# SpyGlass® Auto Verify Rules Reference Guide

Version N-2017.12-SP2, June 2018

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

## Report an Error

The SpyGlass Technical Publications team welcomes your feedback and suggestions on this publication. Please provide specific feedback and, if possible, attach a snapshot. Send your feedback to *spyglass_support@synopsys.com*.

# Contents

# Preface

## About This Book

The SpyGlass® Auto Verify Rules Reference Guide describes concepts and rules of SpyGlass Auto Verify solution.

## Contents of This Book

The SpyGlass Auto Verify Rules Reference Guide consists of the following sections:

| **Section** | **Description** |
|---|---|
| *Using the Rules in the SpyGlass Auto Verify Solution* | Usage concepts and use model features, such as parameters |
| *Rules in SpyGlass Auto Verify* | Various rules of SpyGlass Auto Verify solution |
| *The OVL Support* | About OVL support |
| *Appendix: SGDC Constraints* | SGDC constraints used by SpyGlass Auto Verify |

## Typographical Conventions

This document uses the following typographical conventions:

| To indicate | Convention Used |
|---|---|
| Program code | `OUT <= IN;` |
| Object names | `OUT` |
| Variables representing objects names | *`<sig-name>`* |
| Message | Active low signal name '<sig-name>' must end with _X. |
| Message location | OUT <= IN; |
| Reworked example with message removed | OUT_X <= IN; |
| Important Information | **NOTE:** This rule... |

The following table describes the syntax used in this document:

| Syntax | Description |
|---|---|
| [ ] (Square brackets) | An optional entry |
| { } (Curly braces) | An entry that can be specified once or multiple times |
| \| (Vertical bar) | A list of choices out of which you can choose one |
| . . . (Horizontal ellipsis) | Other options that you can specify |

# Using the Rules in the SpyGlass Auto Verify Solution

SpyGlass® Auto Verify is a *Functional Analysis* solution built on the SpyGlass platform and requires a separate license.

SpyGlass Auto Verify solution has the following features:

- It provides a wide range of rules to validate the functionality of a design. For example, it provides rules for FSM detection, variable range validation, and tristate bus proper functionality validation.
- It performs analysis across hierarchies and sequential elements in a design, and provides parameters to control the scope of analysis to a specific area of a design or for a specific type of analysis.
- It provides multiple engines that are combined to provide fast analysis time and deal with large multi million gate designs.

  You can control both run time and memory utilization through various parameters.

## License Used by SpyGlass Auto Verify

All the rules of SpyGlass Auto Verify and their prerequisite and dependent rules use the `Auto_Verify` license.

# Definitions and Concepts in SpyGlass Auto Verify

This section describes the following topics:

- *Functional Analysis*
- *Property and Property Analysis*
- *Implicit Properties*
- *Standard OVL Properties*
- *Clock Cycle Count and Sequential Depth*
- *Design Virtual Cycle*
- *Initial State*
- *Stuck-Net*

## Functional Analysis

Functional analysis refers to analyzing the functionality of a design as opposed to analyzing the structure of a design or analyzing with regard to a specific domain, such as power-related or SpyGlass DFT solution.

Examples of functional analysis are:

- Search for bus contention
- Checking exclusivity of two signals
- Checking for gray encoding of a vector
- Checking for leachability of states and transition from states for an FSM.

## Property and Property Analysis

A property is a functional characteristic of a design. For example, one hot encoding of an FSM.

Property analysis refers to functional validation of a property in a design.

## Implicit Properties

Implicit properties are the properties that can be automatically extracted from a design without the help from users.

The example of such property is one hot encoding of bus enables. This property is implicit because there is a common predefined rule that two drivers should not drive a bus line simultaneously, and therefore the enables must be one hot encoded. An automatic process can be implemented that searches for such implicit properties.

## Standard OVL Properties

Refer to Accellera Standard Open Verification Library (Version Oct 2002) of properties that can be instantiated in a design as an assertion or a functional constraint.

SpyGlass Auto Verify solution supports OVL for property specification. Any OVL assertion inserted in the RTL code by the user is considered as a user-provided property. Example: `assert_range()`.

## Clock Cycle Count and Sequential Depth

Sequential depth refers to the number of clock cycles a rule takes to start from an initial state and reach the location of rule violation.

In case of a single clock that is active only at posedge or only at negedge, this number is pretty straightforward. However, in a multi-clock environment with the clocks active at posedge, negedge, or both, the cycle count can be interpreted differently.

In order to provide an accurate idea of number of cycles, SpyGlass Auto Verify solution reports two numbers for the sequential depth. These two numbers are the same for a design with a single clock active at a single edge. For all other cases, these two numbers are defined as follows:

1. Number of cycles of fastest clock in the cone of influence of the property being checked

   If a property is applied to a set of nets of a design, then the fastest clock of the relevant nets is extracted and if a message is occurring, then the number of cycles of the fastest clock at the time of rule-violation is reported. In the simple case of a single clock system, this number will be the number of clock cycles of the given clock. In this scheme, a clock cycle is accounted for as soon as one of the edges has occurred; as a consequence, a half cycle is considered as a full cycle.

2. Number of non-overlapping edges

   This number represents both the positive edge count and the negative edge count from the initial state to the rule-violation; where if two edges of two clocks occurred at the exact same time, the counter is incremented only by one. SpyGlass generates a vector signal named `verification_cycle` to represent the counter value. This value is displayed in the Waveform Viewer when an assertion failure occurs.

   Note that both edges are counted regardless if registers are triggered at posedge, negedge, or both. In particular, in a single clock system, this number will be equal to twice the number of clock cycles (~+1 due to the fact that a half cycle is accounted as a full cycle).

   For example, *Figure 1* shows three clocks waveforms. For the given window of time, there are 8 clock cycles (number of cycles of the fastest clock) and there are 17 edges (15 edges for the top waveform, 0 for the middle since all the edges are covered by the top waveform, and 2 edges for the last clock since it is not overlapping with any edges of the previous clocks). Note that although not all edges are active, the counting includes all edges — active or inactive.

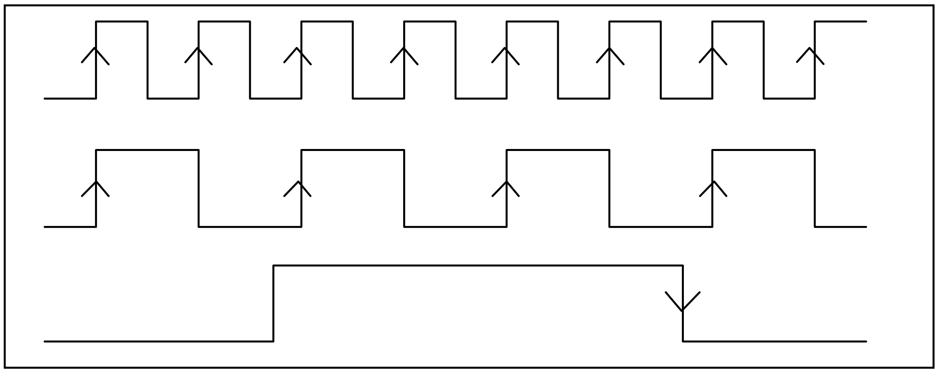


**FIGURE 1.** Clock cycle count and sequential depth

## Design Virtual Cycle

Functional analysis complexity increases with the number of asynchronous clocks in a design. To understand how clock frequencies affect the functional analysis process, consider two clocks running with 17 ns period and 13 ns period respectively. If the rising edges of the two clocks are aligned at time 0 ns, then the next time the rising edges will again be aligned corresponds to 221 ns (LCM of two clock periods). This means that the design behaves asynchronously for 221 ns. Any functional analysis process that would exploit repetition (for proving a property, for example) would have to analyze the design at least for this period of time, which may correspond to many evaluations of logic in the design. We refer to this period as the Design Virtual Cycle. SpyGlass Auto Verify solution reports the virtual design cycle in terms of number of fastest-clock cycles covered by the Design Virtual Cycle as well as the number of non-overlapping edges of all clocks covered by the Design Virtual Cycle.

## Initial State

The initial state is a register-value assignment from which *Functional Analysis* begins.

For example, given a 4-bit counter and a property asserting that the counter will eventually reach 15, this assertion passes in 5 cycles if the counter is initialized with 10, whereas it will pass in 15 cycles if the counter is initialized to 0.

An initial state may or may not be a reset state of a design. A reset state of a design is a register-value assignment obtained by resetting a design using a reset signal (may be user-specified). SpyGlass Auto Verify solution can obtain an initial state in four different ways:

1. Direct register-value assignment using the *reset* constraint
2. State value generated by an external simulation engine as a VCD file and read in to SpyGlass Auto Verify solution using the *vcdfile* parameter
3. Initialization vector that can reset registers using *define_tag* constraint
4. Find an initial state automatically. If you do not provide an initial state and/or do not provide an initialization vector, then SpyGlass Auto Verify solution determines an initial state using the following approach:
   - ❒ Use reset port to reset registers
   - ❒ Random analysis by applying stimulus to the inputs and simulating to find *valid register value assignment*. Although not a reset state, the

assignment is guaranteed to be reached by stimulating the inputs of a design.

- Functional analysis for a valid register-value assignment

Automatic initial state search (case 4) cannot be combined with other cases. Therefore, any register not initialized by the user-specified vector or the initial state remains at “x” which can impact the outcome of functional analysis.

You must validate the initial state of a design before running any functional analysis. SpyGlass Auto Verify solution provides various reports as well as RTL back-annotation and schematic highlight for the initial state exploration. See *Reports and Diagnosis Files in SpyGlass Auto Verify* for more details.

## Stuck-Net

Stuck-at nets (constant nets) in the design can be of three types, as described in the following table:

| **Stuck-at net** | **Description** |
|---|---|
| Globally-stuck-at | A net is globally stuck-at if the net cannot change value, once it is initialized to some initial value. |
| Partially-stuck-at | A net is partially stuck at if the net, once initialized, can change value only one time. For example, a flip-flop can be initialized to 0 and then can change value to 1. In this case, the flip-flop will get stuck at 1. |
| Eventually-stuck-at | A net is eventually stuck-at if the net, once initialized, can change value multiple times, before getting stuck-at a value. |

**NOTE:** *A signal that cannot be initialized to an initial state is, by definition, not stuck-at.*

## Overview of SpyGlass Auto Verify

SpyGlass Auto Verify solution searches for functional problems/bugs in a design through the following ways:

- Automatic checks where SpyGlass Auto Verify solution extracts *Implicit Properties* of a design and checks for their correctness in the context of a specific design.

  Examples of such checks are Bus Contention, Array Bound Violation etc.
- User-specified checks where you provide explicit properties to be validated.

  Examples of such explicit properties are Hand Shaking, Gray Coding of specified signals etc.

*Figure 2* shows the inputs and outputs to SpyGlass Auto Verify solution:

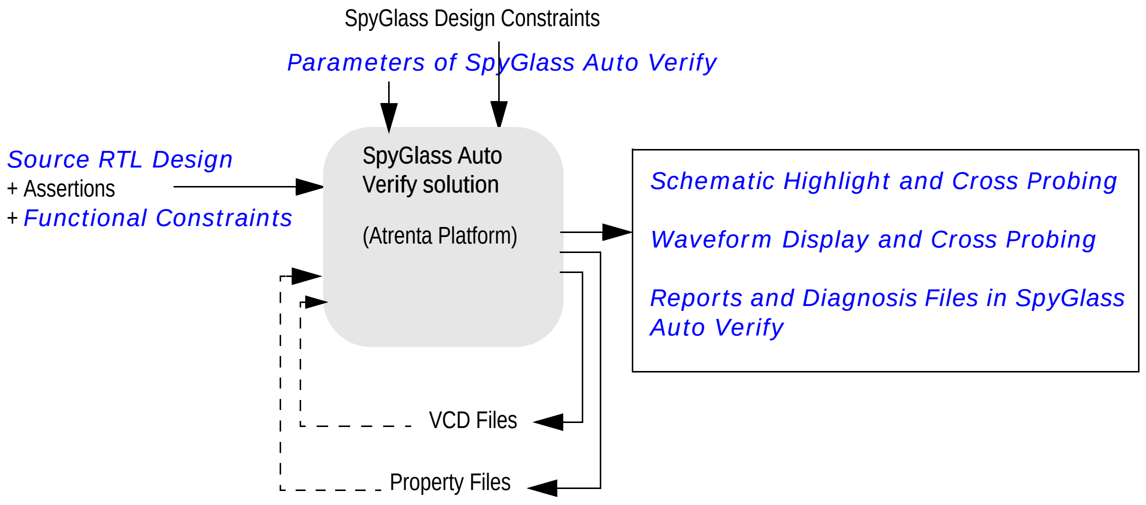

SpyGlass Design Constraints
Parameters of SpyGlass Auto Verify
Source RTL Design
+ Assertions
+ Functional Constraints
SpyGlass Auto Verify solution
(Atrenta Platform)
Schematic Highlight and Cross Probing
Waveform Display and Cross Probing
Reports and Diagnosis Files in SpyGlass Auto Verify
VCD Files
Property Files

**FIGURE 2.** Functional Mode in SpyGlass Auto Verify Solution

To perform in-depth functional analysis, SpyGlass Auto Verify solution uses multiple advanced engines. Unlike simulation, SpyGlass Auto Verify solution performs vector-less static analysis. Most of the analysis within

SpyGlass Auto Verify solution combines the design with the property and explores the combined space in search for a bug. Exploration of this space is conducted across register boundaries for a full sequential analysis. This search can reach depth of hundreds or even thousands of clock cycles. Parameters are provided to control both run-time and sequential depth of analysis.

**NOTE:** *To run SpyGlass Auto Verify solution by selecting the required goal containing rules of SpyGlass Auto Verify solution.*

# Source RTL Design

Since SpyGlass Auto Verify solution analyzes the functionality of a design, it is absolutely necessary to provide the functionality of all blocks for which the analysis is requested. The following items require particular user attention:

- *Library Cells*
- *Black Boxes*
- *Memory Blocks*
- *Bidirectional Ports*
- *Asynchronous Resets*
- *Latches*
- *Tristate Buses*
- *Gated Clocks*
- *Finite-State Machines (FSMs)*

## Library Cells

You can specify library files by using the `read_file -type gateslib <library-file>` command in the project file.

## Black Boxes

A design unit without functionality is considered as a black box during *Functional Analysis*. Examples of such modules are memory blocks not transformed to a register bank and library cells that are not expanded. The inputs of these design units are considered as outputs of the design, the outputs of these modules are considered as inputs of the design. In particular, if a clock port of a register is driven from a black box output, this output is considered as a *primary clock* and needs to be user-defined, or the *default clock* will be attributed to it. Outputs of a black box may be constrained using the functional constraints definition.

## Memory Blocks

For simplicity, the memory blocks should be black boxes. As mentioned in *Black Boxes*, the outputs of such blocks are supposed to generate any combination of data, which is a reasonable assumption for a memory block. The data and address buses and decoders are not part of the memory core; therefore, they participate in the functional analysis.

## Bidirectional Ports

For functional analysis, the bidirectional ports are considered as inputs and/or outputs. This fact is transparent to the user.

## Asynchronous Resets

SpyGlass Auto Verify solution can automatically analyze designs with asynchronous resets. While it is recommended, it is not necessary for you to provide asynchronous reset information through the *reset* constraint. By default, the asynchronous resets are used only for an initial state search and they are disabled during functional analysis. To allow the asynchronous reset usage during functional analysis, use the `reset` constraints to set the reset signal as soft reset. The auto-detection of asynchronous resets can be turned off using the `use_inferred_resets` parameter of SpyGlass CDC solution. Synchronous resets in all cases must be provided through the *reset* constraint in a SpyGlass Design Constraint file.

## Latches

SpyGlass Auto Verify solution can analyze designs with level-sensitive latches. However, for functional analysis purposes, a transparent latch is modeled so that its output is evaluated at each active edge of any clock controlling the data or the enable of the latch. This modeling conforms to a simulation model for a level sensitive latch.

## Tristate Buses

SpyGlass Auto Verify solution can analyze designs with tristate buses. However, the assumption is made that the tristate bus is never floating nor is there contention (this check is performed by SpyGlass Auto Verify solution separately on each tristated bus). In fact, a tristate bus is transformed into an equivalent MUX structure before the functional analysis.

## Gated Clocks

SpyGlass Auto Verify solution supports multiple asynchronous clocks and gated clocks in a design as long as the source clocks are provided which control the register clock pins through functionally analyzable components. You can provide the source clocks using the *clock* constraint.

## Finite-State Machines (FSMs)

SpyGlass Auto Verify solution checks for correctness of FSMs in a design. FSMs are first extracted from a design at the RTL level, then various functional analyses are performed to detect deadlock states, unreachable states etc. However, there are many ways of describing an FSM in a design and therefore an FSM may not be detected by SpyGlass Auto Verify solution even if the FSM is present in the RTL code. The following styles are currently considered:

1. Both single-process FSMs and 2-concurrent processes (for current state and next state assignment) FSMs
2. The core of the FSM computing the next state functions and output functions must use a `case` statement.
3. if-else-if type of FSMs
4. `assign` type FSMs
5. Nested FSMs (one FSM nested inside another FSM)
6. The case statement describing the FSM may have multiple if/else branches embedded in it; some branches may be used to describe the initial state while other branches are used to describe the FSM transition functions.
7. Case select of a case statement describing an FSM can be one of the following only:

- ❒ State variable

**NOTE:** *SpyGlass Auto Verify solution requires you to specify only the simple name (that is, without bit-width specification) for a state variable for the FSM to be inferred.*

- ❒ Bit-select 1'b1 or 1'b0: This is used to describe a one-hot or a one-cold encoding FSM only.

**NOTE:** *Currently, this feature is supported for Verilog designs only.*

8. Next state assignment: RHS must be a constant value. Ideally, it should be one of the case labels.
   - ❒ Tasks/Functions calculating a next state are not supported
   - ❒ Logical/arithmetic operations calculating next state value are not supported
   - ❒ State labels can be of type enum, but the enum literals must be of type `charlit`.

## Parameters of SpyGlass Auto Verify

SpyGlass Auto Verify solution supports the following parameters:

| | | |
|---|---|---|
| *atime* | *av_dump_assertions* | *buscompress* |
| *ieffort* | *audit* | *passfail* |
| *dead_code_scope* | *detect_assign_fsm* | *detect_ifelse_fsm* |
| *detect_nested_fsm* | *reset_convention* | *resetoff* |
| *show_static_latches* | *solvemethod* | *propfile* |
| *modulelist* | *scope* | *vcdtime* |
| *vcdfile* | *vcdfulltrace* | *verbose* |
| *xassign_casedefault* | | |

In addition to the above parameters, SpyGlass Auto Verify solution supports the following parameters of SpyGlass CDC solution:

| | | |
|---|---|---|
| filter_named_resets | use_inferred_clocks | use_inferred_resets |

For details on the above parameters, refer to *SpyGlass CDC Rules Reference Guide*.

**NOTE:** *Unless specified otherwise, all parameters in SpyGlass Auto Verify solution are optional.*

## atime

With `atime` parameter, you can set the runtime limit for the analysis of a single property.

The default value of `atime` depends on the property count. If the property count is high, SpyGlass internally computes and uses a lower runtime limit. Similarly, if the property count is low, SpyGlass internally computes and uses a higher runtime limit in the SpyGlass run.

| | |
|---|---|
| Used by | All functional rules |
| Options | Positive integer value |

| | |
|---|---|
| Default value | Varies depending on the number of properties being checked. |
| Example | |
| *Console/Tcl-based usage* | `set_parameter atime 40` |
| *Usage in goal/source files* | `-atime=40` |

## av_dcode_analysis

Specifies the approach (strict or soft) towards verification of assertions.

By default, the *Av_deadcode01* rule uses the soft approach for verification of assertions. This approach is quick, but it may impact the quality of verification results.

Set this parameter to `strict` to improve the quality of verification results. However, using the strict approach can increase the rule run time.

| | |
|---|---|
| Used by | *Av_deadcode01* |
| Options | soft, strict |
| Default value | soft |
| Example | |
| *Console/Tcl-based usage* | `set_parameter av_dcode_analysis strict` |
| *Usage in goal/source files* | `-av_dcode_analysis=strict` |

## av_dcode_report

Configures the *Av_deadcode01* rule to report violations for the assertions present in all the nested `if-else` blocks. In this case, this rule groups all the violations of the same `if` block, `else-if` block, or `else` block of a nested `if` block (or dependency tree). Within each group, violations are sorted based on the assertions depth within the `if`, `else-if`, or `else` block.

By default, the *Av_deadcode01* rule reports violations for the assertions present only in the top-level `if` block of a nested `if` block (or dependency tree) to reduce violation noise.

For details, see *Message Grouping in the Av_deadcode01 Rule*.

| | |
|---|---|
| Used by | *Av_deadcode01* |
| Options | minimal, all |
| Default value | minimal |
| Example | |
| *Console/Tcl-based usage* | `set_parameter av_dcode_report all` |
| *Usage in goal/source files* | `-av_dcode_report=all` |

## av_dump_assertions

Generates SystemVerilog Assertions (SVA) for partially-proved assertions of the rules specified in the *Used by* section.

When you set this parameter to `sva`, SpyGlass generates the bind file `sva_rules_prop_`*`<top-module-name>`*`_bind.sv` along with simulator-specific (vcs, ncsim, modelsim) assertion files `sva_rules_prop_`*`<top-module-name>`*`_`*`<simulatorName>`*`.sv` in the `wdir/spyglass_reports/auto-verify/assertions` directory.

On passing `av_dump_assertions` with *audit*, the design is run in `audit` mode, and `SVA` for all the assertions are generated.

| | |
|---|---|
| Used by | *Av_deadcode01*, *Av_staticnet01*, *Av_fsm01*, *Av_fsm02*, *Av_case01*, *Av_case02*, *Av_bus01*, *Av_bus02*, *Av_dontcare01*, *Av_range01*, *Av_fsm_analysis*, *Av_divide_by_zero* |
| Options | sva |
| Default value | “” |
| Example | |

| | |
|---|---|
| *Console/Tcl-based usage* | `set_parameter av_dump_assertions “sva”` |
| *Usage in goal/source files* | `-av_dump_assertions=“sva”` |

## av_dump_instance_complexity

Generates instance-based spreadsheet (*Av_complexity01_InstanceBased.csv Tab*) showing cyclomatic complexity of module instances.

This information is similar to the information displayed in the module-based spreadsheet (*Av_complexity01_module.csv Tab*).

The difference between the two spreadsheets is that the instance-based spreadsheet shows information for each instance instead of each module and it additionally displays cumulative complexity of each instance and its level with respect to the top module.

| | |
|---|---|
| Used by | *Av_complexity01* |
| Options | yes, no |
| Default value | no |
| Example | |
| *Console/Tcl-based usage* | `set_parameter av_dump_instance_complexity yes` |
| *Usage in goal/source files* | `-av_dump_instance_complexity = yes` |

## av_dump_liveness

Generates the SystemVerilog Assertions (SVA) in terms of *assert* or *cover* for assertions of the *Av_fsm02* and *Av_fsm02* rules.

By default, SVA assert statements are generated.

**NOTE:** *This parameter is applicable only when the av_dump_assertions parameter is set to sva.*

| | |
|---|---|
| Used by | *Av_fsm01*, *Av_fsm02*, *Av_fsm_analysis* |
| Options | assert, cover |
| Default value | assert |
| Example | |
| *Console/Tcl-based usage* | `set_parameter av_dump_liveness cover` |
| *Usage in goal/source files* | `-av_dump_liveness = cover` |

## av_enable_crpt

Configures SpyGlass Auto Verify rules to generate a spreadsheet showing details of SVA constraints affecting each rule violation.

For information on this spreadsheet, refer to the *Using SystemVerilog Assertions application note*.

| | |
|---|---|
| Used by | Refer to the Using SystemVerilog Assertions application note |
| Options | yes, no |
| Default value | no |
| Example | |
| *Console/Tcl-based usage* | `set_parameter av_enable_crpt yes` |
| *Usage in goal/source files* | `-av_enable_crpt = yes` |

## av_flopcount

Specifies a maximum number of flip-flops so that an input cone of SpyGlass Auto Verify properties can be abstracted by cutting the logic behind the specified number of flip-flops in that cone.

While running SpyGlass Auto Verify analysis on full chips, the cone of SpyGlass Auto Verify properties can be very complex in terms of number of flip-flops. This results in significant time spent on verification.

To circumvent this problem, use this parameter to limit the number of flip-flops to abstract input cones. Using this parameter also increases the chances of getting properties concluded.

It is recommended that you use this parameter only for partially-proved properties because usage of this parameter may help in concluding such properties.

**NOTE:** *If you use this parameter for properties that are failing, such properties may be reported as partially proved. Therefore, it is recommended that you use this parameter only on partially proved properties by using the propfile parameter.*

By default, this parameter is set to `-1`, which indicates that an input cone will not be abstracted by cutting the logic behind a specific number of flip-flops in the cone of SpyGlass Auto Verify properties.

| | |
|---|---|
| Used by | All SpyGlass Auto Verify rules |
| Options | -1, 0, or a positive integer value greater than 0 |
| Default value | -1 |
| Example | |
| *Console/Tcl-based usage* | `set_parameter av_flopcount 2` |
| *Usage in goal/source files* | `-av_flopcount = 2` |

## av_force_soft_reset

Specifies if a reset or set in the design should be forced as a soft reset during *Functional Analysis* while running the *Av_setreset01* and *Av_deadcode01* rules.

By default, the *Av_setreset01* rule considers all resets as soft even when that reset is specified as a hard reset in the SGDC file. Set this parameter to `no` so that this rule considers the resets as hard or soft based on the specifications in the SGDC file.

By default, the *Av_deadcode01* rule uses the specifications in the SGDC file to consider a reset as hard or soft. To force this rule to consider all the resets as soft resets:

- Specify *Av_deadcode01* to the existing parameter specification, or

- Set the value of this parameter to `yes`.

**NOTE:** *By default, all the SpyGlass Auto Verify rules, except the Av_setreset01 rule, uses the hard or soft specification of resets as specified in the SGDC file.*

| | |
|---|---|
| Used by | *Av_deadcode01*, *Av_setreset01* |
| Options | Comma-separated list of the *Av_deadcode01* and *Av_setreset01* rules, yes, no |
| Default value | Av_setreset01 |
| Example | |
| *Console/Tcl-based usage* | `set_parameter av_force_soft_reset no` |
| *Usage in goal/source files* | `-av_force_soft_reset=Av_setreset01,Av_deadcode01` |

## av_ignore_preformal_run_time

Specifies if preformal runtime, such as time for synthesis should be considered while calculating the total runtime for the current run of a SpyGlass Auto Verify goal.

This parameter is used with the *av_run_time* parameter.

| | |
|---|---|
| Used by | All formal rules of SpyGlass Auto Verify |
| Options | yes, no |
| Default value | no |
| Example | |
| *Console/Tcl-based usage* | `set_parameter av_ignore_preformal_run_time yes` |
| *Usage in goal/source files* | `-av_ignore_preformal_run_time = yes` |

## av_msgmode

Specifies the type of assertions (failed, partially proved, and passed) to be reported by the *Av_syncfifo01* rule.

By default, this rule reports failed assertions.

| | |
|---|---|
| Used by | *Av_syncfifo01* |
| Options | fail, pp, pass, all |
| Default value | fail |
| Example | |
| *Console/Tcl-based usage* | `set_parameter av_msgmode all` |
| *Usage in goal/source files* | `-av_msgmode = all` |

## av_run_time

Specifies the total runtime (wall clock time) for current run of a SpyGlass Auto Verify goal.

The current runtime includes preformal runtime, such as time for synthesis. (Use Parameter *av_ignore_preformal_run_time* to ignore preformal time)

By default, total runtime depends on multiple factors, such as the value of the *atime* parameter and number of properties.

| | |
|---|---|
| Used by | All formal rules of SpyGlass Auto Verify |
| Options | <positive-integer>h, <positive-integer>m, or <positive-integer>s<br>Where h, m, and s represent hours, minutes, and seconds, respectively. |
| Default value | No default value |
| Example | |
| *Console/Tcl-based usage* | `set_parameter av_run_time 10h` |
| *Usage in goal/source files* | `-av_run_time = 10h` |

## av_seqdepth

Specifies a maximum sequential depth so that an input cone of SpyGlass Auto Verify properties can be abstracted by cutting the logic behind the specified depth in that cone.

While performing SpyGlass Auto Verify analysis on full chips, the cone of SpyGlass Auto Verify properties can be very complex in terms of a sequential depth. This results in significant time spent for verification.

To circumvent this problem, use this parameter to limit the sequential depth to abstract input cones. Limiting the sequential depth also increases the chances of getting properties concluded.

It is recommended that you use this parameter only for partially-proved properties because usage of this parameter may help in concluding such properties.

**NOTE:** *If you use this parameter for properties that are failing, such properties may be reported as partially proved. Therefore, it is recommended that you use this parameter only on partially proved properties by using the propfile parameter.*

Setting this parameter to `-1` indicates that an input cone will not be abstracted by cutting the logic behind a specific depth in the cone of SpyGlass Auto Verify properties.

| | |
|---|---|
| Used by | All the SpyGlass Auto Verify rules |
| Options | 0, -1, or positive integer value greater than 0 |
| Default value | -1 |
| Example | |
| *Console/Tcl-based usage* | `set_parameter av_seqdepth 2` |
| *Usage in goal/source files* | `-av_seqdepth = 2` |

## av_violation_count

Limits the violation count for the rules of SpyGlass Auto Verify.

When the number of violations specified by this parameter are reported, the remaining properties are not verified formally.

| | |
|---|---|
| Used by | All formal rules of SpyGlass Auto Verify |
| Options | Positive integer value |
| Default value | No default value |
| Example | |
| *Console/Tcl-based usage* | `set_parameter av_violation_count 10` |
| *Usage in goal/source files* | `-av_violation_count = 10` |

## buscompress

Specifies whether the *Av_staticnet01* rule should check single bit or all the bits of a bus signal.

By default, the value of this parameter is set to `yes`, and the Av_staticnet01 rule checks single bit of a bus signal.

Set the value of this parameter to `no` to check all the bits of a bus signal.

| | |
|---|---|
| Used by | *Av_staticnet01* |
| Options | yes, no |
| Default value | yes |
| Example | |
| *Console/Tcl-based usage* | `set_parameter buscompress no` |
| *Usage in goal/source files* | `-buscompress=no` |

## ieffort

This parameter is used to change the effort of the tool put in initial state search during design simulation.

By default, during initial state search, the tool first applies asynchronous set/resets on a design and then performs clocked simulation.

### Clock Simulation

Clocked simulation is performed for a fixed number of cycles (that is 200 cycles) until any of the following occurs:

- A non-X value reaches on all flip-flops.
- No improvement is observed for a fixed number of consecutive cycles (also referred to as waste cycles).

This waste cycle number is 10 for the reset simulation stage, and it is 20 for the data simulation stage.

If you set the value of the `ieffort` parameter to a positive integer value (say `N`), the tool performs the following steps:

1. It multiplies the simulation cycle count and waste cycle count with `N` (which in effect multiplies the time spent in initial state search by `N`).
2. It deactivates sets/resets and performs clocked simulation for the total number of cycles calculated in the above step.

The tool performs the above steps over and above the default behavior of applying asynchronous set/resets on a design and then performing clocked simulation.

Setting the `ieffort` parameter to a negative value (`-1` to `-3`) produces different results, as explained in the following table:

| Value | Result |
|---|---|
| -1 | Complete initial state search is skipped all together |
| -2 | Initial state of all flip-flops to forced to 0 |
| -3 | Initial state of all flip-flops is forced to 1 |

| | |
|---|---|
| Used by | All functional rules |
| Options | -3 to any positive integer value |
| Default value | 0 |
| Example | |
| *Console/Tcl-based usage* | `set_parameter ieffort 2` |
| *Usage in goal/source files* | `-ieffort=2` |

## audit

Use the `audit` parameter to quickly explore the assertion checking opportunities in a design without performing the actual formal analysis.

When the `audit` parameter is set, SpyGlass Auto Verify solution does not perform functional analysis. However, the violation report is still generated. The *Info Rules* are also run unless you explicitly disable them.

When the `audit` parameter is not set (default), SpyGlass Auto Verify solution performs functional analysis.

On passing *av_dump_assertions* with `audit`, the design is run in the `audit` mode and `SVA` for all assertions is generated for the rules specified in the *Used by* section.

| | |
|---|---|
| Used by | All functional rules |
| Options | yes, no |
| Default value | no |
| Default Value in GuideWare2.0 | yes |
| Example | |
| *Console/Tcl-based usage* | `set_parameter audit yes` |
| *Usage in goal/source files* | `-audit=yes` |

## passfail

Specifies whether SpyGlass Auto Verify solution checks the properties for proof only, for failure only, or for both.

The allowed values of the `passfail` parameter are as follows:

| **Value** | **Behavior** |
|---|---|
| pass | Enables pass-centric checking (Select if you expect more properties to pass in your design). |

| Value | Behavior |
|---|---|
| fail | Enables fail-centric checking (Select if you expect more properties to fail in your design) |
| both | (Default) Enables both pass-centric and fail-centric checking |

Use the `passfail` parameter as per your design characteristics. Setting it to `pass` or `fail` may result in improved runtime performance. In all cases, both "Proved" and "Failed" cases are reported.

| | |
|---|---|
| Used by | All functional rules |
| Options | pass, fail, both |
| Default value | both |
| Example | |
| *Console/Tcl-based usage* | `set_parameter passfail pass` |
| *Usage in goal/source files* | `-passfail=pass` |

## dead_code_scope

The `dead_code_scope` parameter specifies the type of constructs to be checked by the *Av_deadcode01* rule.

| | |
|---|---|
| Used by | *Av_deadcode01* |
| Options | Comma-separated list of *Possible Values of the dead_code_scope Parameter*. |
| Default value | if, case_without_default, generate, always |
| Example | |
| *Console/Tcl-based usage* | `set_parameter dead_code_scope 'if,case'` |
| *Usage in goal/source files* | `-dead_code_scope='if,case'` |

### Possible Values of the dead_code_scope Parameter

The `dead_code_scope` parameter accepts the following values:

- `if`

  The *Av_deadcode01* rule checks for the `if-else` constructs only.
- `case`

  The *Av_deadcode01* rule checks for the `case` constructs and `case default` blocks only.
- `case_without_default`

  The *Av_deadcode01* rule checks for the `case` constructs only without checking the `default` label.
- `condasgn`

  The *Av_deadcode01* rule checks for conditional assignments only for **Verilog** and `CONDSIGASGN/SELSIGASGN` statements for **VHDL**.
- `if_case`

  The *Av_deadcode01* rule checks for `if` and `case` blocks.
- `if_case_condasgn`

  The *Av_deadcode01* rule checks for `if` and `case` blocks.

  For **Verilog**, this rule also checks for conditional assignments, as shown in the following example:

  ```
  a = b ? c : d
  ```

  For **VHDL**, this rule checks for `CONDSIGASGN/SELSIGASGN` statements.
- `generate`

  The *Av_deadcode01* rule checks for the `dead-if` and `case_without_default` blocks specified inside `generate` blocks.
- `always`

  The *Av_deadcode01* rule processes the `always` blocks containing the `if` and `case_without_default` blocks.

## detect_assign_fsm

Setting the `detect_assign_fsm` parameter causes the *Av_fsm01*, *Av_fsm02*, *Av_fsminf01*, and *Av_fsminf02* rules to detect assign-style FSMs in addition to detecting case style FSMs (default).

**NOTE:** *The* `detect_assign_fsm` *parameter is available for Verilog designs only.*

| | |
|---|---|
| Used by | *Av_fsm01*, *Av_fsm02*, *Av_fsminf01*, and *Av_fsminf02* |
| Options | yes, no |
| Default value | no |
| Example | |
| *Console/Tcl-based usage* | `set_parameter detect_assign_fsm yes` |
| *Usage in goal/source files* | `-detect_assign_fsm=yes` |

## detect_ifelse_fsm

Enables the *Av_fsm01*, *Av_fsm02*, *Av_fsminf01*, and *Av_fsminf02* rules to detect if-else style FSMs in addition to detecting case style FSMs (default).

| | |
|---|---|
| Used by | *Av_fsm01*, *Av_fsm02*, *Av_fsminf01*, and *Av_fsminf02* |
| Options | yes, no |
| Default value | no |
| Example | |
| *Console/Tcl-based usage* | `set_parameter detect_ifelse_fsm yes` |
| *Usage in goal/source files* | `-detect_ifelse_fsm=yes` |

## detect_nested_fsm

Setting the `detect_nested_fsm` parameter causes the *Av_fsm01*, *Av_fsm02*, *Av_fsminf01*, and *Av_fsminf02* rules to detect nested if-else style FSMs, nested case style FSMs, and assign style FSMs in addition to detecting case style FSMs (default).

1. For assign style FSMs, state-labels are back-referenced to the line where it has been compared with the current state vector. Hence, in the following example, any rule-violation for the `FSM1_ST3` state is indicated on third line of the assign statement as highlighted and not on

the second line where a transition to FSM1_ST3 state has been specified:

```
assign ns =
  (cs == FSM1_IDLE)
  ? (ctl? FSM1_ST2 : FSM1_IDLE) : (cs == FSM1_ST2)
  ? (!ctl? FSM1_ST3 : FSM1_IDLE) : (cs == FSM1_ST3)
  ? FSM1_ST3 : FSM1_IDLE ;
```

**NOTE:** *In case of nested FSMs specified using separate sequential and combinational blocks, if the next state vector is not given a default value, it might lead to ambiguous results as the state vector can then take any value till the FSM actually gets invoked leading to false or missing rule-violations.*

| | |
|---|---|
| Used by | *Av_fsm01*, *Av_fsm02*, *Av_fsminf01*, and *Av_fsminf02* |
| Options | yes, no |
| Default value | no |
| Example | |
| *Console/Tcl-based usage* | `set_parameter detect_nested_fsm yes` |
| *Usage in goal/source files* | `-detect_nested_fsm=yes` |

## fv_dcode_all_inst

By default, the *Av_deadcode01* rule highlights any one instance of a deadcode module. Set this parameter to "yes" to view the schematic and waveform for all the instances of the deadcode module.

| | |
|---|---|
| Used by | *Av_deadcode01* |
| Options | yes, no |
| Default value | no |
| **Example** | |
| *Console/Tcl-based usage* | `set_parameter fv_dcode_all_inst yes` |
| *Usage in goal/source files* | `-fv_dcode_all_inst=yes` |

## fv_parallelfile

Specifies a configuration file for distributed runs of the *Av_sanity06* rule over several machines.

The configuration file is an ASCII text file that contains specific lines for different methods, as discussed below:

- The `lsf` method contains the following lines:

  ```
  LOGIN_TYPE: lsf
  MAX_PROCESSES: <num>
  LSF_CMD: <bsub-command>
  ```

  Details of various arguments and keywords are discussed below:

  - ❒ Specify the value of the `LOGIN_TYPE` keyword as `lsf`.
  - ❒ The *<num>* argument of the `MAX_PROCESSES` keyword specifies the maximum number of processes to be spawned.
  - ❒ The *<bsub-command>* argument of the `LSF_CMD` keyword specifies the LSF invocation command. (default is `bsub`).

  The following table describes the arguments and keywords of the above method:

| **Argument/Keyword** | **Description** |
|---|---|
| <num> | Specifies the maximum number of processes to be spawned. |
| <bsub-command> | Specifies the LSF invocation command. (default is bsub).<br>SpyGlass generates details of the bsub command, which is used in parallel LSF runs, in a log file. This information is useful while debugging.<br>To generate complete information of the bsub command, set the *verbose* parameter to 2. |

**NOTE:** *To know the runtime details of SpyGlass Auto Verify rules that are run on same or different machines, refer to distributed_time report.*

**NOTE:** *In a parallel file specified by the* `fv_parallelfile` *parameter, the* `-I`, `-Ip`, *and* `-Is` *options of the* `bsub` *command are not allowed in the* `LSF_CMD` *keyword. This is because while running the* `bsub` *command, SpyGlass internally passes the* `-K` *option, which is mandatory for running parallel assertion runs. However, the* `bsub` *command does not allow the* `-K` *option along with the* `-I`, `-Ip`, *and* `-Is` *options. Therefore, if you specify these options, parallel assertions are not run and the assertion status may remain partially-proved.*

Following is the example of the `lsf` method:

```
LOGIN_TYPE: lsf
MAX_PROCESSES: 3
LSF_CMD: bsub -q “normal | priority”
```

In the above example, the `-q` option is used to specify the queue as `normal` or `priority`.

The `LSF_CMD` command should contain necessary options required to run the `bsub` command in a particular environment. In most cases, the `bsub` options that are required to launch the main SpyGlass run should be passed through `LSF_CMD` so that child processes launched on `bsub` are run using the same `bsub` options.

- The `rsh` and `ssh` methods contain the following lines:

```
LOGIN_TYPE: rsh | ssh
MAX_PROCESSES: <num>

MACHINES:

<machine1-name>[:<num-processes>]
<machine2-name>[:<num-processes>]
...
```

Details of various arguments and keywords are discussed below:

- ❒ Specify the value of the `LOGIN_TYPE` keyword as `rsh` or `ssh` as per your requirement.
- ❒ The *<num>* argument for the `MAX_PROCESSES` keyword specifies the maximum number of processes to be spawned.
- ❒ The *<machine1-name>*, *<machine2-name>*,... arguments refer to the machine names.

- ❒ The `<num-process>` argument refers to the number of processes to be spawned on the specified machine. By default, the value of this argument is `1`.

**NOTE:** *Each spawned process uses one* `Auto_Verify` *license.*

**NOTE:** *If any issues are found in the parallel file, the Av_sanity06 reports a violation.*

By default, this parameter is not set to any value, and therefore, distributed runs are not enabled.

| | |
|---|---|
| Used by | *Av_sanity06* |
| Options | File name |
| Default value | "" |
| **Example** | |
| *Console/Tcl-based usage* | `set_parameter fv_parallelfile 'machinelist.txt'` |
| *Usage in goal/source files* | `-fv_parallelfile='machinelist.txt'` |

## fv_debug_sim_cycles

Specifies the number of cycles of the slowest clock in property pack for which waveform should be displayed from initial state for failed properties of the Av_deadcode01 and Av_staticnet01 rules.

**NOTE:** *High value of fv_debug_sim_cycles will lead to high runtime for waveform generation.*

| | |
|---|---|
| Used by | *Av_deadcode01*, *Av_staticnet01* |
| Options | Any positive integer |
| Default value | 0 |
| Example | |
| *Console/Tcl-based usage* | `set_parameter fv_debug_sim_cycles 1` |
| *Usage in goal/source files* | `-fv_debug_sim_cycles 1` |

## include_construct

Specifies if the `generate_block` and `always_comb` constructs should be checked by the *Av_deadcode01*, *Av_dontcare01*, and *Av_range01* rules, and if dead code should be checked in the include files by the *Av_deadcode01* rule.

You can set this parameter to the following values:

- `generate`

  Enables the *Av_deadcode01*, *Av_dontcare01*, and *Av_range01* rules to consider the `generate_blocks` constructs for rule-checking.

- `always_comb`

  Enables the *Av_deadcode01*, *Av_dontcare01*, and *Av_range01* rules to consider the `always_comb` constructs for rule-checking.

- `included_file`

  Enables the *Av_deadcode01* rule to check for dead code in the functions in the include files specified by the `'include` directive.

  By default, only the functions in the included file are not checked for dead code

| | |
|---|---|
| Used by | *Av_deadcode01*, *Av_dontcare01*, *Av_range01*, *Av_divide_by_zero* |
| Options | none, generate, always_comb, included_file |
| Default value | generate,always_comb,included_file |
| Example | |
| *Console/Tcl-based usage* | `set_parameter include_construct generate` |
| *Usage in goal/source files* | `-include_construct generate` |

## reset_convention

Specifies the resets to be reported by the *Av_rstinf01* rule.

| | |
|---|---|
| Used by | *Av_rstinf01* |
| Options | Comma or space-separated list of reset names (or Perl regular expressions) |
| Default value | " " |
| Example | |
| *Console/Tcl-based usage* | `set_parameter reset_convention "*rst*,*set*"` |
| *Usage in goal/source files* | `-reset_convention="*rst*,*set*"` |

## resetoff

The `resetoff` parameter disables all user-supplied `reset` constraints.

**NOTE:** *By default, all user-supplied reset constraints are applied.*

| | |
|---|---|
| Used by | All functional rules |
| Options | yes, no |
| Default value | no |
| Example | |
| *Console/Tcl-based usage* | `set_parameter resetoff yes` |
| *Usage in goal/source files* | `-resetoff=yes` |

## show_static_latches

The `show_static_latches` parameter specifies whether the *Av_staticreg02* rule should report static latches in a design.

By default, the static latches are reported in the *Av_staticreg02* rule spreadsheet.

Set this parameter to `no` to stop reporting static latches in the spreadsheet.

| | |
|---|---|
| Used by | *Av_staticreg02* |
| Options | yes, no |

| | |
|---|---|
| Default value | yes |
| Example | |
| *Console/Tcl-based usage* | `set_parameter show_static_latches no` |
| *Usage in goal/source files* | `-show_static_latches=no` |

## solvemethod

Specifies the effort level for property checking.

You can set the `solvemethod` parameter to the following values:

| **Value** | **Description** |
|---|---|
| `1` (default) | • Property verification as per the *passfail* parameter<br>• Cut-based verification is off unless overridden by *atime* parameter |
| `2` | • Property verification as per the *passfail* parameter<br>• Automatic cut-based verification<br>• Different internal engine invocation sequence (from the other two solvemethod settings) to improve coverage based on the property type (proof-dominant or witness-dominant) |
| `3` | • Property verification as per the *passfail* parameter<br>• Automatic cut-based verification<br>• Different internal engine invocation sequence (from the other two solvemethod settings) to improve coverage based on the property type (proof-dominant or witness-dominant) |

| | |
|---|---|
| Used by | All functional rules |
| Options | 1, 2, and 3 |
| Default value | 1 |
| Example | |
| *Console/Tcl-based usage* | `set_parameter solvemethod 2` |
| *Usage in goal/source files* | `-solvemethod=2` |

## staticnet_scope

The `staticnet_scope` parameter specifies the type of nets to be checked by the *Av_staticnet01* rule.

| | |
|---|---|
| Used by | *Av_staticnet01* |
| Options | Comma-separated list of any of the following values:<br>• flop: Specifies that rule-checking is done on flip-flops only.<br>• lhs: Specifies that rule-checking is done on LHS assignment nets only.<br>• rhs: Specifies that rule-checking is done on RHS assignment nets only.<br>• all: Specifies that rule-checking is done on flip-flops, latches, and nets |
| Default value | flop |
| Example | |
| *Console/Tcl-based usage* | `set_parameter staticnet_scope lhs,rhs` |
| *Usage in goal/source files* | `-staticnet_scope=lhs,rhs` |

## propfile

The `propfile` parameter specifies the property file containing properties to be checked.

**NOTE:** *By default, all properties in the design are checked.*

| | |
|---|---|
| Used by | All functional rules |
| Options | property file name |
| Default value | "" |
| Example | |
| *Console/Tcl-based usage* | `set_parameter propfile abc.txt` |
| *Usage in goal/source files* | `-propfile=abc.txt` |

## modulelist

The `modulelist` parameter specifies the design units for which the functional analysis is to be performed.

By default, SpyGlass Auto Verify solution analyzes the user-defined properties and implicit properties for all design units in the user design. If you specify some particular design units with the `modulelist` parameter, SpyGlass Auto Verify solution analyzes only the specified design units at the highest level. Properties at lower levels of the specified design units or in the remaining design units of the user design are not analyzed. However, the complete design is considered while determining the fan-in/fan-out of signals being checked.

| | |
|---|---|
| Used by | All functional rules |
| Options | Comma- or space-separated list of module names enclosed in double quotes |
| Default value | "" |
| Example | |
| *Console/Tcl-based usage* | `set_parameter modulelist "Fsm Fsm_always"`<br>`set_parameter modulelist "Fsm,Fsm_always"` |
| *Usage in goal/source files* | `-modulelist="Fsm Fsm_always"`<br>`-modulelist="Fsm,Fsm_always"` |

## scope

The `scope` parameter defines the scope of functional analysis.

By default, the `scope` parameter is set to `chip` and the complete fan-in cone of the assertion is taken into account.

