in wasted simulation cycles with a low chance of detecting faults.

Dropping Hypertrophic Faults From Simulation

This topic explains how to drop hypertrophic faults from simulation during KaleidoScope fault campaigns to improve simulation performance.

The hypertrophic fault class includes all faults whose effects spread extensively throughout the design, causing divergence from good state machine status for a large percentage of the design. These differences force KaleidoScope to do a large number of calculations, slowing down the simulation. Hypertrophic faults require a large amount of memory and CPU time to calculate their circuit status. To maintain fault simulation performance, KaleidoScope provides support to drop hypertrophic faults from the simulation.

When you specify the required Fusaini options, KaleidoScope checks for hypertrophic faults at regular intervals during fault simulation. Specify the interval with fusaini option hypertrophic_check_interval in terms of simulation time units. For example, if you specify 1000000, KaleidoScope checks for hypertrophic faults at simulation time 1000000, 2000000 and so on. At the time of the check KaleidoScope evaluates whether or not to drop a given fault from future simulation by evaluating two parameters described in the next paragraphs. If you do not specify the hypertrophic_check_interval Fusaini option, KaleidoScope does not check for any hypertrophic faults.

The first parameter for specifying what KaleidoScope considers a hypertrophic fault is defined as the number of deviations caused by this fault. Specify a whole number with fusaini option hypertrophic_sim_multiple to define the upper limit of state elements with deviations. KaleidoScope multiplies this number by 100 and treats any fault that involves a greater number of state elements with deviations as a hypertrophic fault.

For example, if you specify the number 4 with hypertrophic_sim_multiple:

```
hypertrophic_sim_multiple  = 4
```

KaleidoScope treats any fault that involves more than 400 state elements with deviations during simulation as hypertrophic, and does not simulate that fault going forward.

The second parameter for specifying what KaleidoScope considers a hypertrophic fault is defined in terms of the number of deviations propagating from a fault. With this method, you specify a percentage of the total number of start points (including registers, latches, black box outputs, and primary inputs) that KaleidoScope should permit deviations to propagate to before it treats the fault with the original deviation as hypertrophic. For example if you specify hypertrophic_deviation_percent:

```
hypertrophic_deviation_percent = 20
```

and the design has 1000 start points, then KaleidoScope treats a fault as hypetrophic if deviations propagate to over 200 start points from that fault.

KaleidoScope reports the number of hypertrophic faults encountered during simulation in the transcript and the logfile as follows:

```
------------------------------------------------------------------
Faults Summary
--------------------------------------------------------------------------
---
Total Number of Faults         :  3620
Alarm Detected                 :  1687  (46.60%)
Alarm Not Detected             :  1933  (53.40%)
  **  Residual                 :  0
  **  High Fan Out             :  716
  **  HyperTrophic             :  393
  **  Reached BlackBox         :  0
  **  Reached Async Port       :  1
  **  Missing Simulation Data  :  0
  **  Combinational Loop       :  0
  **  No Deviation             :  86
  **  Not Injected             :  737
-------------------------------------------------------------------------
---
```

The Fault Report (*.Krpt*) now also includes the HyperTrophic fault resolution type.

Statistical Fault Sampling

KaleidoScope provides the option to simulate a randomized subset of faults rather than the total number of faults. This approach saves run time by reducing the number of simulated faults, while providing confidence that the fault campaign results are still representative of the original fault space.

You can either specify the size of the subset to randomly select or specify a coverage goal and confidence interval with the rand_fault_select Fusaini option. In the second case, KaleidoScope calculates the size of the subset to randomly sample.

To specify a coverage goal and confidence interval pass both values to the rand_fault_select Fusaini option according to the following form:

```
rand_fault_select = <coverage_goal> : <confidence_interval>
```

- The coverage goal value can range from 0 to 99.9. A value of 100 is not permitted.
- Confidence interval values are restricted to either 90, 95, 99, or 99.9.

KaleidoScope computes the number of faults to sample (n) from the values you specify with rand_fault_select according to the following equations:

- 90% CI = ± [1.645 * stdev + 1/(2n)]
- 95% CI = ± [1.960 * stdev + 1/(2n)]
- 99% CI = ± [2.576 * stdev + 1/(2n)]
- 99.9% CI = ± [3.291 * stdev + 1/(2n)]

Regardless of the confidence interval you specify, stdev is described by the equation:

stdev = sqrt [FPC * c(1-c)/(n-1)]

Where

c is the coverage goal specified with fusaini option rand_fault_select.

n is the size of the randomized fault subset (KaleidoScope calculates this value).

N is the total number of faults in the original fault space.

FPC = 1-n/N. This Finite Population Corrrection (FPC) term provides correction when sample size is a significant percentage of the original fault space for a more precise estimate than when treating the fault space as infinite.

To simply specify a random number of faults, use a single integer value with fusaini option rand_fault_select. For example:

```
rand_fault_select = 3500
```

Specifying the subset size directly in this way is decoupled from any particular confidence interval or coverage goal unless you account for these with your own calculations.

Note:
You can adjust the value of the randomization seed with the randomize_faults_seed Fusaini option.

Support for Combinational Loops

If KaleidoScope detects a combinational loop while simulating a fault, it classifies the fault as Unresolved, assigns it a Combo_Loop fault resolution, and does not simulate the fault further. However, you can enable combinational loop support by specifying Fusaini option simulate_combo_loops.

When you enable combinational loop support KaleidoScope simulates combinational loops. If a particular combinational loop is stable, or stabilizes within the specified iterations, fault simulation continues normally until reaching a fault resolution.

- If the combinational loop is unstable (oscillating), and does not stabilize within the number of iterations given by combo_loop_iterations, KaleidoScope classifies the fault as Unstable_Loop and assign it an Unresolved_Combo_Loop fault resolution.

  The combo_loop_iterations Fusaini option controls how many times KaleidoScope reevaluates the nets in a strongly connected component (SCC) before concluding that a loop is unstable. This option is optional, with a default value of 200.

Note:
You only see the Combo_Loop fault resolution when combinational loop support is off (default). This is because the Combo_Loop resolution indicates that KaleidoScope has detected a combinational loop and will not simulate the fault further.

Example

The following example demonstrates how to specify the simulate_combo_loops and combo_loop_iterations Fusaini options.

```
KaleidoScope
--mode kmanager_single
--filelist inputs/files.f
--clkdef inputs/top.clks
--top dut
--error_inject_inst tb_dut.dut
--fault_list inputs/fault.list
--sim_vcd inputs/vcd.f
--alarm inputs/alarm.list
--ini simulate_combo_loops=true
--ini combo_loop_iterations=50
```

This example contains only a single fault, which is related to an unstable combinational loop. As a result, the Faults Summary output in the transcript shows zero Combinational Loop resolutions, but one Unstable Loop resolution.

```
---------------------------------------------------------------------
Faults Summary
---------------------------------------------------------------------------
-----
Total Number of Faults          :  1
Alarm Detected                  :  0  (0.00%)
Alarm Not Detected              :  1  (100.00%)
  **  Residual                  :  0
  **  High Fan Out              :  0
  **  Reached BlackBox          :  0
  **  Reached Async Port        :  0
  **  Missing Simulation Data   :  0
  **  Combinational Loop        :  0
  **  No Deviation              :  0
  **  Not Injected              :  0
  **  Unstable Loop             :  1
---------------------------------------------------------------------------
-----
```

KaleidoScope also writes a *<design_name>_combo_loops.rpt* to the *Outputs* directory. For example:

```
# These are the 1 combinational loops in the design:
Combo loop 0 of 6 nets:
  Net dut/temp4
  Net dut/long
  Net dut/o7_temp
  Net dut/temp1
  Net dut/temp2
  Net dut/temp3
```

Reducing Runtime With Save and Restore

The KaleidoScope save_design and restore_design Fusaini options allow you to reduce the amount of runtime spent compiling and elaborating design files. This can lead to considerable time savings over the course of multiple runs.

The following example command demonstrates how to specify the save_design Fusaini option:

```
KaleidoScope \
        --mode                  kmanager_distributed           \
        --filelist              ./Inputs_KS/axi_cross_safety.f \
        --clkdef                ./Inputs_KS/axi_cross.clk      \
        --error_inject_inst     tb_top.u_dut.u_axi_cross       \
        --sim_vcd               ./Inputs_KS/vcd_filter.f       \
        --top                    axi_cross_wrapper             \
        --alarm                 ./Inputs_KS/AlarmList.txt      \
        --sim_start_time        150                            \
        --fault_list            ./Inputs_KS/Fault_KMan.list    \
        --ini save_design=      ./ATD_saved_design
```

Because the command specifies the Fusaini option save_design, KaleidoScope writes a *<design>.ATDdb* database to the directory you specify with save_design. In this case the database is named *axi_cross.ADTdb*, because the name of the top level design for this example is axi_cross, and the KaleidoScope writes the database to the *ATD_saved_designs* directory.

The saved database contains the compiled and elaborated design. You can use this file to skip the runtime used for compilation and elaboration of design files on your next analysis run at the same level.

Note:
You can only save and restore at the same design level. For example, if you save with one module specified by the --top KaleidoScope command option, you cannot restore from that database within a run that specifies a different module as --top.

To restore from the database for a new fault campaign on the same top level, run the KaleidoScope command with the same options as before (plus any new options such as new simulation data), except instead of specifying the save_design Fusaini option, specify the restore_design Fusaini option as shown by this next example:

```
KaleidoScope \
        --mode                  kmanager_distributed           \
        --filelist              ./Inputs_KS/axi_cross_safety.f \
        --clkdef                ./Inputs_KS/axi_cross.clk      \
        --error_inject_inst     tb_top.u_dut.u_axi_cross       \
        --sim_vcd               ./Inputs_KS/vcd_filter.f       \
        --top                    axi_cross_wrapper             \
        --alarm                 ./Inputs_KS/AlarmList.txt      \
        --sim_start_time        150                            \
        --fault_list            ./Inputs_KS/Fault_KMan.list    \
        --ini restore_design=      ./ATD_saved_design
```

KaleidoScope completes the fault campaign just as it would have without the restore_design Fusaini option, butt more quickly because loading from the database allows KaleidoScope to skip recompiling and re-elaborating.

Distributed Runs and Multithreading

This section describes how to optimize KaleidoScope performance by leveraging parallel fault campaigns and multithreading.

Efficient Use of Resources and Licenses

To determine the best settings for a distributed KaleidoScope run, it's important to consider the computing resources you have available in combination with the number of licenses available.

Understanding Available Resources

Most simulation environments run jobs on a server grid. Understanding the specifications of the typical machine on your grid plays an important part in setting up a parallel fault campaign.

Important specifications include:

- RAM size
- Number of cores per machine
- Number of jobs allowed to run in parallel
- Whether or not multithreading is allowed on the grid

Considering these factors in addition to the number of available licenses enables you to maximize fault campaign performance.

Determining Maximum Concurrent Faults

The maximum number of concurrent faults you specify for a fault campaign has an important impact on memory usage. Increasing the number of faults increases memory usage, whereas decreasing the value reduces memory usage. In general, a higher number of concurrent faults results in better performance, but this needs to be considered in balance with your available resources.

Specify the maximum number of concurrent faults with the max_concurrent_fault KaleidoScope command option.

Example Configurations

The sections that follow contain example parallel fault campaign and multithreading configurations to help you achieve the best results on a single machine, as well as on a server grid.

Note:
The following examples assume a typical machine with 16 CPUs and 512GB of RAM, and a maximum concurrent fault setting of 2000.