Set the `scope` parameter to `block` to have SpyGlass Auto Verify solution cuts all signals in the fan-in cone (except clocks and resets) at the sub-module boundary in which the assertion is formed. All those signals at the boundary of the sub-module are then treated as primary inputs for functional analysis.

| | |
|---|---|
| Used by | All functional rules |
| Options | chip, block |
| Default value | chip |

| Example | |
|---|---|
| *Console/Tcl-based usage* | `set_parameter scope block` |
| *Usage in goal/source files* | `-scope=block` |

## vcdtime

This parameter is deprecated. Use the *simulation_data* constraint instead of this parameter.

## vcdfile

This parameter is deprecated. Use the *simulation_data* constraint instead of this parameter.

## vcdfulltrace

The `vcdfulltrace` parameter specifies whether all signals or only user signals in the fan-in cone of an assertion are dumped in the VCD file. The default value of the `vcdfulltrace` parameter is `usernets`.

You can set the `vcdfulltrace` rule parameter to the following values:

| **Value** | **Description** |
|---|---|
| no | Only the flip-flop output signals and primary inputs in the fan-in cone of an assertion are written to the VCD file |
| usernets | All the user nets in the fan-in cone of an assertion are written to the VCD file |
| allnets | All internally generated nets along with the user-defined signals in the fan-in cone of an assertion are written to the VCD file |

| | |
|---|---|
| Used by | All functional rules |
| Options | no, usernets, allnets |
| Default value | usernets |

| Example | |
|---|---|
| *Console/Tcl-based usage* | `set_parameter vcdfulltrace allnets` |
| *Usage in goal/source files* | `-vcdfulltrace=allnets` |

## verbose

The `verbose` parameter specifies the verbosity level of the messages printed at the standard output.

You can set the `verbose` parameter to values 0 (default), 1, 2, and 3. Higher the value, more messages are printed.

| | |
|---|---|
| Used by | All functional rules |
| Options | 0, 1, 2, 3, 4 |
| Default value | 0 |
| Example | |
| *Console/Tcl-based usage* | `set_parameter verbose 2` |
| *Usage in goal/source files* | `-verbose=2` |

## xassign_casedefault

The `xassign_casedefault` parameter specifies whether the *Av_dontcare01* rule should check the X-assignment inside the `default` clause of a `case` statement.

By default, this parameter is set to `no`, and the Av_dontcare01 rule does not check the X-assignment inside the `default` clause of a `case` statement. Set this parameter to `yes` to check all the clauses of case statement.

| | |
|---|---|
| Used by | *Av_dontcare01* |
| Options | yes, no |
| Default value | no |

| Example | |
|---|---|
| *Console/Tcl-based usage* | `set_parameter xassign_casedefault yes` |
| *Usage in goal/source files* | `-xassign_casedefault=yes` |

## Functional Constraints

The functional constraints are complex constraints restricting the search space for functional analysis and are defined using assertions. For instance, if you know that an input vector of a sub-block of a design is one hot encoded, you can provide this information as a functional constraint to SpyGlass Auto Verify solution.

## Impact of Constraints on Functional Analysis

Functional constraints are important in two different ways:

- Boundary assumptions

  If SpyGlass Auto Verify solution does not know about the boundary assumptions, it may generate a counter example to show that a property is not holding. However, the counter example may be violating the boundary assumptions.

  For example, as shown in *Figure* , bus contention is reported unless (a + !b) is specified as a functional constraint, the condition a=0, and b=1 may cause contention on the bus. However, if the condition (a || !b) is provided as a functional constraint, then no message is reported. (a || !b) is equivalent to "'a' should be 'high' or else 'b' must be low".

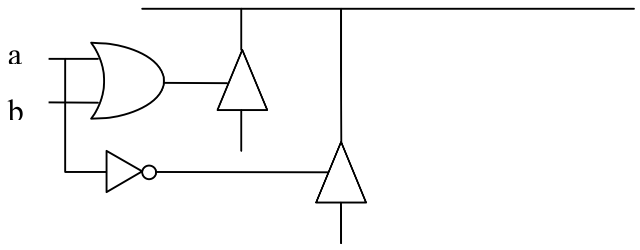


**FIGURE 3.** Bus Contention reported unless (a || !b) is specified as constraint

- By defining constraints, the search space, which is explored to prove or fail a property, is reduced and consequently the run time may be lowered.

## Specifying Functional Constraints

Functional constraints are specified the same way as OVL assertions. An OVL assertion has an option that tells SpyGlass Auto Verify solution whether the assertion is instantiated as a constraint or as an assertion to be validated.

For example, the following Verilog description represents the design in *Figure* with a constraint preventing *Bus Contention*.

```
module BusConstraint(in1,in2,a,b,out);
  input in1,in2,a,b;
  output out;
  wire en1, en2;
  assign en1 = a | b;
  assign en2 = !a;
  assign out= en1  ? in1 : 1'bz;
  assign out = en2 ? in2: 1'bz;
  assert_proposition #(0,1) constraint1(1'b1,(a || !b ));
endmodule
```

The expression (a||!b) will prevent bus contention.

For detailed use of OVL assertions, see *Properties Specification using OVL*.

## Over Constraint

Functional constraints are used to model the environment or help SpyGlass Auto Verify solution in concluding the analysis; in doing so you may introduce constraints that are conflicting between them or with the design itself. This conflict is happening because the design is *over-constrained*. SpyGlass Auto Verify solution reports such scenarios in the *Av_sanity04* rule. Using this rule, you would be able to identify conflicting constraints.

# Properties Specification using OVL

SpyGlass Auto Verify solution supports Open Verification Library (OVL) for user-specified properties. OVL is a predefined set of Verilog and/or VHDL design units that can be instantiated in a design just like a regular design unit. SpyGlass Auto Verify solution uses these instantiations to validate the corresponding functionality. The OVL checks as well as other *implicit properties* supported by SpyGlass Auto Verify solution are described in *Rules in SpyGlass Auto Verify*.

## OVL Assertions Format

In Verilog, an OVL module can be instantiated using the following format:

```
Assertion_identifier [parameter_value_assignment]
module_instance;
```

Where:

- `Assertion_identifier` is the OVL assertion module name. For example, `assert_always`, `assert_next`, etc.
- Parameters are:
  - Severity_level: Ignored by SpyGlass Auto Verify solution
  - Assertion-specific parameters: this is a list of parameters (zero or many) used by the specific assertion only.
  - Options: Is a 32-bit integer optional argument. If ‘0’ the instance must be treated as a *property* to be validated. If ‘1’ the instance is treated as *functional constraint*.
  - Msg: A message string ignored by SpyGlass Auto Verify solution and replaced by regular rule’s message.
- `Module_instance` is an instantiation of the OVL module. This is done the same way as a regular module instantiation in Verilog/VHDL. The interface of these modules depends on the specific assertion, however, the first two ports of all assertions are the same: “clock”, and “reset”.

### Verilog Example

The following Verilog example represents an FSM:

```
// One hot FSM encoding
```

```
`define STATE1 4'b1000
`define STATE2 4'b0100
`define STATE3 4'b0010
`define STATE4 4'b0001

module Fsm(reset, clk, ctl, out);
  input reset, clk, ctl;
  output out;
  reg [0:3] state;

  always @(posedge clk) begin
    if(!reset) state <= `STATE1;
    else begin
      case (state)
        `STATE1 : if(ctl) state <= `STATE2;
        `STATE2 : state <= `STATE3;
        `STATE3 : state <= `STATE4;
        `STATE4 : state <= `STATE1;
        default : ;
      endcase
    end
  end
  assign out = state[0];
  assert_one_hot #(0,4)oh_check(clk, reset, state);
  assert_next #(0, 2, 0) stateTransitionCheck(clk,
    reset, state[1], state[0]);
endmodule
```

Refer the first assertion instantiated at the end of the example:

```
assert_one_hot #( 0, 4 ) oh_check( clk, reset, state );
```

This statement asserts that `state` is a one hot encoded vector. The first parameter, ignored by SpyGlass Auto Verify solution, is the severity_level set to 0. The second parameter (4) is the width of the `state` vector. The instance name is `oh_check`. This assertion module has three inputs namely clock, reset, and the test vector supposed to be one hot encoded (corresponding to instance ports `clk`, `reset`, and `state` respectively).

Now, refer the second assertion instantiated at the end of the example:

```
assert_next #(0, 2, 0) stateTransitionCheck(clk, reset, state[1],
state[0]);
```

This statement asserts that `state[0]` always comes two cycles after `state[1]`. The second parameter is the number of cycles (2), counting from the `state[1]` activation, after which `state[0]` is supposed to happen. The last parameter set to `0` prevents the check for overlapping events on the `state[1]` signal (once an event is detected on the `state[1]` signal, we start the check and a second event would not start a new check till the first check is over).

### VHDL Example

The following VHDL example represents an FSM:

```
library accellera;
 USE accellera.ovl_assert.ALL;
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.Numeric_Std.ALL;

entity top is
  port(clk : in bit;
       rst   : in bit;
       rdReq   : in bit;
       wrReq   : in bit;
       status1  : out unsigned(2 downto 0));
end top;

architecture str of top is
  signal ptr : unsigned(3 downto 0);
  signal status : unsigned(2 downto 0);
  constant NOP : unsigned(2 downto 0) := "000";
  constant READ_REFUSED : unsigned(2 downto 0) := "001";
  constant READ_ACCEPTED : unsigned(2 downto 0) := "010";
  constant WRITE_REFUSED : unsigned(2 downto 0) := "011";
  constant WRITE_ACCEPTED : unsigned(2 downto 0) := "100";
  constant RDWR_NOT_ALLOWED : unsigned(2 downto 0) := "111";
  constant SIZE : unsigned(3 downto 0) := "1111";
  -- ovl module
--  COMPONENT assert_never IS
--  GENERIC (severity_level: INTEGER := 0;
--           options: INTEGER := 0;
--           msg: STRING := "ASSERT NEVER VIOLATION");
--  PORT (clk, reset_n, test_expr: IN std_ulogic);
--  END COMPONENT;
  signal te : boolean;
begin
  process(clk, rst)
  begin
    if(clk'event and clk='1') then
      if(rst='0') then
        status <= NOP;
ptr <= "0000";
      else
        if(rdReq='1' and wrReq='0') then
  if(ptr = 0) then
    status <= READ_REFUSED;
  else
    status <= READ_ACCEPTED;
    ptr <= ptr - 1;
  end if;
elsif(rdReq='0' and wrReq='1') then
  if(ptr = SIZE) then
    status <= WRITE_REFUSED;
  else
    status <= WRITE_ACCEPTED;
    ptr <= ptr + 1;
  end if;
elsif(rdReq='1' and wrReq='1') then
  status <= RDWR_NOT_ALLOWED;
else
  status <= NOP;
end if;
```

```
      end if;
    end if;
  end process;

 te <= true when wrReq='1' and ptr < SIZE and status =
WRITE_REFUSED else false;
  Inst : assert_never generic map(failure,0) port map (clk =>
To_StdULogic(clk),
 reset_n => To_StdULogic(rst),
 test_expr => te);
end str;
```

In the above example, refer to the assertion instantiated at the end of the example, that is:

```
te <= true when wrReq='1' and ptr < SIZE and status
=WRITE_REFUSED else false;
```

```
Inst : assert_never generic map(failure,0) port map (clk
=>To_StdULogic(clk),reset_n => To_StdULogic(rst),test_expr
=> te);
```

This statement asserts that:

- The `te` expression never occurs.
- The first parameter, which is ignored by SpyGlass Auto Verify solution, is the `severity_level`.
- The second parameter (0) specifies this OVL is being used as an assertion.
- The instance name is `Inst`.
- This assertion module has three inputs: clock, reset, and the test expression corresponding to the instance ports `clk`, `rst`, and `te`, respectively.

## Constant Value Control Signals in OVL Assertions/ Assumptions

Normally, you are expected to specify an actual clock signal as the first port of the assertion/assumption (except the `assert_proposition` assertion which is a purely combinational check). In case, the specified clock signal is forced to a constant value in the design or you specify a constant value (0 or 1) instead of a clock signal in the assertion, the corresponding assertion will never reach an error condition and will always be proved.

By definition, all resets in the OVL modules are active low. Therefore, constraints resets all OVL instances to high value (that is, deactivate the reset) for proper functioning of OVL instances. In case, the specified reset signal is forced to a constant low value in the design or you specify a constant value 0 instead of a reset signal in the assertion, SpyGlass Auto Verify solution reports the constraint to be unsatisfied as the reset is being constrained in two different values. Thus, the corresponding assertion will not be checked for pass or fail. In case, the specified reset signal is forced to a constant high value in the design or you specify a constant value 1 instead of a reset signal in the assertion, SpyGlass Auto Verify solution checks the assertion in the normal way.

In case of the assumptions, a constant value clock signal or constant 0 reset will result in the corresponding constraint being ignored/not applied.

## OVL Assertions in Combinational Circuits

All OVL assertions except the `assert_proposition` are intended for sequential circuits and require you to specify a clock signal.

You can still use some of the OVL assertions with combinational circuits by creating a virtual clock and specifying it in an OVL assertion.

Consider the following example:

```
module test(d1, d2, e1, e2, e3, e4, e5, e6, out1);
  input d1, d2, e1, e2, e3, e4, e5, e6;
  output out1;

  reg ena1, ena2;

  always @(e1 or e2 or e3 or e4 or e5 or e6) begin
     ena1 = (e1 & e2 & !e3) | (e2 & e3 & !e4) |
               (e3 & e4 & !e5) | (e4 & e5 & !e6);
     ena2 = (!e1 | !e2 | e3) & (!e2 | !e3 | e4)
                 & (!e3 | !e4 | e5) & (!e4 | !e5 | e6);
    end
endmodule
```

You can now create a virtual clock (say clk) and use them in the OVL assertions as shown below:

```
module test_assertion(d1,d2,e1,e2,e3,e4, e5, e6, out1);
  input   d1, d2, e1, e2, e3, e4, e5, e6;
  output out1;

  wire    clk;
  reg ena1, ena2;

  always @(e1 or e2 or e3 or e4 or e5 or e6) begin
    ena1 = (e1 & e2 & !e3) | (e2 & e3 & !e4)
                | (e3 & e4 & !e5) | (e4 & e5 & !e6);
    ena2 = (!e1 | !e2 | e3) & (!e2 | !e3 | e4) &
                  (!e3 | !e4 | e5) & (!e4 | !e5 | e6);
  end

  assert_one_hot #(0, 2) check_one_hot_pass
                   (clk, 1'b1, {ena1, ena2});
  assert_one_hot #(0, 2) check_one_hot_fail
                     (clk, 1'b1, {!ena1, ena2});
endmodule
```

Now, you can analyze the design for `assert_one_hot` assertions. The first assertion will pass and the second assertion will fail as designed.

## Separate File OVL Support

SpyGlass Auto Verify solution allows OVL insertion inside an HDL module or in a separate file. An OVL assertions file has the following format:

```
attach_properties <module-name>
begin_ovl
// Comments
```

```
<ovl_assertions_instance>;
<ovl_assertions_instance>;
end_ovl
```

Where *`<module-name>`* is the module into which the OVL assertions are instantiated/bound.

Signal names specified in the assertions instantiations should be hierarchical names with respect to the binding module *`<module-name>`* as in the following example:

```
attach_properties test
begin_ovl
assert_always assrtn4 (clk, 1'b1, ( mid1.w2 == 1'b0 ));
end_ovl
```

Here, `w2` is a signal in the instance `mid1` under the binding module `test`.

Mixed-language is supported which means that OVL file may use Verilog format while the design is a VHDL design and vice-versa.

To read an OVL Verilog file in Atrenta Console, specify the following command in a project file:

```
set_option ovl_verilog { OVL-file> }
```

Similarly, to read an OVL VHDL file, specify the following command in a project file:

```
set_option ovl_vhdl { OVL-file> }
```

## Restrictions in Using OVL

Following restrictions are important in using OVL to specify properties for SpyGlass Auto Verify solution:

- The severity_level argument is ignored by SpyGlass Auto Verify solution.
- Assertion message is ignored by SpyGlass Auto Verify solution. It is replaced by corresponding rule’s message in SpyGlass Auto Verify solution.

## Impact of Property and Constraint Modules

The presence of an OVL assertion instance in a design may have impact on any tool loading and working on the corresponding HDL code. The specificity of OVL assertion modules is that they do not feed any other instance or ports of the design. Because of this feature, the OVL assertion instances can be seen as hanging instances in a design. Synthesis tools can literally remove them because they are not observable at the outputs. SpyGlass Auto Verify solution processes OVL assertions as follows:

- Property and Constraint modules are made invisible to all products except SpyGlass Auto Verify solution. These instances are seen by other products as black boxes with the following differences:
  - ❒ Assertion modules are analyzed and any syntax or semantic problem in those modules are reported
  - ❒ Regular products do not perform further checks on them, and therefore, no rule-violations are reported involving these modules.
- Property and Constraint modules will be visible through schematic; you can explore the content of these modules from the Schematic Window.

## Processing Property and Constraint Modules

OVL assertions as defined by Accellera are not synthesizable. To validate the functionality, SpyGlass Auto Verify solution extracts the functionality of the modules. To do so, the following transformations are internally performed:

1. Error report

   In Verilog, OVL assertion failure causes a task call which reports a failure, and increments the fail count. SpyGlass Auto Verify solution ignores the task and processes the failure using its rule-violation report mechanism.

2. Clock and Reset inputs

   All OVL assertions have clock and reset input ports. In case the property is a pure combinational check, SpyGlass Auto Verify solution accepts clock/reset ports that are driven by constant 0/1. In this case these ports are ignored during functional analysis.

3. OVL Verilog Macro Global Variables

   OVL Verilog provides a set of macros (`ASSERT_GLOBAL_RESET`, `ASSERT_MAX_REPORT_ERROR`, etc.). OVL Verilog Macro Global

Variables are ignored by SpyGlass Auto Verify solution.

## Property and Constraint Management

Properties and constraints can be edited through Atrenta Console GUI. Individual checks can be enabled, disabled, or set an explicit property as a constraint. The updated property status can be dumped in the auto_verify.prp file in the current working directory. This file can be read in subsequent runs of SpyGlass Auto Verify solution. This file is referred to as the property file and can be also directly edited and changed.

When a property file is provided using the *propfile* parameter, SpyGlass Auto Verify solution checks only the assertions from that file.

SpyGlass Auto Verify solution provides properties and constraints exploration capabilities. This feature can be used as follows:

1. You can run a goal of SpyGlass Auto Verify solution (with *audit* parameter) to generate a list of all properties and constraints.
2. The list of properties can be visualized in Atrenta Console.
3. The attributes of properties can be modified. You can un-select some rules or mark them as constraints or some of the constraints can be reset as assertions to be checked.
4. The output of this property exploration and editing session can be dumped into a file.
5. A property file can be manually edited and modified.

In particular, the Property file feature can be used for incremental validation purposes. SpyGlass Auto Verify solution can be run with default effort level. The property/constraint file can be simplified so it contains only properties that are partially analyzed. Then, an incremental run can be launched using the modified property file so the previously validated assertions are not re-analyzed.

## Property File Format

The Property file has the rule-wise assertion/constraint information in the following format:

```
RuleName: <rule-name>
<selection> <type> <status> <file-name> <line-num> <hier>
   [ <info> ]
...
```

```
RuleName: <rule-name>
...
```

Where:

- *<rule-name>* is the name of the rule of SpyGlass Auto Verify solution.
- *<selection>* is `on` when the assertion/constraint is enabled or is `off` when the assertion/constraint is disabled.
- *<type>* is the property type — `Assertion` or `Constraint`.
- *<status>* indicates the assertion status:

| **<status> Value** | **Indicates that the assertion ...** |
|---|---|
| PROVED | Proved in the current run |
| FAILED | Failed in the current run |
| Partially-Analyzed | Partially-analyzed in the current run |
| Constraint-Unsatisfied | Constraint-Unsatisfied in the current run |
| Not-Analyzed | Not analyzed in the current run or a previous run |
| [PROVED] | Proved in a previous run |
| [FAILED] | Failed in a previous run |
| [Partially-Analyzed] | Partially-analyzed in a previous run. |
| [Constraint-Unsatisfied] | Constraint-Unsatisfied in a previous run |

**NOTE:** *The status for constraints is indicated as* `NA`*.*

**NOTE:** *The properties with status Partially-Analyzed are reported with selection* `on` *and the properties with all other status are reported with selection* `off`*. You can modify the selection as required for the next run.*

- *<file-name>* and *<line-num>* is the location of the assertion/constraint.
- *<hier>* is the design hierarchy where the assertion/constraint was checked.
- *<info>* is printed for selected rules only. The details are described under the respective rules.

## Property File Example

The contents of a typical property file are as in the following example:

```
RuleName: Av_deadcode01
off Assertion  PROVED    ../src/test.v 203 uart_top.u11
on Assertion Partially-Analyzed  ../src/test.v 211
uart_top.u11
...
```

## Property File Processing

The following actions are taken with regard to the assertions and constraints in a property file:

1. SpyGlass Auto Verify solution exits with a FATAL error if a constraint in the property file is not found in the design.
2. SpyGlass Auto Verify solution skips checking for a rule when a related assertion is not found in the design. A *Av_sanity01* rule message is reported for the rule. No information is printed in the new property file for such rules. The remaining rules are still processed as applicable.
3. The attributes of assertions in the property file override the attributes of assertions in the HDL instances. For instance, if a design assertion (functional constraint) is specified as a functional constraint (assertion) in the property file, SpyGlass Auto Verify solution processed it as a functional constraint (assertion).
4. Any assertion/constraint instantiated in a design that is not present in the property file or is explicitly disabled in the property file is ignored for functional analysis.
5. If SpyGlass Auto Verify solution is run with a property file provided, the new property file generated is based on the existing property file incorporating the results of the current run. Thus, design properties that are not in the original property file do not appear in the new property file.

## Enabling and Disabling Assertions

While debugging SpyGlass Auto Verify issues, you may want to focus on the violations of specific assertions, such as passed, failed, or partially proved.

In such cases, use *The Property Manager* dialog to select and/or deselect assertions so that these changes are saved in the property file. In the next SpyGlass run, pass that property file to run the selected assertions.

For details on this dialog, refer to the SpyGlass CDC Rules Reference Guide.

## Schematic Highlight and Cross Probing

The schematic highlight for a given violation is handled the same way as with other products in SpyGlass. A small gate icon indicates the presence of a schematic link for a given violation. For the detail of schematic highlight for a specific rule, see the rule description.

## Waveform Display and Cross Probing

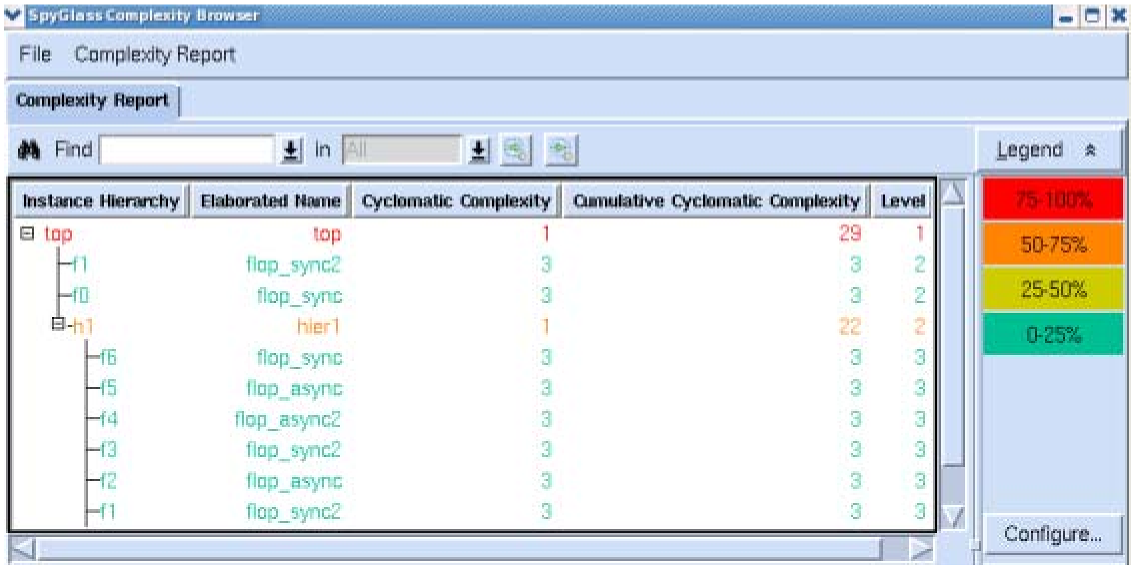
The Waveform Viewer is another analysis tool provided with SpyGlass Auto Verify solution for root cause analysis of a functional bug in a design. In fact, when a property fails, often, SpyGlass Auto Verify solution generates a witness for the failure. This witness is a sequence of events from an initial state of a design/sub-design to the time when the bug appears. This witness can be generated as a set of simulation vectors in VCD format. Each event or time frame in VCD corresponds to an edge of a clock relevant to the violation. By left-clicking on a violation, besides RTL back annotation and schematic highlight, a waveform viewer is launched displaying the VCD content. The presence of a waveform display is indicated by a small waveform icon in front of a violation. For detail waveform information for a specific rule, refer to the rule description.

All the VCD files are dumped in the VCD directory.

## The Complexity Browser

The complexity browser shows a hierarchical structure of module instances and their details, such as cyclomatic complexity, cumulative complexity, and instance depth with respect to the top module.

To view this browser, double-click on the violation of the *Av_complexity01* rule. The following figure shows the complexity browser:



| Instance Hierarchy | Elaborated Name | Cyclomatic Complexity | Cumulative Cyclomatic Complexity | Level |
|---|---|---|---|---|
| ⊟ top | top | 1 | 29 | 1 |
| ├f1 | flop_sync2 | 3 | 3 | 2 |
| ├f0 | flop_sync | 3 | 3 | 2 |
| ⊟-h1 | hier1 | 1 | 22 | 2 |
| ├f6 | flop_sync | 3 | 3 | 3 |
| ├f5 | flop_async | 3 | 3 | 3 |
| ├f4 | flop_async2 | 3 | 3 | 3 |
| ├f3 | flop_sync2 | 3 | 3 | 3 |
| ├f2 | flop_async | 3 | 3 | 3 |
| ├f1 | flop_sync2 | 3 | 3 | 3 |

**FIGURE 4.** SpyGlass Complexity Browser

In the above browser, the legend shows the range of complexity percentage of each instance. The complexity percentage is the ratio of cyclomatic complexity of an instance to the cyclomatic complexity of the top module.

## Configuring the Complexity Percentage

You can configure the complexity percentage by:

- *Changing the Percentage Range*
- *Changing the Percentage-Range Color*

## Changing the Percentage Range

To change the percentage range, perform the following steps:

1. Click the *Configure* button in the *SpyGlass Complexity Browser*.

   The *Configure Highlighting Ranges* dialog appears.
2. Move the mouse pointer between two ranges till a double-sided arrow appears.
3. Click the left mouse button and drag the mouse pointer till you get the desired range.

   The following figure shows the example of changing ranges:

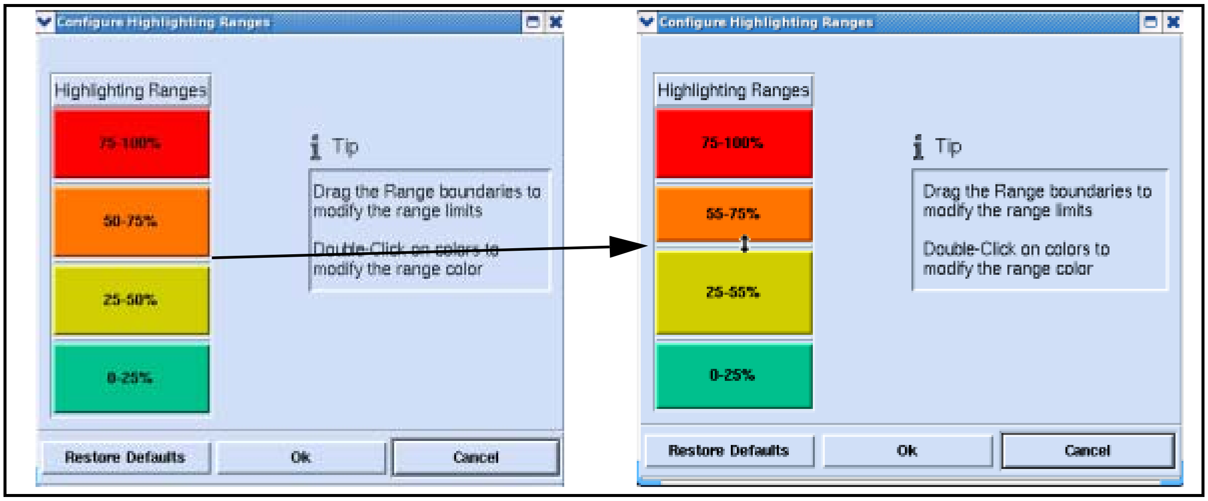

Configure Highlighting Ranges
Highlighting Ranges
75-100%
50-75%
25-50%
0-25%
Tip
Drag the Range boundaries to modify the range limits
Double-Click on colors to modify the range color
Restore Defaults
Ok
Cancel
Configure Highlighting Ranges
Highlighting Ranges
75-100%
55-75%
25-55%
0-25%
Tip
Drag the Range boundaries to modify the range limits
Double-Click on colors to modify the range color
Restore Defaults
Ok
Cancel

**FIGURE 5.** Example of changing the complexity-percentage ranges

4. Release the left mouse button.
5. Click the *OK* button to save the ranges.

## Changing the Percentage-Range Color

To change the color assigned to a complexity-percentage range, perform the following steps:

To change the percentage range, perform the following steps:

1. Click the *Configure* button in the *SpyGlass Complexity Browser*.
   The *Configure Highlighting Ranges* dialog appears.
2. Double-click on a range.
   The *Choose Color for a Range* dialog appears.
3. Set the required color in this dialog.

## Reports and Diagnosis Files in SpyGlass Auto Verify

SpyGlass Auto Verify solution generates report files in the current working directory. Some files are created by default while others are created on user request.

SpyGlass Auto Verify solution generates the following reports and diagnosis files:

| Report Name | Description |
|---|---|
| *auto_verify.rpt* | Describes the functional analysis statistics of a design |
| *ADV-LINT* | Shows a concise summary of the design, design setup, and verification results of SpyGlass Auto Verify solution. |
| *Fsm.info* | Shows information on the FSMs detected in a design by the *Av_fsminf01* rule. |
| *Av_initistate01.csv* | Shows information about the cause of un initialization for sequential elements and a non-default value for each pin. |
| *Av_staticreg02.csv* | Shows information about input pins of static sequential elements after applying case-analysis and VDD/VSS propagation in a design. |
| *Av_complexity01.csv* | Shows information related to modules and FSMs in a design. |
| *auto_verify* | Shows information that enables you to analyze the cause of a bug or to gather functional analysis statistics. |
| *OverConstrainInfo* | Shows information about conflicting constraints. |
| *auto_verify.reg* | Shows information on registers that are relevant for functional analysis. |
| distributed_time | Shows run time details of SpyGlass Auto Verify rules that are run in parallel on same or different machines. This report is similar to the *distributed_time* report generated by SpyGlass CDC. Refer to the SpyGlass CDC documentation for details on this report. |

## Auto Verify Report

The Auto Verify report, auto_verify.rpt, contains the functional analysis statistics of a design.

The following is a sample Auto Verify report:

```
############################################################
#
# Purpose:
#   This report contains the functional analysis statistics
#   of a design.
#
# Format:
#   It contains the following sections:
#   Section A: Run Parameters
#     Lists the parameters specified in the current run
#   Section B: Clock Information
#     Lists the clock information of the design
#   Section C: Reset Information
#     Lists the reset information of the design
#   Section D: Set-Case Analysis Settings
#     Lists the set case analysis settings used in the design
#   Section E: Initial State of the Design
#     Lists the initial-state statistics of the design along
#     with the reset percentage. The initial state of each
#     register can be seen in auto_verify.reg file.
#   Section F: Results Summary (Current)
#     Lists the statistics of the assertions formed for each
#     rule.
#   Section G: Results Summary (Cumulative)
#     Lists the summary of cumulative set of assertions
#     formed in the current run and the information of
#     earlier runs in the property file. This section is
#     printed when you specify a property file using the
#     -propfile command-line option.
#   Section H: Assertion Details
#     Lists the assertion details.
############################################################
```

```
############################################################

Section A: Run Parameters
------------------------

ignore_latches          :     yes
use_inferred_resets     :     yes
verbose                 :     4

############################################################

Section B: Clock Information


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

(clock): (period); (Clock Source); (rising/falling);
(edgeList); (no. of flops on posedge); (no. of flops on
negedge)
------------------------------------------------------------

top.clk: 10.000000; SGDC; Rising; (5.0, 10.0); 8; 0

Design Cycle: 2(1)

############################################################


Section E: Initial State of the Design
--------------------------------------

Total no of sequential elements: 8
No of '1's: 0
No of '0's: 0
No of 'x's: 8
RESET PERCENTAGE for root 'top'(8 sequential elements) is
'0.00%'
```

```
############################################################

Section F: Results Summary (Current)
------------------------------------


-----------------------------------------------------------
RuleName Passed Failed Partially Proved Not Analyzed Others Total
                     (Average Depth)

------------------------------------------------------------
Av_        1      1        0            0         0      2
negative
_shift
------------------------------------------------------------
Total                1         1            0
0                0              2

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

############################################################

Section G: Results Summary (Cumulative)

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

 NOT APPLICABLE (AS NO PROPERTY FILE PASSED)

############################################################

Section H:  Assertion Detail
-----------------------------

RuleName: Av_negative_shift
-----------------------
1. (Hier:top) (b >>> (~d)), test.v, 9,
(Av_negative_shift.1.1.vcd) :  FAILED through depth 1(1)
2. (Hier:top) (b <<< (^ d)), test.v, 9, :  PROVED
```

```
##################################################################
```

## Auto Verify Central Report

The Auto Verify central report (ADV-LINT.rpt) provides a concise summary of the design, design setup, and verification results of SpyGlass Auto Verify solution.

This information is arranged under the following sections in the report:

- *Setup and Design Audit*
- *Analysis and Verification*

## Setup and Design Audit

This section provides the following information:

- Parameters used in the current run that impact run time and quality of results
- SpyGlass run information

  This information includes total run time, total memory consumed, and total peak memory.
- Design statistics

  This information includes the total number of flat instances, flip-flops, latches, sequential library cells, and modules. In addition, it includes the maximum and average cyclomatic complexity if the *Av_complexity01* rule is run.
- Design setup

  This information includes the following:

  - Information on clocks, such as number of user-defined and black box clocks.
  - Information on resets, such as number of user-defined synchronous and asynchronous resets, and black box resets.
  - Information on incomplete clocks definition, which may lead to false positive or negative assertions.

- ❒ Information on incomplete resets definition, which may lead to wrong initial state identification and false functional violations.
- ❒ Information on constraints making assumptions on design execution.
- ❒ Information on improper initial state definition, which may result in missed design bugs or false violations. For details, refer to auto_verify.reg report (*Register Info Report*).
- ❒ Information on FSMs, such as number of FSMs and interacting FSMs detected. For details, refer to Fsm.info file (*Auto Verify-FSM Report*).

Analyze information under this section to ensure that all the setup and design information is as expected.

To see an example of this section, click *here*.

## Analysis and Verification

This section reports current as well as cumulative (only in *propfile* mode) violation counts.

- Current violation count implies the number of failed assertions during formal analysis of the design in the current run.
- Cumulative violation count implies the number of failed violations during formal analysis of the design in the current or previous runs. This column is displayed in the report only when the design is run in *propfile* mode.

Analyze this information to ensure design correction.

To see an example of this section, click *here*.

## Sample Auto Verify Central Report

Following is the sample Auto Verify Central Report:

```
======================================================================
A. Setup and Design audit: To be reviewed for correctness and signoff.
======================================================================
Parameters used in this run that impact run time and quality of results:
----------------------------------------------------------------------
    atime                                                          : 1
    ieffort                                                     : 0
    propfile                                                    : av.prp
    use_inferred_resets                                         : yes
    use_inferred_clocks                                         : yes
    dead_code_scope                                             : generate
    staticnet_scope                                             : flop

Run Information:
----------------------------------------------------------------------
    Total Time (in sec)                                         : 9
    Total Memory (in KB)                                        : 45847
    Peak Memory (in KB)                                         : 205440

Design Statistics:
----------------------------------------------------------------------
    Total Flat Instances                                        : 4399
    Total Flop                                                  : 807
    Total Latches                                               : 15
    Total Sequential Library Cell                               : 0
    Number of Modules                                           : 22
    Maximum Cyclomatic complexity                               : 376
    Average Cyclomatic complexity                               : 47

Design Setup:
----------------------------------------------------------------------
     Clocks:
     ---------------------------------------
       Number of user-defined clocks                            : 1
       Number of Black-box clocks                                : 0

     Incomplete/Modified Clock Constraints:
     ---------------------------------------
       Clocks with undefined periods (period of 10ns assumed)    : 0
       Clocks with undefined edges (50% duty cycle assumed)      : 1
       Clocks with rounded period for functional verification    : 0

     Resets:
     ---------------------------------------
```

```
      Number of user-defined asynchronous resets             : 1
      Number of user-defined synchronous resets              : 0
      Number of black box resets                             : 0

    Incomplete Reset Constraints:
    ---------------------------------------
     Asynchronous resets with undefined active
     value (active value 1 assumed)                          : 0

     Synchronous resets with undefined active
     value (active value 1 assumed)                          : 0

      Constraints making assumptions on design execution that should be
      reviewed
    ----------------------------------------------------------------
      Number of set_case_analysis constraints                : 0
      Number of assume_path constraints                      : 0

    Initialization
    ---------------------------------------
     Number and percentage of flops uninitialized       : 807(100.00 %)
     Number and percentage of latches uninitialized     : 15(100.00 %)

    FSMs
    ------------------------------------------------------------------
      Number of FSMs detected                                 : 6
      Number of interacting FSMs detected                     : 2


=======================================================================
B. Analysis and Verification: To be reviewed for verification signoff.
======================================================================
Analysis and Verification                            : Current Cumulative
----------------------------------------------------------------------
Multiple Overlapping Parallel Case Items (Av_case02) :    1         1
Bus Contention violation (Av_bus01)                  :    2         2
Floating Bus (Av_bus02)                              :    2         3
```

```
Dead FSM Transitions (Av_fsm02)                        :    3        66
Unreachable/Deadlocked FSMs States (Av_fsm01)          :    1        34
Sensitizable X assignments (Av_dontcare01)             :    1        1
Unspecified Full Case Items (Av_case01)                :    0         1
Dead Code Blocks (Av_deadcode01)                       :   6         91
Array Bound violations (Av_range01)                    :   2          2
```

## Auto Verify-FSM Report

The *Av_fsminf01* generates the Auto Verify-FSM report (Fsm.info).

This report provides information on the FSMs detected in a design by the *Av_fsminf01* rule.

The following example shows the sample report:

```
Number of FSMs detected: 1
FileName: test.v, Line num: 23
FSM: test.state
Number of states: 4
Number of transitions: 5
Number of outputs from fsm: 0
Number of inputs to fsm: 2
Fsm encoding style: MINIMUMENCODED
Fsm style: UNKNOWN
Next State logic figures: Simple Assignments
Depth of FSM: 3 (S0 to S3)
Initial State of FSM: S0
Cyclomatic Complexity of FSM: 2
```

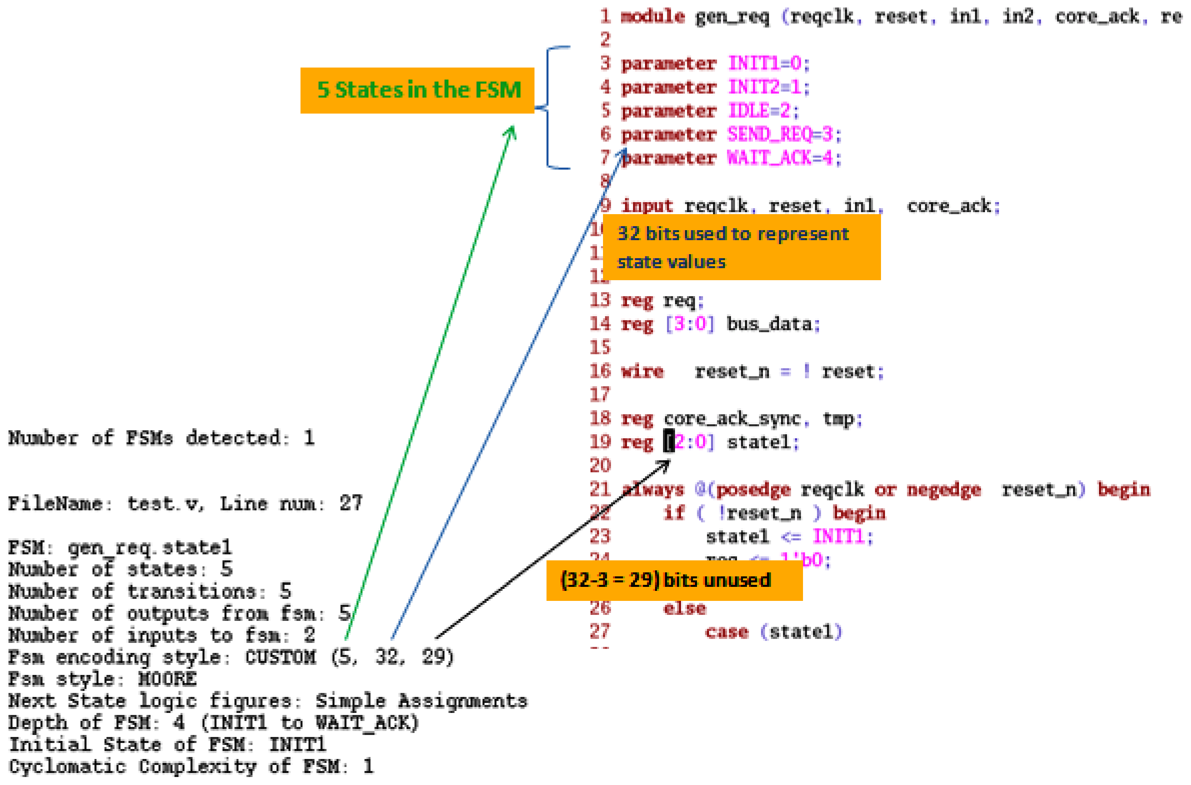
### Custom-Style Encoding

If the *Fsm encoding style* field in the *Auto Verify-FSM Report* is *CUSTOM*, the following three types values appear adjacent to the text *CUSTOM* in this report:

- Number of states in FSM
- Number of bits representing state values

- Number of unused bits

The following figure shows the example of custom encoding style:



```
module gen_req (reqclk, reset, in1, in2, core_ack, re

parameter INIT1=0;
parameter INIT2=1;
parameter IDLE=2;
parameter SEND_REQ=3;
parameter WAIT_ACK=4;

input reqclk, reset, in1,  core_ack;

reg req;
reg [3:0] bus_data;

wire   reset_n = ! reset;

reg core_ack_sync, tmp;
reg [2:0] state1;

always @(posedge reqclk or negedge  reset_n) begin
    if ( !reset_n ) begin
        state1 <= INIT1;
        req <= 1'b0;
    else
        case (state1)
```

```
Number of FSMs detected: 1


FileName: test.v, Line num: 27

FSM: gen_req.state1
Number of states: 5
Number of transitions: 5
Number of outputs from fsm: 5
Number of inputs to fsm: 2
Fsm encoding style: CUSTOM (5, 32, 29)
Fsm style: MOORE
Next State logic figures: Simple Assignments
Depth of FSM: 4 (INIT1 to WAIT_ACK)
Initial State of FSM: INIT1
Cyclomatic Complexity of FSM: 1
```

**FIGURE 6.** Example of custom encoding style

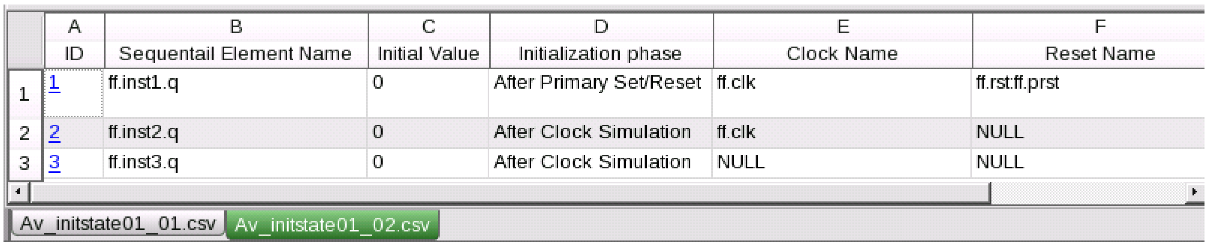
## Uninitialized_Sequential_Elements Spreadsheet Report

The *Av_initstate01* rule generates the Uninitialized_Sequential_Elements.csv file that contains details about the initialized and uninitialized sequential elements. This file also displays a non-default value for each pin.

This spreadsheet contains two tabs to show details of initialized and uninitialized sequential elements. By default, the tab for uninitialized sequential elements is selected.

To open these spreadsheets, double-click on the violation message of the Av_initstate01 rule.

The following figure shows the Uninitialized_Sequential_Elements.csv for initialized sequential elements:

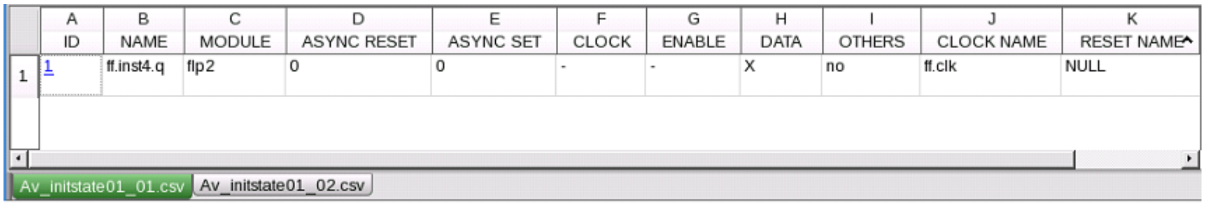


| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| | ID | Sequentail Element Name | Initial Value | Initialization phase | Clock Name | Reset Name |
| 1 | 1 | ff.inst1.q | 0 | After Primary Set/Reset | ff.clk | ff.rst:ff.prst |
| 2 | 2 | ff.inst2.q | 0 | After Clock Simulation | ff.clk | NULL |
| 3 | 3 | ff.inst3.q | 0 | After Clock Simulation | NULL | NULL |

Av_initstate01_01.csv Av_initstate01_02.csv

**FIGURE 7.** Initialized_Sequential_Elements Spreadsheet Report

The following figure shows the Uninitialized_Sequential_Elements.csv for uninitialized sequential elements:



| | A | B | C | D | E | F | G | H | I | J | K |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | ID | NAME | MODULE | ASYNC RESET | ASYNC SET | CLOCK | ENABLE | DATA | OTHERS | CLOCK NAME | RESET NAME |
| 1 | 1 | ff.inst4.q | flp2 | 0 | 0 | - | - | X | no | ff.clk | NULL |

Av_initstate01_01.csv Av_initstate01_02.csv

**FIGURE 8.** Uninitialized_Sequential_Elements Spreadsheet Report

If the reset or set value of a pin is 0 or 1, the corresponding schematic shows the path of that value. For example, if the reset and set value is 0, the schematic will display the path.

To open the schematic, click `1` in the *ID* column in the spreadsheet and then click the button. The following figure displays the schematic highlighting the path:

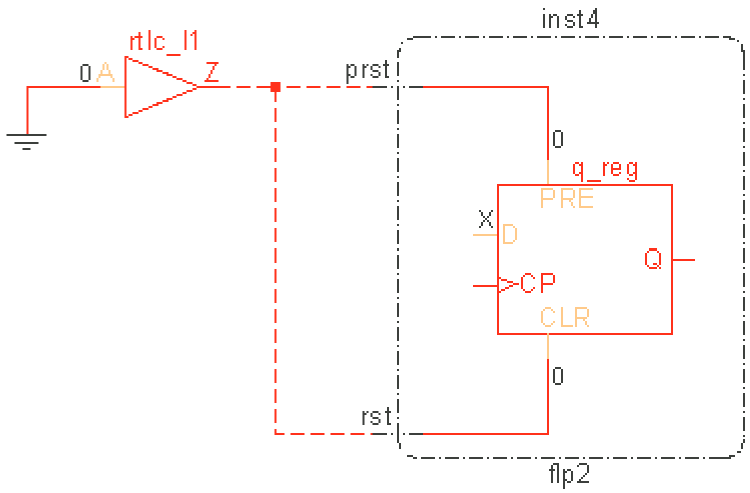


**FIGURE 9.** Av_initstate01 Example

## Details of the Uninitialized_Sequential_Elements Spreadsheet Report

Details of various columns of the Uninitialized_Sequential_Elements Spreadsheet Report are described in the following table:

| Column Name | Description |
|---|---|
| NAME | Specifies a sequential element name.<br>If a sequential cell is a flip-flop, latch, or clock-gating cell, this column displays bus-merged output net name. Else, library instance pins are displayed.<br>This column supports cross-probing to incremental schematic, which highlights a sequential instance along with terminals/pins that have a non-default value in the remaining columns. |
| MODULE | Specifies the name of a leaf-level parent module of a sequential element. |

| Column Name | Description |
|---|---|
| ASYNC RESET | Specifies an asynchronous reset value.<br>This value can be any of the following:<br>• `1`: This value is displayed when the last value on a reset pin of a sequential instance with active low reset is 1 during initial state detection.<br>• `0`: This value is displayed when the last value on a reset pin of a sequential instance with an active high reset is 0 during initial state detection.<br>• `X`: This value is displayed when the last value on a reset pin of a sequential instance is X during initial state detection.<br>• `-`: This is a default value, and it is displayed when a reset pin does not exist or does not have an unexpected value during initial state detection. |
| ASYNC SET | This is similar to the ASYNC RESET column with the difference that in this case, the *Av_initstate01* rule checks asynchronous set pin instead of asynchronous reset pin. |
| CLOCK | This column displays any of the following values based on different conditions:<br>• `-`: This value is displayed if there has been toggling from 0 to 1 or from 1 to 0 during initial state detection.<br>• `1`: This value is displayed when a clock pin is stuck at 1.<br>• `0`: This value is displayed when a clock pin is stuck at 0.<br>• `X`: This value is displayed when a clock pin is stuck at X.<br>• Comma separated list of *<clkPinName>*:*<clkTermValue>* is displayed when a sequential element contains multiple clock pins. |
| ENABLE | This column displays any of the following values based on different conditions:<br>• `1`: This value is displayed when the last value on an enable pin of a sequential instance with active low enable is 1 during initial state detection.<br>• `0`: This value is displayed when the last value on an enable pin of a sequential instance with active high enable is 0 during initial state detection.<br>• `X`: This value is displayed when the last value on an enable pin of a sequential instance is X during initial state detection.<br>• `-`: (Default value): This value is displayed whenever an enable pin does not exist or does not have an unexpected value during initial state detection.<br><br>The *Av_initstate01* rule checks for enable pins of a flip-flop and scan enable pins of sequential library cells. |

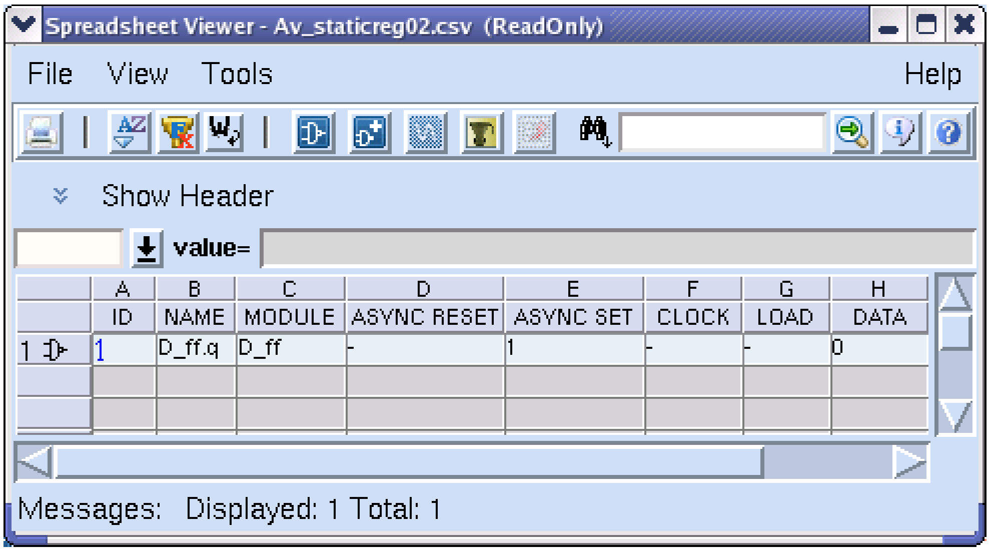
| Column Name | Description |
|---|---|
| DATA | This column displays any of the following values based on different conditions:<br>• X: This value is displayed when the last value on a data pin of a sequential instance is X during initial state detection.<br>• -: (Default value) This value is displayed whenever a data pin does not have an unexpected value during initial state detection.<br>The *Av_initstate01* rule checks for data and scan data pins. |
| OTHERS | This column displays any of the following values based on different conditions:<br>• YES: This value is displayed when a sequential cell is uninitialized and the *Av_initstate01* rule could not find a possible reason in an asynchronous reset, set, clock, load or data field.<br>• NO: This value is displayed when any one of the field is a non-default. |
| CLOCK NAME | Name of the clock that is driving the clock pin of the corresponding sequential element |
| RESET NAME | Name of the reset that is driving the clock pin of the corresponding sequential element |

**NOTE:** *The value* U *instead of* X *is used for hanging pins in the spreadsheet. The* X *value is a default value for simulation through a design is initialized for simulation in the Av_initstate01 rule.*

## Av_staticreg02 Spreadsheet Report

The *Av_staticreg02* rule generates the Av_staticreg02.csv file that contains details about input pins of static sequential elements after applying case-analysis and VDD/VSS propagation in a design.

The following figure displays a sample Av_staticreg02.csv file:


Spreadsheet Viewer - Av_staticreg02.csv (ReadOnly)
File View Tools Help
Show Header
value=
Messages: Displayed: 1 Total: 1

| | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| | ID | NAME | MODULE | ASYNC RESET | ASYNC SET | CLOCK | LOAD | DATA |
| 1 | 1 | D_ff.q | D_ff | - | 1 | - | - | 0 |

**FIGURE 10.** Av_staticreg02 Spreadsheet

Details of various columns in the above spreadsheet are as given in the following table:

| Column Name | Description |
|---|---|
| NAME | Specifies a sequential element name.<br>If a sequential cell is a flip-flop, latch, or clock-gating cell, this column displays bus-merged output net name. Else, library instance name is displayed.<br>This column supports cross-probing to incremental schematic, which:<br>• Highlights a sequential instance along with values (0 or 1) on terminal/pin in the schematic for user debug ability.<br>• Displays sequential elements along with the path from a sequential pin (displayed in spreadsheet) to a fan-in source of constant value in a design. |
| MODULE | Specifies the name of a leaf-level parent module of a sequential element. |

| **Column Name** | **Description** |
|---|---|
| ASYNC RESET | Specifies an asynchronous reset value.<br>This value can be any of the following:<br>• `1`: This value is displayed when a sequential instance with an active high reset receives an active high value, `1`.<br>• `0`: This value is displayed when a sequential instance with an active low reset receives an active low value, `0`.<br>• `-`: This is a default value, and it is displayed whenever a reset is a non-constant, unconnected, unconstrained, etc. |
| ASYNC SET | This is similar to the ASYNC RESET column with the difference that in this case, the *Av_staticreg02* rule checks asynchronous set pin instead of asynchronous reset pin. |
| CLOCK | This column displays any of the following values based on different conditions:<br>• `1`: This value is displayed when a clock pin receives 1 during analysis on a sequential element containing a single clock pin.<br>• `0`: This value is displayed when a clock pin receives 0 during analysis on a sequential element containing a single clock pin.<br>• Comma separated list of *<clkPinName>*:*<clkTermValue>* is displayed when a sequential element contains multiple clock pins.<br>• `-`: This value is displayed for all the other cases. |
| LOAD | This column displays any of the following values based on different conditions:<br>• `1`: This value is displayed when a sequential instance with an active low load receives value, `1`.<br>• `0`: This value is displayed when a sequential instance with an active high load receives value, `0`.<br>• `-`: (Default value): This value is displayed whenever a load is unconnected, unconstrained, etc.<br>The *Av_staticreg02* rule checks for enable pins of a flip-flop, scan enable pins of sequential library cells. |
| DATA | This column displays any of the following values based on different conditions:<br>• `0`: This value is displayed when a data pin receives `0` during analysis of a design.<br>• `1`: This value is displayed when a data pin receives `1` during analysis of a design.<br>• `-`: This value is displayed for the remaining cases.<br>The *Av_staticreg02* rule checks for data and scan data pins. |

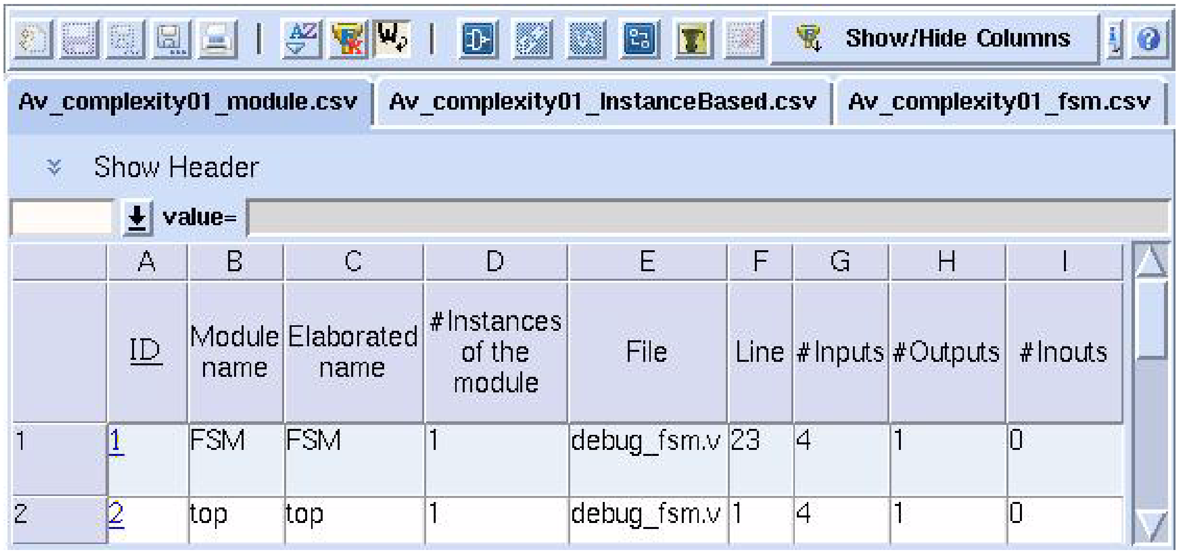
## The Av_complexity01 Spreadsheet Report

The *Av_complexity01* rule generates a spreadsheet to report information related to:

- Modules (*Av_complexity01_module.csv Tab*).
- Module instances (*Av_complexity01_InstanceBased.csv Tab*).
- FSMs (*Av_complexity01_fsm.csv Tab*).

## Av_complexity01_module.csv Tab

Under this tab, information of all modules is displayed. The following figure shows the sample data under this tab:



| | A | B | C | D | E | F | G | H | I |
|---|---|---|---|---|---|---|---|---|---|
| | ID | Module name | Elaborated name | #Instances of the module | File | Line | #Inputs | #Outputs | #Inouts |
| 1 | 1 | FSM | FSM | 1 | debug_fsm.v | 23 | 4 | 1 | 0 |
| 2 | 2 | top | top | 1 | debug_fsm.v | 1 | 4 | 1 | 0 |

**FIGURE 11.** Information under the Av_complexity01_module.csv tab

To cross-probe to the first line in the RTL defining a particular module, click the corresponding cell in the *ID* column of the above spreadsheet.

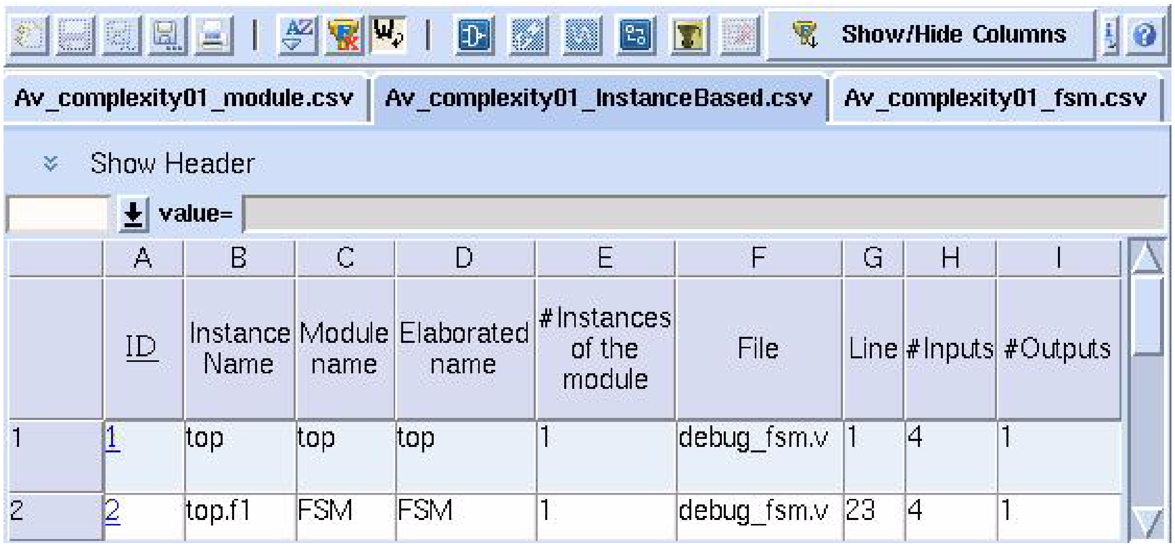
The following table describes each column of the above spreadsheet:

| Column | Description |
|---|---|
| ID | Specifies a unique tag number for a module. Click this cell to cross-probe to the first line of module definition in the RTL. |
| Module name | Specifies RTL design unit name:<br>• For Verilog: module name<br>• For VHDL: Entity. Architecture name |
| Elaborated name | Specifies an elaborated module name.<br>It is relevant for parameterized module. |
| #Instances of the module | Specifies the number of times a given module is instantiated anywhere in a design. |
| File | Specifies the name of a file in which a module is defined. |
| Line | Specifies the starting line number of a module. |
| #Inputs | Specifies the number of input ports of a module. |
| #Outputs | Specifies the number of output ports of a module. |
| #Inouts | Specifies the number of inout ports of a module. |
| #Param | Specifies the number of parameters in a parameterized module. |
| #Lines of code | Specifies the number of code lines inside a module. |
| #Lines of comments | Specifies the number of comment lines inside a module. |
| User defined instances inside the module | Specifies the number of other user-defined modules that are instantiated/present inside a module. |
| #BB Instances | Specifies the number of pure black box instances present in a module. |
| #FSMs | Specifies the number of FSMs detected in a module. |
| #Condition var | Specifies the number of variables involved in conditional statements. |
| #User nets | Specifies the number of signals (excluding ports) defined by a user. |
| #Always/Process | Specifies the number of always and process blocks in a module. |

| Column | Description |
|---|---|
| Cyclomatic complexity | Specifies the number of decision points inside a module plus one. |
| Max of if-else/case depth | Specifies the maximum depth of nested if-else and case conditions. |

## Av_complexity01_InstanceBased.csv Tab

Under this tab, details of module instances are displayed. The following figure shows the sample data under this tab:



| | A | B | C | D | E | F | G | H | I |
|---|---|---|---|---|---|---|---|---|---|
| | ID | Instance Name | Module name | Elaborated name | #Instances of the module | File | Line | #Inputs | #Outputs |
| 1 | 1 | top | top | top | 1 | debug_fsm.v | 1 | 4 | 1 |
| 2 | 2 | top.f1 | FSM | FSM | 1 | debug_fsm.v | 23 | 4 | 1 |

**FIGURE 12.** Information under the Av_complexity01_InstanceBased.csv tab

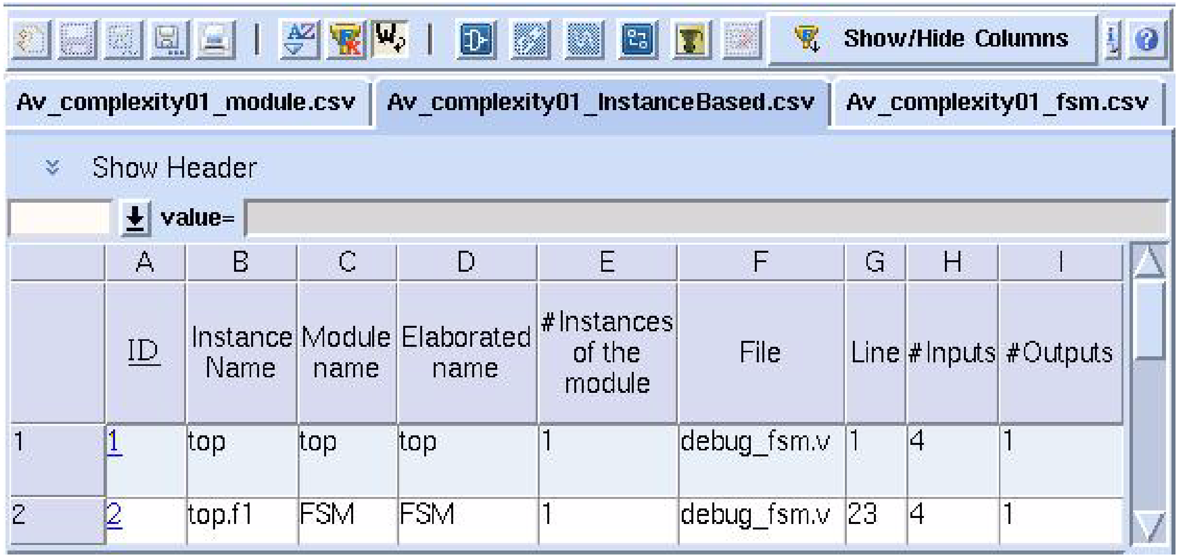
The following table describes each column of the above spreadsheet:

| Column | Description |
|---|---|
| ID | Specifies a unique tag number for a module. Click this cell to cross-probe to the first line of module definition in the RTL. |
| Instance name | Specifies the instance name |
| Module Name | Specifies the module of the instance |
| Elaborated name | Specifies an elaborated module name. It is relevant for parameterized module. |
| #Instances of the module | Specifies the number of times a given module is instantiated anywhere in a design. |
| File | Specifies the name of a file in which a module is defined. |
| Line | Specifies the starting line number of a module. |
| #Inputs | Specifies the number of input ports of a module. |
| #Outputs | Specifies the number of output ports of a module. |
| #Inouts | Specifies the number of inout ports of a module. |
| #Param | Specifies the number of parameters in a parameterized module. |
| #Lines of code | Specifies the number of code lines inside a module. |
| #Lines of comments | Specifies the number of comment lines inside a module. |
| User defined instances inside the module | Specifies the number of other user-defined modules that are instantiated/present inside a module. |
| #BB Instances | Specifies the number of pure black box instances present in a module. |
| #FSMs | Specifies the number of FSMs detected in a module. |
| #Condition var | Specifies the number of variables involved in conditional statements. |
| #User nets | Specifies the number of signals (excluding ports) defined by a user. |
| #Always/Process | Specifies the number of always and process blocks in a module. |
| Cyclomatic complexity | Specifies the number of decision points inside a module plus one. |
| Max of if-else/case depth | Specifies the maximum depth of nested if-else and case conditions. |

| Column | Description |
|---|---|
| Cumulative Complexity | Specifies the cumulative complexity of the instance |
| Instance Level | Specifies the hierarchical level of the instance |

## Av_complexity01_fsm.csv Tab

Under this tab, details of FSMs in a design are displayed. The following figure shows the sample data under this tab:



| | A | B | C | D | E | F | G | H | I |
|---|---|---|---|---|---|---|---|---|---|
| | ID | Instance Name | Module name | Elaborated name | #Instances of the module | File | Line | #Inputs | #Outputs |
| 1 | 1 | top | top | top | 1 | debug_fsm.v | 1 | 4 | 1 |
| 2 | 2 | top.f1 | FSM | FSM | 1 | debug_fsm.v | 23 | 4 | 1 |

**FIGURE 13.** Information under the Av_complexity01_fsm.csv tab

To view the FSM details in the *FSM Viewer* window, click a cell in the *ID* column of the above spreadsheet and then click the *FSM* button ( ) in the main Atrenta Console window.

The following table describes each column of the above spreadsheet:

| **Column** | **Description** |
|---|---|
| FSM ID | Specifies a unique tag number for FSM. When you click this cell and then click the FSM button, the FSM Viewer window appears. |
| FSM | Specifies the de-compiled current state node. |
| File | Specifies the name of the file containing FSM. |
| Line | Specifies the line number of an if/case block containing the current state expression. |
| #States | Specifies the number of states for the FSM. |
| #Transitions | Specifies the number of transitions in the FSM. |
| #Input | Specifies the number of inputs to the FSM. |
| #Output | Specifies the number of outputs from the FSM. |
| Encoding | Specifies the encoding style, such as:<br>• One-Cold<br>• One-Hot<br>• Grey<br>• Minimum<br>• Custom (See *Custom-Style Encoding*) |
| Style | Specifies the type of machine (mealy, moore, or unknown). |
| Next state | Specifies the type of next state assignment (Simple or Non-static. |
| Initial state | Specifies initial state of FSM. |
| Depth | Specifies FSM depth (longest state path in the FSM) |
| Cyclomatic complexity | Specifies the number of decision points in FSM plus one. |

## Functional Analysis Report

This file contains various sections, which can help to analyze the cause of a bug or to gather functional analysis statistics.

This report is saved in the auto_verify.rpt file in the `<curr_working_directory>/spyglass_reports/` directory.

The sections in this report are as follows:

## Report Header

The report header provides a brief overview of the report and a brief summary of all sections in this report.

Following is the content of the header section of this report:

```
############################################################
Purpose:
# This report contains the functional analysis statistics
  of a design.
#
# Format:
# It contains the following sections:
# Section A: Run Parameters
#   Lists the parameters specified in the current run
# Section B: Clock Information
#   Lists the clock information of the design
# Section C: Reset Information
#   Lists the reset information of the design
# Section D: Set-Case Analysis Settings
#   Lists the set case analysis settings used in the design
# Section E: Initial State of the Design
#   Lists the initial-state statistics of the design along
#   with the reset percentage. The initial state of each
#   register can be seen in auto_verify.reg file.
# Section F: Results Summary (Current)
#   Lists the statistics of the assertions formed for each
#   rule
# Section G: Results Summary (Cumulative)
#   Lists the summary of cumulative set of assertions formed
#   in the current run and the information of earlier runs in
#   the property file. This section is printed when you
#   specify a property file using the propfile parameter.
# Section H: Assertion Details
#   Lists the assertion details.

###########################################################
```

## Section A: Run Parameters

This section lists parameters specified in the current run.

## Section B: Clock Information

This section reports a summary of clock definitions as reported in the *Av_clkinf01* rule. The *design virtual cycle* is also reported in this section.

Each clock is reported in the following format:

```
<clk-name>: <clk-period>; <clk-source>; <clk-edge>; <edge-
list>; <num-flops-posedge>; <num-flops-negedge>;
```

Where *<clk-name>* is the clock name, *<clk-period>* is the clock period (specified using the `-period` argument of the `clock` constraint), *<clk-source>* is `SGDC` for clocks specified using the clock constraint in a SpyGlass Constraints file or `Auto-Inferred` for automatically-inferred clocks, *<clk-edge>* is the starting clock edge (`Rising` or `Falling`), *<edge-list>* is the clock edge list (specified using the `-edge` argument of the `clock` constraint), *<num-flops-posedge>* is the number of flip-flops triggered by the clock on the positive edge, and *<num-flops-negedge>* is the number of flip-flops triggered by the clock on the negative edge.

User-specified clocks and auto-inferred clocks are reported under separate headings.

A separate file, named auto_verify.reg, reports controlling clocks for individual registers. See *Register Info Report* for details of the auto_verify.reg file.

This section also contains the Design Virtual Cycle in term of number of fastest clock cycle and in term of non-overlapping edges.

## Section C: Reset Information

This section reports the resets that were used in initial state detection and for functional analysis. User specified resets and auto-inferred resets are reported under separate headings.

Each reset is reported in the following format:

```
<reset-name> ; Active High | Active Low  : [soft reset]
```

All the resets are assumed to be hard resets unless marked as soft resets. All hard resets are deactivated during functional analysis. The soft resets are used only in initial state search and are not deactivated during functional analysis.

A separate file, named auto_verify.reg, reports controlling resets for individual registers. See *Register Info Report* for details of the auto_verify.reg file.

## Section D: Set-Case Analysis Settings

This section reports a summary of *set_case_analysis* constraints that have been applied on the net through the SpyGlass Constraints file.

Each `set_case_analysis` constraint is reported in the following format:

```
<net-name> ; <net-value>
```

Where <*net-name*> is the net's hierarchical name and <*net-value*> is the specified value for the net.

## Section E: Initial State of the Design

This section reports a summary of initial state as reported in the *Av_initstate01* rule.

A separate file, named auto_verify.reg, reports initial state assignments for individual registers. See *Register Info Report* for details of the auto_verify.reg file.

## Section F: Results Summary (Current)

This section lists the statistics of assertions formed for each rule.

The current result summary of each rule is reported in the following format:

```
<Rule-name> ; <Passed> ; <Failed> ; <Partially
-Analyzed> ; <Not-Analyzed> ; <Total>
```

The *Not Analyzed* column has the number of properties that were not run based on user inputs.

## Section G: Results Summary (Cumulative)

This section lists the summary of cumulative set of assertions formed in the current run and the information of earlier runs in the property file. This section is printed when you specify a property file by using the *propfile* parameter.

The cumulative result summary for each rule is reported in the following format:

```
<Rule-name> ; <Passed> ; <Failed> ; <Partially
-Analyzed> ; <Not-Analyzed> ; <Total>
```

The *Not Analyzed* column has the number of properties that were not run based on user inputs.

## Section H: Assertion Details

This section lists the assertion details such as summary of failed checks, partially analyzed checks, and proved checks. This section is further divided into the following sub-sections:

- Report Summary of Failed checks

  Under this section, a detailed report is generated for each rule in the following format:

  ```
  RuleName: <number-of-failed-checks> Failed
  <module-name>, <file-name>, <line-number> (<VCD
  -file-name>): FAILED through depth <d1>(<d2>)
  ```

- Report Summary of Partially Analyzed checks

  Under this section, a detailed report is generated for each rule in the following format:

  ```
  RuleName: <number-of-partially-analyzed-checks>
  Partially Analyzed <module-name>, <file-name>,
  ```

```
<line-number> : Partially-Analyzed through depth
<d1>(<d2>)
```

- Report Summary of Proved checks

  Under this section, a detailed report is generated for each rule in the following format:

  ```
  RuleName: <number-of-proved-checks> Proved <module
  -name>, <file-name>, <line-number> : PROVED
  ```

Module and instance names are truncated to the shortest recognizable module/instance name prefixed by `...`. For example, `...foo` instead of `top.lower1.lower2.foo`.

Depth represents the analysis depth. The first number corresponds to the number of cycles of fastest clock and the second represents the number of edges. This is given only for FAILED or Partially-analyzed properties.

**NOTE:** *The depth reported for a Partially-analyzed property indicates that the property was still being analyzed at the reported depth when the analysis of the property was stopped. Hence, it is possible that a property is reported as Partially-analyzed at a certain depth in a SpyGlass Auto Verify solution run and is reported as FAILED at the same depth in another run with a different set of options in SpyGlass Auto Verify solution.*

## Overconstrain Info File

The SpyGlass Auto Verify solution consolidates all user-specified and generated constraints and applies them together.

However, if any conflicting constraints are found during SpyGlass Auto Verify solution rule-checking, an overconstrain info file is generated that contains the details of conflicting constraints.

In addition, the following message string is appended to the violation messages reported by the rules if a set of conflicting constraints are specified for the design:

```
Status : Other(Constraints-Conflict)
```

The *Av_ovl01* rule and all the *Implicit Properties Rules* except for the Av_staticreg02 rule report the above message.

If you do not fix this violation, the rule does not perform any formal checks and may stop further processing. To fix this violation, identify and resolve the conflicting constraints in the Overconstrain Information File. The file is generated with the <rule-name>.<ID>.OverConstrainInfo name. A sample content of a Overconstrain Information File is shown below.

```
Following constraints can not be satisfied simultaneously at
depth 1(1):
    Reset constraint on net 'top.var1[1]'
```

## Messages Reported in the Overconstrain Info File

This file contains the following messages:

- Comb-loop involving net '<net-name>' is unstable
- Reset constraint on net '<net-name>'

  The reset can be an asynchronous reset or a synchronous reset.
- set_case_analysis constraint on net '<net-name>'
- OVL constraint '<name>' FILE: <file-name>, Line: <line-num>
- Some constraints cannot be satisfied simultaneously at depth *<cycledepth>* (*<depth>*)

  This message appears if the time taken during message generation for the Overconstrain Info file exceeds the time-out limit.

## Property Status Reported during Functional Analysis

During functional analysis, the functional analysis rules may report the status of Properties as any of the following based on whether a property file could be generated for those properties:

- *Passed*
- *Failed*
- *Partially Proved*
- *Others (Internal-Error)*
- *Others (Constraints-Conflict)*
- *Not-Analyzed*

## Passed

A property is considered as passed when SpyGlass is able to generate a property file for that property.

For example, for a property to pass, at least one design state should be reachable in which the property is valid. In all such cases, a sequence of input vectors can be generated (known as witness), which will lead to that particular design state. If it is possible to generate a witness for a property, the property or assertion holds true (or in other words it passes).

## Failed

A property is considered as failed when the property file for that property cannot be generated under any circumstances.

### Witness

A witness is the input sequence that eventually makes Assertions true while satisfying the given constraints throughout the path.

For example, for a property to pass, at least one design state should be reachable in which the property is valid. In all such cases, a sequence of input vectors can be generated, which will lead to that particular design state. This sequence is known as “Witness”. Therefore, if it is possible to generate a witness for a property, the property or assertion holds true, that it is passed.

However, if it is not possible to generate a single witness under the given constraints, the property or the assertion fails.

## Partially Proved

A property is considered as partially-proved when SpyGlass cannot find the property file for that property and also cannot guarantee that generating a property file is not possible.

## Others (Internal-Error)

A property is considered as Others (Internal-Error) when the value of the design cycle for the clocks in that property cone exceeds the threshold value of 65535.

### Others (Constraints-Conflict)

A property is reported as Others (Constraints-Conflict) when constraints in the property cone are not satisfiable.

### Not-Analyzed

A property is reported as Not-Analyzed when it is switched off in the property file and not analyzed during functional verification.

## Register Info Report

Besides the general report described earlier, SpyGlass Auto Verify solution provides a complementary file containing information on registers in a design that are relevant for functional analysis. The register info report is dumped in the auto_verify.reg file. This file contains the following sections:

#### Section A: Clocks in the design

Lists all clocks in the design.

An integer ID number is assigned to each clock signal. Section D reports these Clock ID numbers instead of actual clock names.

#### Section B: Resets in the design

Lists all synchronous and asynchronous resets in the design.

An integer ID number is assigned to each reset signal. Section D reports these Reset ID numbers instead of actual reset names.

#### Section C: Initial State (after primary sets/resets/clock based simulation are applied)

Lists initial state of sequential elements after propagation of design constants, set-case-analysis, reset/set, and clock based simulation in a design.

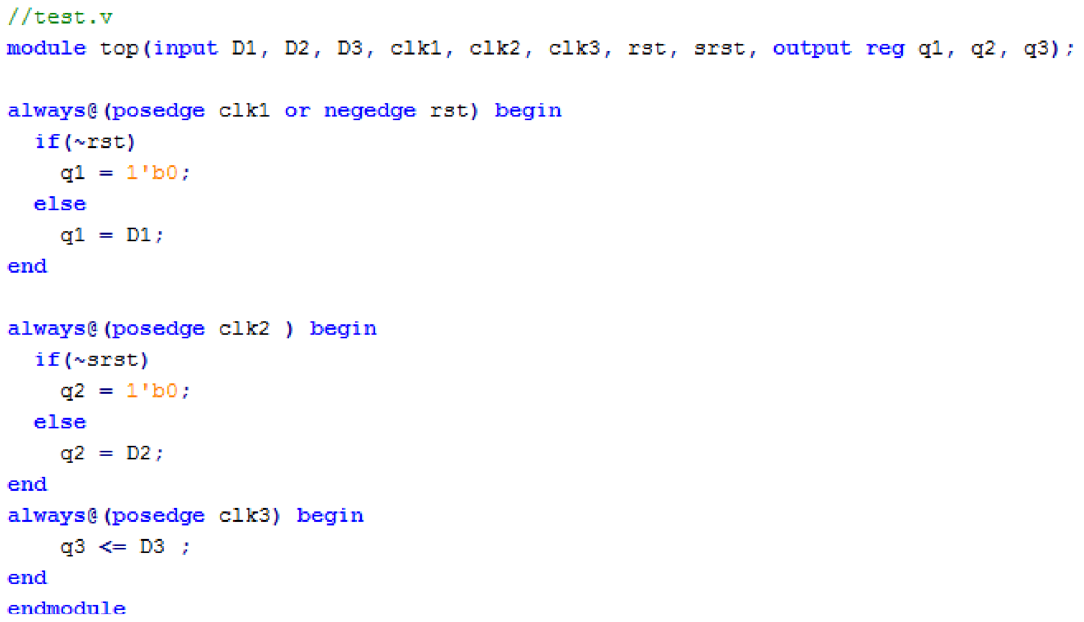
#### Section D: Uninitialized Sequential Elements

Lists uninitialized sequential elements in a design.

This section is equivalent to the *Uninitialized_Sequential_Elements Spreadsheet Report*.

## Sample Register Info Report

Consider the following Verilog file:



```
//test.v
module top(input D1, D2, D3, clk1, clk2, clk3, rst, srst, output reg q1, q2, q3);

always@(posedge clk1 or negedge rst) begin
  if(~rst)
    q1 = 1'b0;
  else
    q1 = D1;
end

always@(posedge clk2 ) begin
  if(~srst)
    q2 = 1'b0;
  else
    q2 = D2;
end
always@(posedge clk3) begin
    q3 <= D3 ;
end
endmodule
```

Now, consider the following SGDC file:

```
//test.sgdc

current_design top
clock -name top.clk1 -period 10
clock -name top.clk2 -period 20
clock -name top.clk3 -period 30
reset -name top.rst -value 0
reset -name top.srst -value 0 -sync
```

The following schematic is generated for the above example:

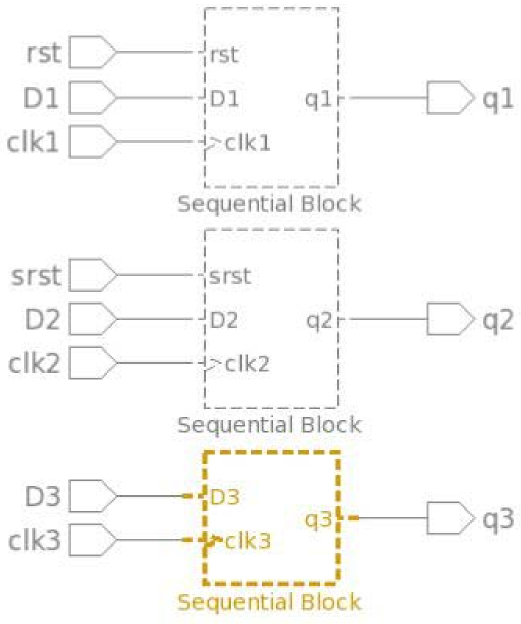
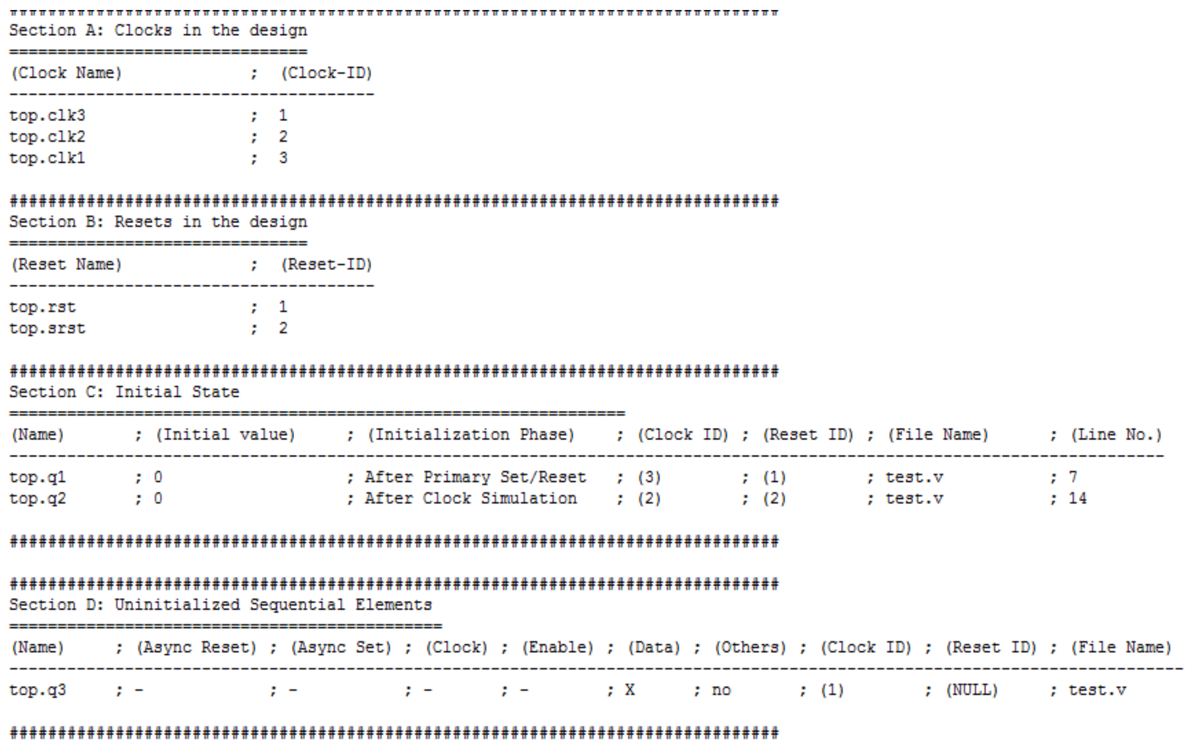


**FIGURE 14.** Design for which the register info report is generated

In addition, the following Register Info report is generated for this example:



```
################################################################################
Section A: Clocks in the design
===============================
(Clock Name)             ;  (Clock-ID)
--------------------------------------
top.clk3                 ;  1
top.clk2                 ;  2
top.clk1                 ;  3

################################################################################
Section B: Resets in the design
===============================
(Reset Name)             ;  (Reset-ID)
--------------------------------------
top.rst                  ;  1
top.srst                 ;  2

################################################################################
Section C: Initial State
================================================================
(Name)       ; (Initial value)     ; (Initialization Phase)     ; (Clock ID) ; (Reset ID) ; (File Name)      ; (Line No.)
------------------------------------------------------------------------------------------------------------------------
top.q1       ; 0                   ; After Primary Set/Reset    ; (3)        ; (1)        ; test.v           ; 7
top.q2       ; 0                   ; After Clock Simulation     ; (2)        ; (2)        ; test.v           ; 14

################################################################################

################################################################################
Section D: Uninitialized Sequential Elements
=============================================
(Name)     ; (Async Reset) ; (Async Set) ; (Clock) ; (Enable) ; (Data) ; (Others) ; (Clock ID) ; (Reset ID) ; (File Name)
--------------------------------------------------------------------------------------------------------------------------
top.q3     ; -             ; -           ; -       ; -        ; X      ; no       ; (1)        ; (NULL)     ; test.v

################################################################################
```

# Rules in SpyGlass Auto Verify

This chapter describes the functional analysis rules in the SpyGlass Auto Verify solution.

The rules of the SpyGlass Auto Verify solution belong to any one of the following categories:

| | | |
|---|---|---|
| *Info Rules* | *Formal Setup Rules* | *Implicit Properties Rules* |
| *Standard Properties Rules* | *Must Rules* | |

# Info Rules

The Info rules report information about the design and its attributes.

While these rules do not report any design problem, they report assumptions under which SpyGlass validates the other rules.

For example, the clock network information provides all clocks in a design along with their frequencies and edges. If the clock definition is incorrect, the other rules may report false violations or some rules may not report certain design problems.

The following table describes the rules under this category:

| Rule | Reports |
|---|---|
| *Av_clkinf01* | Information about clocks in the design |
| *Av_complexity01* | Complexity of a design in terms of characteristics and complexity of RTL modules and FSMs in the design |
| *Av_fsminf01* | FSM statistics for the design |
| *Av_fsminf02* | Interacting FSMs in the design |
| *Av_Info_Case_Analysis* | Case analysis settings |
| *Av_initstate01* | A valid state of the design from which the formal analysis would actually start |
| *Av_report01* | Total number of properties analyzed and number of functional constraints set on the design |
| *Av_rstinf01* | Information about resets in the design |

## Av_clkinf01

**Reports all the clocks in the design.**

## When to Use

Use this rule to check if the clock definitions are as per the design intent.

## Description

The *Av_clkinf01* rule reports the following details of clocks in a design:

- Clock name
- Clock frequency
- Rising and falling edges
- Number of flip-flops working on each edge of the clock.

### Clocks Checked by the Av_clkinf01 Rule

The *Av_clkinf01* rule checks any of the following clocks:

- Clocks defined by using the *clock* constraint.
- Automatically-inferred clocks when the `use_inferred_clocks` parameter of the SpyGlass CDC solution is set to `yes`.

  For such clocks, SpyGlass considers the default time period of 10 ns and 50% duty cycle.

## Parameter(s)

`use_inferred_clocks`: The default value is `no`. Set this parameter to `yes` to use the automatically-generated clock information.

**NOTE:** *This is the parameter of SpyGlass CDC solution.*

## Constraint(s)

*clock* (Optional): Use this constraint to specify clocks signals in a design.

## Messages and Suggested Fix

The following message appears to specify the details of clocks in a design:

```
[INFO] ‘<clk-type>’ Clock ‘<clk-name>’: Period ‘<period>’,
'<num1>' flops on posedge at '<rise-time>', '<num2>' flops on
negedge at '<fall-time>'
```

The details of the arguments of the above violation message are described in the following table:

| **Argument** | **Description** |
|---|---|
| <clk-type> | Specifies the clock type as any of the following:<br>• **User defined** if the clock is specified by using the *clock* constraint.<br>• **Default** if the clock is inferred by SpyGlass after you have set the use_inferred_clocks parameter to yes. |
| <clk-name> | Specifies the clock name. |
| <period> | Specifies the clock period in nano seconds (rounded to nearest 0.5) |
| <num1> | Specifies the number of flip-flops triggered at the posedge of the clock. |
| <num2> | Specifies the number of flip-flops triggered at the posedge of the clock. |
| <rise-time> | Specifies the rise time of the clock. |
| <fall-time> | Specifies the fall time of the clock. |

### *Potential Issues*

This violation appears if your design contains any of the following types of clocks:

- The clocks defined by using the *clock* constraint.
- The clocks inferred after you set the `use_inferred_clocks` parameter to `yes`.

### *Consequences of Not Fixing*

The functionality, and therefore the functional analysis of a design is sensitive to the frequencies and latencies of the clocks in a design.

Therefore, if you do not review these clocks, the functional analysis of the design may get impacted.

### *How to Debug and Fix*

Verify the correctness of the reported clock definitions before investigating the cause of failures of some rules.

Ensure that the clock information is as per the design intent.

## Example Code and/or Schematic

Consider the following files specified for SpyGlass analysis:

```
// Design File

module top (d, q, clk1, clk2);
  input [1:0]d;
  input clk1, clk2;
  output [1:0]q;
  reg [1:0]q;
  always @(posedge clk1)
  q[0] = d[0];
  always @(posedge clk2)
  q[1] = d[1];
endmodule
```

```
// SGDC File

current_design top
clock -name top.clk1 -period 15
  -edge {0 9}
```

```
// Project file command
set_parameter use_inferred_clocks y
```

In the above example, `clk1` is defined by using the *clock* constraint and `clk2` is automatically detected by SpyGlass.

Now when you run the *Av_clkinf01* rule, the following messages appear showing the details of the `clk1` and `clk2` clocks:

```
'Default' Clock 'top.clk2': Period '10.000', '1' flop(s) on
posedge at '5.000', '0' flop(s) on negedge at '10.000'


'User defined' Clock 'top.clk1': Period '15.000', '1' flop(s)
on posedge at '0.000', '0' flop(s) on negedge at '9.000'
```

The following figure shows the schematic of the violation for the `clk2` clock:

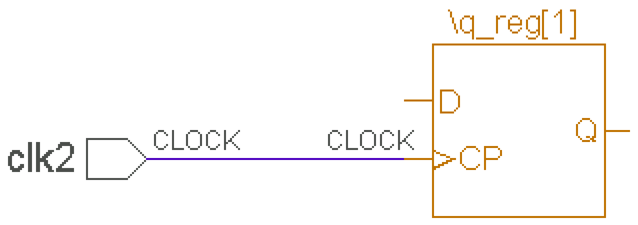


**FIGURE 1.** Schematic of the Av_clkinf01 rule violation

## Default Severity Label

Info

## Reports and Related Files

*Auto Verify Central Report*

## Av_complexity01

**Reports design characteristics and complexity for all the RTL modules and FSMs in the design**

### When to Use

Use this rule to understand the complexity of a design for:

- Modularizing or partitioning an RTL.
- Estimating effort needed for block verification and selecting IPs.

### Description

The *Av_complexity01* rule reports the complexity of a design in terms of characteristics and complexity of RTL modules and FSMs in a design.

This rule shows the complexity information in *The Av_complexity01 Spreadsheet Report* and *The Complexity Browser*.

Understanding the complexity of a design is important for the following reasons:

- For modularizing/partitioning RTL
- For estimating effort required for block verification and IP selection.

### Parameter(s)

*av_dump_instance_complexity*: Default value is `no`. Set this parameter to `yes` to generate instance-based spreadsheet (*Av_complexity01_InstanceBased.csv Tab*).

### Constraint(s)

None

### Messages and Suggested fix

The following message appears to indicate the design complexity:

```
[CMPINFO] [INFO] Design <design-name> has <num-of modules>
modules, <num-of-FSM> FSMs, with <avg-num> average cyclomatic
complexity, and <max-complexity> max cyclomatic complexity for
module <module-name>
```

#### ***Potential Issues***

Not applicable

#### ***Consequences of Not Fixing***

Not applicable

#### ***How to Debug and Fix***

Not applicable

## Example Code and/or Schematic

See *The Av_complexity01 Spreadsheet Report* and *The Complexity Browser*.

## Default Severity Label

Info

## Reports and Related Files

- *The Av_complexity01 Spreadsheet Report*: This is a message-based spreadsheet that shows the complexity of a design.

  A message-based spreadsheet is a spreadsheet that appears when you double-click on a violation message.

**NOTE:** *Certain spreadsheet information, such as lines of codes/comments is calculated at a lexical layer. While running this rule with HDL library files, use the set_option hdllibdu yes command in the project file.*

- *Auto Verify Central Report*

## Av_fsminf01

**Reports all the FSMs in a design.**

## When to Use

Use this rule to view the FSM statistics of a design.

### Prerequisites

Use the *clock* constraint to specify clock signals in a design.

## Description

The *Av_fsminf01* rule reports the following information for each FSM in a design:

- Number of states
- Number of transitions
- Number of inputs and outputs to the FSM
- The encoding style, such as One-Cold, One-Hot, Gray, Minimum, or Custom (See *Custom-Style Encoding*)

**NOTE:** *Two-state FSMs with state labels 01 and 10 are reported as one-hot encoded FSMs.*

- The encoding bit information, such as the number of states, number of bits used, and number of extra bits
- The type of machine, such as mealy, moore, or unknown

  When the SpyGlass Auto Verify solution is unable to detect the FSM output, the machine type is reported as `unknown`.
- The type of next state assignment. For example, simple or non static, such as function calls and arithmetic operations.