Single Machine Examples

Parallel fault campaign with 2000 faults running with eight parallel jobs on a single machine:

- Assuming that a single fault simulation takes 70GB of RAM, running eight parallel jobs requires more memory than the machine has available.
- In this case the simulation is resource limited, assuming that at least eight licenses are available, and can only run a maximum of seven fault campaigns in parallel.

Parallel fault campaign with 2000 faults, four parallel jobs, and multithreading 4 for a total of 8000 concurrent faults:

- This configuration consumes slightly more memory than the first example because of multithreading overhead, but still comes in under the 512GB example limit.
- This configuration requires:
    - 16 cores, one core per thread
    - Five licenses (no additional license requirement for threads)

Parallel fault campaign with 2000 faults, two parallel jobs, and multithreading 8:

- Similar to the previous configuration, multithreading results in a slight increase in memory consumption while remaining below 512GB.
- This configuration requires:
    - 16 cores, one core per thread (eight threads each for two parallel jobs)
    - Three licenses for a total of 4000 faults in parallel

Server Grid Examples

Parallel fault campaign with eight parallel jobs and multithreading 4:

- Distributed run launches on server grid and reserves four cores for each parallel run.
- Requires nine licenses, runs 14,000 faults in parallel, performance boost from multithreading.

Parallel fault campaign with 100 parallel jobs and multithreading 4.

- Distributed run launches on server grid and reserves four cores for each parallel run.
- Requires 101 licenses, runs 200,000 faults in parallel, performance boost from multithreading.

Multithreading With KaleidoScope

KaleidoScope supports multithreading for improved simulation performance by allowing you to split work over multiple threads and cores.

Enable multithreading with fusaini option ks_multi_thread and specify the number of threads with ks_num_threads. This example demonstrates how to specify four threads with the ks_multi_thread option in a Fusani file:

```
ks_multi_thread = true
ks_num_threads = 4
```

Note:
KaleidoScope always uses one thread per core. This means that when you specify multiple threads KaleidoScope attempts to use that many cores. Exceeding the number of available cores results in thread/core sharing and reduced performance. Multithreading does not require an additional license.

Note:
You must specify at least four threads with ks_multi_thread, otherwise KaleidoScope generates an error.

Running a Parallel Fault Campaign

This topic outlines how to run a parallel fault campaign with KaleidoScope.

Prerequisites

- Design files
- Simulation data files
- Clock definition file
- Fault list and Alarm list

Procedure

1. Determine the optimal number of parallel runs for your simulation environment, taking available cores, memory, and licenses into account. See Efficient Use of Resources and Licenses for guidance.
2. Run KaleidoScope in kmanager_distributed mode and specify the number of parallel jobs with the kman_parallel option as follows:

```
KaleidoScope \
        --mode            kmanager_distributed \
        --kman_parallel   4
        --filelist       ./Inputs_KS/axi_cross_safety.f \
        --clkdef         ./Inputs_KS/axi_cross.clk  \
        --sim_vcd        ./Inputs_KS/vcd_filter.f \
        --top
 axi_cross_wrapper_ATD_wrapper_ATD_wrapper_ATD \
        --alarm          ./Inputs_KS/AlarmList.txt \
        --fault_list     ./Inputs_KS/Fault_KMan.list \
```

In this case, KaleidoScope runs four parallel fault campaigns and simulates four times the number of concurrent faults than a run with the default kman_parallel value of one.

Good Machine Simulation

You can run a Good Machine Simulation to confirm that KaleidoScope simulates the same as another functional simulator to prevent false detections. KaleidoScope propagates golden reference data from endpoint to endpoint until the entire design is covered, and report any deviations. Because a Good Machine Simulation does not have fault injections, it does not require a fault list.

Restrictions and Limitations

KaleidoScope Good Machine Simulation does not support derived clocks. You must provide a list of all clocks.

Prerequisites

This task requires:

- HDL files that describe your design
- Clock definition file
- Simulation data files for comparison

Procedure

Run the KaleidoScope command with the following options, in addition to the options corresponding to each prerequisite file:

- --ini ks_good_machine=true

  Specify this option to enable Good Machine Simulation. Each endpoint is simulated until the first deviation is encountered.

- --ini race_check=true

  This option flags any race condition between clocks and primary inputs. It is recommended for Good Machine Simulation, but not required.

- --ini ks_count_num_deviations=true

When you specify this option, KaleidoScope continues counting deviations until the end of stimulus for each endpoint. This option is not required, because encountering a single deviation suffices to show that KaleidoScope simulation is not identical to the golden reference.

- --inix_prop_mode={resolve | noxprop | trap | pass}

  This option supports the X propagation modes of Questa when handling an X value in a multiplexer. Default: resolve.

  The Table that follows describes the operation for each of the X propagation modes.

Table 5. X Propagation Resolution

sel	a	b	no X-prop	trap	resolve	pass
X	0	0	0	0	resolve (a,b)=0	X
X	0	1	1	1	resolve (a,b)=X	X
X	1	0	0	0	resolve (a,b)=X	X
X	1	1	1	1	resolve (a,b)=1	X

- --inix_clk_mode={01 | 0x | 0z}

  This option selects the transition used as rising edge of the clock for the flip-flops. Default: 01.

  All the modes listed for this option also support clock for the latches. In any mode, data passes through the latch (D ⇒ Q) when clock pin is 1.

The following example illustrates how to pass in a set of options to run a Good Machine Simulation:

```
KaleidoScope \
--filelist ./Inputs_KS/axi_cross_safety.f
--clkdef ./Inputs_KS/axi_cross.clk
--error_inject_inst tb_axi_cross_top.u_dut.u_axi_cross
--sim_vcd ./Inputs_KS/vcd_filter.f
--top axi_cross_wrapper_ATD_wrapper_ATD_wrapper_ATD
--ini ks_good_machine=true
--ini race_check=true
```

Results

KaleidoScope writes a Good Machine Summary to the transcript listing the total number of endpoints checked for deviations, and the number of deviations detected. KaleidoScope also writes two reports to your *Outputs* directory after completing a Good Machine Simulation:

- *Kman.GoodMachinePass.rpt* - This report contains a list of all endpoints that passed with no deviation between KaleidoScope simulation and the golden reference.

- *Kman.GoodMachineDeviated.rpt* - This report contains a list of all endpoints that deviated from the golden reference.

Examples

Refer to the *$FUSA_HOME/share/examples/SafeVerification/Good_Machine_Sim* directory in your Austemper software tree for a detailed example demonstrating how to run a Good Machine Simulation.

Four-State Simulation

KaleidoScope propagates X and Z values in fault simulations by default. If you set either the --*four_state_simulation* command line option or the *four_state_simulation* Fusaini option to false, KaleidoScope converts X and Z values in a VCD to 0 (zero) instead.

The functionality is useful with DFT fault simulations or any time you want true X/Z propagation as part of your fault simulation.

Specifying the option *--potential_faults* during DFT fault simulation automatically enables four-state simulation during fault propagation.

Four-state propagation occurs when the VCD has a Z value on an input or an X value on one of the following:

- Outputs
- Flops
- Latches
- Blackbox outputs
- Start points
- End points

Design With Multiple Drivers

KaleidoScope supports designs with multiple drivers by recognizing and resolving situations where multiple instances drive the same net.

Restrictions and Limitations

- KaleidoScope supports multiple drivers only for Verilog and SystemVerilog designs. VHDL is not supported.
- KaleidoScope supports driving strength for Verilog Primitives only, not for continuous assign statements.

Procedure

Set the fusaini option multiple_drivers=true to enable support for multiple drivers.

Examples

The following example sets the required fusaini options to enable support for multiple drivers:

```
KaleidoScope /
--filelist inputs/synthesized_files.f
--clkdef inputs/top.clks
--sim_vcd inputs/vcd.f
--fault_list inputs/fault.list
--alarm inputs/alarm.list
--mode kmanager_single
--error_inject_inst tb_RAM.top
--top top
--ini memory_fault_propagation=true
--ini multiple_drivers=true
```

Run KaleidoScope

Running KaleidoScope to prove a design is safe is a multi-step process. Start by filtering the your simulation data, then run a fault campaign for stuck-at 0 and stuck-at 1 faults. Examine the results. When the KaleidoScope results indicate the design is safe, submit results to the SafetyScope tool to calculate final metrics.

Prerequisites

- Simulation data output (such as FSDB, VCD, or Qwave.db files). Validation relies on robust VCD files that reflect the safety context of the stimulus under which the fault is injected and which is representative of the operating condition of the system while performing a safety-critical function.

Procedure

1. Set up input to run the KaleidoScope tool as outlined in Table 6: Input to Run KaleidoScope and Table 7: Fusaini Initialization File.

Table 6. Input to Run KaleidoScope

Element	Argument	Argument Value
Simulate VCD dump list	--sim_vcd <file with a list of VCD files>	*Inputs_KS/vcd_filter.f*
Simulate FSDB file	--sim_fsdb <file with list of fsdb files> To specify a single file use the fusaini option, fsdb_file = <FSDB file>	*Inputs_KS/test.fsdb*
Error injection instance in VCD	--error_inject_inst vcd_dut_path	tb_axi_cross_top.u_dut.u_axi_cross
Top module name	--top top_mod	axi_cross_wrapper_ATD_wrapper_ATD_wrapper_ATD

Use a fusaini initialization file to set common options When used in a fusaini file, arguments and their parameters are delimited by an equals sign (=) with one or more spaces on each side of the equals sign.

```
mode                 = kmanager_single
uniquify             = 1
filelist             = Inputs_KS/axi_cross_safety.f
syn_to_flop          = 1
clkdef               = Inputs/common/axi_cross.clk
black_box            = Inputs/common/axi_cross.bbox
error_inject_inst    = tb_axi_cross_top.u_dut.u_axi_cross
max_fanout           = 30000
sim_start_time       = 150
FTTI                 = 10000
```

The double hyphen preceding each argument is not specified in the fusaini file as is required when these arguments are specified in a command line.

Table 7. Fusaini Initialization File

Element	Argument	Value
Running mode for KaleidoScope session.	mode	kmanager_single
Uniquify modules.	uniquify	1
Add design source files.	vfiles	*Inputs_KS/axi_cross_safety.f*
Value determines how a 2D array is interpreted. The value 1 treats the 2D array as flip-flops. See --syn_to_flop 0 | 1	syns_to_flop	1
File listing clock port names. See --clkdef clk_ports	ckldef	*Inputs/common/axi_cross.clk*
List of black box modules with the clock port names. See --black_box blackbox_list	black_box	*Inputs/common/axi_cross.bbox*
Verilog design file list. See --vfiles file_list	vfiles	*rtl/rtl_safe/axi_cross.f*
Cost function for High Cost Node in fault simulation. Specifies the maximum number of derived flip-flop events from the fault injection node. See --max_fanout cone_fanout	max_fanout	3000
Specifies an integer that is the start of simulation.	sim_start_time	150

Element	Argument	Value
See --sim_start_time integer		
Specifies the window size in VCD time. See --FTTI integer	FTTI	10000

2. Filter the VCD file to create a VCD file that contains only information that the KaleidoScope tool requires for fault simulation.