**NOTE:** *In case of non-static next state assignments, the number of states and number of transitions reported may not be the same as those in the actual FSM.*

- FSM depth. For details, see *FSM Depth*.
- FSM initial states
- Cyclomatic complexity of FSM

### FSM Depth

The minimum length of a path through which a state can be reached from the initial state of an FSM is known as the state depth from that initial state.

FSM depth refers to the maximum of all the state depths from all initial states.

Consider the FSM shown in the following figure:

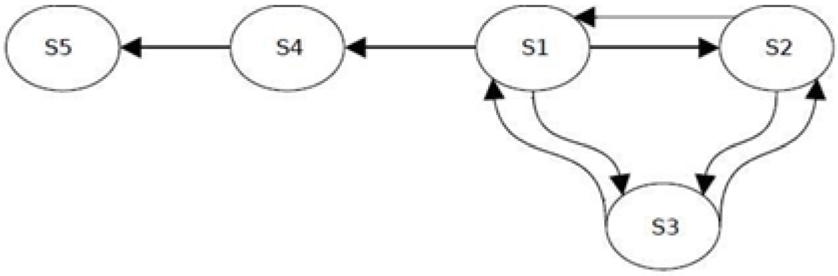


**FIGURE 2.** Example for FSM depth

In the above FSM, `S1` is the initial state. Different state depths in this case are as follows:

- State Depth (`S2`) = 1
- State Depth (`S3`) = 1
- State Depth (`S4`) = 1
- State Depth (`S5`) = 2

In this case, FSM depth is the maximum of all state depths, that is 2.

## Parameter(s)

- *detect_ifelse_fsm*: The default value is `no`. Set this parameter to `yes` to detect the `if-else` style FSMs in addition to the `case` style FSMs.
- *detect_nested_fsm*: The default value is `no`. Set this parameter to `yes` to detect the nested `if-else` style FSMs, the nested `case` style FSMs, and the `assign` style FSMs in addition to detecting the `case` style FSMs.

- *detect_assign_fsm*: The default value is `no`. Set this parameter to `yes` to detect the `assign` style FSMs in addition to detecting the `case` style FSMs.

## Constraint(s)

- *clock* (Mandatory): Use this constraint to specify clock signals in a design.
- *fsm* (Optional): Use this constraint to specify FSM details in a design.

## Messages and Suggested Fix

The following message appears to show the details of an FSM in a design:

```
[INFO] FSM '<fsm-name>' has '<num-fsm-states>' states and
'<num-fsm-transitions>' transitions. Encoding used is
'<encoding-type>'. Refer file: '<file-name>' for details
```

The details of the arguments of the above violation message are described in the following table:

| Argument | Description |
|---|---|
| <fsm-name> | Specifies the FSM name |
| <num-fsm-states> | Specifies the number of FSM states |
| <num-fsm-transitions> | Specifies the number of FSM transitions |
| <file-name> | Specifies the location of the **Auto Verify FSM report** |
| <encoding-type> | Specifies the FSM encoding style, such as s ONE-HOT, ONE-COLD, GRAY, MINIMUM, or CUSTOM (See *Custom-Style Encoding*) |

### *Potential Issues*

Not applicable

### *Consequences of Not Fixing*

Not applicable

### *How to Debug and Fix*

Not applicable

## Example Code and/or Schematic

Consider the following SGDC file and design file specified for SpyGlass analysis:

**// Design File**

```
`define S0 2'b00
`define S1 2'b01
`define S2 2'b10
`define S3 2'b11


module Fsm(clk,ctl, rst, outp);
  input clk, ctl, rst;
  output outp;
  reg outp;
  reg [1:0] state;
  always@(posedge clk or negedge rst)
  begin
  if(!rst)
      state <= `S0;
  else
    case(state) // synopsys full_case parallel_case
    `S0 : state <= `S1;
    `S1 : begin
            state <= `S2;
            outp <= 1'b1;
          end
    `S2 : if (ctl)
           state <= `S3;
    `S3 : if (ctl & !ctl)
              state <= `S1;
      else
          state <= `S3;
   endcase
  end
endmodule
```

```
// SGDC File
current_design Fsm
clock -name Fsm.clk
reset -name Fsm.rst -value 0
```

For the above example, the *Av_fsminf01* rule reports the following violation:

```
FSM 'Fsm.state' has '4' states and '5' transitions. Encoding
used is 'MINIMUMENCODED'.Refer file: 'res_av/spyglass_reports/
auto-verify/Fsm.Info' for details
```

The following figure shows the *FSM Viewer* generated in this case:

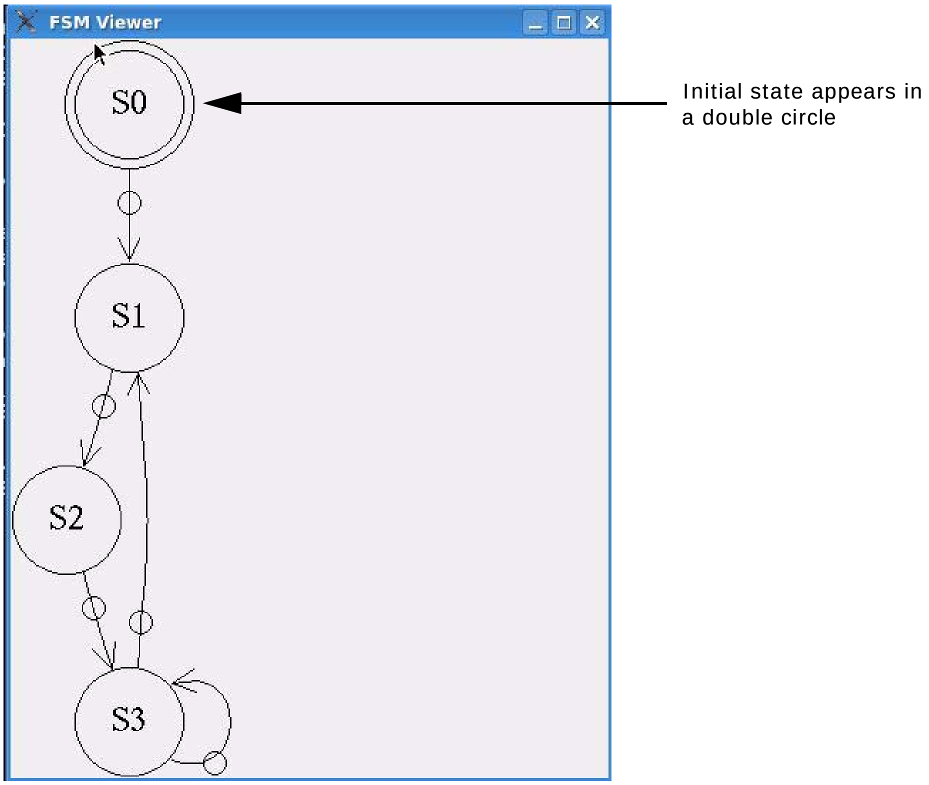


**FIGURE 3.** FSM Viewer

See also *Viewing Conditional Expression of a Transition in the FSM Viewer*.

## Default Severity Label

Info

## Reports and Related Files

- *Auto Verify-FSM Report*
- *Auto Verify Central Report*

## Av_fsminf02

**Reports all the interacting FSMs in a design.**

### When to Use

Use this rule view the dependency between FSMs in a design.

#### Prerequisites

Specify clocks in a design by using the *clock* constraint.

### Description

The *Av_fsminf02* rule reports the interacting FSMs in the design.

### Parameter(s)

- *detect_ifelse_fsm*: The default value is `no`. Set this parameter to `yes` to detect the `if-else` style FSMs in addition to the `case` style FSMs.
- *detect_nested_fsm*: The default value is `no`. Set this parameter to `yes` to detect the nested `if-else` style FSMs, the nested `case` style FSMs, and the `assign` style FSMs in addition to detecting the `case` style FSMs.
- *detect_assign_fsm*: The default value is `no`. Set this parameter to `yes` to detect the `assign` style FSMs in addition to detecting the `case` style FSMs.

### Constraint(s)

- *clock* (Mandatory): Use this constraint to specify clock signals in a design.
- *fsm* (Optional): Use this constraint to specify FSM details in a design.

### Messages and Suggested Fix

The following message appears to report interacting FSMs in a design:

```
[INFO] FSM '<fsm1-name>' output '<out-net-name>' interacts with
FSM '<fsm2-name>' through input '<in-net-name>'
```

The details of the arguments of the above message are described in the following table:

| Argument | Description |
|---|---|
| <fsm1-name> | Preceding FSM state variable name |
| <out-net-name> | Name of the output net of the preceding FSM |
| <fsm2-name> | Succeeding FSM state variable name |
| <in-net-name> | Name of the input net of the succeeding FSM |

#### *Potential Issues*

This violation appears if your design contains interacting FSMs.

#### *Consequences of Not Fixing*

Not applicable

#### *How to Debug and Fix*

Not applicable

### Example Code and/or Schematic

Consider the following file specified for SpyGlass analysis:

```
`define S1_0 2'b00
`define S1_1 2'b01
`define S1_2 2'b10
`define S1_3 2'b11
`define S2_0 2'b11
`define S2_1 2'b10
`define S2_2 2'b01

module Fsm(clk,ctl, rst, outp, outp2);
  input clk, ctl, rst;
  output outp, outp2;
  reg outp, outp2;
  reg [1:0] state;
  reg [1:0] state2;
  always@(posedge clk or negedge rst)
  begin
  if(!rst)
      state <= `S1_0;
  else
    case(state) // synopsys full_case parallel_case
    `S1_0 : state <= `S1_1;
    `S1_1 : begin
            state <= `S1_2;
            outp <= 1'b1;
          end
    `S1_2 : if (ctl)
           state <= `S1_3;
    `S1_3 : if (ctl & !ctl)
             state <= `S1_1;
      else
          state <= `S1_3;
   endcase
  end
  always@(posedge clk or negedge rst)
  begin
  if(!rst)
      state2 <= `S2_0;
  else
    case(state2)
    `S2_0 : state2 <= `S2_2;
    `S2_1 : state2 <= `S2_0;
    `S2_2 : if (outp)
            begin
                state2 <= `S2_1;
                outp2 <= 1'b1;
            end
   endcase
  end
endmodule
```

In the above example contains two FSMs, `state` and `outp`, which are interacting with each other such that the output of one FSM is used as an input by the other FSM. The following figure shows the FSM viewer that displays the interacting FSMs:

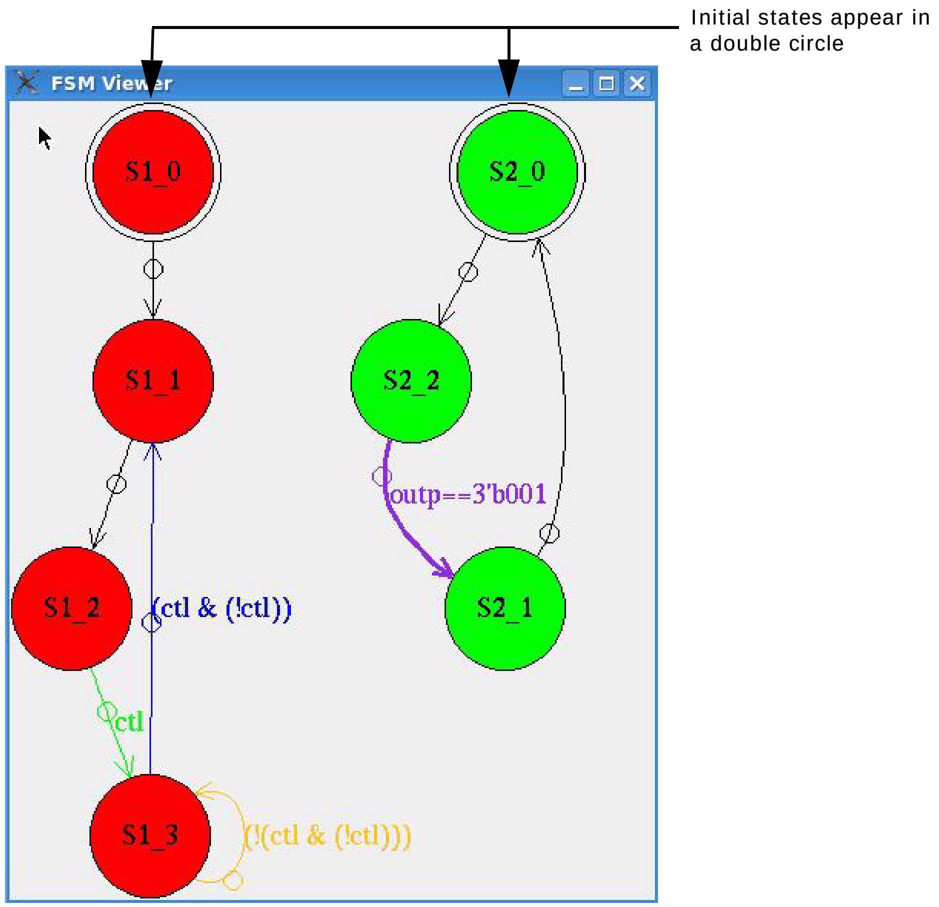


**FIGURE 4.** FSM Viewer showing interacting FSM

For the above example, *Av_fsminf02* rule reports the following violation:

```
FSM 'Fsm.state' output 'Fsm.outp' interacts with FSM
'Fsm.state2' through input 'Fsm.outp'
```

### Viewing Conditional Expression of a Transition in the FSM Viewer

To view the conditional expression of a transition shown in the *FSM Viewer* window, right-click on that transition and select the *label on* option from the shortcut menu. On performing this action:

- The conditional expression appears adjacent to the selected transition.
- The conditional expression and the selected transition appears in bold with a different color.

  The last 14 selected transitions appear in different colors. Only the currently selected transition appears in bold. This is shown in *Figure 4*.
- All the dead transitions reported by the *Av_fsm02* rule appear in the red color.

  When you select the label of such transition by using the *label on* shortcut menu, this transition appears in bold but the color remains red.

## Default Severity Label

Info

## Report and Related Files

*Auto Verify Central Report*

## Av_Info_Case_Analysis

**Highlights case-analysis settings**

### When to Use

Use this rule to view constant values, such as values set by using the *set_case_analysis* SGDC constraint on a terminal/net, supply values, or ground values propagating through a design.

#### Prerequisites

Specify case analysis signals by using the *set_case_analysis* constraint. Based on this information, SpyGlass simulates a design and annotates the simulation value (0 or 1) for each accessible net.

### Description

The *Av_Info_Case_Analysis* rule generates information to highlight case analysis values and power/ground information in a schematic. This power/ground information is inferred from the design.

Information generated by this rule is useful for observing value propagation in a design.

It is recommended to run this rule with other rules as this rule provides valuable debug aid to see how case values are propagating through the design.

#### Performing Value Propagation

If you specify the `set_option enable_const_prop_thru_seq yes` command in the project file, the *set_case_analysis* values propagate beyond sequential elements. Constant propagation from flip-flop-D happens only if one of the following conditions is true:

- Flip-flop does not have preset/clear pin.
- Data is tied to 0, and flip-flop has only clear pin.
- Data is tied to 1, and flip-flop has only preset pin.

While performing value-propagation, SpyGlass generates the following message for each top-level design unit (<*`top-du-name`*>) where case analysis information has been processed:

```
Information for set_case_analysis value propagation for "<top-
du-name>" is displayed
```

### Viewing Case Analysis Settings Along With Rule Violations

While debugging a violation of a rule in the *Incremental Schematic* window, you can view case analysis settings along with the violation of other rules.

To view case analysis settings, perform any of the following actions:

- Select the rule violation.
- Double-click on the rule violation message of the *Av_Info_Case_Analysis* rule while pressing the *<Ctrl>* key.
- Select the rule violation and open the *Incremental Schematic* window.
- Click the Edit -> *Show Case Analysis* menu option in the *Incremental Schematic* window

  For more details, refer to *Atrenta Console User Guide*.

## Parameter(s)

None

## Constraint(s)

None

## Messages and Suggested Fix

This rule reports the following message:

```
[INFO] Information for set_case_analysis value propagation for
design <top-Name> is displayed
```

#### *Potential Issues*

None

#### *Consequences of Not Fixing*

None

#### *How to Debug and Fix*

This rule provides debugging aid to analyze case analysis settings in a design.

View the *Incremental Schematic* of the violation message to see constant value propagation through the design.

## Example Code and/or Schematic

Consider the following schematic of a violation of the *Av_staticnet01* rule:

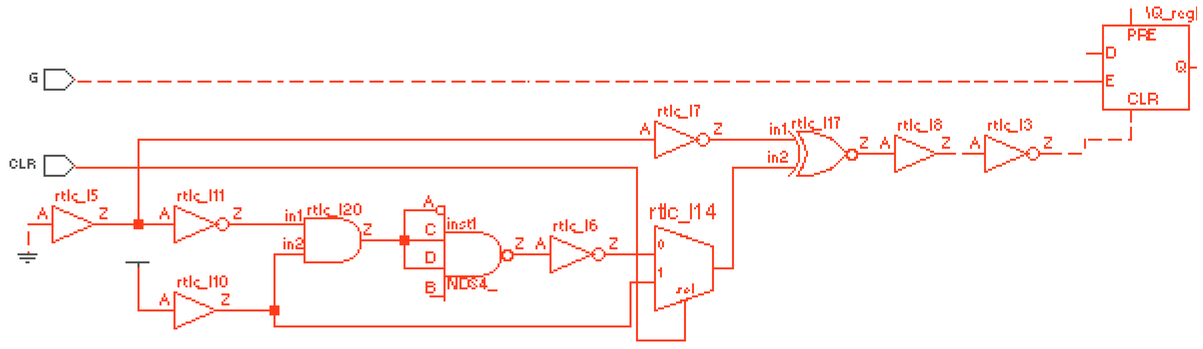


**FIGURE 5.** Schematic of the Av_staticnet01 rule

Now, select the *Edit > Show Info Case Analysis Data* menu option in the schematic window.

The schematic now changes to the following:

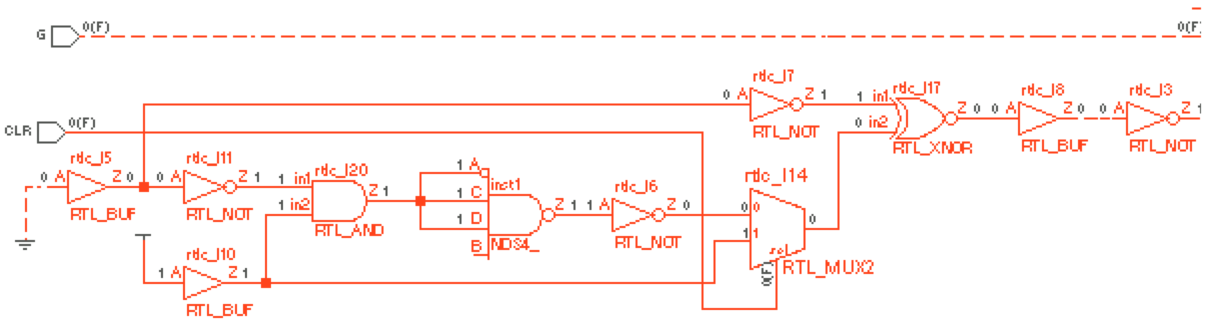


**FIGURE 6.** Constant values in schematic - the Av_Info_Case_Analysis rule

In the above figure, constant values appear in the schematic. This enables you to check the path of constant value propagation.

**Schematic Details**

The *Av_Info_Case_Analysis* rule highlights power ground simulation values and *set_case_analysis* constraints propagated through combinational logic.

## Default Severity Label

Info

## Rule Group

`INFORMATION`

## Reports and Related Files

*Auto Verify Central Report*

## Av_initstate01

**Reports the initial state of a design**

## When to Use

Use this rule to check the initial state of a design from which formal analysis starts.

**NOTE:** *The initial state may not be the reset state of the design.*

### Prerequisites

Specify clock signals in a design by using the *clock* constraint.

## Description

The *Av_initstate01* rule reports the initial state of a design from which formal analysis should start.

### Identifying an Initial State of a Design

The *Av_initstate01* rule identifies the initial state of a design in the following ways (in the given order of priority):

- User-defined initial state where the register value assignment is provided using the *define_tag* constraint.
- State value generated by external simulation engine as a VCD/TCl/FSDB file (use the *simulation_data* constraint to provide the file name)
- Initial state detected by applying a user-defined simulation vector using the *define_tag* constraint in a SpyGlass Design Constraints file.
- Initial state determined by SpyGlass Advanced-Lint solution.

  This search uses the user-specified reset ports (using the *reset* constraint) or auto-detected reset ports and/or may apply proprietary techniques to identify a reachable state of a design.

**NOTE:** *If no reset is present in a design, this rule reports a clock as X for uninitialized sequential elements.*

## Parameter(s)

*ieffort*: The default value is `yes`. Set this parameter to `no` to check all the bits of a bus signal.

## Constraint(s)

- *clock* (Mandatory): Use this constraints to specify clocks in a design.
- *define_tag* (Mandatory): Use this constraint to define a named condition for the application of certain stimulus at the top port or an internal node.
- *reset* (Optional): Use this constraint to specify resets in a design.
- *simulation_data* (Optional): Use this constraint to specify the initial state sequence for a design.

## Messages and Suggested Fix

The following information message appears when the Av_initstate01 rule is run:

```
[ISINFO] [INFO] <num1> percent of sequential outputs are
initialized with sets/resets and <num2> percent sequential
outputs are initialized by data path. Refer file: '<file-name>'
for details
```

The details of the arguments of the above violation message are described in the following table:

| Argument | Description |
|---|---|
| <num1> | Specifies the percentage of sequential outputs initialized with set/reset |
| <num2> | Specifies the percentage of sequential outputs initialized with data path |
| <file-name> | Specifies the path of the auto_verify.reg file. |

### *Potential Issues*

Not applicable.

### *Consequences of Not Fixing*

If the violation message reports zero percentage of initialized sequential elements, the formal analysis can use any random value for analysis. This can result in wrong analysis.

#### *How to Debug and Fix*

When the design contains less than 100% of initialized sequential elements, either specify a VCD file containing initialization data by using the *simulation_data* constraint or specify the following information:

- Specify resets by using the *reset* constraint.
- Use the *ieffort* parameter to specify a higher number of simulation cycles.
- Use the *define_tag* constraint to explicitly specify the initialization value of sequential elements.

### Example Code and/or Schematic

See *Sample Register Info Report*.

### Default Severity Label

Info

### Rule Group

Info

### Reports and Related Files

This rule generates the following files:

- An initial state VCD file that has the simulation vectors applied on primary inputs during initial state search.
- The Advanced Lint-reg report lists registers that could not be initialized. For details on this report, see *Register Info Report*.
- *Uninitialized_Sequential_Elements Spreadsheet Report*
- *Auto Verify Central Report*

## Av_report01

**Reports statistics of properties and functional constraints set on a design.**

### When to Use

Use this rule to view details of analyzed properties and functional constraints set on a design. It provides summarized views of number and status of properties in the design.

#### Prerequisites

Specify clock signals in a design by using the *clock* constraint.

### Description

The *Av_report01* rule reports total number of properties analyzed and the number of functional constraints set on a design.

**NOTE:** *The Av_report01 rule is automatically run when you run any rule of SpyGlass Auto Verify solution.*

#### Dependency on the audit Parameter

| | |
|---|---|
| **audit parameter set to yes** | The rule reports the number of different types of properties in a design.<br>The property file specified by the *propfile* parameter is ignored in this case. |
| **audit parameter set to no** | The rule performs functional analysis and reports the number of properties analyzed, failed, passed, and partially analyzed.<br>The property file specified by the *propfile* parameter is considered in this case. |

### Parameter(s)

- *audit*: The default value is `no`. Set this parameter to `yes` to not perform functional analysis.
- *propfile*: The default value is `NULL`. Set this parameter to the name of the property file containing the properties to be checked.

## Constraints

*clock* (Mandatory): Use this constraint to specify clock signals in a design.

## Messages and Suggested Fix

### Message 1

The following message appears to report the number of properties analyzed and number of functional constraints set on a design when the *audit* parameter is specified:

```
[INFO] Functional analysis not done in audit mode. Design has
'<num>' properties, '<imp-num>' implicit properties, '<ovl-
num>' OVL properties, and '<constr-num>' functional constraints
for top design unit '<du-name>'. Refer file: '<file-name>' for
details
```

The details of the arguments of the above message are described in the following table:

| Argument | Description |
|---|---|
| <num> | Specifies the total number of properties in the design |
| <imp-num> | Specifies the total number of implicit properties |
| <ovl-num> | Specifies the total number of OVL properties |
| <constr-num> | Specifies the total number of functional constraints |
| <du-name> | Specifies the top-level design name |
| <file-name> | Specifies the name of the generated property file |

#### *Potential Issues*

Not applicable.

#### *Consequences of Not Fixing*

Not applicable.

#### *How to Debug and Fix*

Not applicable.

### Message 2

The following message appears to report the number of properties analyzed and number of functional constraints set on a design when the *audit* parameter is not specified:

```
[INFO] Implicit: ‘<imp-analyzed-num>’ implicit properties
analyzed, ‘<imp-failed-num>’ failed, ‘<imp-passed-num>’ passed,
‘<imp-partial-num>’ partially analyzed, <imp-not-analyzed-num>
not analyzed for top design unit ‘<du-name>’. Refer file:
'<file-name>' for details
```

```
Constraints: ‘<constr-num>’ Functional Constraints for top
design unit ‘<du-name>’. Refer file: '<file-name>' for details
```

| Argument | Description |
|---|---|
| <imp-analyzed-num> | Specifies the number of implicit properties analyzed |
| <imp-failed-num> | Specifies the number of implicit properties failed |
| <imp-passed-num> | Specifies the number of implicit properties passed |
| <imp-partial-num> | Specifies the number of implicit properties partially analyzed |
| <imp-not-analyzed-num> | Specifies the number of implicit properties not analyzed |
| <constr-num> | Specifies the total number of functional constraints |
| <du-name> | Specifies the top-level design name |
| <file-name> | Specifies the generated property file name |

#### *Potential Issues*

Not applicable

#### *Consequences of Not Fixing*

Not applicable

#### *How to Debug and Fix*

Not applicable

### Example Code and/or Schematic

Not applicable

### Default Severity Label

Info

### Rule Group

Info

### Report and Related Files

- auto_verify.prp file

  This file contains a list of implicit rules that have been checked. This file is saved in the current working directory.

  For details, see *Property File Format* and *Property File Example*.
- *Register Info Report*
- *Auto Verify Central Report*

## Av_rstinf01

**Reports all the resets in a design.**

### When to Use

Run this rule to find resets in a design.

### Description

The *Av_rstinf01* rule reports synchronous and asynchronous resets in a design.

This rule tries to trace the connected nets to find reset information and accordingly categorize the detected resets, as shown in the following table:

| Traced to | Reset Type |
|---|---|
| Primary inputs | Primary Presets/Clears |
| Black box instances and instances of ASIC cells whose functional description is not available | Black box Presets/Clears |
| Outputs of flip-flops | Derived Presets/Clears |
| Hanging nets | Undriven Presets/Clears |
| Outputs of latches or tristate gates | Gated Preset/Clear |

The rules of SpyGlass Auto Verify solution do not directly use the reset information generated by the *Av_rstinf01* rule. The recommended methodology is to use the *Av_rstinf01* rule to generate autoresets.sgdc and the generated file should then be reviewed and edited by the user for future runs of SpyGlass Auto Verify solution.

#### Resets Ignored by the Av_rstinf01 Rule

The *Av_rstinf01* rule ignores the asynchronous resets that match the following criteria:

- If the name of an asynchronous reset contains the string specified by the `filter_named_resets` parameter of SpyGlass CDC solution and the name of the asynchronous reset does not contain the keyword, `rst`, `set`, `res`, or `reset` as a part of its name

- If the name of the reset does not match with the pattern specified by the *reset_convention* parameter, where the reset pin is driven by multiple sources

  If a reset pin is driven by a single source, the naming convention check is not done and the net is considered as a valid reset.

### Rule Exceptions

The *Av_rstinf01* rule has the following exceptions:

- It does not report presets/clears tied to supply/ground or a constant value due to the *set_case_analysis* constraint.
- It uses a heuristic-based approach based on the RTL structure to determine the synchronous resets. The synchronous resets become a part of the data-line path after synthesis. Therefore, they cannot be detected post-synthesis.

  As a result, this rule might not detect synchronous resets in complex RTL structures, as shown in the following example:

```
always@(posedge clk2)
  if(rst1 & enable) // Expressions are not detected
                    //as synchronous resets
    q <= 1'b0;
  else
    q <= d;
```

## Parameter(s)

- `use_inferred_resets`: The default value is `no`. Set this parameter to `yes` to use automatically-generated reset information.

**NOTE:** *This is the parameter of SpyGlass CDC solution.*

- *reset_convention*: The default value is " ". Set this parameter to a comma or space-separated list of reset names. You may also specify Perl regular expressions.

## Constraint(s)

- *clock* (Mandatory): Use this constraint to specify clocks in a design.
- *reset* (Optional): Use this constraint to specify resets in a design.

## Messages and Suggested Fix

### Message 1

The following message appears to specify the synchronous reset detected in a design:

**[RSTSYNC] [INFO]** `Synchronous <Set | Clear> candidate: <rst-name> of type <rst-type>`

The details of the arguments of the above message are as follows:

- *<rst-name>* specifies the reset name.
- *<rst-type>* specifies the reset type, which can be any of the following:

| | | |
|---|---|---|
| For Set | Primary Set | Black box Set |
| | Derived Set | Undriven Set |
| For Clear | Primary Clear | Generated Clear |
| | Derived Clear | Undriven Clear |

#### *Potential Issues*

Not applicable

#### *Consequences of Not Fixing*

Not applicable

#### *How to Debug and Fix*

Not applicable

### Message 2

The following message appears to specify the asynchronous reset detected in a design:

**[RSTASYNC] [INFO]** `Asynchronous <Set | Clear> candidate: <rst-name> of type <rst-type>`

The details of the arguments of the above message are as follows:

- *<rst-name>* specifies the reset name.
- *<rst-type>* specifies the reset type, which can be any of the following:

| | | |
|---|---|---|
| For Set | Primary Set | Black box Set |
| | Derived Set | Undriven Set |
| For Clear | Primary Clear | Generated Clear |
| | Derived Clear | Undriven Clear |

### *Potential Issues*

Not applicable

### *Consequences of Not Fixing*

Not applicable

### *How to Debug and Fix*

Not applicable

## Example Code and/or Schematic

### Example 1

Consider the following example:

```
assign rst = !rst1;
always@(posedge clk2 or posedge rst)
  if(rst)
    q <= 1'b0;
  else
    q <= d;
```

The following figure represents the above example:

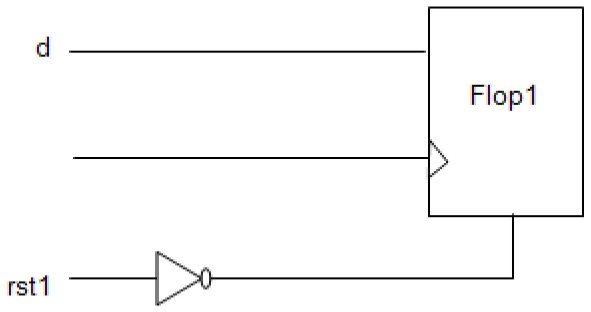


**FIGURE 7.** Reset as a asynchronous clear candidate

For the above example, the *Av_rstinf01* rule detects the `rst1` reset of the type *primary clear*. The following message appears in this case:

```
Asynchronous clear candidate: top.rst1 of type Primary clear
```

In addition, this rule generates the autoresets.sgdc file containing the following constraint:

```
reset -name "top.rst1" -value 0
```

### Example 2

Consider the following example when the *reset_convention* parameter is set to `rst*`:

```
assign syncRst = rst1 & enable;
always@(posedge clk2)
  if(syncRst)
    q <= 1'b0;
  else
    q <= d;
```

The following figure represents the above example:

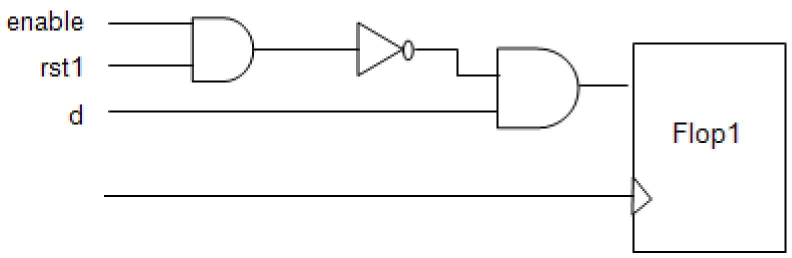


**FIGURE 8.** Reset as a synchronous clear candidate

For the above example, the *Av_rstinf01* rule detects the `rst1` reset of the type *primary clear*. The following message appears in this case:

```
Synchronous clear candidate: top.rst1 of type Primary clear
```

In addition, this rule generates the autoresets.sgdc file containing the following constraint:

```
reset -name "top.rst1" -value 1 -sync
```

## Default Severity Label

Info

## Rule Group

Info

## Report and Related Files

The *Av_rstinf01* rule generates the following files:

- autoresets.sgdc

  This file reports all primary resets and black box presets/clears specified in the SGDC format.

  This file, however, does not report undriven presets/clears.

- generated_resets.sgdc

  This file contains all derived presets/clears.

  Currently, definite and probable asynchronous resets are not categorized in this file.

- *Auto Verify Central Report*

# Formal Setup Rules

The following table lists the formal setup rules:

| **Rule** | **Reports** |
|---|---|
| *Av_sanity03* | Loops in the design |
| *Av_sanity04* | Over-constraining in a design |
| *Av_svasetup01* | Issues in SVA constraints |

## Av_sanity03

**Reports loops in a design**

### When to Use

Use this rule during functional analysis to detect loops in a design.

### Description

The *Av_sanity03* rule reports the following loops in a design:

- All combinational loops
- Loops involving clock to Q, preset to Q, or clear to Q paths of a flip-flop

If you want to check over constraining due to unstable combinational loops, run the *Av_sanity04* rule.

**NOTE:** *By default, this rule is not run.*

### Parameter(s)

None

### Constraint(s)

- *set_case_analysis* (Optional): Use this constraint to specify case analysis conditions.
- *clock* (Optional): Use this constraint to specify clock signals in a design.
- *reset* (Optional): Use this constraint to specify reset signals in a design.

### Messages and Suggested Fix

The following message appears if a loop involving the net *<net-name>* is present in a design:

```
[WARNING] Loops involving net '<net-name>' detected
```

In the above message, name of the first-found user net, *<net-name>*, in an unstable loop is reported. In case of internally generated nets, *<synth_gen_net>* is displayed.

***Potential Issues***

This violation appears if your design contains loops.

### *Consequences of Not Fixing*

Functional analysis cannot be performed in the presence of unstable combinational loops.

### *How to Debug and Fix*

Not applicable

## Example Code and/or Schematic

Consider the following design containing a combinational loop:

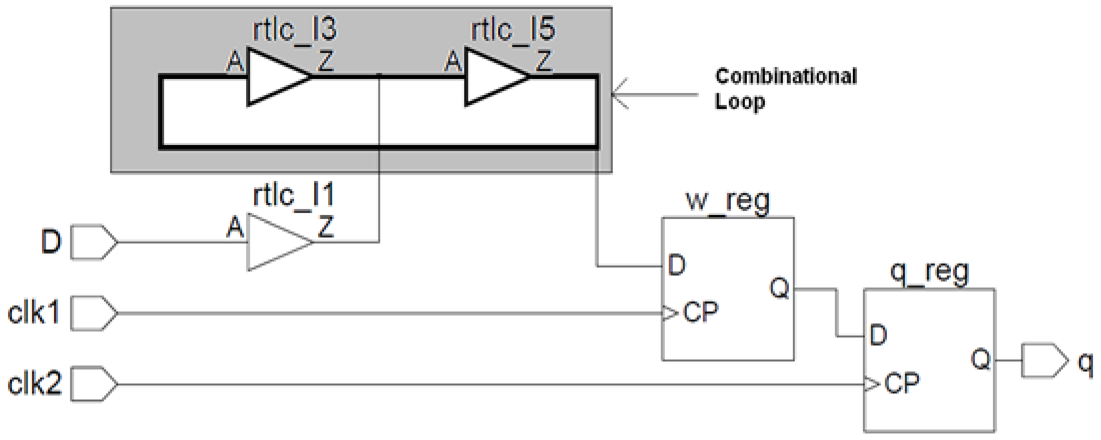


**FIGURE 9.** Ac_sanity03 rule violation

The *Av_sanity03* rule reports a violation in the above case.

## Default Severity Label

Warning

## Rule Group

Sanity

## Reports and Related Files

*Auto Verify Central Report*

## Av_sanity04

**Reports over-constraining in a design**

### When to Use

Use this rule in the pre-layout phase of a design to detect over-constraining in the design.

### Description

The *Av_sanity04* rule reports over-constraining in a design.

SpyGlass Auto Verify consolidates all the user-specified and generated constraints and applies them together. The *Av_sanity04* rule reports conflicting constraints in the *Overconstrain Info File* in the current working directory.

**NOTE:** *By default, this rule is not run.*

### Messages and Suggested Fix

The following message appears to indicate conflicting constraints that are consolidated in a file:

```
[FATAL] There are un satisfiable constraints. Refer
file:'<file-name>' for details
```

#### *Potential Issues*

Not applicable

#### *Consequences of Not Fixing*

If you do not fix this violation, SpyGlass run does not proceed further.

#### *How to Debug and Fix*

Open the file (*Overconstrain Info File*) pointed by the message of this rule and check for the conflicting constraints.

## Example Code and/or Schematic

### Example 1

Consider the following constraints specified for a design:

```
clock -name clk1 -period 5
reset -name clk1 -value 0
```

For the above example, the Av_sanity04 rule reports a violation because of the conflicting *reset* and *clock* constraints on the same net.

### Example 2

Consider the following figure:

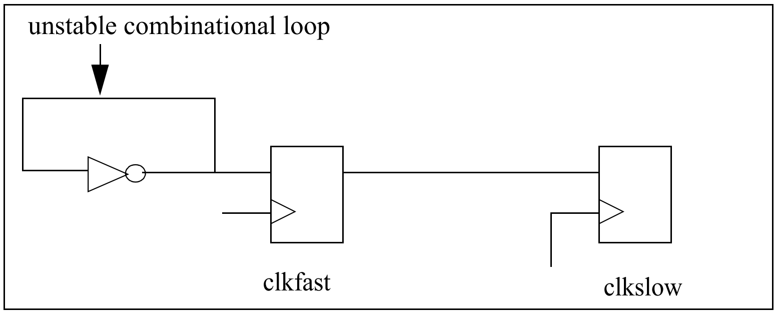


**FIGURE 10.** Av_sanity04 rule violation

For the above example, the Av_sanity04 rule reports a violation because of the presence of an unstable combinational loop.

### Example 3

Consider the following files specified during SpyGlass analysis:

```
//Verilog File

module top(input D,clk1,clk2,rst,output reg q);
reg w;
always @(posedge clk1 or negedge rst)
   if(!rst)
     w<=1'b0;
   else w<=D;
always @(posedge clk2 or negedge rst)
   if(!rst)
     w<=1'b0;
   else q<=w;
assert_proposition #(0,1) constraint (1'b1,!rst); //Ties rst to 0
endmodule
```

```
//SGDC file

current_design top

clock -name clk1 -period 5
clock -name clk2 -period 10
reset -name rst -value 0
```

For the above example, the Av_sanity04 rule reports a violation because of conflicting OVL constraints.

## Default Severity Label

Fatal

## Rule Group

Sanity

## Reports and Related Files

The *Av_sanity04* rule generates the following file(s):

- *Overconstrain Info File*

  This file contains details of conflicting constraints.
- *Auto Verify Central Report*

## Av_svasetup01

**Setup issues in SVA constraints**

### When to Use

Use this rule to parse SVA constraints and report issues related with these constraints.

#### Prerequisites

Specify the following project-file command:

```
set_option enableSVA yes
```

### Description

The *Av_svasetup01* rule parses SVA constraints and reports issues related with these constraints.

For details, refer to the *Using SystemVerilog Assertions* application note.

### Parameter(s)

None

### Constraint(s)

None

### Messages and Suggested Fix

This rule reports different messages based on the issues in SVA constraints. All these messages are described in the *Using SystemVerilog Assertions* application note.

#### Potential Issues

Refer to the *Using SystemVerilog Assertions* application note.

***Consequences of Not Fixing***

Refer to the *Using SystemVerilog Assertions* application note.

***How to Debug and Fix***

Refer to the *Using SystemVerilog Assertions* application note.

## Example Code and/or Schematic

Refer to the *Using SystemVerilog Assertions* application note.

## Default Severity Label

Warning

## Rule Group

SETUP

## Reports and Related Files

No report or related file

## Implicit Properties Rules

Implicit properties are the automatically extracted properties of a design.

Examples of such properties are:

- Avoidance of bus contention and floating bus.
- Avoidance of proper fast to slow clock crossings.

The following table describes the rules under this category:

| Rule | Reports... |
|---|---|
| *Av_bitstuck01* | Nets that are stuck at a constant value after functional analysis |
| *Av_staticnet01* | Globally stuck-at-0 or stuck-at-1 nets in a design |
| *Av_bus01* | Cases where multiple drivers (more than one) are writing into a bus line simultaneously |
| *Av_bus02* | Un-driven bus lines |
| *Av_case01* | `case` constructs with a `fullcase` pragma attribute attached and the `case` items are not complete (not all items are present) |
| *Av_case02* | Overlapping items of a `case` construct when the `case` construct is associated with a parallel case pragma |
| *Av_deadcode01* | Dead code caused by a condition never triggered |
| *Av_dontcare01* | X assignments that are found to be reachable |
| *Av_fsm01* | Unreachable or deadlocked FSM States |
| *Av_fsm02* | Edges between two states of an FSM that cannot be sensitized |
| *Av_range01* | (Verilog) Arrays that can potentially be accessed with an index outside the range of the array |
| *Av_setreset01* | Flop with simultaneous active asynchronous set and asynchronous reset |
| *Av_staticreg01* | All the static registers in the design which do not change its value after attaining it once. |
| *Av_staticreg02* | Static sequential elements in the design |
| *Av_syncfifo01* | Overflow or underflow for synchronous FIFOs in a design |

## Av_bitstuck01

**This rule is deprecated**

The functionality of this rule is covered by the *Av_staticnet01* rule.

## Av_staticnet01

**Reports globally stuck-at-0 or stuck-at-1 nets in a design.**

## When to Use

Use this rule to detect nets that are stuck to a constant value.

### Prerequisites

Specify clocks in a design by using the *clock* constraint.

## Description

The *Av_staticnet01* rule reports globally stuck-at-0 nets (`s-a-0`) or stuck-at-1 nets (`s-a-1`) for the following statements:

- LHS of variable `assign` statement

  In this case, this rule checks if the register (flip-flop or latch) that is generated by the LHS of a variable assignment is `s-a-0` or `s-a-1`.

- LHS of explicit `assign` statement

  In this case, this rule checks if the assigned net in the LHS of an explicit assignment is `s-a-0` or `s-a-1`.

- RHS of explicit `assign` statement

  In this case, this rule checks if the read net in the RHS of an explicit assignment is `s-a-0` or `s-a-1`.

### Rule Exceptions

The *Av_staticnet01* rule has the following exceptions:

- It does not check for the variable assignment in the combinational always block.
- It does not report a violation for the signals that cannot be initialized to an initial state.

  For example, consider the following figure:

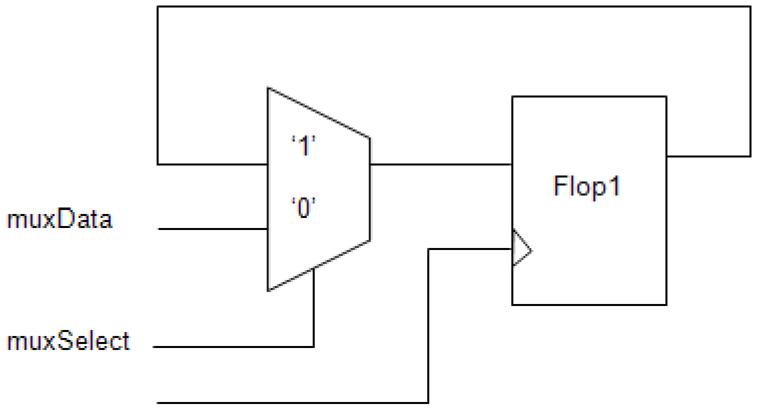


**FIGURE 11.** Non violating case of the Av_staticnet01 rule

In the above figure, when `muxSelect` is tied to an active-high value (as `Flop1` does not have a reset), it leads to an uninitialized flip-flop. Therefore, the *Av_staticnet01* rule does not report a violation in this case.

## Parameter(s)

- *staticnet_scope*: The default value is `flop`. Set this parameter to `lhs` to perform rule-checking only on the LHS assignment nets. The other possible values are `rhs` and `all`.
- *buscompress*: The default value is yes. Set this parameter to `no` to check all the bits of a bus signal.
- `use_inferred_clocks`: The default value is `no`. Set this parameter to `yes` to use automatically-generated clock information.

**NOTE:** *This is the parameter of SpyGlass CDC solution.*

- `use_inferred_resets`: The default value is `no`. Set this parameter to `yes` to use automatically-generated reset information.

**NOTE:** *This is the parameter of SpyGlass CDC solution.*

- *fv_debug_sim_cycles*: The default value is 0. Set this parameter to any positive integer to display waveform from the initial state for the failed properties of the Av_staticnet01 rule.

- *av_dump_assertions*: The default value is “”. Set this parameter to `sva` to generate SystemVerilog Assertions (SVA).

## Constraint(s)

- *clock* (Mandatory): Use this constraint to specify clock signals in a design.
- *formal_analysis_filter* (Optional): Use this constraint to specify the modules or hierarchies on which formal analysis should be ignored or performed.
- *reset* (Optional): Use this constraint to specify reset signals in a design.
- *set_case_analysis* (Optional): Use this constraint to specify case analysis conditions.
- *breakpoint* (Optional): Use this constraint to specify breakpoints in a design where functional analysis should stop.
- *watchpoint* (Optional): Use this constraint to generate a waveform for an internal signal.
- *ip_block* (Optional): Use this constraint to specify IP blocks in your design.
- *meta_design_hier* (Optional): Use this constraint to specify the top-level design name and the hierarchical name of the design with respect to the simulation test bench to be used during SVA dumping of Partially Proved Properties.

## Messages and Suggested fix

The following message appears to specify a net that is globally stuck-at-0 or 1:

```
[ERROR] <expr-type> <net-name> is globally stuck-at-<0|1>
<reason>
```

The details of the arguments of the above violation message are described in the following table:

| Argument | Description |
|---|---|
| <expr-type> | Specifies the expression type, such as RHS net, LHS net, or LHS reg variable. |
| <net-name> | Specifies the net name. |
| <reason> | Specifies the following text:<br>`reason: static nets in fanin cone)`<br><br>The above text appears if the constant value propagation has trivially resulted in a stuck net.<br><br>**Note:** Violations containing this reason appear first followed by the other Av_staticnet01 violations. |

In addition, if a set of conflicting constraints are specified for the design, additional message is appended to the above message string. For details on the message and how to debug and fix this conflict, see the *Overconstrain Info File* section.

### *Potential Issues*

This violation appears if your design contains nets that are stuck to a constant value.

### *Consequences of Not Fixing*

If you do not fix this violation, the functionality of the design is not as expected.

### *How to Debug and Fix*

For cases in which constant value propagation trivially resulted in a stuck net, this rule generates a schematic.

Use the *Av_Info_Case_Analysis* rule to view the constant values in the path in the schematic. For details, see *Example Code and/or Schematic*.

If the nets stuck to a constant value are in an IP and you do not want to report within the IP, specify the *ip_block* constraint for the IP.

## Example Code and/or Schematic

### Example 1

Consider the following example:

```
module testve_staticNet01 (input in1, clk, rst,output w4);
  wire w1, w2;
  wire [1:0] w3;
  reg rseq1;
  assign w4 = w1;
  assign w1 = (in1| w2 | w3[0] | rseq1);
  // RHS may contain any valid operators
  always@(posedge clk or posedge rst)
  if(rst)
  rseq1 <= 1'b0;
  else
  rseq1 <= w4;
endmodule
```

For the above example, the *Av_staticnet01* rule checks for the following nets:

- `rseq1` in the LHS of variable assign statement
- `w1` and `w4` in the LHS of explicit assign statement. This is checked when *staticnet_scope* parameter is set to `lhs`.
- `in1`, `w2`, and `w3 [0]` in the RHS of the explicit `assign` statement. This is checked when *staticnet_scope* parameter is set to `rhs`.