   Using the input setup in Table 6: Input to Run KaleidoScope, start by setting the KaleidoScope tool in filter VCD mode (--mode *vcd_filter*), and then simulate the VCD dump list:

```
KaleidoScope
  --mode              vcd_filter
  --top               axi_cross
  --fusaini           Inputs/common/fusa_safe.ini
  --error_inject_inst tb_axi_cross_top.u_axi_cross
                        _top.u_axi_cross
  --sim_vcd           Outputs/Func_Sim_Safe/vcd.f
  --vcd_out_name      wavedump_filtered
```

3. Validate VCDs to ensure the filtered VCDs contain the required content. Specify --mode *kmanager_single* and --ks_validate_vcd options:

```
KaleidoScope
  --mode              kmanager_single
  --top               axi_cross
  --fusaini           Inputs/common/fusa_safe.ini
  --ks_validate_vcd   1
  --error_inject_inst tb_axi_cross_top.u_axi_cross_top.u_axi_cross
  --fault_list Outputs/FIT_Compute_Safe/axi_cross.PermFault.nodes
  --sim_vcd           Outputs/Func_Sim_Safe/vcd.f
```

4. Run the fault campaign for stuck-at 0 and stuck-at-1 with the filtered VCD to classify each fault in the fault list as SAFE, UNSAFE, DETECTED, or UNRESOLVED.

   Use the input setup in Table 6: Input to Run KaleidoScope. Start with the KaleidoScope tool in fault campaign mode (--mode *kmanager_distributed*), then simulate the VCD file:

```
KaleidoScope
 --mode                 kmanager_distributed
 --top                  axi_cross
 --fusaini              Inputs/common/fusa_safe.ini
 --error_inject_inst    tb_axi_cross_top.u_axi_cross_top
                         .u_axi_cross
 --alarm                 Outputs/Safety_Synthesis/RD/AlarmList.txt
 --fault-list   Outputs/FIT_Compute_Safe/axi_cross.PermFault.nodes
 --sim_vcd              Outputs/Safety_Verification/VCD_Filtering/
                          vcd_filtered.f
 --max_concurrent_fault 50
```

Note:
Optionally denote the number of CPUs on which the distributed fault campaign runs by specifying the option "--kman_parallel integer" on page 99.

5. Review the reports listed in the table that follows:

Table 8. Fault Campaign Output Reports

Report	Description
Kman.AlarmFired.rpt	Reports faults which triggered alarm. (DETECTED)
Kman.AlarmNotFired.rpt	Reports of faults which did not trigger alarm. (UNSAFE)
Kman.NonTriggered..rpt*	Reports faults which were not triggered. (UNRESOLVED)
Kman.SafetyScope.rpt	Fault classification report that the SafetyScope tool analyzes to generate final metrics.

Related Topics

Fault Campaign Input Requirements

KaleidoScope Command

Fault and Alarm Output

KaleidoScope Co-Simulation With Questa SIM

KaleidoScope supports a Co-Simulation workflow with Questa SIM to enable fault propagation through non-synthesizable (black box) parts of a design. Without the Co-Simulation workflow, KaleidoScope cannot propagate faults through behavioral-level models such as a real number model representation of analog blocks. This topic explains the necessary steps to simulate non-synthesizable behavioral models with KaleidoScope.

Restrictions and Limitations

In the current release, KaleidoScope Co-Simulation does not support the following:

- SystemVerilog typedef types in port list of black box
- UDFM fault injection
- VHDL records in black box
- Parameterized modules as a black box
- Multithreading

Prerequisites

- Prepare to run a KaleidoScope fault simulation without the Co-Simulation workflow as described by Run KaleidoScope.
- Ensure that you are using Questa SIM version 2021.4 or later by setting the MODEL_TECH_OVERRIDE environment variable to your Questa SIM 2021.4 installation path.
- A FuSa Common database (File extension *.fdb*) containing fault list information. You can conduct a Co-Simulation workflow without relying on a FuSa Common database, but this is not recommended.
    - Refer to the Austemper Safety Scope User Guide for details on how to save fault list information to a FuSa Common Database
    - Refer to Reading and Writing From a Shared Database With KaleidoScope for additional information.

Note:
Co-Simulation is currently only supported when running KaleidoScope in kmanager_single mode.

Procedure

1. A Co-Simulation run should only target faults that reach black boxes during a fault campaign for efficiency reasons. To obtain a subset of these faults from a fault list, run KaleidoScope in kmanager_distributed mode with the appropriate simulation data as stimulus, and specify the non-synthesizable modules that are the motivation for the Co-Simulation flow in the black box list with the cosim_black_box option. For example:

    ```
    cosim_black_box = ./inputs/bbox.txt
    ```

    a. Specify the module name and clock name on a single line for each module in the Co-Simulation black box file:

    ```
    converter clk
    ```

    b. Save the fault campaign results to a session within a FuSa Common Database by specifying a Fusaini input file containing the following Fusaini options:

    ```
    write_fusa_db     = true
    fusa_db_name      = <db_name.fdb::session_name>
    overwrite_session = true
    ```

    c. Check how many faults propagated to a black box after the simulation completes by checking the Reached BlackBox row of the fault summary as follows:

    ```
    Total Number of Faults             :  282
    Detected                           :  12  (4.26%)
    Potential Detected                   :  0  (0.00%, PT credit = ...)
    Not Detected                       :  270  (95.74%)
    ```

```
   **  Controlled Observed Internal :  56
   **  High Fan Out                 :  0
   **  Reached BlackBox             :  106
   **  Reached Async Port           :  0
```

2. Generate the testbench for analog models. You can combine this step with step 1, but is described separately here for ease of explanation.

    a. Run KaleidoScope with the same setup as step 1, but with the addition of the following Fusaini options (specifying a different output directory for this step may help to organize your workspace):

        - --ini questa_cosim_tb_gen=true

        - --ini questa_cosim_en=true

        - --ini questa_tb_top=Cosim_tb

    When you run KaleidoScope with these options, KaleidoScope dumps a Co-Simulation testbench in the specified output directory named *<top>_KS_CoSim_blackbox.v*. The testbench top is the same as you specified with the --ini questa_tb_top option.

3. Compile and elaborate the design for Questa SIM simulation according to the following sub-steps:

    a. Create a script named *buildcmds* consisting of the following vlog and vopt commands:

```
vlog –work KSBBoxLib/ CosimTB.v -f design_filelist.f –sv –logfile
 vlog.log
vopt Cosim_tb +acc –o Cosim_tb_opt –work KSBBoxLib –logfile
 vopt.log
```

    b. Copy the Co-Simulation testbench to the directory where you will run the *buildcmds* script:

```
cp <Output directory>/<top>_KS_CoSim_blackbox.v CosimTB.v
```

    c. Ensure the relative paths for the design filelist work inside the directory where you will run the *buildcmds* script.

    d. Run the *buildcmds* script to compile and elaborate the design for Questa SIM simulation.

4. Run a fault simulation with Co-Simulation; Use the same setup as for step 1, except with the following adjustments:

    a. Change the KaleidoScope command --mode option from kmanager_distributed to kmanager_single.

    b. Specify the following Fusaini options as shown below:

```
--ini questa_cosim_en=true
--ini questa_tb_top=Cosim_tb
--ini questa_tb_dir=<precompiled design directory>
--ini read_faults_with_resolution=Reached_Cosim_BBox
```

    c. Save the results of this session to a new session within a database.

5. Merge the results stored in the database sessions from step 1 and step 4 with the fusautils utility:

   ```
   fusautils --mode merge --output_dir DB_Results --ini
    fusa_db_list=db.list --ini fusa_db_name=merged.fdb::KS_merged_results
   ```

   Refer to Reading and Writing from a Shared Database with KaleidoScope to review database merging.

6. Analyze the results:

   The log file for each KaleidoScope run captures the fault summary of the simulation activity, including the number of detected vs non-detected faults.

   For a comprehensive analysis of each detected fault node with respect to their actual failure rate contributions, specify the merged database from step 5 to SafetyScope to explore the Co-Simulation results.

   You can also generate a report of the fault campaign results from the merged database with the fusautils utility. For example:

   ```
   fusautils --mode report --output_dir DB_Results --fusa_db_name
    DB_Results/merged.fdb::KS_merged_results
   ```

   The topic Working with the fusautils Database Utility that follows provides additional details on generating reports.

Working With the fusautils Database Utility

KaleidoScope ships with a database utility named fusautils that facilitates merging FuSa Common Database sessions and that can also generate reports from data within a specified session.

Limitations

The following list summarizes the current limitations of the fusautils utility:

- No merging of faults between sessions with different top module, unless the instance in the primary database is given.
- Path delay faults are not yet supported.
- SafetyScope-generated database files are not yet supported in fusautils (except reporting of faults).

fusautils Options and Fusaini options.

- --mode - This required option specifies which mode to run the utility in:
  - merge - This mode merges database sessions into a new session. For example:

    ```
    fusautils \
    -mode merge
    ```

- report - This mode generates text reports based off the data stored in a specified session. For example:

  ```
  fusautils \
  -mode report
  ```

- fusa_db_name = <*db_name.fdb*>::<*session*> - Specifies the database and session name of the merged database that results from running fusautils in merge mode.
- fusa_db_list = <*path*>/<*db_list_file*> - Specifies a file containing a list of sessions within databases that fusautils will merge together. This option only pertains to fusautils merge mode.
- --output_dir = <*path*>/<*directory*>

  This option controls where fusautils will generate output. The fault output directory is *./Outputs*
- --fusaini - This option specifies a file containing Fusaini options. You can use this file to specify which database sessions to read and write from, as well as different types of data to select from within a session.

Fusaini Options

You can specify Fusaini options in a file with the --fusaini option described in the previous section. The primary Fusaini options used with fusautils include:

- fusa_db_name = <*db_name.fdb*>::<*session*> - Specifies the database and session name of the merged database that results from running fusautils in merge mode.
- fusa_db_list = <*path*>/<*db_list_file*> - Specifies a file containing a list of sessions within databases that fusautils will merge together. This option only pertains to fusautils merge mode.

Merging Sessions

Merging results contained in multiple databases into a single database is an important step in certain workflows. For example, in a Co-Simulation workflow, you merge Co-Simulation results corresponding to non-synthesizable blocks in a design with other results from a standard KaleidoScope fault campaign.

The fusautils utility supports merging results from two sessions into a single database session as shown by the example:

```
fusautils --mode merge --output_dir DB_Results --ini fusa_db_list=db.list
 --ini fusa_db_name=merged.fdb::KS_merged_results
```

- Specify the --mode merge option to the fusautils command.
- Specify the output_dir option to the fusautils command to control the output directory for the merged database.
- Specify a list of share databases to merge together with the --fusa_db_list Fusaini option. The database list file must contain a list of databases and sessions according to the form:

```
<DB_path>/<DB_name.fdb::session_name_0>
<DB_path>/<DB_name.fdb::session_name_1>
```

For example:

```
./Outputs_kman_db/cosim.fdb::KS_results
./Outputs_kman_db/cosim.fdb::KS_cosim_results
```

You can also merge sessions from different databases, for example:

```
./Outputs_kman_db/db_a.fdb::KS_results_0
./Outputs_kman_db/db_b.fdb::KS_results_1
```

- Specify the name for the database resulting from the merge with the --fusa_db_name Fusaini option.

Note:
To merge between sessions with different specified top levels you must also include the instance name when specifying the session where a child module is equivalent to the top level of a different session. For example, where a design has a top module named top1_module in DB::session1 and another top module named child_module in DB::session2, pass the instance name in DB::session2, which is the child module in this case. That corresponds to the same module as the one in DB::session1:

```
DB::session1
DB::session2 child_module_inst
```

Generating Reports

You can generate reports based on the data stored in a database session with the fusautils --report mode, as shown by the example:

```
fusautils --mode report --output_dir DB_Results --fusa_db_name
 DB_Results/merged.fdb::KS_dig_results
```

You must specify a single database session for reporting. If you want to include the results from multiple sessions in a report generated by fusautils, merge the sessions together and report from the final merged database session.

Debug Results

Debugging fault issues can be time consuming. Reviewing results from a KaleidoScope run may require you to resolve a dangerous fault, or debug an unexpected undetected fault. This section provides some tasks to help you analyze VCD files to debug and resolve issues.

Debugging With Visualizer

KaleidoScope provides support to assist you in debugging with Visualizer, as explained by these topics.

- Qrun Script for Debugging With Visualizer
- Writing Qwave.db Files

Dump VCD

Create a VCD file that dumps a fault to a VCD file. Use the *--create_vcd_dump 1* option on the command line or in a fusaini configuration file (create_vcd_dump=true) to create the VCD file.

Restrictions and Limitations

- The *vcd_create_dump* option dumps only one fault for each VCD file. If multiple faults exist, specify the fault to be dumped. Not specifying a fault when multiple faults exist causes the process to return the following error message and stops processing.

```
** Error: (fusa-97) VCD Dump is not generated because multiple
** faults exist.
** : Reduce the number of faults to 1.
```

Prerequisites

- Set up the required input files:
    - design files (*axi_cross_safety.f*)
    - alarm list (*AlarmList.txt*)
    - blackbox list (*axi_cross.bbox*)
    - clock safety file (*axi_cross.clk*)
    - error injection instance (*tb_axi_cross_top.u_dut.u_axi_cross*)

- error injection nodes (*fault.list*)
- top module (*axi_cross_wrapper_ATD_wrapper_ATD_wrapper_ATD*)

Procedure

1. Run KaleidoScope with specified inputs, the *--create_vcd_dump 1* option and other relevant options:

```
KaleidoScope
  --mode kmanager_single
  --filelist ./Inputs_KS/axi_cross_safety.f
  --clkdef ./Inputs_KS/axi_cross.clk
  --syn_to_flop 1
  --black_box ./Inputs_KS/axi_cross.bbox
  --error_inject_inst  tb_axi_cross_top.u_dut.u_axi_cross
  --sim_vcd ./Inputs_KS/vcd_filter.f
  --top axi_cross_wrapper_ATD_wrapper_ATD_wrapper_ATD
  --alarm ./Inputs_KS/AlarmList.txt
  --fault_list ./Inputs_KS/fault.list
  --create_vcd_dump 1
  --log_file KS_vcd_dump.log
```

Note:
Access this example with the associated design and input files from the *$FUSA_HOME/share/examples/SafeVerification/VCD_DUMP* directory distributed with the software release.

2. Review the *Faults Summary* in the log file, *KS-Vcd_dump.log*.

```
Faults Summary
-------------------------------------------------------------------
Total Number of Faults       :  1
Number of Faults Resolved    :  1
          Alarm Detected     :  0
          Alarm NotDetected  :  0
          High Fan Out       :  0
          Reached BBox       :  1
          Reached Observe    :  0
          Missing Sim_Data   :  0
Fault Threshold Number       :  1
Fault Threshold Percentage   :  0
-------------------------------------------------------------------
```

Results

KaleidoScope writes the VCD file to your Outputs directory with the name *ATD_Fault_dump.vcd*.

Compare VCDs

Compare two VCD files using the *vcd_compare* mode option to compare the files on an endpoint-to-endpoint basis at each tick of the VCD.

Restrictions and Limitations

- The VCDs must be from the same design.
- VCDs must have the same hierarchy to the top design (*error_inject_inst*).

Prerequisites

- Create VCDs.
- Set up the required input files:
  - design files (*axi_cross_safety.f*)
  - alarm list (*AlarmList.tx*t)
  - black box list (*axi_cross.bbox*)
  - clock file (*axi_cross.clk*)
  - Instance for error injection (*tb_axi_cross_top.u_dut.u_axi_cross*)
  - top module (*axi_cross_wrapper_ATD_wrapper_ATD_wrapper_ATD*)
  - golden file (*GOLD.vcd*)
  - VCD filtered file listing the two VCD files to be compared (*vcd_filter.f*)

Procedure

1. Run the VCD comparison with the following input and options:

```
KaleidoScope
  --mode vcd_compare
  --filelist ./Inputs_KS/axi_cross_safety.f
  --clkdef ./Inputs_KS/axi_cross.clk
  --error_inject_inst  tb_axi_cross_top.u_dut.u_axi_cross
  --sim_vcd ./Inputs_KS/vcd_filter.f
  --top axi_cross_wrapper_ATD_wrapper_ATD_wrapper_ATD
  --log_file KS_vcd_compare.log
```

Restriction:
When specifying *--sim_vcd*, the first file specified is always the *golden* file, the file to which the second file is compared. See the *vcd_filter.f* file in this example:

```
Inputs_KS/GOLD.vcd
Inputs_KS/ATD_Fault_dump.vcd
```

The VCD comparison does a bitwise comparison of buses. Set the *--vcd_compare_bitwise* option to zero to disable bitwise comparison of buses and do a full signal comparison as defined in the VCD header.

Note:
Access this example with the associated design and input files from the *$FUSA_HOME/share/examples/SafeVerification/VCD_COMPARE* directory distributed with the software release.

2. Review the log file and the VCD comparison file, *CompareVCD.txt.vcd*, which contains the VCD comparison information and is in the Outputs directory.

   In the following sample *CompareVCD.txt.vcd*, an ellipsis (...) truncates the fault name to allow the example to fit within the column space.

```
 ErrVal GoldVal ErrNetName
# 0
# 5
# 10
# 15
# 20
# 25
# 30
# 35
# 40
# 45
# 50
0 1 ...u_axi_cross_wr_ch/axi_m_bid_st[7]
0 1 ...u_axi_cross_wr_ch/axi_m_bid_st[6]
0 1 ...u_axi_cross_wr_ch/axi_m_bid_st[5]
0 1 ...u_axi_cross_wr_ch/axi_m_bid_st[4]
0 1 ...u_axi_cross_wr_ch/axi_m_bid_st[3]
0 1 ...u_axi_cross_wr_ch/axi_m_bid_st[2]
0 1 ...u_axi_cross_wr_ch/axi_m_bid_st[1]
0 1 ...u_axi_cross_wr_ch/axi_m_bid_st[0]
0 1 ...u_axi_cross_wr_ch.AustempPGC_5/DParReg
# 55
0 1 ...u_axi_cross_wr_ch/axi_m_bid_st[7]
0 1 ...u_axi_cross_wr_ch/axi_m_bid_st[6]
0 1 ...u_axi_cross_wr_ch/axi_m_bid_st[5]
0 1 ...u_axi_cross_wr_ch/axi_m_bid_st[4]
0 1 ...u_axi_cross_wr_ch/axi_m_bid_st[3]
```

```
0 1 ...u_axi_cross_wr_ch/axi_m_bid_st[2]
0 1 ...u_axi_cross_wr_ch/axi_m_bid_st[1]
0 1 ...u_axi_cross_wr_ch/axi_m_bid_st[0]
0 1 ...u_axi_cross.u_axi_cross_wr_ch.AustempPGC_5/DParReg
```

Chapter 3
Reference

This chapter contains notational conventions, command options, and a glossary of terminology used in this user guide.

Notational Conventions

Notation used in the Austemper KaleidoScope User Guide follows a basic set of conventions. These notational conventions include an extended Bakus-Naur standard syntax notation that incorporates metasyntax notation.

Table 9: Notational Conventions shows basic notational conventions used throughout the document. Table 10: Syntax Conventions shows syntax conventions used for command syntax.

Table 9. Notational Conventions

Notation	Description	Example
italics	In syntax statements: *oblique font* indicates a variable.	[-depth *cycles*]
	In text: *oblique font* indicates: (1) variable (2) code excerpt (3) term being defined	• Specify the black box as *module*. • Both *tx_sig1* and *tx_sig2* converge at *rx_sig*. • A *portconstraint* is a restriction on the clock domains of signals...
<italics in angle brackets>	Italics in angle brackets are used in text to distinguish variables from literals when necessary to reduce confusion.	Specify the reconvergence depth: *-log_file <filename>*.
<angle brackets>	Angle brackets are used in alphanumeric messages from the software to indicate variables.	cdc report scheme <scheme>
italics underline	*Oblique underline font* in text indicates the name of another document.	See the *Questa SIM User Guide* for details about the *vlog* command.

Table 10. Syntax Conventions

<table>
<tr><th>Meta-symbol</th><th>Description</th><th>Example</th></tr>
<tr><td>. . .</td><td>Ellipses indicate a repeatable entry.</td><td>–values value. . .
=> –values ‘b00 ‘b10 ‘b01</td></tr>
<tr><td>[ ]</td><td>Square brackets indicate an optional entry.</td><td>[–black_box blackbox_list]</td></tr>
<tr><td>{ }</td><td>Regular braces indicate a required entry (often used with or-bars or ellipses).</td><td>{-set signal value}...
=> –set sig1 3 –set sig2 4 –set sig5 2</td></tr>
<tr><td>|</td><td>Or-bars separate choices in [ ] and { } entries.</td><td>{violation | caution | evaluation}
=> violation</td></tr>
<tr><td>{ }</td><td>Bold braces are literals to be typed as they appear.</td><td>–prefix_modules {module…}
=> –prefix_modules {A B C D}</td></tr>
</table>

The following replaceable variables in command syntax statements represent the shown object types:

- *signal*: Single-bit register or wire.
- *variable*: Expression that can change value at any time.
- *constant*: Expression that evaluates to a statically constant value.
- “*string*”: String enclosed in double-quotes.

When specifying a command from a command syntax statement, substitute for each meta-variable an HDL identifier, or an expression enclosed in braces, that evaluates to an object of the corresponding type.

Note:
Signals and instance names in VHDL logic and mixed VHDL/SystemVerilog designs use case-insensitive matching because of inferred case insensitivity support.

KaleidoScope Command

This section outlines KaleidoScope command syntax and options. All command options are listed alphabetically in the *Arguments* section.

Syntax

KaleidoScope --mode mode --clkdef clk_ports
{--filelist file | --vfiles file_list | --vfile v_design | --vhdl_file file_list}
[--top top_mod] [--alarm list] [--black_box_inst blackbox_list] [--black_box blackbox_list]
[--create_vcd_dump integer] [--debug_fault [faultspec]] [--design_lib_name design_lib]
[--dft_observe_points filename] [--enable_design_libs 0 | 1] [--eq_mapping_file file]
[--error_inject_inst vcd_dut_path] [--fault_list file] [--fault_file_list file] [--filelist_sort file]
[--FTTI integer] [--fusaini setup_file] [--help] [--ini option_name=option_value]
[--inihelp] [--kman_parallel integer] [--koutname Krpt_string_append]
[--ks_validate_vcd file 0 | 1] [--ks_op_to_alarm = time] [--lib_cell_list file[--log_file file]
[--max_concurrent_fault integer] [--max_fanout cone_fanout] [--mode mode]
[--output_dir directory] [--observe_points filename] [--pdf_file_list pdf file]
[--potential_faults optimistic | pessimistic] [--report_internal_names integer]
[--sim_qwave <qwave.db file>] [--sim_duration_time integer] [--sim_end_time integer]
[--sim_fsdb <file with list of fsdb files>] [--sim_start_time integer]
[--sim_vcd <file with a list of VCD files>] [--sv2009 0 | 1] [--syn_to_flop 0 | 1]
[--transient_window integer] [--udfm_models_list file] [--uniquify 0 | 1]
[--vcd_compare_all_eps 1 | 0] [--vcd_compare_bitwise 1 | 0] [--vcd_out_name string]
[--version] [--vhdl_std vhdl_1987 | vhdl_1993 | vhdl_2008] [--vlibcell library_cell_list]

Note:
Options appear in alphabetical order, regardless of case, in the *Arguments* section.

Arguments

- --alarm *list*

  File specifies the name of the alarm list. Type: string.