Now consider that `w2` is stuck-at-1 in the above example. This case results in a chain of stuck-at-1,that is, w2=>w1=>w4, and the *Av_staticnet01* rule reports all the nets in the chain.

### Example 2

Consider the example shown in the following figure:

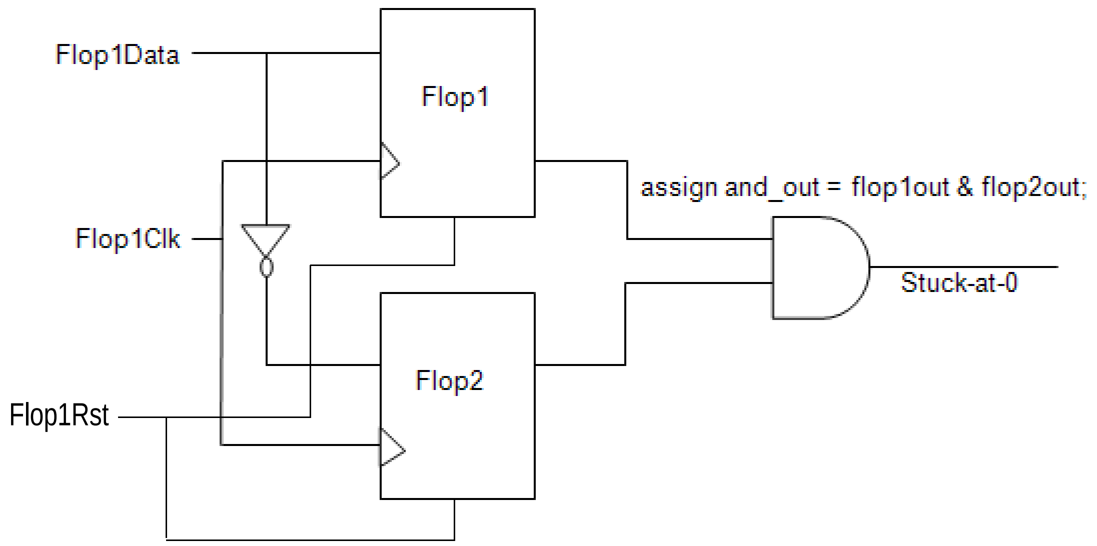


**FIGURE 12.** Example of the Av_staticnet01 rule violation

For the above example, the *Av_staticnet01* rule reports the `and_out` signal that is stuck-at-0.

## Default Severity Label

Error

## Rule Group

Implicit-Properties

## Report and Related Files

- *Register Info Report*
- Av_staticnet01.<ID>.OverConstrainInfo: This file contains details of conflicting constraints. For details, see *Overconstrain Info File*.
- *Auto Verify Central Report*

## Av_bus01

**Reports all the bus contentions in a design.**

### When to Use

Use this rule to detect cases resulting in bus contention.

#### Prerequisites

Specify clocks in the design by using the *clock* constraint.

### Description

The *Av_bus01* rule reports a violation when multiple drivers write to a bus-line simultaneously.

### Parameter(s)

*av_dump_assertions*: The default value is “”. Set this parameter to `sva` to generate SystemVerilog Assertions (SVA).

### Constraint(s)

- *clock* (Mandatory): Use this constraint to specify clock signals in a design.
- *formal_analysis_filter* (Optional): Use this constraint to specify the modules or hierarchies on which formal analysis should be ignored or performed.
- *reset* (Optional): Use this constraint to specify reset signals in a design.
- *set_case_analysis* (Optional): Use this constraint to specify case analysis conditions.
- *breakpoint* (Optional): Use this constraint to specify breakpoints in a design where functional analysis should stop.
- *watchpoint* (Optional): Use this constraint to generate a waveform for an internal signal.
- *ip_block* (Optional): Use this constraint to specify IP blocks in your design.

## Messages and Suggested Fix

The following message appears when multiple drivers write into a bus-line simultaneously:

`[WARNING]` `There is contention writing into Bus line '<bus-line-name>'`

In addition, if a set of conflicting constraints are specified for the design, additional message is appended to the above message string. For details on the message and how to debug and fix this conflict, see the *Overconstrain Info File* section.

### *Potential Issues*

This violation appears if your design contains drivers that write to a bus-line simultaneously.

### *Consequences of Not Fixing*

If you do not fix this violation, conflicting data on the bus can result in contention.

### *How to Debug and Fix*

To debug this violation, double-click on the violation and check the following to determine the cause of the violation:

- The rule-violating line highlighted in the *HDL Viewer* pane.
- The schematic.
- The waveform viewer to check the witness signals.

To fix this violation, modify the RTL so that multiple drivers are not active simultaneously. Else, use the *set_case_analysis* constraint to apply case analysis on the enables that may not be active as per the design.

If the multiple drivers that are active simultaneously are in an IP and you do not want to report within the IP, specify the *ip_block* constraint for the IP.

## Example Code and/or Schematic

Consider the following example:

```
//test.v

`define STATE1 4'b1000
`define STATE2 4'b0100
`define STATE3 4'b0010
`define STATE4 4'b0011

module Fsm(reset, clk, ctl, in1, in2, out);
  input reset, clk, ctl, in1, in2;
  output out;
  reg [0:3] state;
  always @(posedge clk) begin
    if(reset) begin
      state <= `STATE1;
      end
    else begin
      case (state)
        `STATE1 : if(ctl) state <= `STATE2;
        `STATE2 : state <= `STATE3;
        `STATE3 : state <= `STATE4;
        `STATE4 : state <= `STATE1;
        default : ;
      endcase
       end
    end
//Busline with tristates enabled by one-hot encoded FSM
  assign out = state[0] ? in1 & in2 : 1'bz;
  assign out = state[1] ? in1 | in2 : 1'bz;
  assign out = state[2] ? in1 ^ in2 : 1'bz;
  assign out = state[3] ? !in1 | in2 : 1'bz;
endmodule
```

**Project File:**

```
set_parameter use_inferred_clocks yes
```

In the above example, STATE4 will enable state[2] and state[3] simultaneously, which results in contention on the out signal.

In this case, SpyGlass generates the schematic displaying contentious bus with multiple enables of the tristate buffers being active simultaneously. See the following figure:

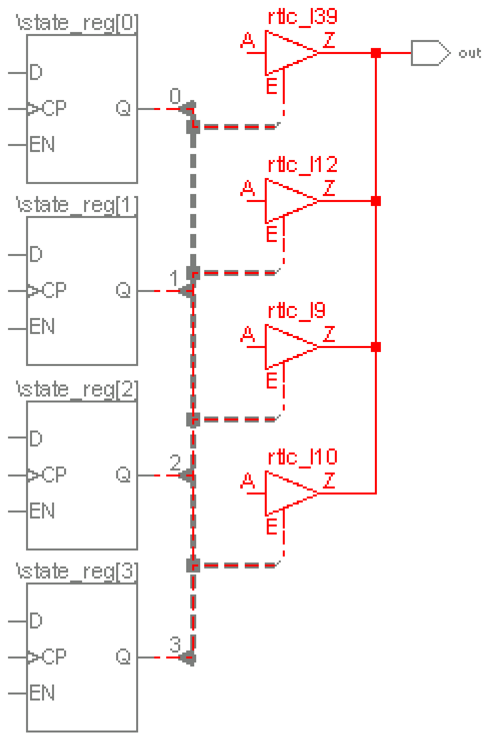


**FIGURE 13.** Schematic of the Av_bus01 rule violation

In addition, the waveform viewer of the above violation displays the enables that are active at the same time, resulting in contention on the output.

The following figure shows the waveform generated in this case:

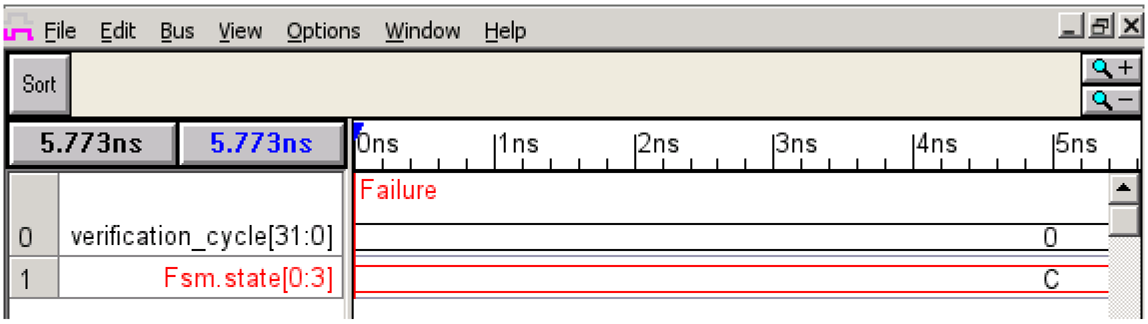

File Edit Bus View Options Window Help
Sort
5.773ns
5.773ns
0ns
1ns
2ns
3ns
4ns
5ns
Failure
0 verification_cycle[31:0]
1 Fsm.state[0:3]
0
C

**FIGURE 14.** Waveform of the Av_bus01 rule violation

The above waveform shows the enables that are active at the same time, resulting in the contention on output.

To fix this violation, ensure that tri-state drivers are not active in the design simultaneously.

## Default Severity Label

Warning

## Rule Group

Implicit-Properties

## Report and Related Files

- *Register Info Report*
- Av_bus01.<ID>.OverConstrainInfo: This file contains details of conflicting constraints. For details, see *Overconstrain Info File*.
- *Auto Verify Central Report*

## Av_bus02

**Reports all the floating buses in the design.**

### When to Use

Use this rule to ensure that all the enables driving a bus are not inactive simultaneously.

#### Prerequisites

Specify clocks in a design by using the *clock* constraint.

### Description

The *Av_bus02* rule reports undriven bus lines.

### Parameter(s)

*av_dump_assertions*: The default value is “”. Set this parameter to `sva` to generate SystemVerilog Assertions (SVA).

### Constraint(s)

- *clock* (Mandatory): Use this constraint to specify clock signals in a design.
- *formal_analysis_filter* (Optional): Use this constraint to specify the modules or hierarchies on which formal analysis should be ignored or performed.
- *reset* (Optional): Use this constraint to specify reset signals in a design.
- *set_case_analysis* (Optional): Use this constraint to specify case analysis conditions.
- *breakpoint* (Optional): Use this constraint to specify breakpoints in a design where functional analysis should stop.
- *watchpoint* (Optional): Use this constraint to generate a waveform for an internal signal.
- *ip_block* (Optional): Use this constraint to specify IP blocks in your design.

## Messages and Suggested Fix

The following message appears to report undriven bus lines in a design:

`[WARNING] Bus line '<bus-line-name>' may be floating`

In addition, if a set of conflicting constraints are specified for the design, additional message is appended to the above message string. For details on the message and how to debug and fix this conflict, see the *Overconstrain Info File* section.

### *Potential Issues*

This violation appears if your design contains undriven bus lines.

A bus line is undriven when all the enables driving the bus line are inactive simultaneously.

### *Consequences of Not Fixing*

If you do not fix this violation, the design nets are floating. This impacts the design functionality.

### *How to Debug and Fix*

To debug this violation, trace the waveform viewer to check if all the enables driving the reported bus are inactive simultaneously.

If all the enables are inactive simultaneously, adjust the enables in the design such that reported nets are appropriately driven.

If the undriven bus lines are in an IP and you do not want to report within the IP, specify the *ip_block* constraint for the IP.

## Example Code and/or Schematic

Consider the following example in which:

- A bus line driven by four tristates.
- The enables of the tristates are controlled from an FSM, which is one-hot encoded.
- One state of the register is encoded with all registers at '0' causing a violation of this rule.

**// test.v**

```
`define STATE1 4'b1000
`define STATE2 4'b0100
`define STATE3 4'b0010
`define STATE4 4'b0000


module Fsm(reset, clk, ctl, in1, in2,  out);
  input  reset, clk, ctl, in1, in2;
  output out;
  reg [0:3] state;
  always @(posedge clk) begin
    if(reset) begin
      state <= `STATE1;
      end
    else begin
      case (state)
        `STATE1 : begin
          if(ctl) state <= `STATE2;
          end
        `STATE2 : begin
            state <= `STATE3;
          end
        `STATE3  : begin
            state <= `STATE4;
          end
        `STATE4  : begin
            state <= `STATE1;
       end
        default     : ;
      endcase
    end
  end
// Busline with tristates enabled by
//one-hot encoded FSM
  assign out = state[0] ? in1 & in2 : 1'bz;
  assign out = state[1] ? in1 | in2 : 1'bz;
  assign out = state[2] ? in1 ^ in2 : 1'bz;
  assign out = state[3] ? !in1 | in2 : 1'bz;
endmodule
```

**Project File:**

```
set_parameter use_inferred_clocks yes
```

In the above example, none of the enables are active in STATE4. As a result, the out signal is floating in the design. Therefore, the *Av_bus02* rule reports a violation in this case.

To debug this violation, double-click the violation and open the *Incremental Schematic* window. The following figure shows the *Incremental Schematic* window in this case:

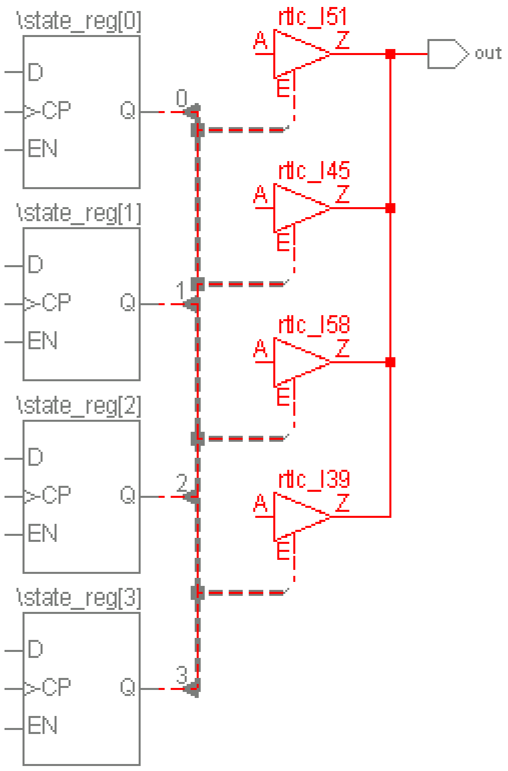


**FIGURE 15.** Schematic of the Av_bus02 rule violation

In the above schematic, notice that the `out` net is floating.

To view the condition resulting in this floating net, view the waveform of the violation. Such condition occurs when all the enables of the tristate drivers are inactive.

The following figure shows the waveform showing this condition:

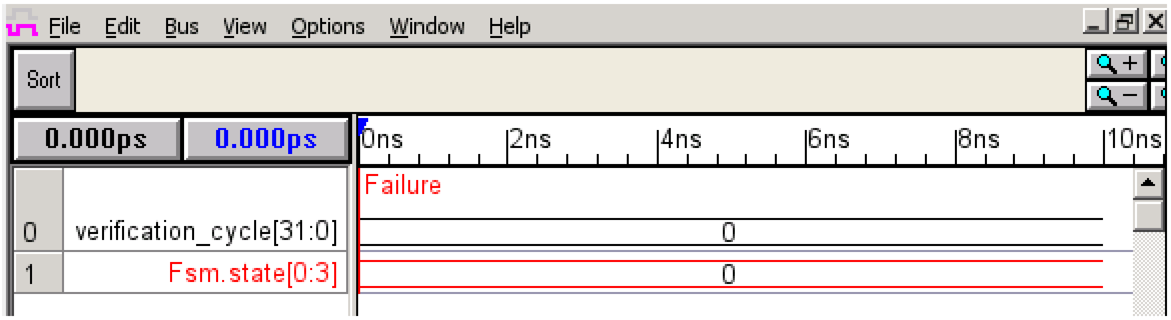


**FIGURE 16.** Waveform of the Av_bus02 rule violation

To fix this violation, ensure that at least one driver is always active for the reported bus.

**NOTE:** *This rule does not require user specified properties.*

## Default Severity Label

Warning

## Rule Group

Implicit-Properties

## Report and Related Files

- Av_bus02.<ID>.OverConstrainInfo: This file contains details of conflicting constraints. For details, see *Overconstrain Info File*.
- *Auto Verify Central Report*

## Av_case01

**Reports reachable case items that are not specified.**

## When to Use

Use this rule to check for coding issues related to `case` statements.

### Prerequisites

Specify clock signals in the design by using the *clock* constraint.

## Description

The *Av_case01* rule reports reachable `case` items that are missing in the `case` statement on which the `full_case` pragma or the `priority` modifier is attached.

### Points to be Noted

Please note the following points:

- Note that when a function containing a case statement is called multiple times, this rule reports a violation at the line containing the case expression for each violating function call. Such violations may appear duplicate.

  In such cases, use the schematic of a violation to analyze the corresponding function call.
- In case of the `unique` case without a `default` label, both the *Av_case01* and *Av_case02* rules are checked.

### Rule Exceptions

The *Av_case01* rule has the following exceptions:

- It does not report violation for the `case` statements that have a default branch.
- It does not report a violation if the missing (uncovered) `case` item can never be executed. Such items are known as unreachable case items.

- It does not check functions that are without a `begin-end` block and contain a `case` statement in which the expression is a `case-select` expression, containing an operator, over the inputs of the function.

## Parameter(s)

*av_dump_assertions*: The default value is “”. Set this parameter to `sva` to generate SystemVerilog Assertions (SVA).

## Constraint(s)

- *clock* (Mandatory): Use this constraint to specify clock signals in a design.
- *formal_analysis_filter* (Optional): Use this constraint to specify the modules or hierarchies on which formal analysis should be ignored or performed.
- *reset* (Optional): Use this constraint to specify reset signals in a design.
- *set_case_analysis* (Optional): Use this constraint to specify case-analysis conditions.
- *breakpoint* (Optional): Use this constraint to specify breakpoints in a design where functional analysis should stop.
- *watchpoint* (Optional): Use this constraint to generate a waveform for an internal signal.
- *ip_block* (Optional): Use this constraint to specify IP blocks in your design.

## Messages and Suggested Fix

This rule reports the following message:

```
[WARNING] Case statement over '<sig-name-list>' is not a full-
case (Uncovered Item: '<case-item-name>')
```

The details of the arguments of the above violation message are described in the following table:

| **Argument** | **Description** |
|---|---|
| <sig-name-list> | Specifies the case statement sensitivity signals |
| <case-item-name> | Specifies the case item that is not covered |

In addition, if a set of conflicting constraints are specified for the design, additional message is appended to the above message string. For details on the message and how to debug and fix this conflict, see the *Overconstrain Info File* section.

### *Potential Issues*

This violation appears if your design file contains a `case` statement with the `full_case` pragma or a `priority` modifier, and one of the case items to be executed is missing.

### *Consequences of Not Fixing*

If you do not fix this violation, the output from the case statement cannot be determined.

### *How To Debug and Fix*

To fix this violation, perform the following steps:

1. Double-click on the violation message.

   SpyGlass highlights the rule-violating line in the *HDL Viewer* pane.
2. Review the `case` statement and check if the missing `case` item is required in the `case` statement.

   If it not required, ignore the violation.

   However, if it is required, add the missing `case` item to avoid any unexpected results.

If the `case` statement is present in an IP and you do not want to report within the IP, specify the *ip_block* constraint for the IP.

## Example Code and/or Schematic

Consider the following example:

```
module FullCase(clk, in1, in2, out);
  input  clk, in1, in2;
output reg out;
  wire [0:1] bs;
  assign bs = {in1,in2};
  always @(posedge clk) begin
    case (bs) // synopsys full_case
      2'b00: out <= 0;
      2'b01: out <= 0;
      2'b10: out <= 1;
    endcase
  end
endmodule
```

In the above example, the `full_case` pragma is attached to the `case` statement.

Now consider that during the execution of the above code, `bs` attains the value `2'b11`. However, this value is not covered by any of the `case` items. Therefore, the *Av_case01* rule reports a violation.

## Default Severity Label

Warning

## Rule Group

Implicit Properties

## Report and Related Files

- Av_case01.<ID>.OverConstrainInfo: This file contains details of conflicting constraints. For details, see *Overconstrain Info File*.
- *Auto Verify Central Report*

## Av_case02

**Reports overlapping case items of the case statement that have the parallel_case pragma or the unique modifier attached.**

## When to Use

Use this rule to check for coding issues related to `case` statements.

### Prerequisites

Specify clock signals in the design by using the *clock* constraint.

## Description

The *Av_case02* rule reports *Overlapping Case Items* of the `case` statement that have the `parallel_case` pragma or the `unique` modifier attached.

Note that when a function containing a case statement is called multiple times, this rule reports a violation at the line containing the case expression for each violating function call. Such violations may appear duplicate. In such cases, use the schematic of a violation to analyze the corresponding function call.

### Overlapping Case Items

Overlapping case items occur in any of the following situations:

- When the same case items are repeated twice in a `case` statement, as shown in the following example:

```
casex (bs)
  2'b11: out <= 0;
  2'b11: out <= 1;  //Case item 2'b11 repeated twice
endcase
```

- When a case item covers another case item, as shown in the following example:

```
 casex (bs)
   2'bx1: out <= 0;
   2'b11: out <= 1;  // Overlaps with x1
```

```
    endcase
```

### Rule Exception

The *Av_case02* rule does not check functions that are without a `begin-end` block and contain a `case` statement in which the expression is a `case-select` expression, containing an operator, over the inputs of the function.

## Parameter(s)

*av_dump_assertions*: The default value is “”. Set this parameter to `sva` to generate SystemVerilog Assertions (SVA).

## Constraint(s)

- *clock* (Mandatory): Use this constraint to specify clock signals in a design.
- *formal_analysis_filter* (Optional): Use this constraint to specify the modules or hierarchies on which formal analysis should be ignored or performed.
- *reset* (Optional): Use this constraint to specify reset signals in a design.
- *set_case_analysis* (Optional): Use this constraint to specify case-analysis conditions.
- *breakpoint* (Optional): Use this constraint to specify breakpoints in a design where functional analysis should stop.
- *watchpoint* (Optional): Use this constraint to generate a waveform for an internal signal.
- *ip_block* (Optional): Use this constraint to specify IP blocks in your design.

## Messages and Suggested Fix

The following message appears if a `case` statement has *Overlapping Case Items*:

```
[WARNING] Case statement has overlapping items ‘<case-item-list>’ (Expanded Label: ‘<label>’)
```

The details of the arguments of the above violation message are described in the following table:

| Argument | Description |
|---|---|
| <case-item-list> | Specifies the overlapping items for each overlapping pair |
| <label> | Specifies the expanded label. |

In addition, if a set of conflicting constraints are specified for the design, additional message is appended to the above message string. For details on the message and how to debug and fix this conflict, see the *Overconstrain Info File* section.

### *Potential Issues*

This violation appears if the design file contains *Overlapping Case Items* in the `case` statement on which the `parallel_case` pragma or the `unique` modifier is attached.

### *Consequences of Not Fixing*

If you do not fix this violation, the output of the reported `case` statement cannot be determined.

### *How to Debug and Fix*

To fix this violation, perform the following steps:

1. Double-click on the violation message.

   SpyGlass highlights the rule-violating line in the *HDL Viewer* pane.
2. Modify the *Overlapping Case Items* such that they are mutually exclusive.

If the `case` statement is present in an IP and you do not want to report within the IP, specify the *ip_block* constraint for the IP.

## Example Code and/or Schematic

Consider the following example:

```
// Parallel case pragma validation
module PrllCase(clk, in1, in2, out);
  input  in1, in2, clk;
  output reg out;
  wire [0:1] bs;
  assign bs = {in1, in2};
// Case statement using bs assignments as case items
  always @(posedge clk) begin
    casex (bs) // synopsys parallel_case
      2'b00: out <= 0;
      2'bx1: out <= 0;
      2'b11: out <= 1;  // Overlaps with x1
    endcase
  end
endmodule
```

In the above example, `2'bx1` and `2'b11` are overlapping case items. Therefore, the *Av_case02* rule reports the following violation:

```
Case statement has overlapping items x1, 11 (Expanded Label:
'11')
```

## Default Severity Label

Warning

## Rule Group

Implicit Properties

## Report and Related Files

- Av_case02.<ID>.OverConstrainInfo: This file contains details of conflicting constraints. For details, see *Overconstrain Info File*.
- *Auto Verify Central Report*

## Av_case03

**Reports overlapping case items of the case statement without the parallel_case pragma attached.**

### When to Use

Use this rule to check for coding issues related to `case` statements.

#### Prerequisites

Specify clock signals in the design by using the *clock* constraint.

### Description

The *Av_case03* rule reports a violation when the `case` statement does not have the `parallel_case` pragma attached, and there are *Overlapping Case Items* in the `case` statement.

### Parameter(s)

None

### Constraint(s)

- *clock* (Mandatory): Use this constraint to specify clock signals in a design.
- *reset* (Optional): Use this constraint to specify reset signals in a design.
- *set_case_analysis* (Optional): Use this constraint to specify case-analysis conditions.
- *breakpoint* (Optional): Use this constraint to specify breakpoints in a design where functional analysis should stop.
- *watchpoint* (Optional): Use this constraint to generate a waveform for an internal signal.
- *formal_analysis_filter* (Optional): Use this constraint to specify the modules or hierarchies on which formal analysis should be ignored or performed.

### Messages and Suggested Fix

The following message appears for the `case` statement that does not have the `parallel_case` pragma attached, and there are *Overlapping Case Items*.

**[WARNING]** Case statement has overlapping items <case-item-list> (Expanded Label: '<label>')

The details of the arguments of the above violation message are described in the following table:

| Argument | Description |
|---|---|
| <case-item-list> | Specifies the overlapping items for each overlapping pair |
| <label> | Specifies the expanded label resulting from the overlap between the case labels |

In addition, if a set of conflicting constraints are specified for the design, additional message is appended to the above message string. For details on the message and how to debug and fix this conflict, see the *Overconstrain Info File* section.

#### *Potential Issues*

This violation appears if your design file contains *Overlapping Case Items* in the `case` statement on which no `parallel_case` pragma is attached.

#### *Consequences of Not Fixing*

If you not fix this violation, a wrong value can propagate in the design.

#### *How to Debug and Fix*

To fix this violation, perform the following steps:

1. Double-click on the violation message.

   SpyGlass highlights the rule-violating line in the *HDL Viewer* pane.
2. Modify the *Overlapping Case Items* such that they are mutually exclusive.

### Example Code and/or Schematic

Consider the following example:

```
always @(posedge clk2) begin
  casex (bs)
    2'b00: out <= 0;
    2'bx0: out <= 0;
    2'b10: out <= 1; //Overlaps with above label 2'bx0
  endcase
end
```

In the above example, `x0` and `10` are overlapping `case` items. In both the states, the value of `out` at one place is `0` and it is `1` at the other place.

## Default Severity Label

Warning

## Rule Group

Implicit Properties

## Report and Related Files

*Auto Verify Central Report*

## Av_deadcode01

**Reports redundant logic in the design.**

## When to Use

Use this rule to detect redundant logic in a design due to the presence of dead code.

### Prerequisites

Specify clock signals by using the *clock* constraint.

## Description

The *Av_deadcode01* rule reports dead codes caused by a condition that is never triggered.

This rule does not report redundancies due to re-convergent paths. It only detects the RTL code that can never be exercised due to a branching that is stuck at a false value.

### Message Grouping in the Av_deadcode01 Rule

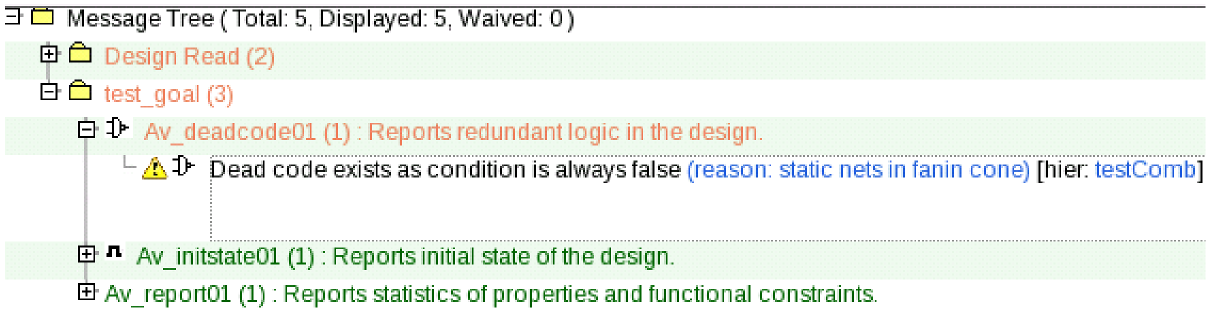
By default, the *Av_deadcode01* rule reports violations for only the top-level `if` statement of a nested `if-else` block (or dependency tree).

For example, consider the following code snippet showing the nested `if` block:

```
always @(in1 or in2)
begin
  if (rst)
    w2 = s1;
  else if (in1 && in2) begin
    w2 = !s1;
    if (in1 | in2) begin
      w2 = w3;
      if (!in1 && in2) begin
        w2 = w3;
        if (in1 || in2) begin
          w2 = w3;
          if (in2) begin
```

```
              w2 = w4;
            end
          end
        end
      end
    end
    else
      w2 = 1'b0;
    end

  always @(posedge clk or posedge rst)
    if (rst)
      out = 1'b0;
    else
      out = w2;
```

For the above example, the *Av_deadcode01* rule reports violation for the assertions in the top-level `if` block. The following figure shows the violation in this case:

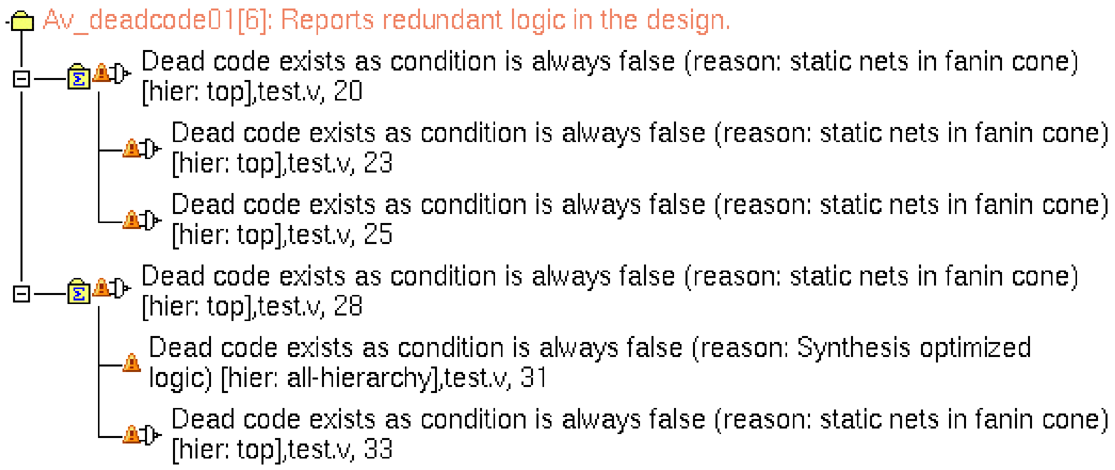


```
Message Tree ( Total: 5, Displayed: 5, Waived: 0 )
  Design Read (2)
  test_goal (3)
    Av_deadcode01 (1) : Reports redundant logic in the design.
      Dead code exists as condition is always false (reason: static nets in fanin cone) [hier: testComb]
    Av_initstate01 (1) : Reports initial state of the design.
    Av_report01 (1) : Reports statistics of properties and functional constraints.
```

**FIGURE 17.**

To configure the *Av_deadcode01* rule to report violations for the assertions present in all the nested `if-else` blocks, set the *av_dcode_report* parameter to `all`. In this case, the *Av_deadcode01* rule groups all the violations of the same `if` block, `else-if` block, or `else` block of a nested `if` block (or dependency tree). Within each group, violations are sorted based on the assertions depth within the `if`, `else-if`, or `else` block.

The following figure shows the grouped violations of the *Av_deadcode01* rule:



Av_deadcode01[6]: Reports redundant logic in the design.

Dead code exists as condition is always false (reason: static nets in fanin cone) [hier: top],test.v, 20

Dead code exists as condition is always false (reason: static nets in fanin cone) [hier: top],test.v, 23

Dead code exists as condition is always false (reason: static nets in fanin cone) [hier: top],test.v, 25

Dead code exists as condition is always false (reason: static nets in fanin cone) [hier: top],test.v, 28

Dead code exists as condition is always false (reason: Synthesis optimized logic) [hier: all-hierarchy],test.v, 31

Dead code exists as condition is always false (reason: static nets in fanin cone) [hier: top],test.v, 33

**FIGURE 18.** Grouping of the Av_deadcode01 violations

## Parameter(s)

- *av_dcode_analysis*: The default value is `soft`. Set this parameter to `strict` to use the strict approach for verification of assertions.
- *av_dcode_report*: The default value is `minimal`. Set this parameter to `all` to group violations of the same `if` block, `else-if` block, or `else` block of a nested `if` block.
- *av_force_soft_reset*: The default value is `Av_setreset01`. Set this parameter to `no` to consider a reset as a hard reset.
- *dead_code_scope*: The default values are `if`, `case_without_default`, `generate`, and `always`. Set the value of this parameter to specify the constructs to be checked. Other possible values are `case`, `if_case`, `condasgn`, and `if_case_condasgn`.

- *fv_debug_sim_cycles*: The default value is 0. Set this parameter to any positive integer to display waveform from the initial state for the failed properties of the Av_deadcode01 rule.
- *fv_dcode_all_inst*: The default value is `no`. Set this parameter to "yes" to view the schematic and waveform for all the instances of the dead code module.
- *include_construct*: The default value is `none`. Set this parameter to `generate` to check the `generate_block` constructs. Other possible values are `always_comb`, `included_file`, and `none`.
- *av_dump_assertions*: The default value is “”. Set this parameter to `sva` to generate SystemVerilog Assertions (SVA).

## Constraint(s)

- *clock* (Mandatory): Use this constraint to specify clock signals in a design.
- *formal_analysis_filter* (Optional): Use this constraint to specify the modules or hierarchies on which formal analysis should be ignored or performed.
- *reset* (Optional): Use this constraint to specify reset signals in a design.
- *set_case_analysis* (Optional): Use this constraint to specify case-analysis conditions.
- *breakpoint* (Optional): Use this constraint to specify breakpoints in a design where functional analysis should stop.
- *watchpoint* (Optional): Use this constraint to generate a waveform for an internal signal.
- *ip_block* (Optional): Use this constraint to specify IP blocks in your design.
- *meta_design_hier* (Optional): Use this constraint to specify the top-level design name and the hierarchical name of the design with respect to the simulation test bench to be used during SVA dumping of Partially Proved Properties (under the *av_dump_assertions* parameter).

## Messages and Suggested Fix

### Message 1

The following message appears if the design contains dead code:

`[DEADCODE] [WARNING]` Dead code exists as condition is always false [Hier:<hier-name>]

Where, *<hier-name>* is the name of the module in which the dead code is detected. This information is not available when the dead code is found to be statically unreachable within the module scope.

In addition, if a set of conflicting constraints are specified for the design, additional message is appended to the above message string. For details on the message and how to debug and fix this conflict, see the *Overconstrain Info File* section.

#### *Potential Issues*

This violation appears if your design file contains dead code.

#### *Consequences of Not Fixing*

If you do not fix this violation, the dead code may result in the generation of extra silicon area.

#### *How to Debug and Fix*

To fix this violation, remove the logic or condition that is resulting in the dead code.

If the dead code is present in an IP and you do not want to report within the IP, specify the *ip_block* constraint for the IP.

### Message 2

The following message appears if constant value propagation has trivially resulted in the dead code:

`[DEADCODE] [WARNING]` Dead code exists as condition is always false (reason: static nets in fanin cone) [Hier:<hier-name>]

In addition, if a set of conflicting constraints are specified for the design, additional message is appended to the above message string. For details on the message and how to debug and fix this conflict, see the *Overconstrain Info File* section.

#### *Potential Issues*

This violation appears if your design file contains dead code.

#### *Consequences of Not Fixing*

If you do not fix this violation, the dead code may result in the generation of extra silicon area.

#### *How to Debug and Fix*

To debug this violation, view the schematic after running the *Av_Info_Case_Analysis* rule to view the constant values in the path.

To fix this violation, remove the logic or condition that is resulting in the dead code.

If the dead code is present in an IP and you do not want to report within the IP, specify the *ip_block* constraint for the IP.

### Message 3

The following message appears if optimization due to synthesis results in the dead code:

```
[DEADCODE] [WARNING] Dead code exists as condition is always
false (reason: synthesis optimized logic) [Hier:<hier-name>]
```

#### *Potential Issues*

This violation appears if your design file contains dead code due to optimization done by synthesis.

#### *Consequences of Not Fixing*

If you do not fix this violation, the design functionality is not as expected.

#### *How to Debug and Fix*

Revisit the RTL.

### Message 4

The following message appears if a function is declared but not instantiated in any module:

```
[DEADFUNC] [WARNING] Function <function-name> declared but not
used in module <module-name>
```

#### *Potential Issues*

Not applicable.

#### *Consequences of Not Fixing*

Such functions are not checked for any dead code violations.

#### *How to Debug and Fix*

To fix this violation, either instantiate the reported function in a module or remove the function declaration.

## Example Code and/or Schematic

### Example 1

Consider the following example:

```
assign sel = 2'b00;
always @(in1 or in2 or sel)
begin
    case(sel)
    2'b00: out = in1[0];
    2'b01: out = in1[1];      //dead code reported
    2'b10: out = in2[0];      //dead code reported
    default: out = in2[1];      //dead code reported
    endcase
end
```

In the above example, `sel` is assigned the value `2'b00` before the `case` construct is executed.

Now, during the execution of the `case` construct, the conditions in which the value of `sel` is `2'b01` and `2'b10` are never reached as `sel` is already assigned the value `2'b00`.

As a result, the lines highlighted in red in the above code are never executed. These lines are therefore considered as dead code and such situation is reported by the *Av_deadcode01* rule.

**NOTE:** *For a particular select pin, this rule reports only one message.*

Consider another example. If the dead code is due to non-static nets, schematic from enabling condition of the dead code block until the first RTL net in the fan-in cone is displayed as shown below:

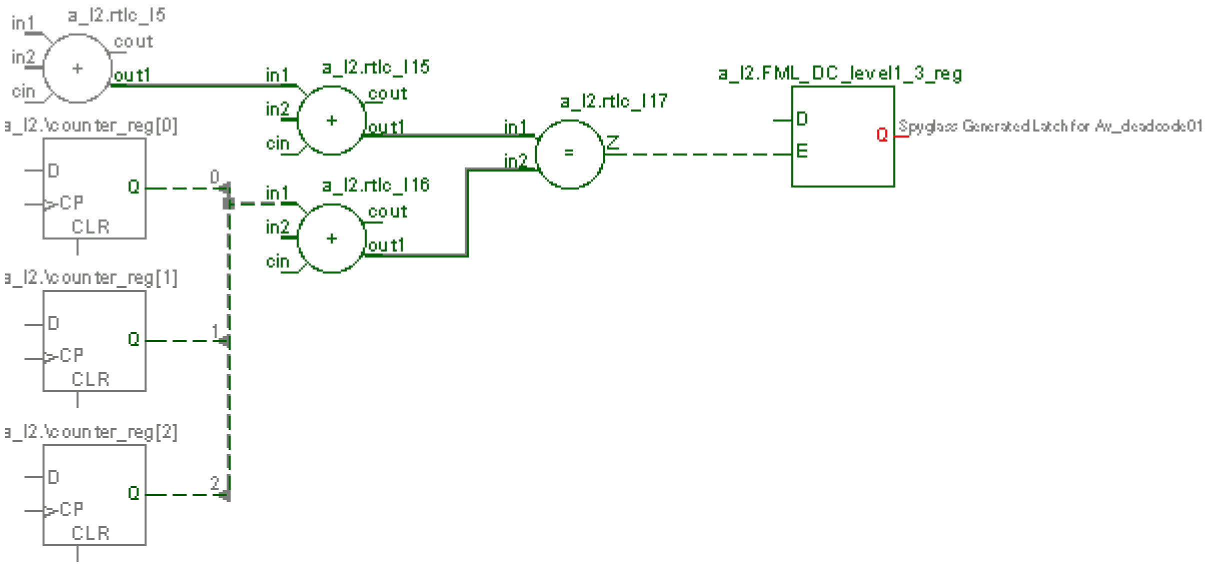


**FIGURE 19.** Schematic of the Av_deadcode01 rule violation

The waveform, as shown below, is displayed if the *fv_debug_sim_cycles* parameter is specified:

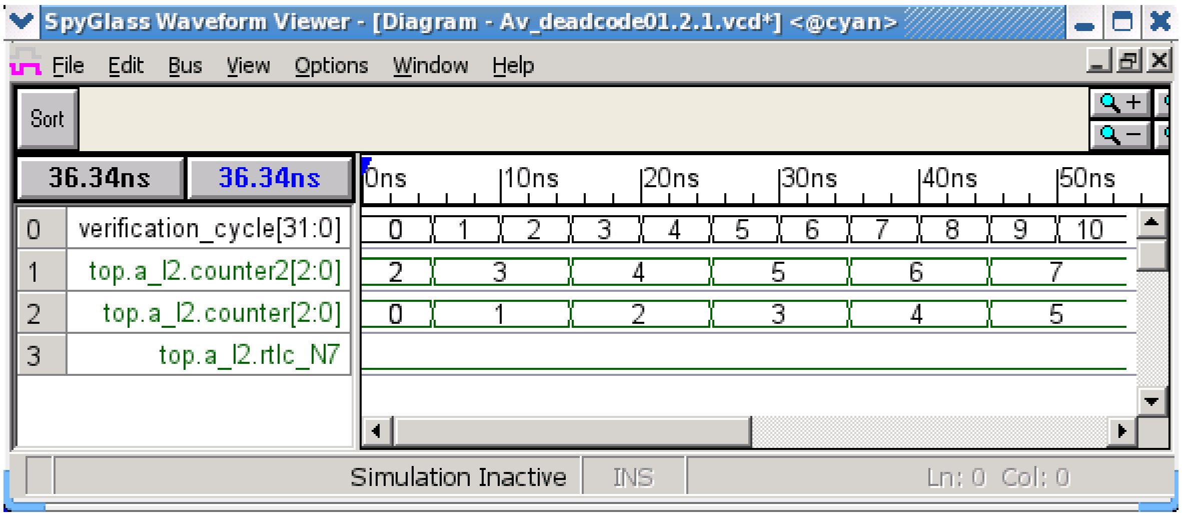

SpyGlass Waveform Viewer - [Diagram - Av_deadcode01.2.1.vcd*] <@cyan>
File Edit Bus View Options Window Help
Sort
36.34ns 36.34ns
0ns 10ns 20ns 30ns 40ns 50ns
0 verification_cycle[31:0]
1 top.a_l2.counter2[2:0]
2 top.a_l2.counter[2:0]
3 top.a_l2.rtlc_N7
0 1 2 3 4 5 6 7 8 9 10
2 3 4 5 6 7
0 1 2 3 4 5
Simulation Inactive INS Ln: 0 Col: 0

**FIGURE 20.** Waveform of the Av_deadcode01 rule violation

### Example 2

Consider the following example:

```
module top(in1, in2, cond, q);
  input in1, in2,  cond;
  output reg q;
  always
    if(cond == cond) // Always true
      q = in1;
    else
      q = in2 ;      // dead code
endmodule
```

In the above example, the expression of the `if` condition (`cond == cond`) is always true. Therefore, the `else` block will never get executed. This is reported as dead code.

### Example 3

Consider the following example:

```
always@(posedge clk or posedge rst or posedge set)
     if(rst)
              out <= 1'b0;
    else if(set)
             out <= 1'b1;
    else
            out <= in;
```

In the above example, the highlighted portion is reported as dead code. To remove this violation, set the *av_force_soft_reset* parameter to `Av_setreset01,Av_deadcode01`.

## Default Severity Label

Warning

## Rule Group

Implicit Properties

## Report and Related Files

- OverConstrainInfo: This file contains details of conflicting constraints. For details, see *Overconstrain Info File*.
- *Auto Verify Central Report*

## Av_dontcare01

**Reports sensitizable X-assignments in the design.**

### When to Use

Use this rule to understand the conditions under which X-assignments are reachable.

#### Prerequisites

Specify clock signals by using the *clock* constraint.

### Description

The *Av_dontcare01* rule reports X assignments that are reachable.

### Parameter(s)

- *xassign_casedefault*: The default value is no. Set this parameter to yes to check for X-assignments inside the `default` clause of the `case` statement.
- *include_construct*: The default value is `none`. Set this parameter to `generate` to check the `generate_block` constructs. Other possible values are `always_comb` and `none`.
- *av_dump_assertions*: The default value is "". Set this parameter to `sva` to generate SystemVerilog Assertions (SVA).

### Constraint(s)

- *clock* (Mandatory): Use this constraint to specify clock signals in a design.
- *formal_analysis_filter* (Optional): Use this constraint to specify the modules or hierarchies on which formal analysis should be ignored or performed.
- *reset* (Optional): Use this constraint to specify reset signals in a design.
- *set_case_analysis* (Optional): Use this constraint to specify case-analysis conditions.
- *breakpoint* (Optional): Use this constraint to specify breakpoints in a design where functional analysis should stop.

- *watchpoint* (Optional): Use this constraint to generate a waveform for an internal signal.
- *ip_block* (Optional): Use this constraint to specify IP blocks in your design.

## Messages and Suggested Fix

The following message appears if your design contains X-assignments that are reachable:

```
[WARNING] X assignment may be executed
```

In addition, if a set of conflicting constraints are specified for the design, additional message is appended to the above message string. For details on the message and how to debug and fix this conflict, see the *Overconstrain Info File* section.

### *Potential Issues*

This violation appears when the RHS of an assignment that has a static value containing X gets executed.

### *Consequences of Not Fixing*

If you do not fix this violation, the simulation results over pre synthesis and post synthesis netlist may mismatch because simulators treat X as literal X whereas synthesis engines optimize it to logical 0 or 1.

If the reported X assignment is present in an IP and you do not want to report within the IP, specify the *ip_block* constraint for the IP.

## Example Code and/or Schematic

Consider the following example:

```
always@( sel1) begin
  if(sel1 == 2'b10 || sel1 == 2'b01)
     out1 <= in1 ;
  else
     out1 <= 1'bx ;
end
```

In the above example, the user may assume that the X assignment in the above code is not reachable. As a result, the user may consider that the `sel1` signal can attain the values 1 and 2 only.

However, since the X assignment is reachable in this case, `sel1` can also toggle to the values 0, 1, 2, and 3. Therefore, this rule reports a violation at the `X` assignment.

## Default Severity Label

Warning

## Rule Group

Implicit-Properties

## Report and Related Files

- Av_dontcare01.<ID>.OverConstrainInfo: This file contains details of conflicting constraints. For details, see *Overconstrain Info File*.
- *Auto Verify Central Report*

## Av_fsm_analysis

**Reports FSM related issues in the design**

## When to Use

Use this rule to detect FSM issues in a design.

## Description

The *Av_fsm_analysis* rule reports the following FSM issues in a design:

- *Unreachable State of an FSM*
- *Deadlocked State of an FSM*
- *Dead Transition of an FSM*
- *Live Locks in an FSM*

### Unreachable State of an FSM

An unreachable state of an FSM can be one of the following states in the RTL code:

- There are no transitions to reach to that FSM state.
- There are transitions that cannot be exercised by the logic controlling that FSM state.

In *Figure 21*, the nodes in red form unreachable states.

### Deadlocked State of an FSM

A deadlocked state of an FSM can be any of the following reachable states:

- There are no out going transitions.
- The out going transitions cannot be exercised due to a control logic.

When a state machine reaches the deadlocked state, it cannot transition to a different state.

In *Figure 21*, the node in grey form a deadlocked state.

### Dead Transition of an FSM

It refers to a state machine transition that is present in the RTL code but cannot be exercised.

Dead transitions may result in *Unreachable State of an FSM* or *Deadlocked State of an FSM*.

### Live Locks in an FSM

A group of reachable FSM states creates a live-lock scenario when these states create a loop for which the both the following conditions hold true:

- The loop size is smaller than the size of total reachable states in an FSM.

  A loop size is the number of states involved in a live lock. For example, in *Figure 21*, the loop size is 4.
- All the outgoing edges of the loop are dead.