- --black_box_inst *blackbox_list*

  Specifies a file containing a list of instance names to black box along with clock names (clock can be NULL). This is different than the option for specifying a list of black box modules, which is --black_box

  Example:

  ```
  BBOX_INSTANCE_NAME CLK
  ```

- --black_box *blackbox_list*

  Blackbox module list of blackbox modules with their clock port names. Type: filename.

  Example:

  ```
  BBOX_NAME CLK
  ```

Note:
Only one primary clock is supported for a black box.

- --clkdef *clk_ports*

  File that lists top-level clock port names. Derived clocks are allowed, as shown in the example. Type: filename.

  Example file (*./inputs/clkdef.txt*)

```
## clkdef.txt
top_clk
top.A.B.derived_clk
## top_clk is found at top level of design.
## "derived_clk" is found in hierarchy at
"top.A.B"
```

- --create_vcd_dump *integer*

  Creates VCD dump. Type: integer. Default: 0.

- --debug_fault [*faultspec*]

  Debugs a single fault from the input fault list. If you do not provide the optional fault specification *faultspec*, this option matches the first fault from the fault set you provided. See Example Fault Lists for information on how to specify a fault.

- --design_lib_name *design_lib*

  Specifies the design library name. Type: string.

- --dft_observe_points *filename*

  Specifies the name of a file that contains a list of DFT observe points. Type: filename.

- --enable_design_libs 0 | 1

  When 1, this option supports multiple design libraries. Default:0. Type: integer.

- --eq_mapping_file *file*

  Use this option when running gate-level netlist fault campaigns with RTL simulation data as stimulus. The mapping file maps gate level netlist signals to their equivalent RTL signals. Each line in the mapping file has the following format:

```
<gate signal>  <rtl signal>
```

- --error_inject_inst *vcd_dut_path*

  String for vcd_dut_path specifies the hierarchical name of the DUT in the testbench. This is the DUT instance name in the given VCD. Type: string.

- --fault_list *file*

  Specifies a fault file with a list of hierarchical names of signals to inject with faults, with one signal per line. Port faults are injected by default.

  To specify more than one fault file, use *--fault_file_list file*.

- --fault_file_list *file*

Specifies a file that contains a list of fault file names. Those fault files specify hierarchical names of signals to inject with faults, with one signal per line. Port faults are injected by default.

To specify a single fault file, use *--fault_list file*.

- --filelist *file*

Questa list parser. Files must be listed in the order required for accessing and processing the files listed. Table 11: File Parsing Options for filelist and filelist_sort lists file parsing options supported for this --filelist option.

Table 11. File Parsing Options for filelist and filelist_sort

Option	Description
+define	Definition of MACROs to be used within the RTL.
-f / -F / -file	Specifies a file that contains a list of pathnames to source files and compile-time options.
+incdir	Specifies the directories that contain the files specified with the `include compiler directive. More that one directory can be specified if each pathname is separated with the + character.
+libext	Specify suffix of files in library directory.
-sverilog	Enables the use of the Verilog language extensions in SystemVerilog specification.
+systemverilogext	Specifies a filename extension for source files containing SystemVerilog source code.
-u	Changes all characters in identifiers to uppercase.
-v	Specify Verilog source library file.
+v2k	Enables the use of new Verilog constructs in the 1364-2001 standard.
+verilog1995ext	Specifies a filename extension for source files containing Verilog 1995 source code.
+verilog2001ext	Specifies a filename extension for source files containing Verilog 2001 source code.
-y	Specify Verilog source library directory.

- --filelist_sort *file*

Files listed in this file are sorted automatically such that files and any associated libraries are accessed in the required order. Reference Table 11: File Parsing Options for filelist and filelist_sort for file parsing options supported by *--filelist_sort*. Type: filename.

- --FTTI *integer*

Specifies window size in VCD time. This is the time for the alarm to detect the fault after a failure. If the fault propagates to a state element and does not reach the alarm within the FTTI, then KaleidoScope does not detect the fault, hence the fault is dangerous. Type: Integer. Default: -1, which indicates the full VCD.

- --fusaini *setup_file*

  Specifies an initialization setup file containing command options. Using the *fusaini* initialization file not only documents the options for a specific run, but also provides a flexible way to modify options and more efficiently rerun the tool instead of typing all the options on the command line.

  You can also set the environment variable FUSAINI to point to the fusaini initialization file.

  Restriction:
  Some options do not work within a fusaini initialization file. The fusaini file does not support the --log_file file option. It does not work from within a fusaini file.

  ◦ See the following *fusa_setup.txt* initialization file example.

```
# Example initialization file: fusa_setup.txt
top          = dut
syn_to_flop  = 1
mode = kmanager_single
alarm        = AlarmList.txt
clkdef       = ./inputs/axi.s
filelist     = ./inputs/files.f
```

  Applying the options from this fusaini initialization file requires only this command:

```
prompt> KaleidoScope –fusaini fusa_setup.txt.
```

  This *fusaini* option and initialization file is much more efficient than retyping this command at the command line to rerun a session:

```
prompt> KaleidoScope --mode kmanager_single
                     --filelist ./inputs/files.f
                     --alarm AlarmList.txt
                     --clkdef ./inputs/axi.s
                     --syn_to_flop 1
                     --top dut
```

  ◦ Enable the fusaini initialization file by setting the FUSAINI environment variable:

```
Setenv FUSAINI <path>/my.ini
```

  The tool reads the *my.ini* file in the previous example as if it were passed in the command line. When the FUSAINI environment variable is set, it is parsed first *before* any other fusaini file; these values can be overridden by a subsequent fusaini file.This can be useful to run a set of tests/regressions that run non-default options without changing the fusaini file for each test.

KaleidoScope can read and use environment variables specified in the fusaini initialization file. You can specify the environment via the Fusaini *mode* option in one of the supported formats:

- $VARNAME
- ${VARNAME}
- $(VARNAME)

See the following *fusa_setup.txt* initialization file example

```
# Example initialization file with environment variable
ks_num_threads  = 4
ks_multi_thread = true
kman_parallel   = 4
mode            = ${VARNAME}
```

- --help

  Displays a list of arguments and descriptions.

- --ini *option_name*=*option_value*

  Specifies a fusaini file option pair at the command line using the format:

  --ini *option_name*=*option_value*

  Multiple *--ini* options can be specified. All *--ini* option-value pairs are processed after all fusaini files, but before other command line options. Specifying the *--ini* option-value pairs on the command line allows previously specified fusaini file options to be overridden, which is useful for overriding centralized fusaini file options.

  See also “--fusaini setup_file” on page 98.

- --ini dft_simulate_internally_observed *true* | *false*

  Set the value of this option to true to continue simulation of DFT Controlled_Observed_Internal faults (see “KaleidoScope DFT Fault Resolution” on page 39) across multiple simulation data files. Simulation continues in DFT kmanager_distributed mode until all faults are detected.

  Default: false. Simulation ends for faults resolved to Controlled_Observed_Internal in DFT KaleidoScope distributed mode.

- --ini simulate_residual *true* | *false*

  Set the value of this option to true to continue simulation of Undetected_Observed faults (see “Specifying Observe Points” on page 37) across multiple simulation data files. Simulation continues in DFT kmanager_distributed mode until all faults are detected.

  Default: false. Simulation ends for faults resolved to Undetected_Observed faults in KaleidoScope distributed mode.

- --inihelp

  Prints fusaini options and descriptions.

- --kman_parallel *integer*

  Enables and sets the number of parallel runs for a KaleidoScope distributed run. Use this option with the --mode *kmanager_distributed* option to specify parallel simulations across multiple CPUs. Type: Integer. Default: 1.

- --koutname *Krpt_string_append*

String appended to the fault campaign report results. This string is appended to the output: [*modulename*_<*KrptString*>.Krpt]. Ideally this should be the testcase name from which the simulation VCD is generated. Input is "*string*". Type: string. Default: KMAN outputs [*modulename*.Krpt].

- --ks_validate_vcd file *0* | *1*

  Validate given VCD. Recommend using this option when running with --mode *vcd_filter*. Type: integer.

- --ks_op_to_alarm = *time*

  Specifies the amount of time that KaleidoScope simulates a fault after it propagates to a primary output port. Time is an integer number and acceptable time units are sec, ms, us, ns, ps. For example:

```
ks_op_to_alarm = 2560ps
```

CAUTION:
If you specify a fractional number, KaleidoScope truncates the time value.

By default, KaleidoScope uses the value you specify with --FTTI integer; if FTTI is not set, the default is *256000* with the same time unit used in the stimulus file.

- --lib_cell_list *file*

  Specifies a file containing a list of modules to treat as library cells. Specify this option when you have library cells, but no accompanying liberty file. The library cell list file format specifies a space-separated module/value pair on each line. The first column contains module names, the second column takes one of three values:

  - 0 = Combinational Logic
  - 1 = Flop/Register
  - 2 = Latch

  For example:

```
EDFQD1BWP12T30P140 1
EDFQD2BWP12T30P140 1
EDFQD4BWP12T30P140 1
FA1D0BWP12T30P140 0
FA1D0BWP12T30P140 0
```

- --log_file *file*

  Specifies the name of log file. Type: string.

Restriction:
This option cannot be specified in a fusaini initialization file.

- --max_concurrent_fault *integer*

Specifies the maximum number of concurrent faults per run. Type: Integer. Default: 150.

• --max_fanout *cone_fanout*

Defines the maximum cone fanout determined by the maximum number of derived flip-flop events from the fault injection node. Specify an integer value for the maximum number of second order errors to define the maximum fanout. Type: integer. Default: 99.

• --mode *mode*

Runs KaleidoScope tool in a specific mode based on the option specified, as shown in the table that follows.

Table 12. mode Options and KaleidoScope Actions

Mode	Action
generate_fault_list	Generates a fault list.
kmanager_distributed	Fault campaign manager mode session that can run on one or more CPUs.
kmanager_single	Fault campaign mode that runs multiple fault campaigns in one session.
report_design_stats	Displays design statistics.
sim_grade	Grades VCD or FSDB files with given fault list for design.
vcd_compare	Compares two VCD endpoints of the design.
vcd_filter	Filters VCD and retain only the endpoints of the design.
vcd_mapping	Maps RTL Structs, UnpackedArrrays to VCD signals.
$VARNAME	Uses environment specified.

• --observe_points *filename*

Specifies the name of a file that contains a list of observe points. Type: filename.

• --output_dir *directory*

Specifies an alternative output directory to the default *Outputs* directory. Type: directory name. Default: Outputs.

• --pdf_file_list *pdf file*

Specifies a path delay file. Without proper timing, the file results in an error/warning. This argument requires that you also set --ini dft=true.

• --potential_faults *optimistic* | *pessimistic*

Classifies potentially detected faults and turns on X propagation. Type literal. Default: optimistic.

◦ optimistic — X value on Alarm result in PD.

◦ pessimistic — X value on Alarm result in UO.

• --report_internal_names *integer*

Reports internal nodes in reports output when KaleidoScope runs in kmanager_distributed mode. Type integer. Default: 1.

- --sim_duration_time *integer*

  Window of simulation. Specifying -1 is the total VCD time. Default: -1.

- --sim_end_time *integer*

  End time of simulation. Default: LLONG_MAX.

  LLONG_MAX, the maximum value for an object of type long long int, can be 9233372036854775807 (2^{63}-1) or greater.

- --sim_fsdb <file with list of fsdb files>

  Specifies a file with a list of FSDB files for input. If you have only a single file for input, you can use the *fsdb_file=<FSDB file>* fusaini option instead. If you specify both the --sim_fsdb option and the fsdb_file fusaini option, the fusaini option takes precedence.

  You must specify the verdi_install_path=$(VERDI_HOME) fusaini option to use this option.

- --sim_qwave <qwave.db file>

  Specifies a file containing *qwave.db* file names with simulation data for fault campaigns the file can contain multiple *qwave.db* files, each on a separate line.

  - Lines in this file that begin with the # character are interpreted as comments and ignored.

  - You can specify simulation parameters in space separated columns after the name of each *qwave.db* file, see Reading Qwave.db files as Simulation Data for additional detail.

- --sim_start_time *integer*

  Denotes the start of simulation. Type: integer. Default: 0.

  The KaleidoScope tool starts simulation at the specified *sim_start_time*. If an error is injected before *sim_start_time,* the error is injected at *sim_start_time*. If an error is injected after *sim_start_time,* the error is injected at the time specified in the fault list.

- --sim_vcd <file with a list of VCD files>

  Specifies a file with VCD filenames from the RTL or GLS simulation waveforms.

- --sv2009 0 | 1

  When set (1), enables SV2009 compilation. Default: 0.

- --syn_to_flop 0 | 1

  The value 0 treats memories as memory. The value 1 analyzes memory structures as a 2D array of flip-flops. Default: 0.

- --top *top_mod*

  String specifies top module name. Required. Type: string.

- --transient_window *integer*

  Specifies the transient window time when running a fault campaign in distributed mode (--mode *kmanager_distributed*). Default: -1.

- --udfm_models_list *file*

Specifies the filename of a file that contains a list of UDFM model names. Type: filename. This argument requires that you also set --ini dft=true.

- --uniquify 0 | 1

  When 1, uniquifies the design during read. Default: 0.

- --vcd_compare_all_eps *1* | *0*

  Compares all endpoints in each time tick (only in vcd_compare mode; see “Table 12: mode Options and KaleidoScope Actions” on page 101). Type integer. Default: 1.

- --vcd_compare_bitwise *1* | *0*

  Performs a bitwise compare of buses (only in vcd_compare mode; see “Table 12: mode Options and KaleidoScope Actions” on page 101). Default: 1.

- --vcd_out_name *string*

  Output name for filtered VCD. Specify only a string without a file extension. The KaleidoScope tool appends the string specified to *_n.vcd* in which *n* is an assigned number and creates the output file *<string>_n.vcd* in the *Outputs* directory. File numbers are assigned starting with 0 and increment in value based on the order that the file is created and the number of existing files. You can override the placement of the this VCD file to the *Outputs* directory by specifying *output_dir* with a new directory path in the *fusaini* file. Type: string. Default: *./Outputs/<string>_n.vcd*

- --version

  Prints version information and then exits as shown:

  # Version 2019.2-external linux_x86_64 10 Jun 2019

- --vfile *v_design*

  Verilog design file. Type: filename.

- --vfiles *file_list*

  Verilog design file list. Type: filename.

- --vhdl_file *file_list*

  File contains VHDL file list. Type: filename.

- --vhdl_std *vhdl_1987* | *vhdl_1993* | *vhdl_2008*

  Applies the specified VHDL standard *vhdl_1987* | *vhdl_1993* | *vhdl_2008*. Type: string. Default: *vhdl_1993*.

- --vlibcell *library_cell_list*

  File that lists the library cells. Type: filename.

KaleidoScope Fusaini Options

Fusaini options are a superset of KaleidoScope command options.

Syntax

Fusaini Initialization File Format

option_name = *option argument*

Command Line

--ini *option_name*=*option_argument*

Note:
Syntax differs based on whether you specify the option in an initialization file with the --fusaini option or on the command line with the --ini option.

- Fusaini Initialization File: delimit the option name and parameter with an equals sign (=) that includes one or more spaces on each side of the equals sign.
- Command Line: Specify the --ini option followed by a fusaini option-argument pair denoted by an equals sign without spaces on either side.

[alarm = list] [black_box = blackbox_list] [bbox_wildcard] [clkdef = clk_ports] [combo_loop_iterations = integer] [cosim_black_box filename] [cosim_questa_args = <Questa arguments>] [create_vcd_dump = integer] [debug_fault [faultspec]] [debug_qwave_stim = integer] [default_fault_injection_value = value] [default_TDF_duration = integer] [default_tessent_udfm_type = string] [design_lib_name = design_lib] [dft = true | false] [dft_observe_points = string] [dft_report = true | false] [distribute_cmd = LSF command | SGE command] [distribute_cmd_attempts = <N>] [enable_design_libs = true | false] [enable_observe_nodes = true | false] [error_inject_inst = vcd_dut_path] [fault_exclude_inst = file] [fault_exclude_sig = file] [fault_list = filefile] [fault_file_list file] [filelist = file] [filelist_sort = file] [fsdb_file = <FSDB file>] [FTTI = integer] [generate_design_bin_script = true | false] [hypertrophic_sim_multiple = <integer>] [hypertrophic_deviation_percent = <integer>] [ignore_translate_off = true | false] [kman_parallel = integer] [ks_async_sim = true | false] [ks_count_num_deviations = true | false] [ks_fault_collapse = true | false] [ks_good_machine = true | false] [ks_memory_module_format = false | true] [ks_multi_thread = false | true] [ks_num_threads = integer] [ks_op_to_alarm = clock_cycles] [lib_cell_mapping = true | false] [liberty_file = file] [liberty_file_list = file] [map_fault_class = fault_resolution new_fault_class_name] [max_concurrent_fault = integer] [max_fanout = cone_fanout] [mem_init_file = filename] [memory_data_file_has_explict_addresses = true | false] [memory_fault_propagation = true | false] [merge_safesim_reports = true | false] [message_severity = message severity] [mode = mode] [multiple_drivers = true | false] [outputs_are_observe = true | false] [parser_messages = true | false] [potential_faults = optimistic | pessimistic] [potential_credit = integer] [preserve_user_nets = true | false] [print_blackbox_rpt = true | false] [print_sim_interval = -1 | interval] [questa_cosim_en = true | false] [questa_cosim_tb_gen = true | false] [questa_tb_top = <testbench_top>] [questa_tb_dir = <path>/<tb_directory>] [qwave_file = <path>/<qwave.db>] [race_check = true | false] [rand_fault_select = integer | <coverage_goal> : <confidence_interval>] [randomize_faults = true | false] [randomize_faults_seed = integer] [report_internal_names = true | false] [restore_design = <path>/<ATDdb_directory>] [save_design = <path>/<ATDdb_directory>] [save_design_libs = true | false] [sim_duration_time = integer] [sim_end_time = integer] [sim_force_list = filename] [sim_start_time = integer] [simulate_combo_loops = false | true] [sim_fsdb <file with list of fsdb files>] [sim_qwave <qwave.db file>] [sim_vcd = sim_vcd] [sv2009 = 0 | 1] [syn_to_flop = 0 | 1] [tessent_udfm_fault_list = udfm_instances_fault_list] [tessent_udfm_fault_type =

all | delay | static] [tessent_udfm_node_faults = true | false] [tessent_udfm_spec_files = cell_udfm_definition_list] [top = top_mod] [transient_window = integer] [udfm_models_list = file] [uniquify = 0 | 1] [vcd_compare_all_eps = 0 | 1] [vcd_name_print] [vcd_out_name = string] [verdi_install_path=$(VERDI_HOME)] [vfile = verilog_design] [vfiles = file_list] [vhdl_file = file_list] [vlibcell = library_cell_list]

Reference "Specify Fusaini Options" on page 45 for details about specifying fusaini options. Fusaini options are listed alphabetically in the Arguments section.

Arguments

- alarm = *list*

  File specifies the name of the alarm list. Type: string.

- black_box = *blackbox_list*

  Blackbox module list of blackbox modules with their clock port names. Type: filename.

  Example:

  ```
  BBOX_NAME CLK
  ```

  Note:
  Only one primary clock is supported for a black box.

- bbox_wildcard

  Specify this fusaini option to enable KaleidoScope to locate the modules associated with each black box instance.

- clkdef = *clk_ports*

  File that lists top-level clock port names. Derived clocks are allowed, as shown in the example. Type: filename.

  Example file (*./inputs/clkdef.txt*)

  ```
  ## clkdef.txt
  top_clk
  top.A.B.derived_clk
  ## top_clk is found at top level of design.
  ## "derived_clk" is found in hierarchy at
  "top.A.B"
  ```

- combo_loop_iterations = *integer*

  Specifies the number of iterations to continue simulating a combinational loop to see if it stabilizes. Default: 200.

- cosim_black_box *filename*

  Specifies a file containing a list of non-synthesizable modules to black box during Co-Simulation. Specify each module name and clock name on a single line for each module in the Co-Simulation black box file.

- cosim_questa_args = <Questa arguments>

  Specifies extra arguments passed to Questa for Co-Simulation. Use this syntax with the --ini option and separate the arguments with spaces.

- create_vcd_dump = *integer*

  Create VCD dump. Type: integer. Default: 0.

- debug_fault [*faultspec*]

  Debugs a single fault from the input fault list. If you do not provide the optional fault specification *faultspec*, this option matches the first fault from the fault set you provided.

- debug_qwave_stim = *integer*

  Controls the number of debug messages for *qwave.db* stimulus. A value of zero results in no debug messages. The maximum value, corresponding to the maximum amount of debug messages, is 3.

- default_fault_injection_value = *value*

  User-specified default for fault injection value. Type: string.

  If unspecified in the *fault_list_file*, the default_fault_injection_value is used to inject faults. If the default_fault_injection_value is not specified and the fault injection value is not identified in the *fault_list_file*, the tool uses a default fault injection value of SA1.

  For example, the following injects both stuck-at 1and stuck-at 0 faults:

  ```
  default_fault_injection_value=SA1:SA0
  ```

  This example injects a stuck-at 0 fault:

  ```
  default_fault_injection_value=SA0
  ```

- default_TDF_duration = *integer*

  Specifies the default for Time Delay Fault duration. Type: long long integer. Default: 0.

- default_tessent_udfm_type = *string*

  User-specified default for Tessent UDFM type for faults. Type: string. Default: intra_cell_defects

- design_lib_name = *design_lib*

  Specifies the design library name. Type: string.

- dft = *true* | *false*

  Enables or disables DFT fault grading. Type: boolean. Default: false.

  - true — Enables fault grading.
  - false — Disables fault grading.

- dft_observe_points = *string*

  Specifies a file containing a list of observe points in the DFT flow. Type: string.

  You must specify the dft = true Fusaini option to enable this.

- dft_report = *true* | *false*

  Specifies creating a DFT report. Type: boolean. Default: false.

  - true — The tool creates a DFT report.
  - false — The tool does not create a DFT report.

- distribute_cmd = *LSF command* | *SGE command*

  Specifies commands to launch LSF/SGE just as you would on the command line. KaleidoScope prepends your string to the KaleidoScope command before submitting the command. Type: string.

  Note:
  It is recommended to not use double quotes (" ") to encapsulate the string value. The tool captures the entire line after the equals sign.

  For example.

  ```
  distribute_cmd = qsub -b y -V -j y -r y -t 1-1 -w n -p -50 -wd $PWD $QSUB_ARG
  ```

- distribute_cmd_attempts = <*N*>

  Specifies the number of attempts made to run a child job. The default value of 1 results in no resubmission of the job. If there is a low probability of failure on the grid, a value of 2 is suggested. Otherwise, a value of 3 is recommended.

- enable_design_libs = *true* | *false*

  Determines multiple design library support. Type: boolean. Default: false.

  - true — Supports multiple design libraries.
  - false — Disables multiple design libraries.

- enable_observe_nodes = *true* | *false*

  When true, marks a fault as failure only if it reaches the Observe Point. Type: boolean. Default: false.