In *Figure 21*, the nodes in blue form a live lock state.

## Parameter(s)

*av_dump_liveness*: The default value is `assert`. Set this parameter to `cover` to generate the SystemVerilog Assertions (SVA) in terms of *cover*.

## Constraint(s)

- *clock* (Mandatory): Use this constraint to specify clock signals in a design.
- *formal_analysis_filter* (Optional): Use this constraint to specify the modules or hierarchies on which formal analysis should be ignored or performed.
- *fsm* (Optional): Use this constraint to specify FSM details in a design.
- *reset* (Optional): Use this constraint to specify reset signals in a design.
- *set_case_analysis* (Optional): Use this constraint to specify case-analysis conditions.
- *breakpoint* (Optional): Use this constraint to specify breakpoints in a design where functional analysis should stop.
- *watchpoint* (Optional): Use this constraint to generate a waveform for an internal signal.
- *ip_block* (Optional): Use this constraint to specify IP blocks in your design.

## Messages and Suggested Fix

### Message 1

This rule reports the following message:

```
[WARNING] FSM '<FSM-name>' has <num-livelocks> Livelocks,
<num-unreachable-states> Unreachable States,
<num-deadlock-states> Deadlock States and <dead-transitions>
Dead Transitions
```

#### *Potential Issues*

This violation appears if the design contains the following issues:

- *Unreachable State of an FSM*
- *Deadlocked State of an FSM*
- *Dead Transition of an FSM*
- *Live Locks in an FSM*

#### *Consequences of Not Fixing*

If you do not fix this violation, the design may have redundant or incorrect functionality.

#### *How to Debug and Fix*

To debug and fix this violation:

- Analyze the FSM in *FSM Viewer*.
- Analyze the expression containing fan-in cone nets, which would have triggered the transition.
- Perform appropriate actions based on the following conditions:
  - ❒ **Condition:** No transition occurs to an unreachable state or no transition occurs from a deadlocked state of the FSM.

    **Action:** Modify the design to introduce a transition to an unreachable state or from a deadlocked state.
  - ❒ **Condition:** Transition occurs.

    **Action:** Analyze the RTL to determine the cause of the dead transition.

If the FSM that is unreachable or deadlocked is present in an IP and you do not want to report within the IP, specify the *ip_block* constraint for the IP.

### Message 2

This rule reports the following message:

```
[WARNING] FSM '<FSM-name>' not analyzed (Reason:
Constraint-Conflict).
```

#### *Potential Issues*

Not applicable

#### *Consequences of Not Fixing*

If you do not fix this violation, SpyGlass run does not proceed further.

#### *How to Debug and Fix*

Open the file (*Overconstrain Info File*) pointed by the message of this rule and check for the conflicting constraints.

## Example Code and/or Schematic

Consider the following files (*test.v* and *constr.sgdc*) specified for SpyGlass analysis:

**test.v**

```
`define S0  5'b00000
`define S1  5'b00001
`define S2  5'b00010
`define S3  5'b00011
`define S4  5'b00100
`define S5  5'b00101
`define S6  5'b00110
`define S7  5'b00111
`define S8  5'b01000
```

```
`define S9  5'b01001
module Fsm(input rst, clk, in, en1, en2, en3, en4, output reg
out);
reg [15:0] counter ;
reg [4:0] state;
always @(posedge clk or posedge rst) begin
    if(rst) begin
        counter <= 0;
        out <= 1'b0;
    end
    else begin
        counter <= counter + 1;
        out<= state[3] & in;
    end
end
always @(posedge clk or posedge rst) begin
    if(rst) begin
        state <= `S0 ;
    end
    else begin
        case(state)
            `S0:
                if(en1) begin
                    state <= `S1;
                end
                else if(en2) begin
                    state <= `S2;
                end
                else begin
                    state <= `S3;
                end
            `S1:
                if(counter >=16'b0011111111111111) begin
                    state <= `S4;
                end
            `S2:
                    state<=`S5;
            `S3:
                if(en3) begin
                    state<=`S5;
                end
            `S5:
                if(en3 == 0) begin
                    state<=`S6;
                end
                else if(en3 == 1) begin
                    state <= `S9;
                end
                else begin
                    state<=`S0;
                end
            `S6:
                state<=`S7;
            `S7:
                state<=`S8;
            `S8:
                if(en3) begin
                    state<=`S6;
                end
            `S9:
                if(en1 || en2) begin
                state <=`S4;
                end
        endcase
    end
end
//assume property(@(posedge clk) (state == `S5 |-> (en3 |->
en4)));
endmodule
```

**constr.sgdc**

```
current_design Fsm
clock -name Fsm.clk -period 10
reset -name Fsm.rst -value 1
set_case_analysis -name Fsm.en1 -value 0
set_case_analysis -name Fsm.en2 -value 0
```

After running SpyGlass with the *test.v* and *constr.sgdc* files, the following *FSM Viewer* is generated by the *Av_fsm_analysis* rule:

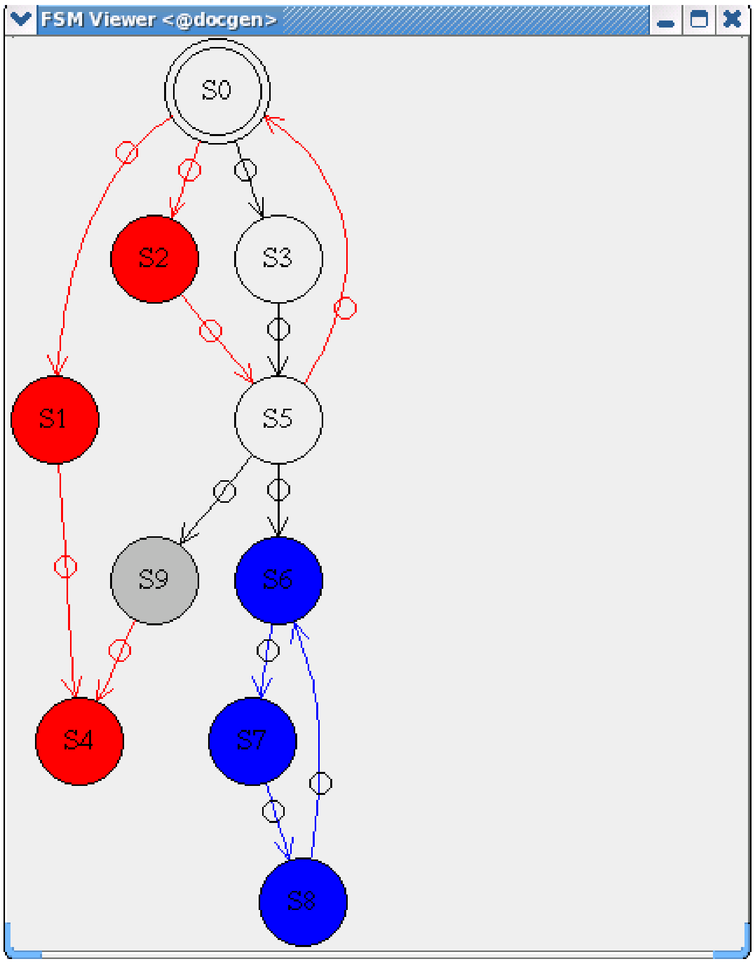


**FIGURE 21.** FSM generated by the Av_fsm_analysis rule

Based on the above FSM, the *Av_fsm_analysis* rule generates the spreadsheet described in the *Reports and Related Files* section.

## Default Severity Label

Warning

## Rule Group

Implicit-Properties

## Reports and Related Files

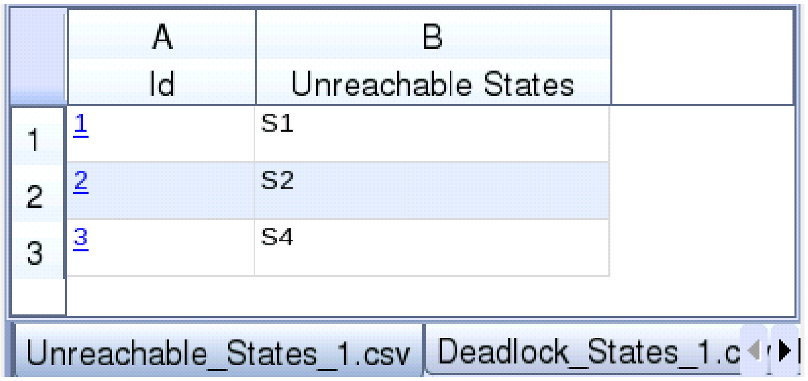
The *Av_fsm_analysis* generates a consolidated spreadsheet showing details of different types of FSM issues under the following tabs:

- *The Unreachable_States Tab*
- *Deadlock_States Tab*
- *The Livelock_States Tab*
- *The Dead_Transitions Tab*

### The Unreachable_States Tab

Each row under this tab shows the name of one unreachable state.

For example, based on the *FSM Viewer* shown in *Figure 21*, the following figure shows the contents under this tab:



| | A<br>Id | B<br>Unreachable States |
|---|---|---|
| 1 | 1 | S1 |
| 2 | 2 | S2 |
| 3 | 3 | S4 |

**FIGURE 22.** The Unreachable_States Tab

In the above spreadsheet, the state with the minimal depth from the initial state is displayed first. The states with the same depth can appear in any order.

For example, based on the *FSM Viewer* in *Figure 21*, S1 and S2 can appear in any order as they are at the same depth. However, S1 or S2 should come before S4.

On clicking an unreachable state in the above spreadsheet:

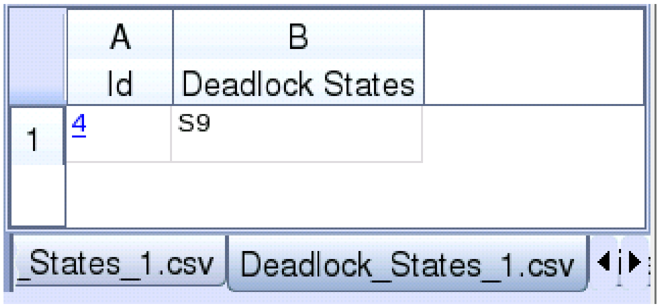
- The corresponding state is highlighted in the *FSM Viewer*.
- The corresponding FSM state net is highlighted in the schematic.

**NOTE:** *A state that is reported as unreachable is not reported as a deadLock state.*

### Deadlock_States Tab

Each row under this tab shows the name of one dead state.

For example, based on the *FSM Viewer* shown in *Figure 21*, the following figure shows the contents under this tab:



|   | A | B |
|---|---|---|
|   | Id | Deadlock States |
| 1 | 4 | S9 |

**FIGURE 23.** The Dead_States Tab

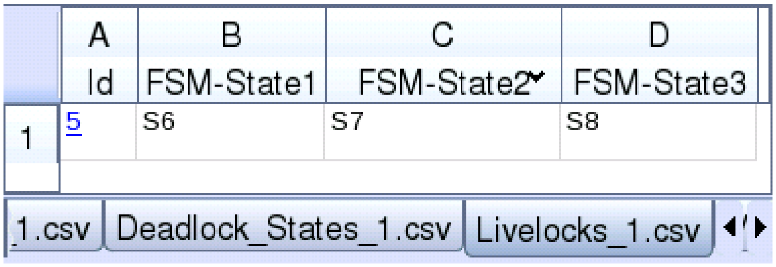
On clicking a deadlock state in the above spreadsheet:

- The corresponding state is highlighted in the *FSM Viewer*.
- The corresponding FSM state net is highlighted in the schematic.

### The Livelock_States Tab

Each row under this tab shows one loop in a livelock state.

For example, based on the *FSM Viewer* shown in *Figure 21* (the green node in this figure are in a livelock state), the following figure shows the contents under this tab:



| | A | B | C | D |
|---|---|---|---|---|
| | Id | FSM-State1 | FSM-State2 | FSM-State3 |
| 1 | 5 | S6 | S7 | S8 |

1.csv | Deadlock_States_1.csv | Livelocks_1.csv

**FIGURE 24.**

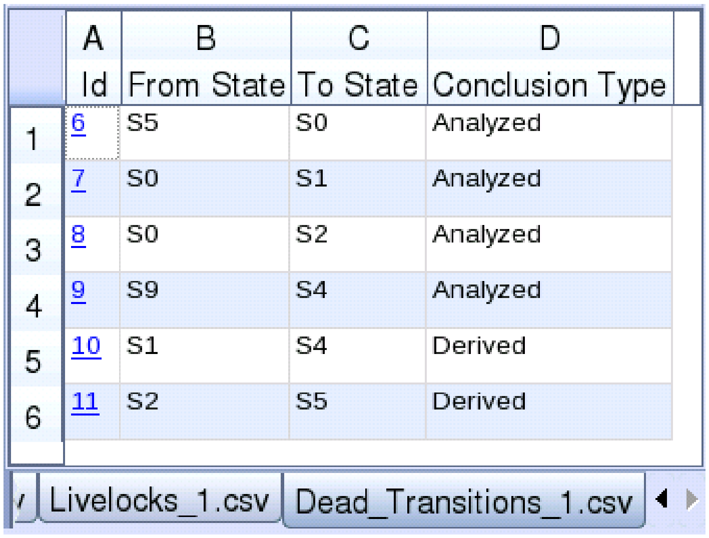
On clicking any state in a row in the above spreadsheet:

- The corresponding loop is highlighted in the *FSM Viewer*.
- The corresponding FSM state net is highlighted in the schematic.

### The Dead_Transitions Tab

The spreadsheet under this tab shows all the dead transitions of an FSM.

For example, based on the *FSM Viewer* shown in *Figure 21*, the following figure shows the contents under this tab:



|  | A | B | C | D |
|---|---|---|---|---|
|  | Id | From State | To State | Conclusion Type |
| 1 | 6 | S5 | S0 | Analyzed |
| 2 | 7 | S0 | S1 | Analyzed |
| 3 | 8 | S0 | S2 | Analyzed |
| 4 | 9 | S9 | S4 | Analyzed |
| 5 | 10 | S1 | S4 | Derived |
| 6 | 11 | S2 | S5 | Derived |

**FIGURE 25.** The All_Dead_Transitions tab

In the above spreadsheet, the transitions marked as *Analyzed* in the *Conclusion Type* column are the dead transitions. However, a *Derived* transition is the transition for which all the incoming transitions to the state in the *From State* column are dead.

On clicking a dead transition in the above spreadsheet:

- The corresponding transition is highlighted in the *FSM Viewer*.
- The corresponding FSM state net is highlighted in the schematic.

## Av_divide_by_zero

**Reports divide / modulo by zero violation**

## When to Use

Use this rule to detect cases where a non-constant signal is used as a divisor.

## Description

The *Av_divide_by_zero* rule reports the division operator (/) or modulo operator (`%`) used in an expression in which a divisor can become zero.

**NOTE:** *This rule is applicable for Verilog designs only.*

### Language

Verilog

## Parameter(s)

- *include_construct*: The default value is `none`. Set this parameter to `generate` to check the `generate_block` constructs. Other possible values are `always_comb` and `none`.
- *av_dump_assertions*: The default value is “”. Set this parameter to `sva` to generate SystemVerilog Assertions (SVA).

## Constraints

- *clock* (Mandatory): Use this constraint to specify clock signals in a design.
- *formal_analysis_filter* (Optional): Use this constraint to specify the modules or hierarchies on which formal analysis should be ignored or performed.
- *reset* (Optional): Use this constraint to specify reset signals in a design.
- *set_case_analysis* (Optional): Use this constraint to specify case-analysis conditions.
- *breakpoint* (Optional): Use this constraint to specify breakpoints in a design where functional analysis should stop.

- *watchpoint* (Optional): Use this constraint to generate a waveform for an internal signal.
- *ip_block* (Optional): Use this constraint to specify IP blocks in your design.

## Messages and Suggested Fix

This rule reports the following message:

**[WARNING]** `<Division | Modulus> by zero in <expression> [Hier: <hierarchy-name>]`

### *Potential Issues*

A non-constant signal is used as a divisor.

### *Consequences of Not Fixing*

If you do not fix this violation, division or modulo by zero can cause chip failure.

### *How to Debug and Fix*

To debug this violation, double-click on the violation and check the following to determine the cause of the violation:

- The rule-violating line highlighted in the *HDL Viewer* pane.
- The schematic.
- The waveform viewer to check the witness signals.

To fix this violation, modify the RTL so that divisor cannot be zero. If multiple drivers that are active simultaneously are in an IP and you do not want to report within the IP, specify the *ip_block* constraint for the IP.

## Example Code and/or Schematic

Consider the following files specified for SpyGlass analysis:

**// test.v**

```
module top(clk, a, b, c, d, e);
 input clk, a, b, c, d;
 output reg e;
 always @ (posedge clk) begin
     e <= c + 1/b;
 end
 always @ (posedge clk) begin
     e = c + 1/b;
 end
endmodule
```

**constr.sgdc**

```
current_design top
clock -name clk -period 10
```

For the above example, the *Av_divide_by_zero* reports the following violations:

```
Division by zero in (1 / b)[Hier:top]

Division by zero in (1 / b)[Hier:top]
```

When you select the first violation and open the *Incremental Schematic*, the violating net along with its fan-in cone till the first level RTL net appears in the schematic. The following figure shows the schematic:

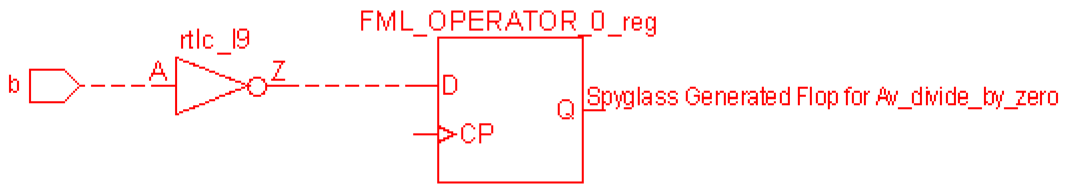


**FIGURE 26.**

To debug the violation, open the waveform. The following figure shows the waveform generated for the first violation:

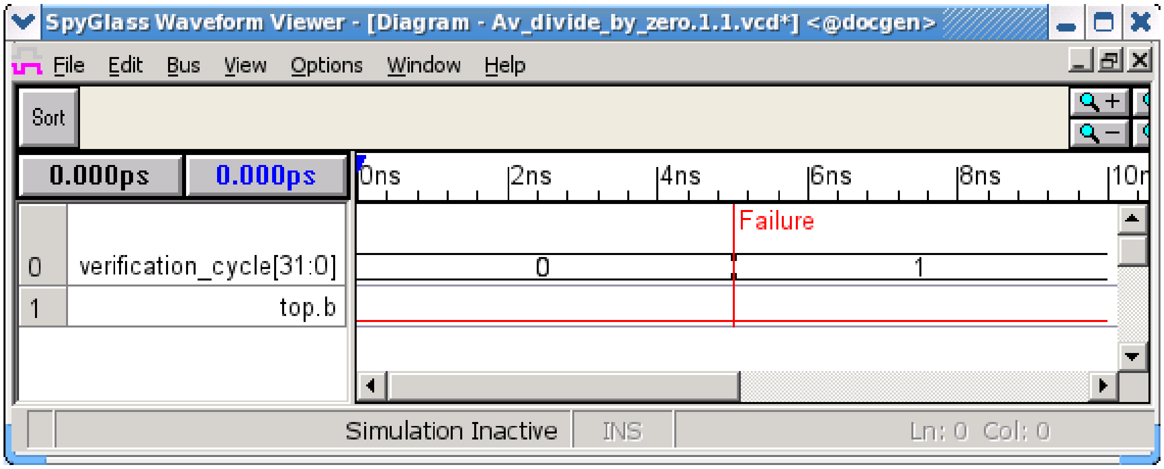

SpyGlass Waveform Viewer - [Diagram - Av_divide_by_zero.1.1.vcd*] <@docgen>
File Edit Bus View Options Window Help
Sort
0.000ps
0.000ps
0ns
2ns
4ns
6ns
8ns
10n
0
verification_cycle[31:0]
1
top.b
Failure
0
1
Simulation Inactive
INS
Ln: 0 Col: 0

**FIGURE 27.**

The waveform shows the direct fan-in of the violating net and the fan-in RTL net. It will have pop-up help and consistency of colors with schematic.

## Default Severity Label

Warning

## Rule Group

Implicit Property

## Reports and Related Files

- Av_divide_by_zero.<ID>.OverConstrainInfo: This file contains details of conflicting constraints. For details, see *Overconstrain Info File*.
- *Auto Verify Central Report*

## Av_negative_shift

**Reports arithmetic shift by negative value violations**

### When to Use

Use this rule check for the presence of a negative value in the RHS of all the arithmetic shift operators.

### Description

The Av_negative_shift rule checks for a negative RHS value in the right-hand side of all the arithmetic shift operators. If a negative value is found, the Av_negative_shift rule reports a violation message for the shift operator.

**NOTE:** *This rule is applicable for Verilog designs only.*

#### Language

Verilog

### Parameter(s)

- *include_construct*: The default value is `none`. Set this parameter to `generate` to check the `generate_block` constructs. Other possible values are `always_comb` and `none`.
- *av_dump_assertions*: The default value is “”. Set this parameter to `sva` to generate SystemVerilog Assertions (SVA).

### Constraints

- *clock* (Mandatory): Use this constraint to specify clock signals in a design.
- *formal_analysis_filter* (Optional): Use this constraint to specify the modules or hierarchies on which formal analysis should be ignored or performed.
- *reset* (Optional): Use this constraint to specify reset signals in a design.
- *set_case_analysis* (Optional): Use this constraint to specify case-analysis conditions.

- *breakpoint* (Optional): Use this constraint to specify breakpoints in a design where functional analysis should stop.
- *watchpoint* (Optional): Use this constraint to generate a waveform for an internal signal.
- *ip_block* (Optional): Use this constraint to specify IP blocks in your design.

## Messages and Suggested Fix

This rule reports the following message:

```
[WARNING] <operator_position> arithmetic shift by negative
value for <value> in <expression>
```

### *Arguments*

- Left or Right, to denote the right arithmetic shift or left arithmetic shift, <operator_position>
- Value of the net at the failing point or failing sequence, <value>
- expression, <expression>

### *Potential Issues*

A negative value exists in the RHS of the arithmetic shift operator

### *Consequences of Not Fixing*

If you do not fix this violation, negative shift value can cause chip failure.

### *How to Debug and Fix*

To debug this violation, double-click on the violation and check the following to determine the cause of the violation:

- The rule-violating line highlighted in the *HDL Viewer* pane.
- The schematic.
- The waveform viewer to check the witness signals.

To fix this violation, modify the RTL so that the RHS of the arithmetic shift operator does not have a negative value. If multiple drivers that are active simultaneously are in an IP and you do not want to report within the IP, specify the *ip_block* constraint for the IP.

## Example Code and/or Schematic

Consider the following files specified for SpyGlass analysis:

```
test.v

module top(clk, a, b, c, d, e);

input clk;
input [7:0] a, b;
output reg signed [7:0] c, d;
output reg [7:0] e;

always @ (posedge clk)
    e <= a || (b >>> ~d) ? c : b <<< ^d;

endmodule
```

For the above example, the *Av_divide_by_zero* reports the following violations:

```
(Hier:top) (b >>> (~d)), test.v, 9, (Av_negative_shift.1.1.vcd)
FAILED through depth 1(1)
```

```
Hier:top) (b <<< (^ d)), test.v, 9, :  PROVED
```

```
Av_negative_shift@@@@Warning@@test.v@@9@@1@@10@@Right
Arithmetic shift by negative value for 'd = 0' in '(b >>>
(~d))'@@0
```

To debug the violation, open the *Auto Verify Report*. This report displays the functional analysis statistics of a design.

## Default Severity Label

Warning

## Rule Group

Implicit Property

## Reports and Related Files

*Auto Verify Report*

## Av_fsm01

**Reports unreachable or deadlocked states of an FSM.**

## When to Use

Use this rule to verify the functionality of an FSM in a design.

### Prerequisites

Specify clock signals by using the *clock* constraint.

## Description

**NOTE:** *This rule will be deprecated in a future SpyGlass release. Use the Av_fsm_analysis rule instead of this rule.*

The *Av_fsm01* rule reports a violation in the following cases:

- *Unreachable State of an FSM*
- *Deadlocked State of an FSM*

The *Av_fsm01* rule works only on the *Finite-State Machines (FSMs)* that identified by SpyGlass Auto Verify solution.

### Unreachable State of an FSM

An unreachable state of an FSM can be one of the following states in the RTL code:

- You have not created any transitions to reach that FSM state.
- You have created transitions that cannot be exercised by the logic controlling that FSM state.

### Deadlocked State of an FSM

A deadlocked state of an FSM can be any of the following states:

- The state from which no out-going transitions exist.
- The state in which the out-going transitions cannot be exercised due to a control logic.

When a state machine reaches the deadlocked state, it cannot transition to a different state.

## Parameter(s)

- *av_dump_assertions*: The default value is “”. Set this parameter to `sva` to generate SystemVerilog Assertions (SVA).
- *av_dump_liveness*: The default value is `assert`. Set this parameter to `cover` to generate the SystemVerilog Assertions (SVA) in terms of *cover*.

## Constraint(s)

- *clock* (Mandatory): Use this constraint to specify clock signals in a design.
- *formal_analysis_filter* (Optional): Use this constraint to specify the modules or hierarchies on which formal analysis should be ignored or performed.
- *reset* (Optional): Use this constraint to specify reset signals in a design.
- *set_case_analysis* (Optional): Use this constraint to specify case-analysis conditions.
- *breakpoint* (Optional): Use this constraint to specify breakpoints in a design where functional analysis should stop.
- *watchpoint* (Optional): Use this constraint to generate a waveform for an internal signal.
- *ip_block* (Optional): Use this constraint to specify IP blocks in your design.

## Messages and Suggested Fix

The following message appears to indicate the FSM that is in the unreachable or the deadlocked state:

```
[WARNING] The state <state-name> is '<Unreachable |
DeadLocked>' for FSM '<fsm-name>'
```

In addition, if a set of conflicting constraints are specified for the design, additional message is appended to the above message string. For details on the message and how to debug and fix this conflict, see the *Overconstrain Info File* section.

#### *Potential Issues*

This violation appears if your design contains an FSM that is unreachable or deadlocked.

For information on these states, see *Unreachable State of an FSM* and *Deadlocked State of an FSM*.

### *Consequences of Not Fixing*

If you do not fix this violation, the design may have redundant or incorrect functionality.

### *How to Debug and Fix*

To debug and fix this violation, analyze the FSM in *FSM Viewer* window.

Perform appropriate actions based on the following conditions:

- No transition occurs to an unreachable state or no transition occurs from a deadlocked state of the FSM

  **Action:** Modify the design to introduce a transition to an unreachable state or from a deadlocked state.
- Transition occurs

  **Action:** Analyze the RTL to determine the cause of the dead transition.

If the FSM that is unreachable or deadlocked is present in an IP and you do not want to report within the IP, specify the *ip_block* constraint for the IP.

## Example Code and/or Schematic

### Example 1 - Unreachable State of an FSM

Consider the following example:

```
// Unreachable FSM states
// FSM1 and FSM2 are interacting FSMs
// FSM2 is launched when FSM1 reaches a specific state.
// But FSM2 has an unreachable state due to
// non-sensitizable transition


`define FSM1S1 4'b1000
`define FSM1S2 4'b0100
`define FSM1S3 4'b0010
```

```
`define FSM1S4 4'b0001

`define FSM2S1 4'b1000
`define FSM2S2 4'b0100
`define FSM2S3 4'b0010
`define FSM2S4 4'b0001

module unreachable(reset, clk1, clk2, ctl, out);
  input  reset, clk1, clk2, ctl;
  output out;
  reg [0:3] fsm1s, fsm2s;

  // FSM1
  always @(posedge clk1) begin
    if(reset)
      fsm1s <= `FSM1S1;
    else
      case(fsm1s)  // synopsys full_case
                   // synthesis parallel_case
        `FSM1S1: fsm1s <= `FSM1S2;
        `FSM1S2: fsm1s <= `FSM1S3;
        `FSM1S3: fsm1s <= `FSM1S4;
        `FSM1S4:
           if(ctl) fsm1s <= `FSM1S1;
      endcase
  end

  // FSM2

  always @(posedge clk2) begin
    if(fsm1s == `FSM1S4) // FSM1 initializes FSM2
      fsm2s <= `FSM2S1;
    else
      case(fsm2s)  // synopsys full_case
                   // synthesis parallel_case
        `FSM2S1:
          if(fsm1s == 4'b1100)
```

```
            fsm2s <= `FSM2S2; // Condition always false
          else
            fsm2s <= `FSM2S3;
        `FSM2S2: fsm2s <= `FSM1S3;
        `FSM2S3: fsm2s <= `FSM1S4;
        `FSM2S4:
           if(ctl) fsm2s <= `FSM1S1;
      endcase
  end
  assign out = fsm2s[2];
endmodule
```

In the above example, the state `4'b0010` is unreachable or the FSM `fsm2s`. Therefore, the *Av_fsm01* rule reports a violation in this case.

The following figure shows the FSM viewer generated in this case:

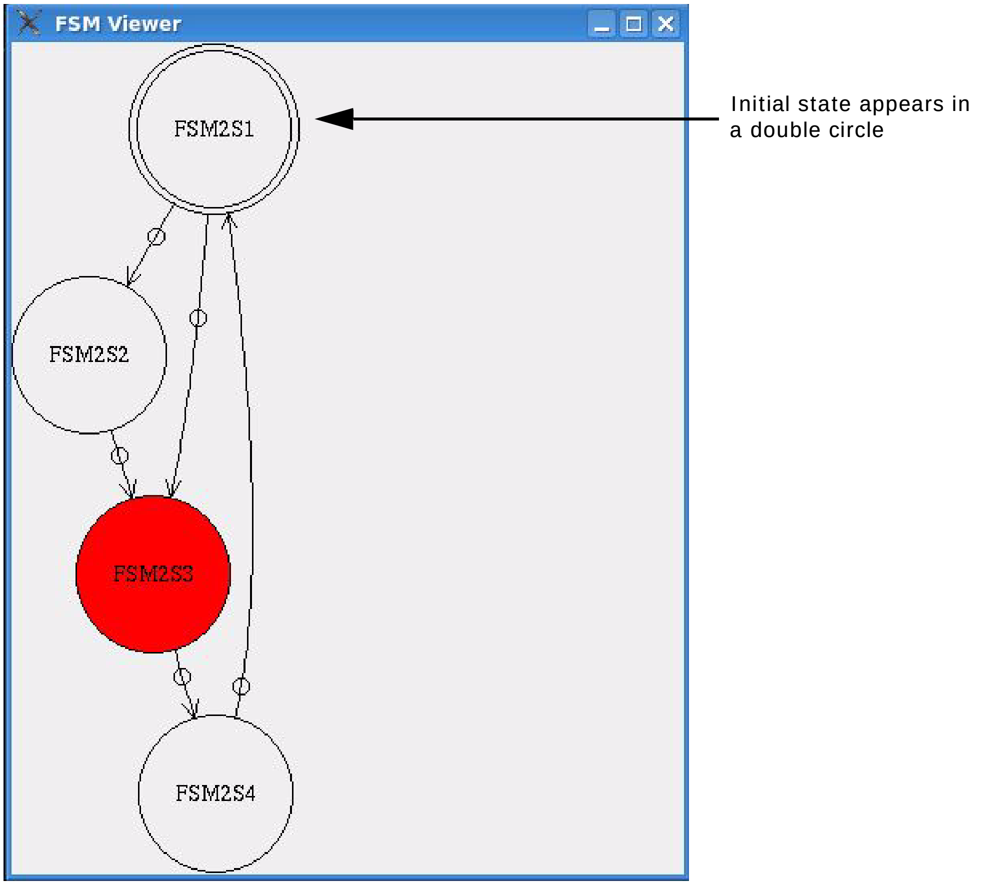


**FIGURE 28.** Unreachable State of an FSM

To fix this violation, modify the RTL to introduce a transition to the reported unreachable state.

See also *Viewing Conditional Expression of a Transition in the FSM Viewer*.

### Example 2 - Deadlocked State of an FSM

Consider the following example:

```
`define S0 2'b00
`define S1 2'b01
```

```
`define S2 2'b10
`define S3 2'b11

module Fsm(clk, ctl, rst);
  input clk, ctl, rst;
  reg [1:0] state;
  always@(posedge clk or negedge rst)
  begin
  if(!rst)
      state <= `S0;
  else
    case(state) // synopsys full_case parallel_case
    `S0 : state <=  `S1;
    `S1 : state <=  `S2;
    `S2 : if (ctl) state <= `S3;
    `S3 : if (ctl & !ctl) state <=  `S1;
          else state <=  `S3;
   endcase
  end
endmodule
```

In the above example, the state machine reaches `STATE3` and is deadlocked. Therefore, the *Av_fsm01* rule reports a violation in this case.

The following figure shows the FSM viewer generated in this case:

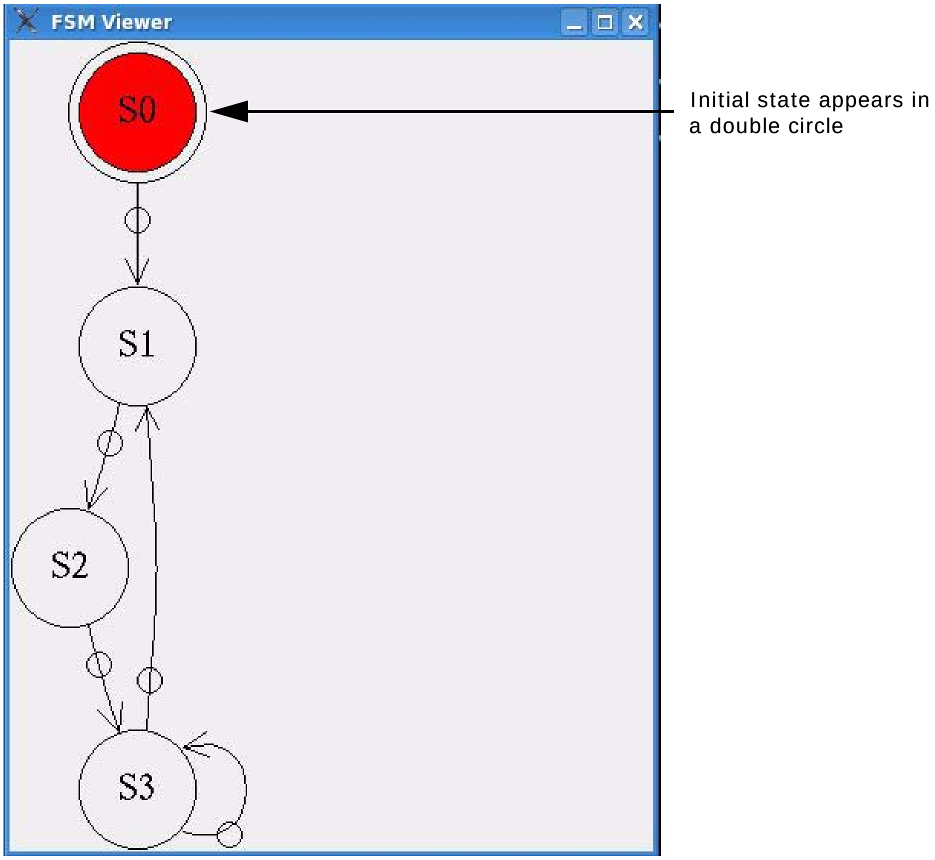


**FIGURE 29.** Deadlocked State of an FSM

In the above example, analyze the RTL to determine the cause of the deadlocked state and modify the RTL accordingly.

See also *Viewing Conditional Expression of a Transition in the FSM Viewer*.

## Default Severity Label

Warning

## Rule Group

Implicit-Properties

## Report and Related Files

- Av_fsm01.<ID>.OverConstrainInfo: This file contains details of conflicting constraints. For details, see *Overconstrain Info File*.
- *Auto Verify Central Report*

## Av_fsm02

**Reports the dead transition of an FSM.**

## When to Use

Use this rule to verify the functionality of FSM in the design.

### Prerequisites

Specify clock signals by using the *clock* constraint.

## Description

**NOTE:** *This rule will be deprecated in a future SpyGlass release. Use the Av_fsm_analysis rule instead of this rule.*

The *Av_fsm02* rule reports the *Dead Transition of an FSM*.

### Dead Transition of an FSM

It refers to a state machine transition that is present in the RTL code but cannot be exercised.

Dead transitions may result in *Unreachable State of an FSM* or *Deadlocked State of an FSM*.

Each transition of an FSM in the design is separately checked using formal verification techniques to determine whether the transition is dead.

### Unreachable State of an FSM

An unreachable state of an FSM can be one of the following states in the RTL code:

- You have not created any transitions to reach that FSM state.
- You have created transitions that cannot be exercised by the logic controlling that FSM state.

### Deadlocked State of an FSM

A deadlocked state of an FSM can be any of the following states:

- The state from which no out-going transitions exist.

- The state in which the out-going transitions cannot be exercised due to a control logic.

When a state machine reaches the deadlocked state, it cannot transition to a different state.

### Rule Exceptions

In case of multiple transitions, SpyGlass first merges the transitions before running this rule. Consequently, this rule does not report an inactive transition from state A to state B if there are other active transitions from the same state A into state B.

## Parameter(s)

- *av_dump_assertions*: The default value is “”. Set this parameter to `sva` to generate SystemVerilog Assertions (SVA).
- *av_dump_liveness*: The default value is `assert`. Set this parameter to `cover` to generate the SystemVerilog Assertions (SVA) in terms of *cover*.

## Constraints

- *clock* (Mandatory): Use this constraint to specify clock signals in a design.
- *formal_analysis_filter* (Optional): Use this constraint to specify the modules or hierarchies on which formal analysis should be ignored or performed.
- *reset* (Optional): Use this constraint to specify reset signals in a design.
- *set_case_analysis* (Optional): Use this constraint to specify case-analysis conditions.
- *breakpoint* (Optional): Use this constraint to specify breakpoints in a design where functional analysis should stop.
- *watchpoint* (Optional): Use this constraint to generate a waveform for an internal signal.
- *ip_block* (Optional): Use this constraint to specify IP blocks in your design.

## Messages and Suggested Fix

The following message appears to report a dead transitions from the state <*state1-name*> to the state <*state2-name*> of the FSM <*fsm-name*>:

```
[WARNING] FSM ‘<fsm-name>’ has a dead transition ‘<state1-name>
=> <state2-name>’
```

In addition, if a set of conflicting constraints are specified for the design, additional message is appended to the above message string. For details on the message and how to debug and fix this conflict, see the *Overconstrain Info File* section.

### *Potential Issues*

This violation appears if your design contains an FSM that is in a dead transition state. For details, see *Dead Transition of an FSM*.

### *Consequences of Not Fixing*

If you do not fix this violation, the following issues may appear in the design:

- *Unreachable State of an FSM*
- *Deadlocked State of an FSM*

This may result in redundant or incorrect functionality of the design.

### *How to Debug and Fix*

To debug and fix this violation:

- Analyze the FSM in *FSM Viewer*.
- Analyze the expression containing fan-in cone nets, which would have triggered this transition.

If the FSM that is in a dead transition state is present in an IP and you do not want to report within the IP, specify the *ip_block* constraint for the IP.

## Example Code and/or Schematic

Consider the following example:

```
`define S0 2'b00
`define S1 2'b01
`define S2 2'b10
`define S3 2'b11

module test(clk, rst, state);
  input clk, rst;
  output [1:0] state;
  reg [1:0] state;
  reg a, b ;
  wire int_clk;

  assign int_clk = !clk;
  always@(posedge int_clk)
      b <= 0;
  always@(posedge clk)
      a <= 1;
  always@(posedge clk)
  if(rst) state <= `S0;
  else begin
    case(state)
    `S0 : state <= `S1;
    `S1 : if (a) state <= `S2;
    `S2 : if (!b) state <= `S3;
    `S3 : begin
         if (a & b) state <= `S1;
         if (a & !b) state <= `S0;
         end
    default     : ;
  endcase
  end
endmodule
```

In the above example, the transition from STATE3 to STATE1 is a dead transition.

The signal a is always tied high whereas the signal b is always tied low.

Therefore, the conditional expression is `(a && !b)` on which the transition from `S3` to `S1` can occur. But as the condition is always false, this rule reports the transition from `S3` to `S1` as the dead transition.

The following figure shows the FSM viewer in this case:

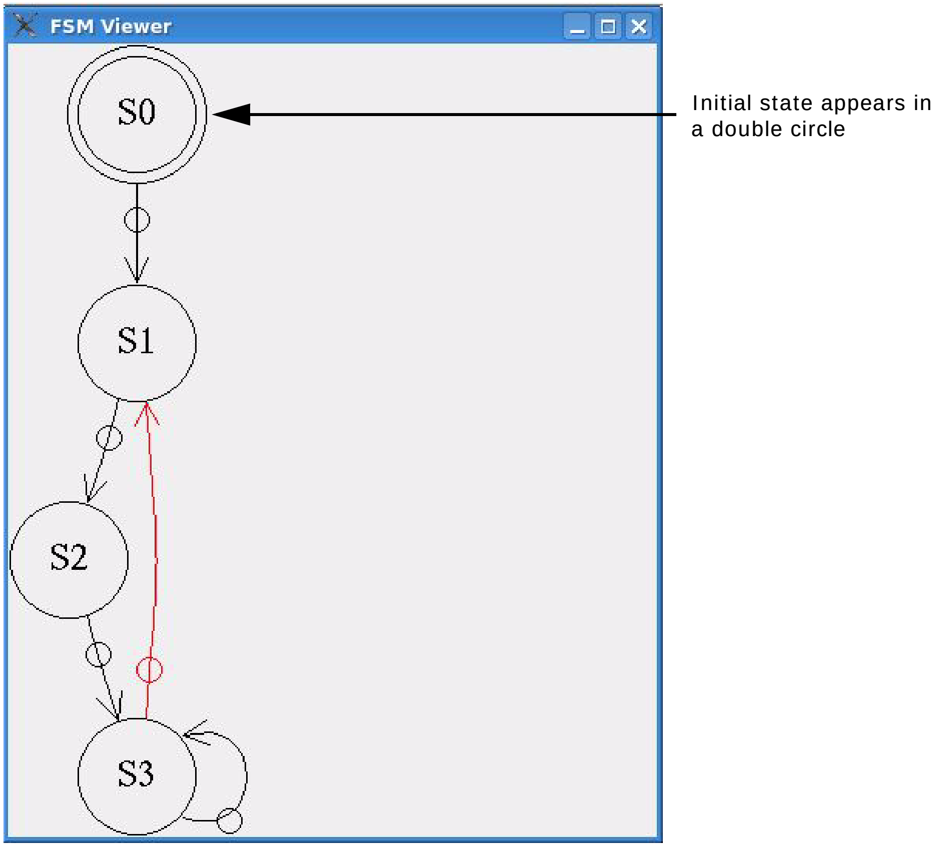


**FIGURE 30.** Dead transition of an FSM

To fix the above violation, correct the fan-in cone nets in the condition expression or the expression itself to fix the dead state transition.

See also *Viewing Conditional Expression of a Transition in the FSM Viewer*.

## Default Severity Label

Warning

## Rule Group

Implicit-Properties

## Report and Related Files

- Av_fsm02.<ID>.OverConstrainInfo: This file contains details of conflicting constraints. For details, see *Overconstrain Info File*.
- *Auto Verify Central Report*

## Av_range01

**Reports array bound violation.**

## When to Use

Use this rule to detect the arrays in the RTL that are accessed out of the range assigned to them.

### Prerequisites

Specify clock signals in a design by using the *clock* constraint.

## Description

The *Av_range01* rule reports arrays that can be accessed with an index that is outside the range of the array.

**NOTE:** *The Av_range01 rule is applicable for Verilog designs only.*

## Parameter(s)

- *include_construct*: The default value is `none`. Set this parameter to `generate` to check the `generate_block` constructs. Other possible values are `always_comb` and `none`.
- *av_dump_assertions*: The default value is “”. Set this parameter to `sva` to generate SystemVerilog Assertions (SVA).

## Constraints

- *clock* (Mandatory): Use this constraint to specify clock signals in a design.
- *formal_analysis_filter* (Optional): Use this constraint to specify the modules or hierarchies on which formal analysis should be ignored or performed.
- *reset* (Optional): Use this constraint to specify reset signals in a design.
- *set_case_analysis* (Optional): Use this constraint to specify case-analysis conditions.
- *breakpoint* (Optional): Use this constraint to specify breakpoints in a design where functional analysis should stop.

- *watchpoint* (Optional): Use this constraint to generate a waveform for an internal signal.

## Messages and Suggested Fix

This rule reports the following message:

```
[WARNING] Array bound violation observed for <sig-name>=<value>
for dimension <dimension> of variable <var-name> where allowed
range is <allowed-range> (Hier: <hier>)
```

The details of the arguments of the above violation message are described in the following table:

| **Argument** | **Description** |
|---|---|
| <sig-name> | Specifies the name of the signal |
| <value> | Specifies the value of the signal <sig-name> for which the array bound violation occurs |
| <dimension> | Specifies the dimension of the variable |
| <var-name> | Specifies the variable name |
| <allowed-range> | Specified the allowed range of the variable |
| <hier> | Specifies the hierarchical name of the module in which the violation occurred. |

In addition, if a set of conflicting constraints are specified for the design, additional message is appended to the above message string. For details on the message and how to debug and fix this conflict, see the *Overconstrain Info File* section.

### *Potential Issues*

This violation appears if your design file contains an array that is accessed with an index outside the range of the array.

### *Consequences of Not Fixing*

If you do not fix this violation, improperly designed index logic can go out of bound causing chip failure.

#### *How to Debug and Fix*

If the reported array is present in an IP and you do not want to report within the IP, specify the *ip_block* constraint for the IP.

### Example Code and/or Schematic

Consider the following example:

```
module test (out1, out2, in1, in2, clk, reset);
  output  out1;
  output [2:0] out2;
  input clk, reset;
  input [2:0] in1, in2;

  reg [2:0] out2;
  reg [1:0] count;

  assign out1=in1[count+1]; // Message: 2+1 > [2:0]

  always @(posedge clk)
    begin
      if(reset == 0) count= 0;
      else if(count == 2'b10) count = 0;
      else count=count+1;
    end

   always @(posedge clk)
    out2[count+1]=in2[count]; //Message: 2+1 > [2:0]

endmodule
```

In the above example, `count` can toggle in the values 0, 1, or 2.

Now when the value 2 is reached, `count +1` results in an array access of `index=3` in the `in1` and `out2` arrays.

In this case, you need to allow `count` to toggle values between 0 and 1.

## Default Severity Label

Warning

## Rule Group

Implicit-Properties

## Report and Related Files

- Av_range01.<ID>.OverConstrainInfo: This file contains details of conflicting constraints. For details, see *Overconstrain Info File*.
- *Auto Verify Central Report*

## Av_setreset01

**Reports flip-flop with simultaneous active asynchronous set and asynchronous reset**

### When to Use

Use this rule to check if your design contains flip-flops with both reset and set.

#### Prerequisites

Specify clock signals in the design by using the *clock* constraint.

### Description

The *Av_setreset01* rule reports a violation when flip-flops with both asynchronous set and asynchronous reset are asserted at the same time.

For example, consider the following figure:

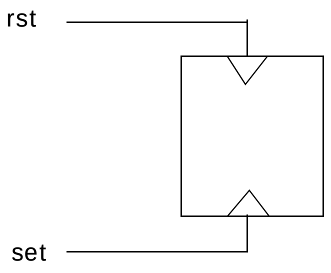


**FIGURE 31.** Example of the Av_setreset01 violation

During optimization, some tools assume that reset takes priority over set in the above flip-flop resulting in possible active reset and set simultaneously. This may result in the following situations in the design:

- When the reset is asserted on a positive edge, the reset or preset may reach first. If the preset reaches first, a glitch may be generated at the output of the above flip-flop.
- When reset is de-asserted followed by set de-asserted, the reset or set may reach first. In this case, the flip-flop will come out of the reset in state "1" (instead of "0") since preset is deasserted last.

The following figure shows the waveform pertaining to the above cases:

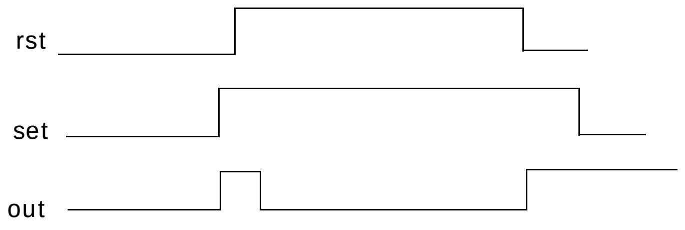


**FIGURE 32.** Waveform of the Av_setreset01 violation

## Parameter(s)

*av_force_soft_reset*: The default value is `Av_setreset01`. Set this parameter to `no` to consider a reset as a hard reset.

## Constraint(s)

- *clock* (Mandatory): Use this constraint to specify clock signals in a design.
- *formal_analysis_filter* (Optional): Use this constraint to specify the modules or hierarchies on which formal analysis should be ignored or performed.
- *reset* (Optional): Use this constraint to specify reset signals in a design.
- *set_case_analysis* (Optional): Use this constraint to specify case-analysis conditions.
- *breakpoint* (Optional): Use this constraint to specify breakpoints in a design where functional analysis should stop.
- *watchpoint* (Optional): Use this constraint to generate a waveform for an internal signal.
- *ip_block* (Optional): Use this constraint to specify IP blocks in your design.

## Messages and Suggested Fix

The following message appears to report flip-flops that are asserted with both asynchronous set and asynchronous reset at the same time

```
[WARNING] Flop <flop-name> detected with simultaneously active
```

`set and reset`

In addition, if a set of conflicting constraints are specified for the design, additional message is appended to the above message string. For details on the message and how to debug and fix this conflict, see the *Overconstrain Info File* section.

### *Potential Issues*

This violation appears if your design contains flip-flops that are asserted with a set and reset simultaneously.

### *Consequences of Not Fixing*

If you do not fix this violation, there may be glitches in the design during set or reset activation.

### *How to Debug and Fix*

To fix this violation, perform the following steps:

1. Double-click on the violation and open the schematic.

   The schematic shows the set and reset which are active at the same time and trigger the reported flip-flop.
2. Open the *Waveform Viewer* window to view the condition when set and reset are active simultaneously.

If the reported flip-flops are in an IP and you do not want to report within the IP, specify the *ip_block* constraint for the IP.

## Example Code and/or Schematic

Consider the following example:

```
//test.v
module test(input D, rst, prst, clk, output reg out);
always @(posedge clk or negedge prst or posedge rst)
    begin
      if (rst)
        out <=  1'b0;
      else if (!prst)
        out <=  1'b1;
      else
        out <=  D;
     end
endmodule
```

**Project File:**

```
set_parameter user_inferred_clocks yes
```

In the above example, if `rst` and `prst` become active at the same time and `prst` appears first then the out flip-flop will be high first and then it will be driven low as soon as rst goes high. This will create a glitch at the output of the flop.

The following figure shows the schematic of the out flip-flop in this case:

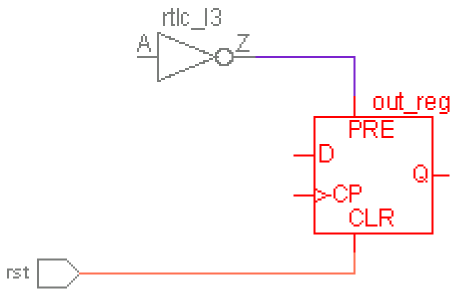


**FIGURE 33.** Schematic of the Av_setreset01 violation

The waveform of the above violation displays the condition when set and reset of the `out` flip-flop are active simultaneously. See the following figure:

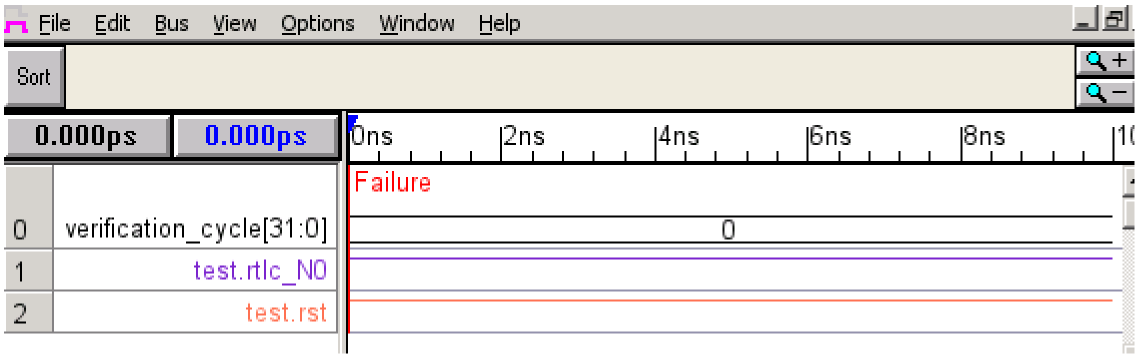


**FIGURE 34.** Waveform of the Av_setreset01 violation

## Default Severity Label

Warning

## Rule Group

Implicit-Properties

## Report and Related Files

- Av_setreset01.<ID>.OverConstrainInfo: This file contains details of conflicting constraints. For details, see *Overconstrain Info File*.
- *Auto Verify Central Report*

## Av_staticreg01

**This rule is deprecated.**

This rule is not being used by SpyGlass Auto Verify.

## Av_staticreg02

**Reports static sequential elements in a design.**

### When to Use

Use this rule to check if sequential elements have any input tied to a constant value.

#### Prerequisites

Specify clock signals by using the *clock* constraint.

### Description

The *Av_staticreg02* rule reports a summary of *Static Sequential Elements* in a design.

This rule identifies sequential elements as static when any of the following conditions hold true after applying case-analysis (by using the *set_case_analysis* constraint) and VDD/VSS (power/ground) propagation in a design:

- Always active reset/clear
- Constant clock
- Inactive load
- Constant data

For library cells, this rule reports an instance as static if any of the data input is static. For multiple clocks, this rule reports an instance as static if any of the clocks is static.

**NOTE:** *This rule does not initialize flip-flops by using resets.*

#### Static Sequential Elements

These are the sequential elements that have at least one of the input tied to a constant value.

### Parameter(s)

*show_static_latches*: The default value is `yes`. Set this parameter to `no` to stop reporting static latches in the spreadsheet.

## Constraint(s)

- *clock* (Mandatory): Use this constraint to specify clock signals in a design.
- *formal_analysis_filter* (Optional): Use this constraint to specify the modules or hierarchies on which formal analysis should be ignored or performed.
- *reset* (Optional): Use this constraint to specify reset signals in a design.
- *set_case_analysis* (Optional): Use this constraint to specify case analysis conditions.
- *breakpoint* (Optional): Use this constraint to specify breakpoints in a design where functional analysis should stop.
- *watchpoint* (Optional): Use this constraint to generate a waveform for an internal signal.

## Messages and Suggested Fix

### Message 1

The following message appears if no sequential elements have any input tied to a constant:

```
[SR2INF] [INFO] Top design unit <top-du-name> : No sequential
elements have any input tied to constant
```

#### *Potential Issues*

This violation appears if your design does not contain any static sequential element.

#### *Consequences of Not Fixing*

Not applicable.

#### *How to Debug and Fix*

Not applicable.

### Message 2

The following warning message appears to report the percentage of sequential elements that have at least one of the inputs tied to a constant:

```
[SR2WRN] [WARNING] Top design unit <top-du-name> :
<seq-elements-percentage> percent of sequential elements have
at least one of the inputs tied to constant
```

#### *Potential Issues*

This violation appears if your design contains sequential elements that have at least one of the inputs tied to a constant value.

#### *Consequences of Not Fixing*

If you do not fix this violation, you may see some unexpected results. For example, if a reset is always on, the data may never be transferred.

#### *How to Debug and Fix*

To fix this violation, remove the condition due to which the sequential elements are becoming static.

## Example Code and/or Schematic

### Example 1

Consider the following Verilog and SGDC files:

```
//test.v
module D_ff(q,d,clk,reset,preset);
output q;
input d,clk,reset,preset;
reg q;
wire w1, w2;
assign w1 = 1 'b1;

assign w2 = d & w1;

 always@(posedge clk or posedge reset or posedge preset)
if(reset)
  q <= 1 'b0;

else if(preset)
  q <= 1 'b1;

else
  q <= w2;
endmodule
```

```
//constraints.sgdc
current_design D_ff
clock -name D_ff.clk -period 10
reset -name D_ff.preset -value 1
```

For the above example, the *Av_staticreg02* rule generates the following schematic:

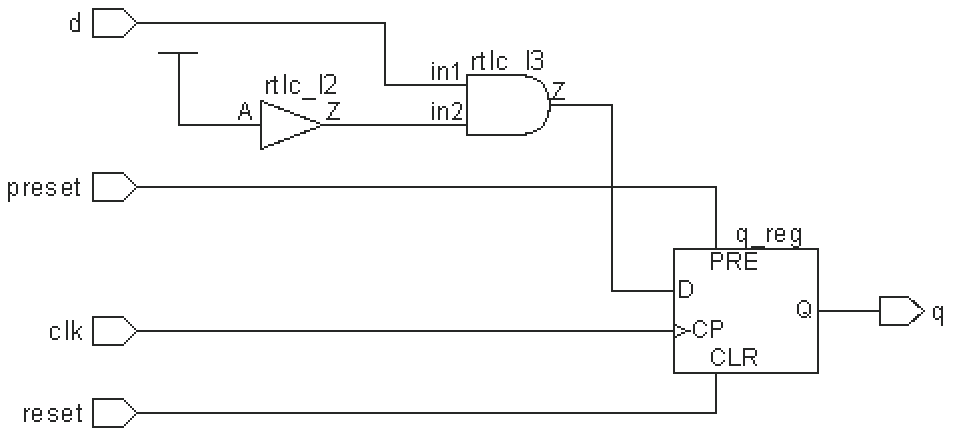


**FIGURE 35.** Example of the Av_staticreg02 violation

In the above example, no pin of the `q_reg` instance is static. Therefore, this rule reports an informational message indicating that there are no static sequential elements in the design.

### Example 2

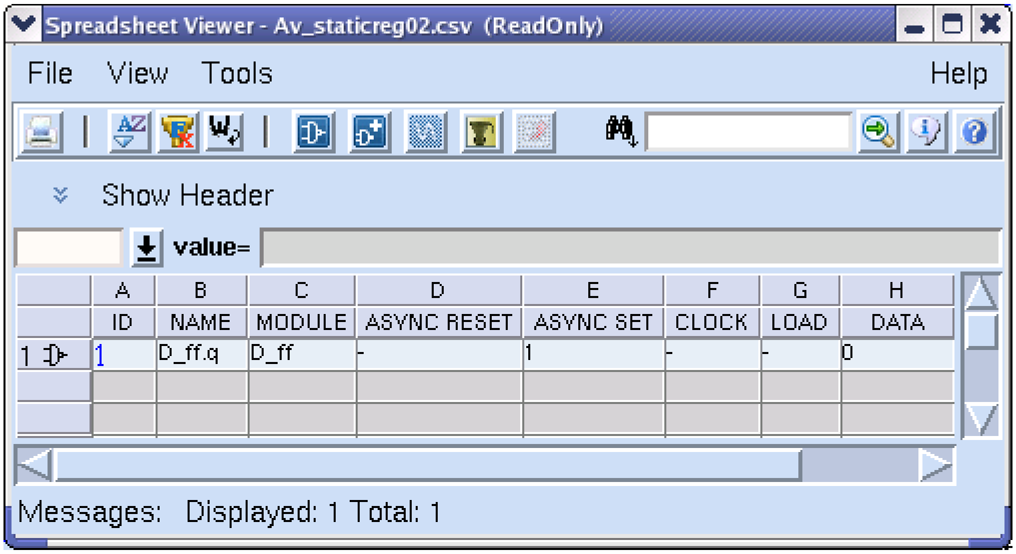
Consider the following Verilog and SGDC files:

```
// test.v

module D_ff(q,d,clk,reset,preset);
output q;
input d,clk,reset,preset;
reg q;
wire w1, w2;
assign w1 = 1
'b0;
assign w2 = d & w1;
always@(posedge clk or posedge reset or posedge preset)
if(reset)
  q <= 1'b0;
else if(preset)
  q <= 1'b1;
else
  q <= w2;
endmodule
```

```
// Constraints.sgdc

current_design D_ff
clock -name D_ff.clk -period 10
set_case_analysis -name D_ff.preset
-value 1
```

In the above example, the pins of the sequential element `q` are static. Therefore, the *Av_staticreg02* rule reports a violation.

When you double-click on the violation of this rule, the following spreadsheet appears:



| | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| | ID | NAME | MODULE | ASYNC RESET | ASYNC SET | CLOCK | LOAD | DATA |
| 1 | 1 | D_ff.q | D_ff | - | 1 | - | - | 0 |

Messages: Displayed: 1 Total: 1

FIGURE 36. Spreadsheet generated by the Av_staticreg02 rule

In the above spreadsheet, click 1 in the *ID* column, and open the *Modular Schematic* window. The following schematic appears:

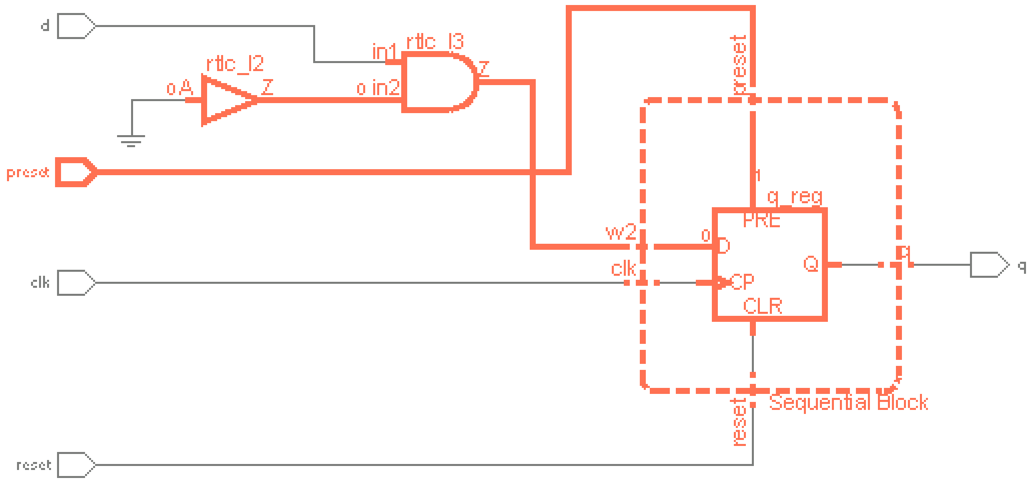


FIGURE 37. Schematic the Av_staticreg02 rule violation

In the above schematic, the portion highlighted in red is the rule-violating portion.

To fix this violation, perform the following actions:

- Change the value of `w1` to `1'b1`.
- Remove the *set_case_analysis* constraint from `preset`.

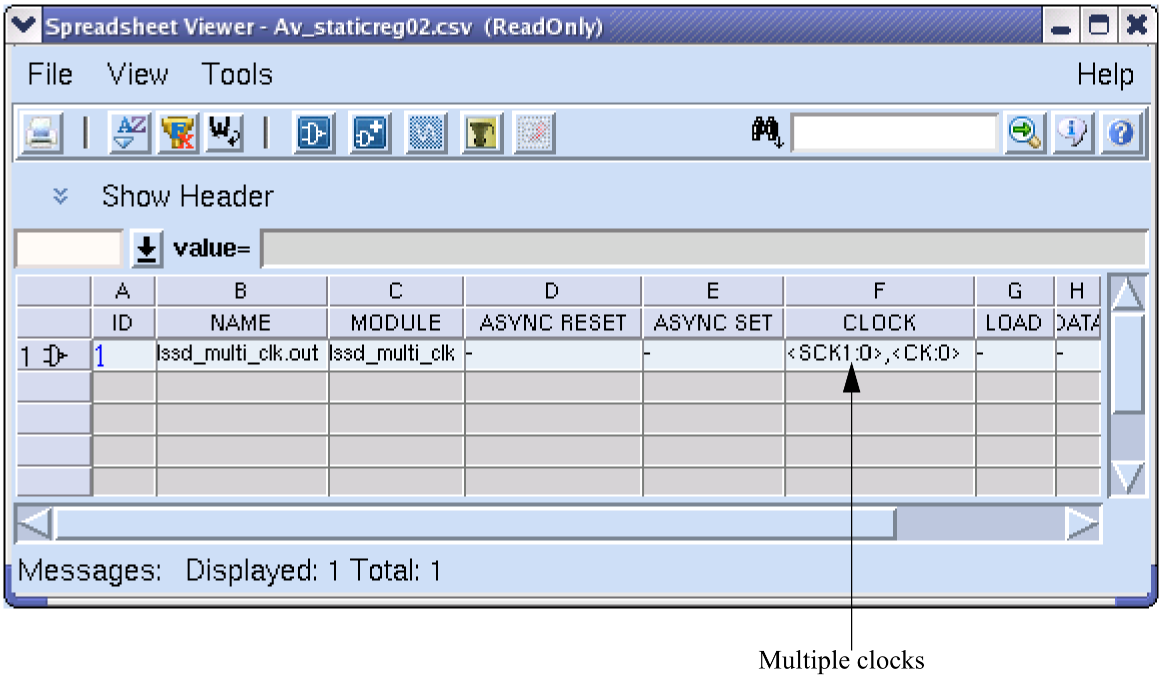
### Example 3

Consider the following Verilog and SGDC files:

```
// test.v
module lssd_multi_clk(clk, master_clk, slave_clk, d1, scan_d1, out, scan_out);
    input clk, master_clk, slave_clk, d1, scan_d1;
    output out, scan_out;

  FC3S2AQHV33
  lssd1(.D(d1),.CK(clk),.SDI(scan_d1),.SCK1(master_clk),.SCK2(slave_clk),
  .Q(out),.SDON(scan_out));
endmodule


// Constraints.sgdc
current_design lssd_multi_clk
clock -name lssd_multi_clk.slave_clk -period 10
set_case_analysis -name lssd_multi_clk.master_clk -value 0
set_case_analysis -name lssd_multi_clk.clk -value 0
```

In the above example, the *Av_staticreg02* rule reports a violation for the `lssd1` instance as multiple clocks (`CK`, `SCK1`, `SCK2`, `CK`, and `SCK1`) of this instance are tied to a constant value.

When you double-click on the violation of this rule, the following spreadsheet appears:


Spreadsheet Viewer - Av_staticreg02.csv (ReadOnly)
File View Tools Help
Show Header
value=
Messages: Displayed: 1 Total: 1
Multiple clocks

|   | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
|   | ID | NAME | MODULE | ASYNC RESET | ASYNC SET | CLOCK | LOAD | DATA |
| 1 | 1 | lssd_multi_clk.out | lssd_multi_clk | - | - | <SCK1:0>,<CK:0> | - | - |

**FIGURE 38.** Spreadsheet generated by the Av_staticreg02 rule

In the above spreadsheet, click 1 in the *ID* column and open the *Modular Schematic* window.

The following figure shows the schematic of this example:

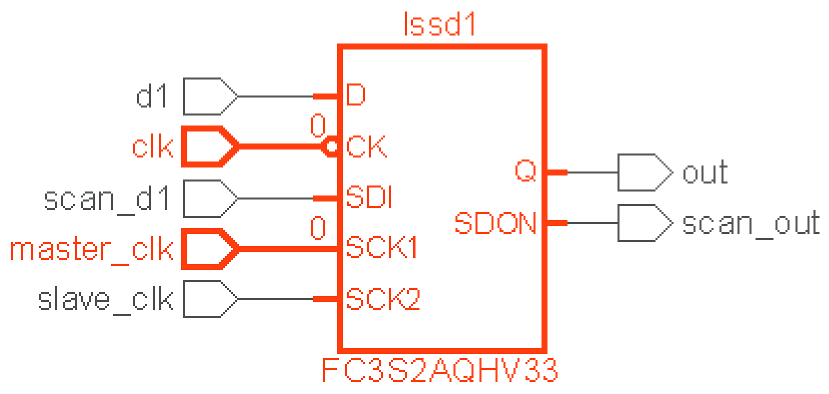


**FIGURE 39.** Schematic the Av_staticreg02 rule violation

To fix the above violation, remove the *set_case_analysis* constraint from the SGDC file.

## Default Severity Label

Info | Warning

## Rule Group

Implicit-Properties

## Reports and Related Files

- *Av_staticreg02 Spreadsheet Report*
- *Auto Verify Central Report*

## Av_syncfifo01

**Checks overflow and underflow of synchronous FIFOs in a design**

### When to Use

Use this rule to check overflow and underflow of synchronous FIFOs in a design.

### Description

The *Av_syncfifo01* rule performs overflow and underflow checks for synchronous FIFOs. These checks are performed by using read/write pointers of FIFO.

Note that if a FIFO is implemented with a library memory cell, overflow and underflow checks are not performed and such checks are reported as DISABLED.

#### Restrictions Applied on FIFO Extraction

The following restrictions apply to FIFO extraction:

- The *Av_syncfifo01* rule detects only synthesized FIFOs, that is, FIFOs for which memory is instantiated and synthesized.

  By default, only memories up to 4096 bits are synthesized. To specify a higher threshold, specify the following project file command:

  ```
  set_option mthresh <threshold-value>
  ```

  For details, refer to the *FIFO Synchronization Scheme* topic of *SpyGlass CDC Rules Reference Guide*.
- The *Av_syncfifo01* rule detects FIFOs for which read-pointer and write pointer increment (by 1) upon the read and write operations, respectively.

#### Assumptions Applied on FIFO Verification

The following assumptions apply to FIFO verification:

- Initially, a FIFO is assumed empty and read/write pointers are set to zero.
- The FIFO is not cleared or flushed after reset.

If this assumption is violated, that is, the FIFO can be cleared after reset, a false violation about overflow and underflow may get reported.

You can avoid this issue by specifying a FIFO reset signal in an SGDC file, as shown in the following example:

```
reset -name fifo_flush -value 1
```

In the above example, the FIFO reset signal `fifo_flush` is specified to be active high. Therefore, SpyGlass -Auto Verify solution controls the FIFO flush function accordingly.

## Parameter(s)

- *audit*: Default value is `no`. Set this parameter to `yes` to not perform functional analysis.
- *av_msgmode*: Default value is `fail`. Set this parameter to `all` to report all types of assertions (pass, fail, partially proved). Other possible values are `pass` and `pp`.
- `distributed_fifo`: Default value is `no`. Set this parameter to `yes` to detect FIFOs with distributed memories.

**NOTE:** *This is a SpyGlass CDC parameter.*

- `delayed_ptr_fifo`: Default value is `no`. Set this parameter to `yes` when the read/write pointers are delayed and the multiplexer inside the memory is one-hot or implemented using gates.

**NOTE:** *This is a SpyGlass CDC parameter.*

- `filter_named_resets`: Default value is `clk`, `clock`, `scan`. Specify a list of strings to auto-infer asynchronous resets that do not match the specified strings.

**NOTE:** *This is a SpyGlass CDC parameter.*

## Constraint(s)

- *set_case_analysis* (Optional): Use this constraint to specify case analysis conditions.
- *formal_analysis_filter* (Optional): Use this constraint to specify the modules or hierarchies on which formal analysis should be ignored or performed.
- *clock* (Optional): Use this constraint to specify clocks signals in a design.

- *reset* (Optional): Use this constraint to specify reset signals in a design.

## Messages and Suggested Fix

### Message 1

The following message appears when FIFOs are identified:

```
[FIFOERR] [ERROR] FIFO with memory '<memory-name>', read
pointer '<read-pointer-name>' and write pointer '<write-
pointer-name>' detected. '<type>' check: <FAILED | Others
(Constraints-Conflict)>
```

The arguments of the above message are explained below:

| Argument | Description |
|---|---|
| <memory-name> | FIFO memory name |
| <read-pointer-name> | FIFO read pointer name |
| <write-pointer-name> | FIFO write pointer name |
| <type> | The problem type as overflows or underflows |

#### *Potential Issues*

This violation appears if your design contains completely recognized FIFOs.

#### *Consequences of Not Fixing*

In case overflow and underflow, if the status is reported as FAILED, it may result in functional issues in the design if the failure is not due to incorrect setup.

#### *How to Debug and Fix*

If the status is `FAILED` or `Others (Constraints-Conflict)`, open the *Waveform Viewer* window corresponding to the message, and check the marker that appears on the waveform.

This marker is positioned at a transition where overflow or underflow problem occurs. Therefore, this specific transition is a *witness* to the failure.

Some of the reasons that may cause false failures are as follows:

- Presence of potential reset/clear signal causing such violation.

  In this case, provide the reset/clear in the constraint file as a reset.
- The setup (clocks, resets, *set_case_analysis*, and input constraints) is not correct and complete.
- The initial state values in the Waveform Viewer window are incorrect.

  In this case, provide correct initial state in the constraints file or provide a VCD file from which an initial state can be loaded.

### Message 2

The following message appears when FIFOs are identified:

```
[FIFOWRN] [WARNING] FIFO with memory '<memory-name>', read
pointer '<read-pointer-name>' and write pointer '<write-
pointer-name>' detected. '<type>' check: Partially-Proved
```

The arguments of the above message are explained below:

| Argument | Description |
|---|---|
| <memory-name> | FIFO memory name |
| <read-pointer-name> | FIFO read pointer name |
| <write-pointer-name> | FIFO write pointer name |
| <type> | The problem type as overflows or underflows |

#### *Potential Issues*

This violation appears if your design contains completely recognized FIFOs.

#### *Consequences of Not Fixing*

If you do not fix this violation, your design may contain functional issues.

#### *How to Debug and Fix*

The `Partially-Proved` status appears when SpyGlass is not able to either fail or pass FIFO verification in the given time. In this case, you need to help the tool to complete the analysis. You may try the following options for better results:

- Increase assertion run-time by using the *atime* parameter.

- Use incremental analysis approach by using the *propfile* parameter.

### Message 3

The following message appears when FIFOs are identified:

**[FIFOINF] [INFO]** `FIFO with memory '<memory-name>', read pointer '<read-pointer-name>' and write pointer '<write-pointer-name>' detected. '<type>' check: PASSED`

The arguments of the above message are explained below:

| Argument | Description |
|---|---|
| <memory-name> | FIFO memory name |
| <read-pointer-name> | FIFO read pointer name |
| <write-pointer-name> | FIFO write pointer name |
| <type> | The problem type as overflows or underflows |

#### *Potential Issues*

Not applicable

#### *Consequences of Not Fixing*

Not applicable

#### *How to Debug and Fix*

Not applicable

### Message 4

The following message is reported when FIFOs with library memory cells are identified:

**[FIFOLIBMEM] [INFO]** `FIFO with library memory '<memory-name>', read pointer '<read-pointer-name>' and write pointer '<write-pointer-name>' detected. '<type>' check:<status>`

The arguments of the above message are explained below:

| Argument | Description |
|---|---|
| <memory-name> | FIFO memory name |
| <read-pointer-name> | FIFO read pointer name |
| <write-pointer-name> | FIFO write pointer name |
| <type> | The problem type as overflows or underflows |
| <status> | Assertion status - DISABLED |

### *Potential Issues*

This violation appears if your design contains library memory cell based FIFOs.

### *Consequences of Not Fixing*

None

### *How to Debug and Fix*

If FIFO memory is a library cell, no functional check is performed. It is an informational message.

## Example Code and/or Schematic

Consider the following schematic of a violation of this rule:

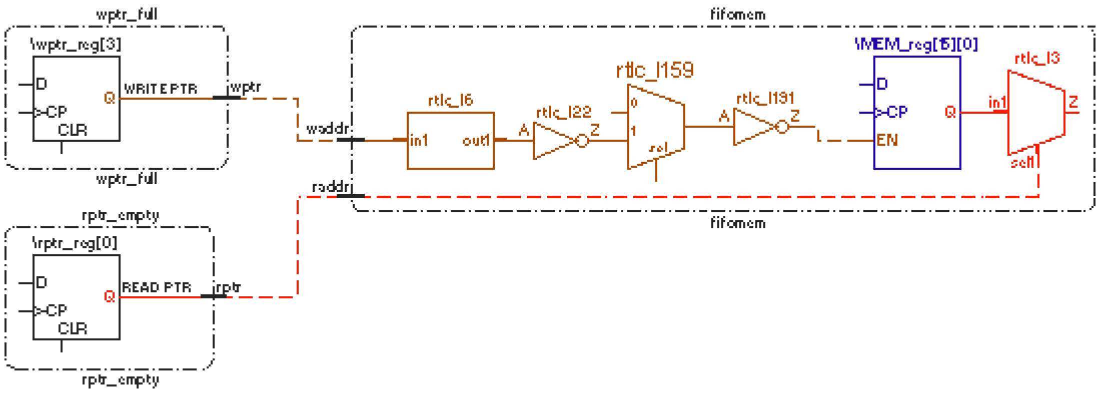


**FIGURE 40.** Schematic the Av_syncfifo01 rule violation

In the above scenario, the *Av_syncfifo01* rule reports a violation as the

`MEM` memory is a part of the synchronous FIFO where the `raddr` read pointer and the `waddr` write pointer are from the same clock domain.

### Schematic Highlight

The *Av_syncfifo01* rule highlights the following information in a schematic:

- Read pointers
- Write pointers
- Memory

For the FIFOs implemented with library memories, only memory is highlighted.

## Default Severity Label

The rule severity varies according to the assertion status as follows:

- FAILED: Error
- Partially-Proved: Warning
- PASSED/DISABLED: Info
- Others (Constraints-Conflict): Error

## Rule Group

Implicit-Properties

## Reports and Related Files

*Av_syncfifo01.<ID>.OverConstrainInfo*: This file contains details of conflicting constraints. For details, see *Overconstrain Info File*.

## Standard Properties Rules

The Standard Properties rules check for the standard OVL v. 02.10.16 assertions.

A violation of a standard OVL property corresponds to the violation of the OVL assertion specified in code. As a general rule to investigate the cause of a standard OVL assertion failure, you should verify the correctness of the OVL assertion specification, especially those assertions that have complex parameters that impact the validation of the rule.

Inputs to the assertion modules are generic Verilog/VHDL expressions. You must consider the following points while writing and analyzing OVL assertions:

- You can provide synthesizable logic for property check without having to use them within the design. For example, in a FSM, you may have an output that is activated when you are in three different states of the design. You can provide a OR of the three states to a property and request a check to assure that the output becomes true whenever we reach these states. The OR of the three states won't be part of the design but will be relevant for the property validation.
- These expressions, if directly provided to the assertion module instantiation, won't have a user name associated with them. An internal name will be assigned to them. If this is a problem for proper debugging an internal wire can be created that will generate the expression and the wire can be provided as input to the OVL instance. SpyGlass Auto Verify solution will preserve all user defined wires and naming.

OVL properties can be seen as HDL modules that are monitoring design's activities. In general these modules monitor its inputs at each positive edge of the clock provided as input to the OVL module. There are exceptions to this rule, for instance assert_proposition() monitors the test expression at all time (not only at the edge of the clock, in fact this property does not take clock as input). Rule descriptions of SpyGlass Auto Verify solution do not explicitly mention the clock edge presence for a check; unless otherwise indicated the test expressions are monitored at each positive edge of the clock. Some of the OVL assertions are also waiting for some events to start monitoring a test expression; in this case the event comes on conjunction with the clock, both event and clocks need to be activated in order to start monitoring.

For any violation of the Av_ovl01 rule, the waveform viewer shows all signals at the boundary of the OVL module (for example, test expressions). Other signals in the fan-in cone can be loaded gradually through the Waveform Viewer user interface.

The following table describes the rules under this category:

| **Rule** | **Reports** |
|---|---|
| *Av_ovl01* | OVL checks in the design. |

## Av_ovl01

**Reports OVL checks in a design.**

## When to Use

Use this rule to validate design functionality that is specified by using OVL assertions.

### Prerequisites

Specify clock signals in a design by using the *clock* constraint.

## Description

The *Av_ovl01* rule validates OVL assertions and assumptions in a design.

OVL assertion represents the expected functionality of the design.

For the list of OVL assertions checked by this rule, see *The OVL Support*.

To understand how OVL assertions are asserted in a design, see *Properties Specification using OVL*.

## Parameter(s)

None

## Constraint(s)

- *breakpoint* (Optional): Use this constraint to specify breakpoints in a design where functional analysis should stop.
- *watchpoint* (Optional): Use this constraint to generate a waveform for an internal signal.
- *special_module* (Optional): Use this constraint to define property and constraint modules.
- *clock* (Mandatory): Use this constraint to specify clocks in a design.
- *reset* (Optional): Use this constraint to specify resets in a design.
- *set_case_analysis* (Optional): Use this constraint to specify case-analysis conditions in a design.

## Messages and Suggested Fix

The following message appears if the OVL property *<OVL-property>*

fails:

```
[WARNING] OVL check failed for '<OVL-property'
```

In addition, if a set of conflicting constraints are specified for the design, additional message is appended to the above message string. For details on the message and how to debug and fix this conflict, see the *Overconstrain Info File* section.

#### *Potential Issues*

This violation appears if an OVL property defined for your design fails.

#### *Consequences of Not Fixing*

If you do not fix this violation, the design behavior becomes inconsistent with the expected functionality as captured by OVL assertions.

### Default Severity Label

Warning

### Rule Group

Standard-Properties

### Report and Related Files

- Av_ovl01.<ID>.OverConstrainInfo: This file contains details of conflicting constraints. For details, see *Overconstrain Info File*.
- *Auto Verify Central Report*

# Must Rules

These rules are always run.

The following table describes the rules under this category:

| Rule | Flags |
|---|---|
| *Av_license01* | For license failure |
| *Av_init01* | When all clocks in the design are not specified using the clock constraint and the `use_inferred_clocks` parameter is also not set. |
| *Av_initseq01* | When all `define_tag` constraints with the `-tag initSeq` argument specified do not have the same length sequence specified with the `-value` argument |
| *Av_multitop01* | When the design has multiple top-level design units |
| *Av_sanity01* | Issues with the user-specified property files |
| *Av_sanity02* | Non-tristated nets that have multiple drivers |

## Av_license01

**Reports license failure**

### When to Use

This rule is run automatically.

### Description

The *Av_license01* rule reports a violation in the following cases:

- If the `Auto_Verify` license key is unavailable when the Auto Verify rules are run.

  *Message 1* is reported in this case.
- If the `SVA_GEN` license key is unavailable when the *av_dump_assertions* parameter is used.

  *Message 2* is reported in this case.
- If the `sva` license key is unavailable when the set_option enableSVA yes project file command is specified.

  *Message 3* is reported in this case.

### Parameter(s)

None

### Constraint(s)

None

### Messages and Suggested Fix

The following message appears if

#### Message 1

The following message appears when the `Auto_Verify` license is unavailable:

```
[FATAL] Advanced Lint Policy not run due to unavailability of
Auto_Verify license feature
```

#### *Potential Issues*

Not applicable

#### *Consequences of Not Fixing*

If you do not fix this violation, SpyGlass run does not proceed further.

#### *How to Debug and Fix*

Specify the `Auto_Verify` license.

### Message 2

The following message appears when the `Auto_Verify` license is unavailable:

```
[ERROR] SVA generation feature not run due to unavailability of
SVA_GEN license feature
```

#### *Potential Issues*

Not applicable

#### *Consequences of Not Fixing*

If you do not fix this violation, the feature on generating SVA is disabled.

#### *How to Debug and Fix*

Specify the `SVA_GEN` license.

### Message 3

The following message appears when the `sva` license is unavailable:

```
[ERROR] SVA constraints not read due to unavailability of sva
license feature
```

#### *Potential Issues*

Not applicable

#### *Consequences of Not Fixing*

If you do not fix this violation, the SVA assume properties are not honored for functional verification.

#### *How to Debug and Fix*

Specify the `sva` license.

### Example Code and/or Schematic

Not applicable

### Default Severity Label

Fatal | Error

### Reports and/or Related Files

None

## Av_init01

**Reports initial setup issues of a design.**

### When to Use

Use this rule to detect incorrect initial inputs to a design.

### Description

The *Av_init01* rule reports a violation in the following cases:

- If no clock is specified for the design

  For details on fixing this violation, see *How to Debug and Fix*.
- If the OVL constraint specified in a property is invalid
- If SpyGlass is unable to locate or open the VCD file specified by the *vcdfile* parameter.

  This VCD file contains the details of the initial state of a design for functional analysis.

### Parameter(s)

- `use_inferred_clocks` of SpyGlass CDC solution: The default value is `no`. Set this parameter to `yes` to use the automatically-generated clock information.

**NOTE:** *This is the parameter of SpyGlass CDC solution.*

- *vcdfile*: The default value is `NULL`. Specify a VCD file name that SpyGlass can use to extract an initial state for functional analysis.

### Constraint(s)

*clock* (Optional): Use this constraint to specify clock signals in a design.

### Messages and Suggested Fix

#### Message 1

The following message appears if clocks for all the flip-flops in the design are not specified:

```
[FATAL] Could not find clocks for all the flops. Please add
clocks to design
```

#### *Potential Issues*

This violation appears if no clocks are detected in the design.

#### *Consequences of Not Fixing*

If you do not fix this violation, SpyGlass run does not proceed further.

#### *How to Debug and Fix*

To fix this violation, specify clocks for the design in any of the following ways:

- By using the *clock* constraint
- By setting the `use_inferred_clocks` parameter of SpyGlass CDC solution to `yes`.

For better results, run the *Clock_info03* rule of the SpyGlass CDC solution that reports unconstrained flip-flops. Based on the violation reported by this rule, specify clocks for the reported flip-flops by using the *clock* constraints.

### Message 2

The following message appears if the constraints defined in the property file are missing in the design:

**[FATAL]** `Some constraints specified in property file not found in design. Refer '<file-name>' for more details`

#### *Potential Issues*

This violation appears if your design is not constrained by the constraints specified in the property file.

#### *Consequences of Not Fixing*

If you do not fix this violation, SpyGlass run does not proceed further.

#### *How to Debug and Fix*

To fix this violation, review the constraints specified in the property file and update them as per the design requirement.

### Message 3

The following message appears if SpyGlass is not able to open the VCD file specified by the *vcdfile* parameter:

```
[FATAL] Unable to open vcd file for initial state
```

#### *Potential Issues*

This violation will appear when VCD file specified by the *vcdfile* parameter is not present in the specified location.

#### *Consequences of Not Fixing*

If you do not fix this violation, SpyGlass run does not proceed further.

#### *How to Debug and Fix*

To fix this violation, specify a correct VCD file.

### Message 4

The following message appears if you use the *av_run_time* parameter when the *audit* parameter is set to `yes`:

```
[WARNING] Parameter "av_run_time" ignored in "audit" mode
```

#### *Potential Issues*

This violation appears if you use the *av_run_time* parameter when the *audit* parameter is set to `yes`.

#### *Consequences of Not Fixing*

If you do not fix this violation, upper bound on runtime is ignored and

SpyGlass runs normally.

***How to Debug and Fix***

Do not use these parameters together.

## Example Code and/or Schematic

Consider that you do not:

- Specify a clock by using the *clock* constraint or
- Set the `use_inferred_clocks` parameter to yes,

Now consider the following file:

```
// test.v
module test (input in, clk, output reg q);
  always @(posedge clk)
  q <= d;
endmodule
```

When you specify the above file to SpyGlass and run any rule of the SpyGlass Auto Verify solution, the Av_init01 rule reports *Message 1*.

## Default Severity Label

Fatal

## Rule Group

Setup

## Report and Related Files

*Auto Verify Central Report*

## Av_initseq01

**Initialization sequences of multiple signals should be of the same length.**

## When to Use

Use this rule to detect setup issues due to different initialization sequences specified by the *define_tag* constraint.

### Prerequisites

Specify signals by using the *define_tag* constraint.

## Description

The *Av_initseq01* rule reports a violation if the initialization sequences specified by the `-value` argument of the *define_tag* constraints (specified with the `-tag initSeq` argument) are not of the same length.

## Parameter(s)

None

## Constraint(s)

*define_tag* (Mandatory): Use this constraint to define a named condition for the application of certain stimulus at the top port or an internal node.

## Messages and Suggested Fix

The following message appears if the initialization sequence of signals is not of the same length:

```
[WARNING] Initialization sequences provided by the 'define_tag
-tag initSeq' constraint are all not of the same length
```

#### *Potential Issues*

This violation appears if your design contains signals that have initialization sequence of different lengths.

#### *Consequences of Not Fixing*

If you do not fix this violation, the design may be simulated by an undefined value.

When you specify simulation vectors on signals by using the *define_tag* constraint, the design is simulated by using these vectors to find a valid initial state. However, if the sequences applied on the signals are of different lengths (specified by the `-value` argument of the *define_tag* constraint), the design is simulated by an undefined value for the remaining cycles of smaller sequences. This may not be desirable.

#### *How to Debug and Fix*

To fix this violation, specify the same length of sequence for the signals specified by the *define_tag* constraint with the `-tag initSeq` argument.

### Example Code and/or Schematic

Consider the following *define_tag* constraints:

```
define_tag -tag initSeq -name top.reset1 -value 1 1 1 x x x
define_tag -tag initSeq -name top.reset2 -value x x 1 1
```

For the first *define_tag* constraint specification, the length of sequence is six and it is four for the second specification. Therefore, the *Av_initseq01* rule reports a violation.

To fix this violation, modify the second specification as below:

```
define_tag -tag initSeq -name top.reset2 -value x x 1 1 1 1
```

### Default Severity Label

Warning

### Rule Group

Must rule

### Report and Related Files

*Auto Verify Central Report*

## Av_multitop01

**Reports a violation in case of multiple top-level design units**

### When to Use

Use this rule to check if your design contains multiple top-level design units.

### Description

The *Av_multitop01* rule reports a violation if your design contains multiple top-level design units.

For details on these design units, refer to the violation of the *DetectTopDesignUnits* Built-In rule.

### Parameter(s)

None

### Constraint(s)

None

### Messages and Suggested Fix

The following message appears if multiple top-level design units are present in a design:

```
[FATAL] Detected '<num>' top level design units. Please specify
a single top level design unit
```

Where *<num>* is the total number of top-level design units.

#### *Potential Issues*

This violation appears if your design contains multiple top-level design units.

#### *Consequences of Not Fixing*

If you do not fix this violation, SpyGlass run does not proceed further.

#### *How to Debug and Fix*

To fix this violation, specify a single top-level design unit by using the following command in the project file:

```
set_option top <du-name>
```

## Example Code and/or Schematic

Consider the following example:

```
module seq_block (input d1, d2, clk, rst, output reg q1, q2);
always @(posedge clk or posedge rst)
    if (rst)
      q1 <= 1'b0;
    else
      q1 <= d1;
always @(posedge clk)
    q2 <= d2;
endmodule

module flop (input d, clk, output reg q);
always @(posedge clk)
  q<= d;
endmodule
```

In the above example, both the modules `seq_block` and `flop` are considered as the top-level design units.

Specify one of these modules as a top-level design unit by using the `set_option top <du-name>` project file command.

## Default Severity Label

Fatal

## Rule Group

Sanity

## Report and Related Files

*Auto Verify Central Report*

## Av_sanity01

**Reports an error if there is any issue in the property file.**

## When to Use

Use this rule to detect setup issues due to incorrect property file.

### Prerequisites

Specify a property file by using the *propfile* parameter.

## Description

The *Av_sanity01* rule reports issues in the property files specified using the *propfile* parameter.

## Parameter(s)

*propfile*: Specify the name of the property file containing properties to be checked.

## Constraint(s)

*clock* (Optional): Use this constraint to specify clock signals in a design.

## Messages and Suggested Fix

The following message appears if the specified property file contains issues:

```
[ERROR] Some Assertions specified in property file (<prop-file-
name>) not found in design. Refer '<error-log>' for more
details
```

### *Potential Issues*

This violation appears if your design does not contain assertions that are mentioned in the property file specified by the *propfile* parameter.

This may happen if the design or the design view has changed due to changes in the commands of SpyGlass Auto Verify solution. As a result, the assertions specified in the property file become invalid in the context of the changed design.

### *Consequences of Not Fixing*

If you do not fix this violation, SpyGlass ignores the invalid assertions specified in the property file.

### *How to Debug and Fix*

To fix this violation, review the assertions reported in the *propfile_Assertion_<rule-name>.errorlog* file and update them as per the design.

## Example Code and/or Schematic

Consider the following files specified for SpyGlass analysis:

**// test.v**

```
`define STATE1 2'b01
`define STATE2 2'b11
module fsm (input clk, rst, d1, d2,
sel, output out);
  reg [1:0] state;
  always @ (posedge clk or posedge rst)
    if (rst)
      state <= 2'b00;
    else if (sel)
      state <= `STATE1;
    else
      state <= `STATE2;
  assign out = state[0] ? d1 : 1'bz;
  assign out = state[1] ? d2 : 1'bz;
endmodule
```

**Property file:**

```
RuleName: Av_bus01
off   Assertion   FAILED              test.v  16  "fsm"  [out]
off   Assertion   FAILED              test.v  16  "fsm"  [out1]

RuleName: Av_bus02
```

```
off  Assertion   FAILED          test.v  16  "fsm"  [out]
off  Assertion   FAILED          test.v  16  "fsm"  [out2]
```

In the above example, assertions are specified in the property file for the `out1` and `out2` nets for the *Av_bus01* and *Av_bus02* rules, respectively.

However `out1` and `out2` do not exist in the design. Therefore, the *Av_sanity01* rule reports a violation corresponding to each rule.

To fix this violation, review the *propfile_Assertion_<rule-name>.errorlog* file corresponding to each rule and update the assertions to avoid any mismatch with the design.

## Default Severity Label

Error

## Rule Group

Sanity

## Report and Related Files

- propfile_Assertion_<rule-name>.errorlog

  This file contains the details of incorrect assertions. Here, <rule-name> refers to the rule that is not run due to invalid assertions.
- *Auto Verify Central Report*

## Av_sanity02

**Reports nets that have multiple drivers**

### When to Use

Use this rule during functional analysis to detect nets with multiple drivers.

### Description

The *Av_sanity02* rule reports non-tristate nets that have multiple drivers.

Such nets are considered as primary inputs for functional analysis.

### Parameter(s)

None

### Constraint(s)

- *set_case_analysis* (Optional): Use this constraint to specify case analysis conditions.
- *clock* (Optional): Use this constraint to specify clock signals in a design.
- *reset* (Optional): Use this constraint to specify reset signals in a design.

### Messages and Suggested Fix

The following message appears when a non-tristate net *<net-name>* is present in a design with multiple drivers:

```
[WARNING] Net '<net-name>' is not tristate and has multiple
simultaneous drivers
```

#### *Potential Issues*

This violation appears if your design contains a non-tristate net with multiple drivers.

#### *Consequences of Not Fixing*

Not applicable

### *How to Debug and Fix*

To fix this violation, provide tristate nets in the design.

## Example Code and/or Schematic

Consider the following figure:

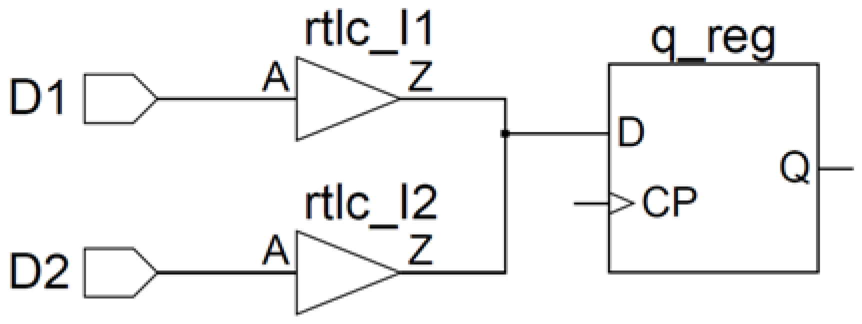


**FIGURE 41.** Example of the Av_sanity02 rule violation

For the above example, the *Av_sanity02* rule reports a violation because of the presence of a non-tristate net with multiple drivers `D1` and `D2`.

### Schematic Details

The *Av_sanity02* rule highlights the non-tristate net that has multiple drivers.

## Default Severity Label

Warning

## Rule Group

Sanity

## Reports and Related Files

*Auto Verify Central Report*

## Av_sanity06

**Reports issues found in the distributed computing flow**

### When to Use

Use this rule to detect issues in the distributed computing flow.

### Description

The *Av_sanity06* rule reports a violation in the following cases:

- If parse errors are found in the parallel file specified by the *fv_parallelfile* parameter.
- If an error occurs while accessing any of the machines specified in the parallel file.
- If there are insufficient number of licenses for the SpyGlass Auto Verify solution.