  - true — Marks a fault as failed only if it reaches the Observe Point.
  - false — Disables observe point checking for faults.

- error_inject_inst = *vcd_dut_path*

  String for *vcd_dut_path* specifies the hierarchical name of the DUT in the testbench. This is the DUT instance name in the given VCD. Type: string.

- fault_exclude_inst = *file*

  Specifies a file containing instances to exclude from fault list generation when running KaleidoScope in generate_fault_list mode. Each string in the specified file supports wildcard (*). KaleidoScope issues a warning if the specified list contains unmatched entries.

- fault_exclude_sig = *file*

  Specifies a file containing signals to exclude from fault list generation when running KaleidoScope in generate_fault_list mode. Each instance path in the specified file supports wildcard (*). KaleidoScope issues a warning if the specified list contains unmatched entries.

- fault_list = *filefile*

  Specifies a fault file with a list of hierarchical names of signals to inject with faults. Port faults are injected by default.

  To specify more than one fault file, use *fault_file_list file*.

- fault_file_list *file*

  Specifies a file that contains a list of fault filenames. Those fault files specify hierarchical names of signals to inject with faults. Port faults are injected by default.

  To specify a single fault file, use *--fault_list file*.

- filelist = *file*

  Questa list parser. Files must be listed in the order required for accessing and processing the files listed. The table that follows lists file parsing options supported for this --filelist option.

Table 13. File Parsing Options for filelist and filelist_sort

Option	Description
+define	Definition of MACROs to be used within the RTL.
-f / -F / -file	Specifies a file that contains a list of pathnames to source files and compile-time options.
+incdir	Specifies the directories that contain the files specified with the `include compiler directive. More that one directory can be specified if each pathname is separated with the + character.
+libext	Specify suffix of files in library directory.
-sverilog	Enables the use of the Verilog language extensions in SystemVerilog specification.
+systemverilogext	Specifies a filename extension for source files containing SystemVerilog source code.
-u	Changes all characters in identifiers to uppercase.
-v	Specify Verilog source library file.
+v2k	Enables the use of new Verilog constructs in the 1364-2001 standard.
+verilog1995ext	Specifies a filename extension for source files containing Verilog 1995 source code.
+verilog2001ext	Specifies a filename extension for source files containing Verilog 2001 source code.
-y	Specify Verilog source library directory.

- filelist_sort = *file*

  Files listed in this file are sorted automatically such that files and any associated libraries are accessed in the required order. Reference Table 13: File Parsing Options for filelist and filelist_sort for file parsing options supported by *--filelist_sort*. Type: filename.

- fsdb_file = <FSDB file>

  Specifies a single FSDB file for input. To specify multiple files, use the *KaleidoScope --sim_fsdb <file with FSDB files>* command line option. If you specify both the --sim_fsdb option and the fsdb_file fusaini option, the fusaini option takes precedence.

  You must specify the verdi_install_path=$(VERDI_HOME) fusaini option to use this option.

- FTTI = *integer*

  Specifies window size in VCD time. This is the time for the alarm to detect the fault after a failure. If the fault propagates to a state element and does not reach the alarm within the FTTI, then KaleidoScope does not detect the fault, hence the fault is dangerous. Type: Integer. Default: -1, which indicates the full VCD.

- generate_design_bin_script = *true* | *false*

  Controls whether or not KaleidoScope writes a *generate_design_bin* Qrun script to the *Outputs* directory that you can run to create a *design.bin* file for Visualizer.

  KaleidoScope will error out unless you specify the --filelist and --filelist_sort options along with the generate_design_bin fusaini option.

- hypertrophic_sim_multiple = <*integer*>

  Specifies a whole number with fusaini option hypertrophic_sim_multiple to define the upper limit of state elements with deviations. KaleidoScope multiplies this number by 100 and treats any fault that involves a greater number of state elements with deviations as a hypertrophic fault.

- hypertrophic_deviation_percent = <*integer*>

  Specifies a percentage of the total number of start points (including registers, latches, black box outputs, and primary inputs) that KaleidoScope should permit deviations to propagate to before it treats the fault with the original deviation as hypertrophic.

- ignore_translate_off = *true* | *false*

  Ignores *translate_off* pragma. Type: boolean. Default: false.

  - true — Ignores *translate_off* pragma.

  - false — Processes *translate_off* pragma.

- kman_parallel = *integer*

  Enables and sets the number of parallel runs for a KaleidoScope distributed run. Use this option with the --mode *kmanager_distributed* option to specify parallel simulations across multiple CPUs. Type: Integer. Default: 1.

- ks_async_sim = *true* | *false*

  By default KaleidoScope propagates faults through async ports. If you set this option to false, KaleidoScope faults do not propagate through async ports, and faults that reach async ports are listed in the *.Krpt* reports with a Reached_Async resolution. Default: true.

- ks_count_num_deviations = *true* | *false*

  When you specify this option during for Good Machine Simulation, KaleidoScope continues counting deviations until the end of stimulus for each endpoint. This option is not required, because encountering a single deviation suffices to show that KaleidoScope simulation is not identical to the golden reference.

- ks_fault_collapse = *true* | *false*

Specifies fault collapsing when generating a fault list. Type: boolean. Default: false.

See “Fault Collapsing and Fault List Generation” on page 18.

- ks_good_machine = *true* | *false*

  Specify this option to enable “Good Machine Simulation” on page 76.

- ks_memory_module_format = *false* | *true*

  This option controls the behavior of files you specify when initializing memory values with mem_init_file = filename. When you specify a value of true:

  - The file you specify with mem_init_file must contain module names instead of instance names.
  - The memory initialization file does not require addresses to pair with values, because KaleidoScope assumes the list of values begins at address zero.

  Otherwise the file you specify with mem_init_file must contain instance names, and the memory initialization files specified within that file must contain pairs of hexadecimal addresses and values on each row. See Initializing Memory Arrays for an example.

- ks_multi_thread = *false* | *true*

  Set this option to true to enable multithreading.

- ks_num_threads = *integer*

  Specifies an integer number of threads for multithreading. You must also enable the ks_multi_thread Fusaini option.

- ks_op_to_alarm = *clock_cycles*

  Specifies the number of clock cycles that KaleidoScope simulates the fault after incurring a dangerous fault. Type: integer. By default, KaleidoScope uses the value you specify with FTTI = integer; if FTTI is not set, the default is 25600 clock cycles.

- lib_cell_mapping = *true* | *false*

  Specifies KaleidoScope to automatically map start points buried inside library cells, for example in primitive cells whose instances do not have names. Type: boolean. Default: true.

- liberty_file = *file*

  Specifies a liberty file for library cells with its full path. For example:

```
--ini liberty_file=inputs/tcbn28hpcplusbwp12t30p140ffg0p88v0c.lib
```

- liberty_file_list = *file*

  Specifes a file containing a list of liberty files for library cells. For example:

```
--ini liberty_file_list=inputs/liberty.f
```

  Where the contents of liberty.f are:

```
input/tcbn28hpcplusbwp12t30p140ffg0p88v0c.lib
```

- map_fault_class = *fault_resolution new_fault_class_name*

  Renames the default FaultResolution value to a custom name the tool reports in the KaleidoScope fault report (Krpt). The option uses the following syntax:

  - *fault_resolution* — Name of the ISO fault classification you want to change.
  - *new_fault_class_name* — New name for the fault classification.

  Example:

```
map_fault_class = Detected_Observed MPF1
```

- max_concurrent_fault = *integer*

  Specifies the maximum number of concurrent faults per run. Type: Integer. Default: 150.

- max_fanout = *cone_fanout*

  Defines the maximum cone fanout determined by the maximum number of derived flip-flop events from the fault injection node. Specify an integer value for the maximum number of second order errors to define the maximum fanout. Type: integer. Default: 99.

- mem_init_file = *filename*

  Specifies a file that associates instances/module names with arrays and another file containing hexadecimal initialization values/address-value pairs on each row according the following format:

```
<instance/module name> <array name> <initialization file name>
```

  The ks_memory_module_format = false | true option controls whether this file must contain instance or module names in the first column, as well as whether the files specified in the third column contain address/value pairs or only values. See Initializing Memory Arrays for an example.

- memory_data_file_has_explict_addresses = *true* | *false*

  Enables the memory file format where an address precedes every data word. Type: boolean. Default: false.

  This option is not recommended, see Initializing Memory Arrays.

- memory_fault_propagation = *true* | *false*

  Enables (true) or disables (false) memory fault injection and fault propagation. Default: false. See “Memory Fault Use Model” on page 13 for complete details.

  Note:
  When memory_fault_propagation = true, then *syn_to_flop* must be 0.

- merge_safesim_reports = *true* | *false*

  Merge SafeSim text reports if true. Type: boolean. Default: false.

  - true — Merges SafeSim text reports.
  - false — Does not merge SafeSim reports.

- message_severity = *message severity*

  Specifies the severity of messages. One of the following severity levels can be specified:

- INFO
- WARNING
- ERROR
- FATAL

The severity may also be specified as OFF to suppress the message completely. See “Change Message Severity Levels” on page 118 for more information.

- mode = *mode*

Runs KaleidoScope tool in a specific mode, based on the operating mode specified. See Table 14: mode Options and KaleidoScope Actions.

Table 14. mode Options and KaleidoScope Actions

Mode	Action
generate_fault_list	Generates a fault list.
kmanager_distributed	Fault campaign manager mode session that can run on one or more CPUs.
kmanager_single	Fault campaign mode that runs multiple fault campaigns in one session.
report_design_stats	Displays design statistics.
sim_grade	Grade VCD or FSDB files with given fault list for design.
vcd_compare	Compares two VCD endpoints of the design.
vcd_filter	Filters VCD and retain only the endpoints of the design.
vcd_mapping	Maps RTL Structs, UnpackedArrrays to VCD signals.
$VARNAME	Uses environment specified.

- multiple_drivers = *true* | *false*

Enables support for multiple drivers. This functionality is only supported for Verilog and SystemVerilog, requires four-state-simulation, and only supports driving strength for Verilog primitives. See “Design With Multiple Drivers” on page 78

- outputs_are_observe = *true* | *false*

Designates all primary outputs as observe points. Type: boolean. Default: false.

- true — Designates all primary outputs as observe points.
- false — Primary outputs are not observe points.

- parser_messages = *true* | *false*

Generates info and warning messages. Type: boolean. Default: false.

- true — Generate info and warning messages.
- false — Does not generate info and warning messages.

• potential_faults = *optimistic* | *pessimistic*

Classifies potentially detected faults and turns on X propagation. Type literal. Default: optimistic.

- optimistic — X value on observe point results in PT.
- pessimistic — X value on observe point results in UO.

• potential_credit = *integer*

Specifies an integer value between 0 and 100 (representing a percent) for credit assigned to potential faults.

• preserve_user_nets = *true* | *false*

Preserve all user nets during RTL elaboration and in the netlist. Type: boolean. Default: true.

- true — Preserves all user nets during elaboration and in the netlist.
- false — Does not preserve user-nets during elaboration and in the netlist.

• print_blackbox_rpt = *true* | *false*

Prints blackbox report if true. Type: boolean. Default: false.

- true — Prints blackbox report.
- false — Does not print blackbox report

• print_sim_interval = *-1* | *interval*

Specifies the interval between which KaleidoScope prints simulation time. Type: Integer. Default: -1.

- -1 — Does not print simulation time.
- <*interval*> — Prints simulation time each specified interval.

• questa_cosim_en = *true* | *false*

Specifies to enable or disable Co-Simulation with Questa SIM as part of the Co-Simulation workflow. Specify this option both when generated a testbench for Co-Simulation, and while running the Co-Simluation run itself.

• questa_cosim_tb_gen = *true* | *false*

Specifies whether or not to generate a testbench for analog models for use in a Co-Simulation flow.

• questa_tb_top = <*testbench_top*>

This option specifies the name for the top level of the testbench generated by KaleidoScope as part of the Co-Simulation flow. This option also requires that you set the Fusaini options questa_cosim_tb_gen and questa_cosim_en to true.

- questa_tb_dir = *<path>/<tb_directory>*

  Specifies a directory containing a compiled and optimized design for Co-Simulation with Questa SIM.

- qwave_file = *<path>/<qwave.db>*

  Specifies a *qwave.db* containing simulation data for fault campaign stimulus.

- race_check = *true* | *false*

  Flags any race condition between clocks and primary inputs. The Rational column of the KaleidoScope fault report (*Krpt*) lists primary inputs with race conditions.

- rand_fault_select = *integer* | *<coverage_goal>* : *<confidence_interval>*

  This option enables you to either specify a single integer as the sample size for statistical fault sampling, or a pair of colon-separated values to specify a coverage goal and confidence interval.

  - The coverage goal value can range from 0 to 99.9. A value of 100 is not permitted.
  - Confidence interval has four allowed values: 90, 95, 99, and 99.9.

  See “Statistical Fault Sampling” on page 69 for full detail on the behavior of this option.

- randomize_faults = *true* | *false*

  Set this option to true to randomize the order faults are simulated. Type: boolean. Default: false.

- randomize_faults_seed = *integer*

  Specifies an integer seed value for statistical fault sampling. Default: 20201125.

- report_internal_names = *true* | *false*

  Reports internal nodes in report output when KaleidoScope runs in kmanager_distributed mode. Type: boolean. Default: true.

  - true — Reports internal nodes when KaleidoScope runs in kmanager_distributed mode.
  - false — Does not report internal nodes when KaleidoScope runs in kmanager_distributed mode.

- restore_design = *<path>/<ATDdb_directory>*

  Specifies a directory from which to load a compiled and elaborated top level design saved with the save_design option.

- save_design = *<path>/<ATDdb_directory>*

  Specifies a directory to save the compiled and elaborated top level design for reloading in subsequent analysis or fault campaign runs.

- save_design_libs = *true* | *false*

  Saves design libraries if true. Type: boolean. Default: false.

  - true — Saves design libraries.
  - false — Does not save design libraries.

- sim_duration_time = *integer*

  Specifies window of simulation. Specifying -1 is the total VCD time. Default: 0.

- sim_end_time = *integer*

  End time of simulation. Default: LLONG_MAX.

  LLONG_MAX, the maximum value for an object of type long long int, can be 9233372036854775807 (2^{63}-1) or greater. Default: 0.

- sim_force_list = *filename*

  Specifies a list of signals to force. KaleidoScope picks up values from simulation data instead of using computed values for simulation. The specified file does not contain values. Format each signal to force as *<top_module>.<hier_path>.<signal_name>*.

- sim_start_time = *integer*

  Start of simulation. Type: integer. Default: 0.

- simulate_combo_loops = *false* | *true*

  This option controls whether or not to simulate combinational loops during fault campaigns. The fault value is false.

- sim_fsdb <file with list of fsdb files>

  Specifies a file with a list of FSDB files for input. If you have only a single file for input, you can use the *fsdb_file=<FSDB file>* fusaini option instead. If you specify both the --sim_fsdb option and the fsdb_file fusaini option, the fusaini option takes precedence.

  You must specify the verdi_install_path=$(VERDI_HOME) fusaini option to use this option.

- sim_qwave <qwave.db file>

  Specifies a file containing *qwave.db* filenames with simulation data for fault campaigns the file can contain multiple *qwave.db* files, each on a separate line.

  - Lines in this file that begin with the # character are interpreted as comments and ignored.

  - You can specify simulation parameters in space separated columns after the name of each *qwave.db* file, see Reading Qwave.db files as Simulation Data for additional detail.

- sim_vcd = *sim_vcd*

  File lists VCD stimulus consisting of either VCD filenames from the RTL or GLS simulation waveforms.

- sv2009 = *0* | *1*

  When set (1), enables SV2009 compilation. Default: 0.

  - 0 — Compiles with SV 2000.

  - 1 — Compiles with SV 2009.

- syn_to_flop = *0* | *1*

  The value 0 treats memories as memory. The value 1 analyzes memory structures as a 2D array of flip-flops. Default: 0.

    - 0 — Treats memories as memory.
    - 1 — Treats memory as 2D array f flip-flops.
- tessent_udfm_fault_list = *udfm_instances_fault_list*

  Specifies a list of files containing Tessent UDFM instances and faults. Type: string array.

  See Tessent UDFM Fault Flow.
- tessent_udfm_fault_type = *all* | *delay* | *static*

  Specifies the fault type to run in the Tessent UDFM tests specification. Type: string. This option requires that you also set --ini dft=true.

    - all — The tool processes both delay and static faults in the Tessent UDFM fault list.
    - delay — The tool processes only the delay faults in the Tessent UDFM fault list.
    - static — The tool processes only the static faults in the Tessent UDMM fault list.

  See Tessent UDFM Fault Flow.
- tessent_udfm_node_faults = *true* | *false*

  Enables or disables node faults for the Tessent UDFM KaleidoScope fault simulation. Type: boolean. Default: false.

    - true — Enables the fault simulation.
    - false — Disables the fault simulation.

  See Tessent UDFM Fault Flow.
- tessent_udfm_spec_files = *cell_udfm_definition_list*

  Specifies a list of files with Tessent UDFM definitions for cells. Type: string array. This option requires that you also set --ini dft=true.

  See Tessent UDFM Fault Flow.
- top = *top_mod*

  String specifies top module name. Required. Type: string.
- transient_window = *integer*

  Specifies the transient window time when running a fault campaign in distributed mode (--mode *kmanager_distributed*). Default: -1.
- udfm_models_list = *file*

  Specifies the filename of a file that contains a list of UDFM model names. Type: filename. This argument requires that you also set --ini dft=true.
- uniquify = *0* | *1*

  When 1, uniquifies the design during read. Default: 0.

    ◦ 0 — Does not uniquify the design during read.

    ◦ 1 — Uniquifies the design during read.

• vcd_compare_all_eps = *0* | *1*

  Compares all endpoints in each time tick (only in vcd_compare mode; see “Table 14: mode Options and KaleidoScope Actions” on page 112). Type integer. Default: 1.

    ◦ 0 — Does not compare endpoints.

    ◦ 1 — Compares all endpoints in each time tick when operating mode is vcd_compare.

• vcd_name_print

  Prints the VCD filename. Type: string.

```
#ErrorNet                           ErrorVal  TriggerTime  FailTime  AlarmTime  FaultResolution  Rational  VCDFileName
Memory_wrapper_ATD.u_Memory.DataOut[7]  0     30           30        0          0                -         rtl/out.vcd
Memory_wrapper_ATD.u_Memory.DataOut[6]  0     30           30        0          0                -         rtl/out.vcd
Memory_wrapper_ATD.u_Memory.DataOut[5]  0     30           30        0          0                -         rtl/out.vcd
Memory_wrapper_ATD.u_Memory.DataOut[4]  0     40           40        0          0                -         rtl/out.vcd
Memory_wrapper_ATD.u_Memory.DataOut[3]  0     30           30        0          0                -         rtl/out.vcd
Memory_wrapper_ATD.u_Memory.DataOut[2]  0     40           40        0          0                -         rtl/out.vcd
Memory_wrapper_ATD.u_Memory.DataOut[1]  0     30           30        0          0                -         rtl/out.vcd
Memory_wrapper_ATD.u_Memory.DataOut[0]  0     30           30        150        0                -         rtl/out.vcd
```

• vcd_out_name = *string*

  Output name for filtered VCD created when the --mode *vcd_filter* option is specified.

  Use this *vcd_out_name* option to change the name of the default *FilterOut_O.vcd* file in the output directory to a user-defined name. Specify only a string without a file extension. The KaleidoScope tool appends the string specified to *_n.vcd* in which *n* is an assigned number and creates the output file *<string>_n.vcd* in the *Outputs* directory. File numbers are assigned starting with 0 and increment in value based on the order that the file is created and the number of existing files. Type: string. Default: *./Outputs/<string>_n.vcd*

• verdi_install_path=$(VERDI_HOME)

  Specifies the Verdi install path, which is required to run KaleidoScope with FSDB files as input, (the --sim_fsdb command line option or fsdb_file=<fsdb file> fusaini option).

• vfile = *verilog_design*

  Verilog design file. Type: filename.

• vfiles = *file_list*

  Verilog design file list. Type: filename.

• vhdl_file = *file_list*

  File contains VHDL file list. Type: filename.

• vlibcell = *library_cell_list*

  File that lists the library cells. Type: filename.

Qrun Script for Debugging With Visualizer

KaleidoScope supports writing a Qrun script that you can, in turn, run to create a *design.bin* file for use when running Visualizer for debugging purposes.

When you specify fusaini option generate_design_bin as true, KaleidoScope writes a script named *generate_design_bin* to the *Outputs* directory.

- KaleidoScope will error out unless you specify the --filelist and --filelist_sort options along with the generate_design_bin fusaini option.

The resulting script has the format:

```
fusa_qrun -f <filelist_name> -designfile <bin file_name> -top
 <top_level_DUT_name> -optimize
```

Note:
fusa_qrun is a wrapper script that ensures the script uses the Qrun version shipped within the Functional Safety install tree to avoid compatibility issues.

Change Message Severity Levels

You can change severity levels of all fusa, known fusa_Veri-*, fusa-VDB-*, and fusa-LIB-* messages, except FATAL. You can also suppress their display by setting them as on or off with the message_severity directive. Only parser messages are turned off by default. However, your log file displays a warning message just after RTL processing and when all analysis completes telling you the type and number of each type of message that was suppressed.

You can change the severity level of a message by specifying the message_severity option in the FUSA initialization file.

Specifying these options does not overwrite a previous severity level, but adds to existing options if the message ID for each option is unique. When you specify the same message ID multiple times using the message_severity option, then the tool uses the last option specified.

All messages except the ones with the FATAL severity level can be promoted to a severity level above the current one, but the message severity level cannot be demoted. The table that follows outlines the promotion options.

Table 15. Valid Severity Level Changes

Default Severity Level	Turn On/Off Messages	Promoted Only Cannot be Demoted
INFO	Yes	WARNING, ERROR, or FATAL
WARNING	Yes	ERROR or FATAL
ERROR	Yes	FATAL
FATAL	No	Cannot be changed

Message Severity Examples

The following message_severity directives specified in the *fusaini* initialization file change the level of the fusa -41 message from a WARNING to an ERROR, displays any fusa_42 messages, and turns off all fusa-7 messages.

```
message_severity = fusa-41 ERROR
message_severity = fusa-42 on
message_severity = fusa-7 off
```

For additional information refer to the Functional Safety Reference Manual.

Global Customer Support and Success

A support contract with Siemens Digital Industries Software is a valuable investment in your organization’s success. With a support contract, you have 24/7 access to the comprehensive and personalized Support Center portal.

Support Center features an extensive knowledge base to quickly troubleshoot issues by product and version. You can also download the latest releases, access the most up-to-date documentation, and submit a support case through a streamlined process.

https://support.sw.siemens.com

If your site is under a current support contract, but you do not have a Support Center login, register here:

https://support.sw.siemens.com/register

Third-Party Information

Details on open source and third-party software that may be included with this product are available in the *<your_software_installation_location>/legal* directory.