### Parameter(s)

*fv_parallelfile*: By default, this parameter is not set to any value. Specify a configuration file to this parameter. This file is used for distributed runs over several machines.

### Constraint(s)

- *set_case_analysis* (Optional): Use this constraint to specify case analysis conditions.
- *clock* (Optional): Use this constraint to specify clock signals in a design.
- *reset* (Optional): Use this constraint to specify reset signals in a design.

### Message Details

#### Message 1

The following message appears if parse errors are found in the parallel file:

```
[SW01] [FATAL] Could not open parallel run file '<file-name>'
```

***Potential Issues***

This violation appears if there is any error in the parallel run file.

***Consequences of Not Fixing***

If you do not fix this violation, SpyGlass run does not proceed further.

***How to Debug and Fix***

To fix this violation, specify a correct parallel run file.

### Message 2

The following message appears for an invalid login type:

`[SW02] [FATAL] <type> is not a supported login type`

***Potential Issues***

This violation appears if you specify an invalid login type in the parallel file.

***Consequences of Not Fixing***

If you do not fix this violation, SpyGlass run does not proceed further.

***How to Debug and Fix***

To fix this violation, specify a correct login type.

### Message 3

The following message appears if the value of the `MAX_PROCESSES` keyword is equal to or less than `1` or if it is equal to or greater than `500`:

`[SW03] [FATAL] Value of MAX_PROCESSES should be between 1 and 500`

***Potential Issues***

This violation appears if you specify an incorrect value for the `MAX_PROCESSES` keyword in the parallel file.

***Consequences of Not Fixing***

If you do not fix this violation, SpyGlass run does not proceed further.

***How to Debug and Fix***

To fix this violation, specify a value between 1 and 500 for the `MAX_PROCESSES` keyword.

### Message 4

The following message appears for the unsuccessful LSF run because of invalid options in the LSF command:

```
[SW04] [FATAL] Lsf run with specified command is not successful
```

***Potential Issues***

This violation appears if you specify invalid options with the LSF command in the parallel file.

***Consequences of Not Fixing***

If you do not fix this violation, SpyGlass run does not proceed further.

***How to Debug and Fix***

To fix this violation, specify correct options for the LSF command.

### Message 5

The following message appears if process count is not a positive integer value in the parallel file:

```
[SW05] [FATAL] Process count in parallel file must be a
positive integer
```

***Potential Issues***

This violation appears if you specify an invalid integer value to the process count in the parallel file. The process count accepts only a positive integer value.

***Consequences of Not Fixing***

If you do not fix this violation, SpyGlass run does not proceed further.

***How to Debug and Fix***

To fix this violation, specify a positive integer value for the process count in the parallel file.

### Message 6

The following message appears if none of the specified machines in the parallel file is accessible:

```
[SW06] [FATAL] None of the machines specified in parallel file
is accessible
```

***Potential Issues***

This violation appears if none of the machines specified in a parallel file is accessible.

#### *Consequences of Not Fixing*

If you do not fix this violation, SpyGlass run does not proceed further.

#### *How to Debug and Fix*

To fix this violation, specify the names of accessible machines in the parallel file.

### Message 7

The following message appears to report the machines that are not accessible:

**[SW07] [FATAL]** `Machines '<machines>' are not accessible`

#### *Potential Issues*

This violation appears if none of the machines specified in a parallel file is accessible.

#### *Consequences of Not Fixing*

If you do not fix this violation, SpyGlass run does not proceed further.

#### *How to Debug and Fix*

To fix this violation, specify the names of accessible machines in the parallel file.

### Message 8

The following message appears if the `LOGIN_TYPE` keyword is not specified in the parallel file:

**[SW08] [FATAL]** `'LOGIN_TYPE' is not specified in parallel file`

#### *Potential Issues*

This violation appears if you do not specify the `LOGIN_TYPE` keyword in the parallel file.

#### *Consequences of Not Fixing*

If you do not fix this violation, SpyGlass run does not proceed further.

#### *How to Debug and Fix*

To fix this violation, specify the `LOGIN_TYPE` keyword in the parallel file.

### Message 9

The following message appears if the `MAX_PROCESSES` keyword is not specified in the parallel file:

```
[SW09] [FATAL] 'MAX_PROCESSES' is not specified in parallel
file
```

#### *Potential Issues*

This violation appears if you do not specify the `MAX_PROCESSES` keyword in the parallel file.

#### *Consequences of Not Fixing*

If you do not fix this violation, SpyGlass run does not proceed further.

#### *How to Debug and Fix*

To fix this violation, specify the `MAX_PROCESSES` keyword in the parallel file.

### Message 10

The following message appears if the `MACHINES` keyword is not specified for the `rsh/ssh` login type in the parallel file:

```
[SW10] [FATAL] 'MACHINES' not specified for login type rsh/ssh
in parallel file
```

#### *Potential Issues*

This violation appears if you do not specify the `MACHINES` keyword for the `rsh/ssh` login type in the parallel file.

#### *Consequences of Not Fixing*

If you do not fix this violation, SpyGlass run does not proceed further.

#### *How to Debug and Fix*

To fix this violation, specify the `MACHINES` keyword for the `rsh/ssh` login type in the parallel file.

### Message 11

The following message appears if an error occurs while running the `lsf bsub` command:

**[SW11] [FATAL]** `Error executing lsf bsub command`

#### *Potential Issues*

This violation appears if you specify invalid options, such as `-I`, `-Ip`, and `-Is` with the `bsub` command. These options are not allowed with the `LSF_CMD` keyword in the parallel file.

#### *Consequences of Not Fixing*

If you do not fix this violation, SpyGlass run does not proceed further.

#### *How to Debug and Fix*

To fix this violation, specify valid the options with the `bsub` command.

### Message 12

The following message appears to indicate a missing solver executable:

**[SW12] [FATAL]** `Solver executable '<executable>' not found`

#### *Potential Issues*

This violation appears if the solver executable file is not found in the SPYGLASS_HOME/lib/ path of SpyGlass release area. The name of this file is of the format solver.<*platform*>. For example, solver.SunOS5.

#### *Consequences of Not Fixing*

If you do not fix this violation, SpyGlass run does not proceed further.

#### *How to Debug and Fix*

To fix this violation, add the missing solver executable file in the SPYGLASS_HOME/lib/ path of SpyGlass release area.

### Message 13

The following message appears to indicate missing licenses of SpyGlass Auto Verify solution for distributed computing flow:

**[SW13] [FATAL]** `No Advanced Lint licenses available for`

`Distributed Computing Flow`

#### *Potential Issues*

This violation appears if you do not specify licenses for the SpyGlass Auto Verify solution.

#### *Consequences of Not Fixing*

If you do not fix this violation, SpyGlass run does not proceed further.

#### *How to Debug and Fix*

To fix this violation, provide licenses of SpyGlass Auto Verify solution for the distributed computing flow.

### Message 14

The following message appears to indicate inadequate SpyGlass Auto Verify licenses available for distributed computing flow:

```
[SW14] [WARNING] Only '<num>' Advanced Lint licenses available
for Distributed Computing
```

#### *Potential Issues*

This violation appears if you specify insufficient number of licenses for SpyGlass Auto Verify solution.

#### *Consequences of Not Fixing*

If `n` licenses for SpyGlass Auto Verify solution are available, only `n-1` licenses are used for distributed computing as one of the license is used by the main process.

#### *How to Debug and Fix*

To fix this violation, specify adequate number of licenses of SpyGlass Auto Verify solution for distributed computing flow.

### Message 15

The following message appears if you specify an invalid option *`<option-name>`* in the `LSF_CMD` keyword in a parallel file:

```
[SW15] [WARNING] Unsupported option '<option-name>' specified
in LSF_CMD field is ignored for Distributed Computing Flow
```

#### *Potential Issues*

This violation appears if you specify an invalid option *`<option-name>`* in the `LSF_CMD` keyword in a parallel file.

#### *Consequences of Not Fixing*

If you do not fix this violation, distributed computing does not run.

#### *How to Debug and Fix*

To fix this violation, specify supported options with the `LSF_CMD` keyword in a parallel file.

### Example Code and/or Schematic

Not applicable

### Default Severity Label

Fatal | Warning

### Rule Group

Sanity

### Reports and Related Files

*Auto Verify Central Report*

## SGDC_av_meta_design_hier01

**Checks the presence of constraint meta_design_hier**

### When to Use

Use this rule to check if the `meta_design_hier` constraint is specified in a VHDL or mixed design that has the *av_dump_assertions* parameter set to `sva`.

### Description

The *SGDC_av_meta_design_hier01* rule reports the absence of the `meta_design_hier` constraint when the *av_dump_assertions* parameter is set to `sva` in a VHDL or mixed design.

If the `meta_design_hier` constraint is not specified for a top module, the rule uses the default name `TB.DESIGN_INST` for *`<testbench name>.<top-module-name>`*.

### Parameter(s)

- *av_dump_assertions*: The default value is “”. Set this parameter to `sva` to generate SystemVerilog Assertions (SVA) for partially-proved assertions of rule Av_staticnet01 and Av_deadcode01.

### Constraint(s)

- *meta_design_hier*: Use this constraint to specify a top-level design and a hierarchical design for VHDL and mixed designs.

### Messages and Suggested Fix

The following message appears if you do not specify the *meta_design_hier* constraint when the *av_dump_assertions* parameter is set to `sva` in a VHDL or mixed design:

```
[ERROR] Constraint "meta_design_hier" not specified for design
'<design-name>'. Default test-bench and design name will be
used in generated SVA
```

***Potential Issues***

This violation appears if you do not specify the *meta_design_hier* constraint when the *av_dump_assertions* parameter is set to `sva` in a VHDL or mixed design.

#### *Consequences of Not Fixing*

If you do not fix this violation, the *SGDC_av_meta_design_hier01* rule uses the default name `TB.DESIGN_INST` for *`<testbench name>.<top-module-name>`*.

Because of the default test bench and design name used in generated assertions, syntax errors may appear during the simulation run.

#### *How to Debug and Fix*

To fix this violation, perform the following actions:

- Specify the *meta_design_hier* constraint when the *av_dump_assertions* parameter is set to `sva` in a VHDL or mixed design.
- Replace the default names with the actual test bench and design instance name before running simulation.

### Example Code and/or Schematic

Not applicable

### Default Severity Label

Error

### Rule Group

Non-fatal must rule

### Reports and Related Files

*Auto Verify Central Report*

## SGDC_fsm_setup01

### When to Use

Use this rule to perform sanity checks on the *fsm* constraint.

#### Prerequisites

Specify the *fsm* constraint.

### Description

The *SGDC_fsm_setup01* rule reports different types of issues in the *fsm* constraint. For information on the types of issues reported, see *Messages and Suggested Fix*.

### Parameter(s)

None

### Constraint(s)

*fsm* (Mandatory): Use this constraint to specify FSM details in a design.

### Messages and Suggested Fix

#### Message 1

The following message appears:

```
[WARNING] Fsm constraint with logical name '<fsm-name>' ignored
(Reason: Missing module name and state variable in the first
specification)
```

***Potential Issues***

This violation appears if the first *fsm* constraint specified with a logical name (`-name`) is not specified with a module name (`-module`) and a state variable (`-state_variables`).

See *Example 1 - Missing Module and State Variable for an FSM*.

***Consequences of Not Fixing***

The reported constraint is ignored from SpyGlass analysis.

#### *How to Debug and Fix*

To fix this violation, specify a module (`-module`) and a state variable (`-state_variables`) for the reported *fsm* constraint.

### Message 2

The following message appears:

```
[WARNING] Fsm constraint with logical name '<fsm-name>' mapped
to state variable '<state-variable>' of module '<module-name>'
(All previous specifications of Fsm constraints with same
logical name ignored)
```

#### *Potential Issues*

This violation appears if the same logical name (`-name`) is specified to multiple *fsm* constraints of different module names (`-module`) or state variables (`-state_variables`).

See *Example 2 - Same Logical Name for Multiple FSMs*.

#### *Consequences of Not Fixing*

If you do not fix this violation, all the *fsm* constraints for which both the following conditions hold true are ignored from SpyGlass analysis:

- *fsm* constraints that have the same logical name as that of the reported constraint.
- *fsm* constraints that are declared before the reported constraint

#### *How to Debug and Fix*

To fix this violation, use a unique combination of logical name (`-name`), module name (`-module`), and state variables (`-state_variables`) to uniquely identify an FSM.

### Message 3

The following message appears:

```
[WARNING] State value '<state-value>' specified in
-state_values field of Fsm constraint with state variable
'<state-variable>' (Module: <module-name>) ignored (Reason:
Invalid state value)
```

#### Potential Issues

This violation appears if an invalid state value is specified to the `-state_values` argument of the *fsm* constraint.

See *Example 3 - Invalid State Value of an FSM*.

#### Consequences of Not Fixing

If you do not fix this violation, the reported constraint is ignored from SpyGlass analysis.

#### How to Debug and Fix

Specify a valid state to the `-state_values` argument of the reported *fsm* constraint.

### Message 4

The following message appears:

```
[WARNING] State value '<state-value>' specified in
-state_values field of Fsm constraint with state variable
'<state-variable>' (Module: <module-name>) is to be
<appended|removed> (All previous specification for same state
value ignored)
```

#### Potential Issues

This violation appears if a state value specified by the `-state_values` argument of the *fsm* constraint is already specified to be removed (`-remove`) or appended (`-append`) by a previous *fsm* constraint.

See *Example 4 - FSMs with Duplicate State Values*.

#### Consequences of Not Fixing

If you do not fix this violation then other than the reported *fsm* constraint, all the *fsm* constraints declared before the reported constraint with the same state value are ignored from SpyGlass analysis.

#### How to Debug and Fix

Remove the *fsm* constraint with duplicate state value.

### Message 5

The following message appears:

```
[WARNING] Transition specified with state value '<state-value>'
in -from_state_value field of Fsm constraint with state
variable '<state-variable>' (Module: <module-name>) ignored
(Reason:  Invalid state value)
```

#### *Potential Issues*

This violation appears if an invalid state value is specified to the `-from_state_value` argument of the *fsm* constraint.

See *Example 5 - Invalid State Value To the -from_state_value Argument*.

#### *Consequences of Not Fixing*

If you do not fix this violation, the reported constraint is ignored from SpyGlass analysis.

#### *How to Debug and Fix*

Specify a valid state to the `-from_state_value` argument of the reported *fsm* constraint.

### Message 6

The following message appears:

```
[WARNING] Transition specified with state value '<state-value>'
in -to_state_value field of Fsm constraint with state variable
'<state-variable>' (Module: <module-name>) ignored (Reason:
Invalid state value)
```

#### *Potential Issues*

This violation appears if an invalid state value is specified to the `-to_state_value` argument of the *fsm* constraint.

See *Example 6 - Invalid State Value To the -to_state_value Argument*.

#### *Consequences of Not Fixing*

If you do not fix this violation, the reported constraint is ignored from SpyGlass analysis.

#### *How to Debug and Fix*

Specify a valid state to the `-to_state_value` argument of the reported *fsm* constraint.

### Message 7

The following message appears:

```
[WARNING] Transition specified from state value '<state-
value1>' to state value '<state-value2>' for Fsm constraint
with state variable '<state-variable>' (Module: <module-name>)
is to be <removed|appended> (All previous specification for
same transition ignored)
```

#### *Potential Issues*

This violation appears if a state value specified by the `-state_values` argument of the *fsm* constraint is already specified to be removed (`-remove`) or appended (`-append`) by a previous *fsm* constraint.

See *Example 7 - FSMs with Duplicate State Values*.

#### *Consequences of Not Fixing*

If you do not fix this violation then other than the reported *fsm* constraint, all the *fsm* constraints declared before the reported constraint with the same state value are ignored from SpyGlass analysis.

#### *How to Debug and Fix*

Remove the *fsm* constraint with duplicate state value.

### Message 8

The following message appears:

```
[WARNING] Fsm constraint with state variable '<state-variable>'
(Module: <module-name>) completely ignored (Reason: State
variable not found in the module)
```

#### *Potential Issues*

This violation appears if the state variable specified by the `-state_variables` argument of the *fsm* constraint is missing in the design.

See *Example 8 - Missing State Variable*.

#### *Consequences of Not Fixing*

The reported constraint is ignored from SpyGlass analysis.

#### *How to Debug and Fix*

Specify a valid state variable that exists in the design.

## Message 9

The following message appears:

```
[WARNING] Fsm constraint with state variable '<state-variable>'
(Module: <module-name>) partially ignored (Reason: State Value
'<state-value>' specified to be appended is SpyGlass auto-
detected or duplicate user-specification)
```

### *Potential Issues*

This violation appears if the FSM state value to be added is automatically detected by SpyGlass or different width state values evaluating to the same logical value are specified by the user.

See *Example 9 - State Variable Automatically Detected By SpyGlass*.

### *Consequences of Not Fixing*

The reported constraint is partially ignored from SpyGlass analysis.

`-state_values` can take multiple inputs, but this violation is reported only for individual input. Therefore, the constraint is partially ignored for a particular state value.

### *How to Debug and Fix*

Remove the reported state variable from the FSM.

## Message 10

The following message appears:

```
[WARNING] Fsm constraint with state variable '<state-variable>'
(Module: <module-name>) partially ignored (Reason: State Value
'<state-value>' specified to be removed is not SpyGlass auto-
detected or duplicate user-specification)
```

### *Potential Issues*

This violation appears if the FSM state value to be removed is not automatically detected by SpyGlass or different width state values evaluating to the same logical value are specified by the user.

See *Example 10 - State Variable Not Detected Automatically By SpyGlass*.

#### *Consequences of Not Fixing*

The reported constraint is partially ignored from SpyGlass analysis.

`-state_values` can take multiple inputs, but this violation is reported only for individual input. Therefore, the constraint is partially ignored for a particular state value.

#### *How to Debug and Fix*

The reported constraint is partially ignored from SpyGlass analysis.

#### *How to Debug and Fix*

Remove the reported state variable from the FSM.

### Message 11

The following message appears:

```
[WARNING] Fsm constraint with state variable '<state-variable>'
(Module: <module-name>) partially ignored (Reason: Transition
from  state value '<state-value1>' to state value '<state-
value2>' specified to be appended is SpyGlass auto-detected or
duplicate user-specification)
```

#### *Potential Issues*

This violation appears if the FSM transition to be added is automatically detected by SpyGlass or different width state values of transition evaluating to the same logical transition are specified by the user.

See *Example 11 - FSM Transition Automatically Detected By SpyGlass*.

#### *Consequences of Not Fixing*

The reported constraint is partially ignored from SpyGlass analysis.

There can be multiple transitions, but this violation is reported only for individual transition. Therefore, the constraint is partially ignored for a particular transition.

#### *How to Debug and Fix*

Remove the reported FSM transition from the *fsm* constraint.

## Message 12

The following message appears:

**[WARNING]** `Fsm constraint with state variable '<state-variable>' (Module: <module-name>) partially ignored (Reason: Transition from  state value '<state-value1>' to state value '<state-value2>' specified to be removed is not SpyGlass auto-detected or duplicate user-specification)`

### *Potential Issues*

This violation appears if the FSM transition to be removed is not automatically detected by SpyGlass or different width state values of transition evaluating to the same logical transition are specified by the user.

See *Example 12 - FSM Transition Not Automatically Detected By spyGlass*.

### *Consequences of Not Fixing*

The reported constraint is partially ignored from SpyGlass analysis.

There can be multiple transitions, but this violation is reported only for individual transition. Therefore, the constraint is partially ignored for a particular transition.

### *How to Debug and Fix*

Remove the reported FSM transition from the *fsm* constraint.

## Message 13

The following message appears:

**[WARNING]** `Fsm constraint with logical name '<fsm-name>' mapped to state variable '<state-variable>' of module '<module-name>' (All previous specifications of Fsm constraints with same state variable and module name ignored)`

### *Potential Issues*

This violation appears if duplicate *fsm* constraint specifications with the same state variable and module.

See *Example 13 - FSM Logical Name Mapped to a State Variable*.

### *Consequences of Not Fixing*

The last specification is considered and all the remaining duplicate specifications are ignored from SpyGlass analysis.

#### *How to Debug and Fix*

Remove duplicate *fsm* constraint specifications with the same state variable and module.

### Message 14

The following message appears:

`[WARNING]` `Fsm constraint ignored (Reason: Missing logical name, module name and state variable)`

#### *Potential Issues*

This violation appears if an *fsm* constraint is defined without the `-name` argument or without the combination of the `-module` and `-state_variables` arguments.

See *Example 14 - Missing Mandatory Arguments*.

#### *Consequences of Not Fixing*

The reported constraints are ignored from SpyGlass analysis.

#### *How to Debug and Fix*

Specify either the `-name` argument or the combination of the `-module` and `-state_variables` arguments to the *fsm* constraint.

### Message 15

The following message appears:

`[WARNING]` `Fsm constraint with state variable '<state-variable>' (Module: <module-name>) completely ignored (Reason: All state variables specified in constraint are not outputs of sequential element)`

#### *Potential Issues*

This violation appears if all the state variables specified to an *fsm* constraint are not the output of any sequential element.

See *Example 15 - FSM State Variable is Not the Output of a Sequential Element*.

#### *Consequences of Not Fixing*

The reported constraints are ignored from SpyGlass analysis.

#### *How to Debug and Fix*

Update the `-state_variables` argument of the reported *fsm* constraint to specify the state variables that are the output of a sequential element.

### Message 16

The following message appears:

```
[WARNING] Fsm constraint with state variable '<state-variable>'
(Module: <module-name>) completely ignored (Reason: FSM
specified to be removed is not auto-detected by SpyGlass)
```

#### *Potential Issues*

This violation appears if the *fsm* constraint is ignored from SpyGlass analysis because the specified FSM is not automatically detected by SpyGlass.

#### *Consequences of Not Fixing*

The reported constraint is ignored from SpyGlass analysis.

#### *How to Debug and Fix*

Remove the reported *fsm* constraint.

### Message 17

The following message appears:

```
[WARNING] Fsm constraint with state variable '<state-variable>'
(Module: <module-name>) completely ignored (Reason: FSM
specified to be added has no valid state or transition
specification)
```

#### *Potential Issues*

This violation appears if all the state variables specified to an *fsm* constraint is ignored from SpyGlass analysis because the specified *fsm* constraint does not have a valid state/transition specification.

#### *Consequences of Not Fixing*

The reported constraint is ignored from SpyGlass analysis.

#### *How to Debug and Fix*

Add a valid state/transition specification to the reported *fsm* constraint.

## Example Code and/or Schematic

This section has the following examples:

- *Example 1 - Missing Module and State Variable for an FSM*
- *Example 2 - Same Logical Name for Multiple FSMs*
- *Example 3 - Invalid State Value of an FSM*
- *Example 4 - FSMs with Duplicate State Values*
- *Example 5 - Invalid State Value To the -from_state_value Argument*
- *Example 6 - Invalid State Value To the -to_state_value Argument*
- *Example 7 - FSMs with Duplicate State Values*
- *Example 8 - Missing State Variable*
- *Example 9 - State Variable Automatically Detected By SpyGlass*
- *Example 10 - State Variable Not Detected Automatically By SpyGlass*
- *Example 11 - FSM Transition Automatically Detected By SpyGlass*
- *Example 12 - FSM Transition Not Automatically Detected By spyGlass*
- *Example 13 - FSM Logical Name Mapped to a State Variable*
- *Example 14 - Missing Mandatory Arguments*
- *Example 15 - FSM State Variable is Not the Output of a Sequential Element*

### Example 1 - Missing Module and State Variable for an FSM

Consider the following constraints:

```
fsm -name myFsm_3 -remove
fsm -name myFsm_3 -module block_fsm -state_variables state
```

For the above specifications, the *SGDC_fsm_setup01* rule reports *Message 1* because of a missing module name and a state variable in the first constraint specification.

### Example 2 - Same Logical Name for Multiple FSMs

Consider the following constraints:

```
fsm -name myFsm_3 -module top -state_variables state
-from_state_value 10 -to_state_value 11
```

```
fsm -name myFsm_3 -module block_fsm -state_variables state
-remove
```

For the above specifications, the *SGDC_fsm_setup01* rule reports *Message 2*, as shown below:

```
Fsm constraint with logical name 'myFsm_3' mapped to state
variable 'state' of module 'block_fsm' (All previous
specifications of Fsm constraints with same logical name
ignored)
```

### Example 3 - Invalid State Value of an FSM

Consider the following constraints:

```
fsm -name myFsm_1 -module Fsm -state_variables state1[2]
state1[1] state1[0]
```

```
fsm -name myFsm_1 -state_values d2 d4
```

```
fsm -name myFsm_1 -from_state_value d7 -to_state_value d3
```

For the above specifications, the *SGDC_fsm_setup01* rule reports the following messages (*Message 3*), as shown below:

```
State value 'd2' specified in -state_values field of Fsm
constraint with state variable 'state1[2] state1[1] state1[0]'
(Module: Fsm) ignored (Reason: Invalid state value)
```

```
State value 'd4' specified in -state_values field of Fsm
constraint with state variable 'state1[2] state1[1] state1[0]'
(Module: Fsm) ignored (Reason: Invalid state value)
```

### Example 4 - FSMs with Duplicate State Values

Consider the following constraints:

```
fsm -name myFsm_1 -module Fsm -state_variables state1[2]
state1[1] state1[0] -state_values 010
```

```
fsm -name myFsm_1 -state_values 111
```

```
fsm -name myFsm_1 -state_values 111 -remove
```

For the above specifications, the *SGDC_fsm_setup01* rule reports the following message (*Message 4*):

```
State value '111' specified in -state_values field of Fsm
constraint with state variable 'state1[2] state1[1] state1[0]'
(Module: Fsm) is to be removed (All previous specification for
same state value ignored)
```

### Example 5 - Invalid State Value To the -from_state_value Argument

Consider the following constraints:

```
fsm -name myFsm_1 -module Fsm -state_variables state1[2]
state1[1] state1[0]
```

```
fsm -name myFsm_1 -state_values d2 d4
```

```
fsm -name myFsm_1 -from_state_value d7 -to_state_value d3
```

For the above specifications, the *SGDC_fsm_setup01* rule reports the following message (*Message 5*):

```
Transition specified with state value 'd7' in -from_state_value
field of Fsm constraint with state variable 'state1[2]
state1[1] state1[0]' (Module: Fsm) ignored (Reason:  Invalid
state value)
```

### Example 6 - Invalid State Value To the -to_state_value Argument

Consider the following constraints:

```
fsm -name myFsm_1 -module Fsm -state_variables state1[2]
state1[1] state1[0]
```

```
fsm -name myFsm_1 -state_values "AB"
fsm -name myFsm_1 -state_values 11ABCDEF0
fsm -name myFsm_1 -from_state_value FF -to_state_value 110
fsm -name myFsm_1 -from_state_value 110 -to_state_value 250A
fsm -name myFsm_1 -from_state_value 110 -to_state_value S456
fsm -name myFsm_1 -from_state_value F1 -to_state_value 110
fsm -name myFsm_1 -from_state_value F1 -to_state_value
WACHIRA
```

For the above specifications, the *SGDC_fsm_setup01* rule reports the following message (*Message 6*):

```
Transition specified with state value '250A' in -to_state_value
field of Fsm constraint with state variable 'state1[2]
state1[1] state1[0]' (Module: Fsm) ignored (Reason: Invalid
state value)

Transition specified with state value 'S456' in -to_state_value
field of Fsm constraint with state variable 'state1[2]
state1[1] state1[0]' (Module: Fsm) ignored (Reason: Invalid
state value)
```

### Example 7 - FSMs with Duplicate State Values

Consider the following constraints:

```
fsm -name myFsm_1 -module Fsm -state_variables state1[2]
state1[1] state1[0]
fsm -name myFsm_1 -from_state_value 000 -to_state_value 111
fsm -name myFsm_1 -from_state_value 000 -to_state_value 111
-remove
```

For the above specifications, the *SGDC_fsm_setup01* rule reports the following message (*Message 7*):

```
Transition specified from state value '000' to state value
'111' for Fsm constraint with state variable 'state1[2]
state1[1] state1[0]' (Module: Fsm) is to be removed (All
previous specification for same transition ignored)
```

### Example 8 - Missing State Variable

Consider the following constraints:

```
fsm -name myFsm_3 -module block_fsm -state_variables
s_t_a_t_e -remove
```

For the above specifications, the *SGDC_fsm_setup01* rule reports the following message (*Message 8*):

```
Fsm constraint with state variable 's_t_a_t_e' (Module:
block_fsm) completely ignored (Reason: State variable not found
in the module)
```

#### Example 9 - State Variable Automatically Detected By SpyGlass

Consider the following files specified for SpyGlass analysis:

**input_RTL.v**

```
`define S0 3'b001
`define S1 3'b010
`define S2 3'b100
`define S3 3'b000
module Fsm(clk1, clk2,ctl, rst, state,
           state1, out);
  input clk1, clk2,ctl, rst;
  output [3:0] state, state1;
  output out;
  reg [3:0] state, state1;
  reg a, b, out ;
  always@(posedge clk1)
      b <= 0;
  always@(posedge clk2)
      a <= 1;
  always@(posedge clk2 or negedge rst)
  if(!rst)
     begin
      state1 <= `S0;
      out <= 0;
     end
  else begin
    case({state1[2],state1[1],state1[0]})
    `S0 : state1 <= `S1;
    `S1 : out <= 1;
    `S2 : if (ctl)
           state1 <= `S1;
    `S3 : begin
          if (ctl)
              state1 <= `S2;
  else
      state1 <= `S0;
  end
     default     : ;    // Do nothing
   endcase
  end
  endmodule
```

**SGDC**

```
....
...
fsm -name myFsm_1 -module Fsm -state_variables
state1[2] state1[1] state1[0]
-state_values 010 1010
fsm -name myFsm_1 -module Fsm -state_variables
 state1[2] state1[1] state1[0] -state_values111

fsm -name myFsm_1 -module Fsm -state_variables
 state1[2] state1[1] state1[0] -state_values001
 -remove
```

For the above example, the *SGDC_fsm_setup01* rule reports the following message (*Message 9*):

```
Fsm constraint with state variable 'state1[2] state1[1]
state1[0]' (Module: Fsm) partially ignored (Reason: State Value
'010' specified to be appended is auto-detected by SpyGlass)
```

### Example 10 - State Variable Not Detected Automatically By SpyGlass

Consider the following constraints:

```
fsm -name myFsm_3 -module block_fsm -state_variables state
-remove

fsm -name myFsm_3 -state_values 11

fsm -name myFsm_3 -state_values 00 100 -remove

fsm -name myFsm_3 -from_state_value 11 -to_state_value 01

fsm -name myFsm_3 -from_state_value 10 -to_state_value 10
-remove

fsm -name myFsm_4 -module new_fsm -state_variables state3
-remove

fsm -name myFsm_4 -state_values 11

fsm -name myFsm_4 -state_values 01 -remove

fsm -name myFsm_4 -from_state_value 10 -to_state_value 11
-append

fsm -name myFsm_4 -from_state_value 00 -to_state_value 00
-remove
```

For the above specifications, the *SGDC_fsm_setup01* rule reports the following messages (*Message 10*):

```
Fsm constraint with state variable 'state' (Module: block_fsm)
partially ignored (Reason: State Value '00' specified to be
removed is not auto-detected by SpyGlass)

Fsm constraint with state variable 'state3' (Module: new_fsm)
partially ignored (Reason: State Value '01' specified to be
removed is not auto-detected by SpyGlass)
```

### Example 11 - FSM Transition Automatically Detected By SpyGlass

Consider the following constraints:

```
fsm -name myFsm_3 -module block_fsm -state_variables state

fsm -name myFsm_3 -from_state_value 10 -to_state_value 10
fsm -name myFsm_3 -from_state_value 110 -to_state_value 010
fsm -name myFsm_4 -module new_fsm -state_variables state3
fsm -name myFsm_4 -from_state_value 00 -to_state_value 00
```

For the above specifications, the *SGDC_fsm_setup01* rule reports the following messages (*Message 11*):

```
Fsm constraint with state variable 'state' (Module: block_fsm)
partially ignored (Reason: Transition from  state value '10' to
state value '10' specified to be appended is auto-detected by
SpyGlass)

Fsm constraint with state variable 'state3' (Module: new_fsm)
partially ignored (Reason: Transition from  state value '00' to
state value '00' specified to be appended is auto-detected by
SpyGlass)
```

### Example 12 - FSM Transition Not Automatically Detected By spyGlass

Consider the following constraints:

```
fsm -name myFsm_3 -module block_fsm -state_variables state
fsm -name myFsm_3 -from_state_value 11 -to_state_value 01
-remove
fsm -name myFsm_3 -from_state_value 111 -to_state_value 101
-remove
fsm -name myFsm_4 -module new_fsm -state_variables state3
fsm -name myFsm_4 -from_state_value 10 -to_state_value 11
-remove
```

For the above specifications, the *SGDC_fsm_setup01* rule reports the following messages (*Message 12*):

```
Fsm constraint with state variable 'state' (Module: block_fsm)
partially ignored (Reason: Transition from  state value '11' to
state value '01' specified to be removed is not auto-detected
by SpyGlass)

Fsm constraint with state variable 'state3' (Module: new_fsm)
partially ignored (Reason: Transition from  state value '10' to
state value '11' specified to be removed is not auto-detected
by SpyGlass)
```

### Example 13 - FSM Logical Name Mapped to a State Variable

Consider the following constraints:

```
fsm -name myFsm_1 -module Fsm -state_variables state

fsm -name myFsm_2 -module Fsm -state_variables state
```

For the above specifications, the *SGDC_fsm_setup01* rule reports the following messages (*Message 13*):

```
Fsm constraint with logical name 'myFsm_2' mapped to state
variable 'state' of module 'Fsm' (All previous specifications
of Fsm constraints with same state variable and module name
ignored)
```

### Example 14 - Missing Mandatory Arguments

Consider the following constraints:

```
fsm

fsm -remove

fsm -append

fsm -state_values 001

fsm -state_values 001 -remove

fsm -state_values 001 -append

fsm -from_state_value 001 -to_state_value 110

fsm -from_state_value 001 -to_state_value 110 -append

fsm -from_state_value 001 -to_state_value 110 -remove
```

For each of the above specifications, the *SGDC_fsm_setup01* rule reports the following message (*Message 14*):

```
Fsm constraint ignored (Reason: Missing logical name, module
name and state variable)
```

#### Example 15 - FSM State Variable is Not the Output of a Sequential Element

Consider the following constraints:

```
fsm -name myFsm_1 -module Fsm -state_variables sys_if_state
```

For the above specification, the *SGDC_fsm_setup01* rule reports the following messages (*Message 15*):

```
Fsm constraint with state variable 'sys_if_state' (Module: Fsm)
completely ignored (Reason:Is Not  All state variables
specified in constraint are not outputs of sequential element)
```

## Default Severity Label

Warning

## Reports and Related Files

None

# The OVL Support

This section describes the OVL assertions as checked by the *Av_ovl01* rule.

It covers the following topics:

- *Common Assertion Arguments*
- *OVL Assertions*

## Common Assertion Arguments

SpyGlass Auto Verify solution supports both OVL 1.0 and OVL 2.0. The following table lists commonly used OVL parameters:

| **Parameters** | **Purpose** |
|---|---|
| *severity_level* | Severity of the failure with default value of 0. |
| *options* | Vendor options. Currently, the only supported option is *options*=1, which defines the assertion as a constraint on formal tools. The default value is *options*=0, or no options specified.<br>**Note:** In OVL 2.0, the options parameter has been renamed to property_type. |
| *msg* | Error message that will be printed if the assertion fires. |

**NOTE:** *In OVL 2.0, the* `flag` *parameter has been renamed to* `action_on_new_start`*.*

The following table lists the commonly used OVL ports:

| **Ports** | **Purpose** |
|---|---|
| *clk* | Triggering or clocking event that monitors the assertion. |
| *reset_n* | Signal indicating completed initialization (for example, a local copy of *reset_n* of a global reference to *reset_n*). |
| *test_expr* | Expression being verified at the positive edge of *clk*. |

## OVL Assertions

This section describes the following commands:

| | | |
|---|---|---|
| *assert_always* | *assert_always_on_edge* | *assert_change* |
| *assert_cycle_sequence* | *assert_decrement* | *assert_delta* |
| *assert_even_parity* | *assert_fifo_index* | *assert_frame* |
| *assert_handshake* | *assert_implication* | *assert_increment* |
| *assert_never* | *assert_next* | *assert_no_overflow* |
| *assert_no_transition* | *assert_no_underflow* | *assert_odd_parity* |
| *assert_one_cold* | *assert_one_hot* | *assert_proposition* |
| *assert_quiescent_state* | *assert_range* | *assert_time* |
| *assert_transition* | *assert_unchange* | *assert_width* |
| *assert_win_change* | *assert_win_unchange* | *assert_window* |
| *assert_zero_one_hot* | | |

## assert_always

### Declaration

```
assert_always
  [#(severity_level, options, msg)]
  inst_name(clk, reset_n, test_expr);
```

### SpyGlass Handling

The *Av_ovl01* rule reports test expressions *test_expr* that are not always evaluated TRUE at every positive edge of the triggering event or clock *clk*.

## assert_always_on_edge

### Declaration

```
assert_always_on_edge
  [#(severity_level, edge_type, options, msg)]
  inst_name (clk, reset_n, sampling_event, test_expr);
```

Where:

- *edge_type*:
  - 0: no edge
  - 1: positive edge
  - 2: negative edge
  - 3: any edge
- *sampling_event*: Expression that defines when to evaluate *test_expr*. Transition of *sampling_event* must match transition selected by *edge_type* in order for *test_expr* to be evaluated.

### SpyGlass Handling

The *Av_ovl01* rule reports test expressions *test_expr* that are not always evaluated TRUE at the edge of an event.

## assert_change

### Declaration

```
assert_change
   [#(severity_level, width, num_cks, flag, options, msg)]
  inst_name(clk, reset_n, start_event, test_expr);
```

Where:

- *width*: width of test expression *test_expr*
- *num_cks*: is the number of cycles, from a start signal activation, within which the test expression is supposed to change value
- *flag*:
  - 0: Ignore any subsequent start while monitoring the test signal starting from a first start signal (default).
  - 1: Each start signal occurrence will cause the monitoring to restart.
  - 2: The property fails when a start occurs while monitoring a test signal due to a previous activation of the start signal.
- *start_event*: Event that triggers monitoring of the *test_expr*.

### SpyGlass Handling

The *Av_ovl01* rule reports those test expressions, *test_expr*, that do not change value (0 to 1 or 1 to 0) within specified number of cycles, *num_cks*, of a starting event.

Check the width within which the test expression has changed value for correctness.

## assert_cycle_sequence

### Declaration

```
assert_cycle_sequence
   [#(severity_level, num_cks, necessary_condition, options,
msg)]
  inst_name (clk, reset_n, event_sequence);
```

Where:

- *num_cks*: The number of cycles the analysis is covering.

  The maximum number supported clock cycles is 64.
- *necessary_condition*: 2'b00, 2'b01, 2'b10, or 2'b11. The default is 2'b00.
- *event_sequence*: A Verilog or VHDL concatenation expression, where each bit represents an event.

### SpyGlass Handling

The *Av_ovl01* rule reports user-defined sequencing of events *event_sequence* that are not followed correctly during functional checking.

The `assert_cycle_sequence` assertion checks the following:

- If *necessary_condition* is 2'b00 or 2'b10:

  This assertion checks to ensure that if all *num_cks-1* first events of *event_sequence* are true, then the last one (`event_sequence[0]`) must occur. The check is done in pipe-lined mode.
- If *necessary_condition* is 2'b01:

  This assertion checks to ensure that once the first event (`event_sequence[num_cks-1]`) occurs, all the remaining events occur. The check is done in pipe-lined mode.
- If *necessary_condition* is 2'b11:

  This assertion checks to ensure that once the first event (`event_sequence[num_cks-1]`) occurs, all the remaining events occur. The check is done in non pipe-lined mode.

**NOTE:** *In pipe-lined mode, the first events are checked repeatedly for a match. If a new match is found, a check is started again.*

## assert_decrement

### Declaration

```
assert_decrement
  [#(severity_level, width, value, options, msg)]
  inst_name(clk, reset_n, test_expr);
```

Where:

- *width*: width of test expression *test_expr*
- *value*: The value by which the *test_expr* is supposed to decrease.

### SpyGlass Handling

The *Av_ovl01* rule reports test expressions *test_expr* whose actual value is decreased by any number other than the specified decrement value.

## assert_delta

### Declaration

```
assert_delta
  [#(severity_level, width, min, max, options, msg)]
  inst_name(clk, reset_n, test_expr);
```

Where:

- *width*: width of test expression *test_expr*
- *min*: Minimum change in value
- *max*: Maximum change in value

### SpyGlass Handling

The *Av_ovl01* rule reports test expressions *test_expr* whose actual value changes by a number outside the specified range.

## assert_even_parity

### Declaration

```
assert_even_parity
  [#(severity_level, width, options, msg)]
  inst_name (clk, reset_n, test_expr);
```

Where:

- *width*: width of test expression *test_expr*

### SpyGlass Handling

The *Av_ovl01* rule reports test expressions *test_expr* where odd number of bits are asserted at any time.

## assert_fifo_index

### Declaration

```
assert_fifo_index
  [#(severity_level, depth, push_width, pop_width, options,
msg)]
  inst_name(clk, reset_n, push, pop);
```

Where:

- *depth*: Depth of the FIFO, which is the maximum number of pushes allowed
- *push_width*: The width of the push signal. Maybe greater than one if multiple pushes are allowed in one cycle
- *pop_width*: The width of the pop signal. Maybe greater than one if multiple pops are allowed in one cycle
- *push*: The value of *push* indicates the number of writes that are occurring on that particular clock cycle. The *push_width* defines the width of the push expression. By default, only a single write can be performed on a particular clock cycle.
- *pop*: The value of *pop* indicates the number of reads that are occurring on that particular clock cycle. The *pop_width* defines the width of the pop expression. By default, only a single read can be performed on a particular clock cycle.

### SpyGlass Handling

The *Av_ovl01* rule reports FIFOs that either overflow or underflow.

The Av_ovl01 rule is not validating a FIFO but rather the environment of FIFO for compliance with FIFO's attributes. The Av_ovl01 rule is violated in both cases when there is a write into a FIFO which is already full, or when a read from an empty FIFO is requested.

## assert_frame

### Declaration

```
assert_frame
  [#(severity_level, min_cks, max_cks, flag, options, msg)]
  inst_name(clk, reset_n, start_event, test_expr);
```

Where:

- *min_cks*: the test signal must occur after *min_cks* cycles. If *min_cks* is 0 then the check ensures that test signal is occurring before *max_cks*, however it may happen at the same time as start.
- *max_cks*: the test signal must occur before max_clk cycles. If 0 then test signal must occur at the same time as the start signal. *max_cks* must be greater than or equal to *min_cks*.
- *flag*: if 0 then ignore any subsequent start while monitoring the test signal starting from a first start signal (default). If 1 then each start signal occurrence will cause the monitoring to restart. If 2 then the assertion fails when a start occurs while monitoring a test signals due to a previous activation of the start signal.
- *start_event*: Starting event that triggers monitoring of the *test_expr*. The *start_event* is a cycle transition from 0 to 1.

### SpyGlass Handling

The *Av_ovl01* rule reports test expressions *test_expr* that are not activated within a minimum *min_cks* and maximum *max_cks* number of cycles when the start signal is high (*start_event*).

## assert_handshake

### Declaration

```
assert_handshake
  [#(severity_level, min_ack_cycle, max_ack_cycle,
      req_drop, deassert_count, max_ack_length, options,
msg)]
  inst_name(clk, reset_n, req, ack);
```

Where:

- *min_ack_cycle*: ack is expected to occur at or after min_ack_cycle cycles.
- *max_ack_cycle*: ack is expected to occur at or before max_ack_cycle cycles.
- *req_drop*: Check if req is active until ack occurs
- *deassert_count*: req is expected to be deactivated deassert_count cycles after ack arrival.
- *max_ack_length*: ack is expected to be max_ack_cycle cycles wide or less. Also check if req is active for the entire deassert_count cycles after ack.
- *req*: Signal that starts the transaction.
- *ack*: Signal that terminates the transaction.

### SpyGlass Handling

The *Av_ovl01* rule reports handshaking problems with the request and acknowledge signals of a protocol.

The check ensures that acknowledge is occurring within a defined range of cycles after a request has been sent; checks the acknowledge width against its spec; if required by the user, ensures that the request is active until arrival of acknowledge and remain active for specified number of cycles after arrival of acknowledge; also ensures that the request is inactivated within a specified number of cycles from acknowledge activation. Both request and acknowledge signals must go inactive before a handshake validation starts.

## assert_implication

### Declaration

```
assert_implication
  [#(severity_level, options, msg)]
  inst_name(clk, reset_n, antecedent_expr, consequent_expr);
```

Where

- *antecedent_expr*: Expression verified at the positive edge of *clk*.
- *consequent_expr*: Expression verified if *antecedent_expr* is TRUE.

### SpyGlass Handling

The *Av_ovl01* rule reports “consequence” expressions *consequent_expr* that are not evaluated TRUE after an “antecedent” expression *antecedent_expr* has become TRUE.

Please note that the `assert_implication` assertion can be also validated using the following statement:

```
assert_always
  imply(clk, consequent_expr || !antecedent_expr);
```

## assert_increment

### Declaration

```
assert_increment
  [#(severity_level, width, value, options, msg)]
  inst_name(clk, reset_n, test_expr);
```

Where:

- *width*: width of test expression *test_expr*
- *value*: The value by which the *test_expr* is supposed to increase.

### SpyGlass Handling

The *Av_ovl01* rule reports test expressions *test_expr* whose actual value is increased by any number other than the specified increment value.

## assert_never

### Declaration

```
assert_never
  [#(severity_level, options, msg)]
  inst_name (clk, reset_n, test_expr);
```

### SpyGlass Handling

The *Av_ovl01* rule reports test expressions *test_expr* that are evaluated to be TRUE.

## assert_next

### Declaration

```
assert_next
  [#(severity_level, num_cks, check_overlapping, only_if,
      options, msg)]
  inst_name(clk, reset_n, start_event, test_expr);
```

Where:

- *num_cks*, Number of clock cycles after start event at which test expression must be evaluated ‘1’
- *check_overlapping*, if ‘1’, allows another check to start upon a new start pulse while the first check is continuing.
- *only_if*, if ‘1’ causes a failure if the test expression is evaluated true without a prior start event.
- *start_event*: Starting event that triggers monitoring of the *test_expr*.

### SpyGlass Handling

The *Av_ovl01* rule reports test expressions *test_expr* where no event happens exactly after the specified number of cycles *num_cks* counted from the start event *start_event*.

## assert_no_overflow

### Declaration

```
assert_no_overflow
  [#(severity_level, width, min, max, options, msg)]
  inst_name (clk, reset_n, test_expr);
```

Where:

- *width*: width of test expression *test_expr*
- *min*: Lower bound value below which the test expression cannot take value while it transitions from max value.
- *max*: Upper bound value above which the test expression cannot take value while it transitions from max value.

### SpyGlass Handling

The *Av_ovl01* rule reports test expressions *test_expr* that changes value from a maximum value to another value outside the *min*-*max* range.

No message is reported if the value outside the *min*-*max* range is reached from a value different from the *max* value.

## assert_no_transition

### Declaration

```
assert_no_transition
  [#(severity_level, width, options, msg)]
  inst_name (clk, reset_n, test_expr, start_state,
next_state);
```

Where:

- *width*: width of test expression *test_expr*
- *start_state*: Source state of unwanted transition
- *next_state*: Destination transition of unwanted transition.

### SpyGlass Handling

The *Av_ovl01* rule reports test expressions *test_expr* that changes value from a given state *start_state* to another state *next_state*.

## assert_no_underflow

### Declaration

```
assert_no_underflow
  [#(severity_level, width, min, max, options, msg)]
  inst_name (clk, reset_n, test_expr);
```

Where:

- *width*: width of test expression *test_expr*
- *min*: Lower bound value below which the test expression cannot take value while it transitions from the *min* value.
- *max*: Upper bound value above which the test expression cannot take value while it transitions from the *min* value.

### SpyGlass Handling

The *Av_ovl01* rule reports test expressions *test_expr* that changes value from the *min* value to another value outside the *min*-*max* range.

No message is reported if the value outside the *min*-*max* range is reached from a value different from the *min* value.

## assert_odd_parity

### Declaration

```
assert_odd_parity
  [#(severity_level, width, options, msg)]
  inst_name (clk, reset_n, test_expr);
```

Where:

- *width*: width of test expression *test_expr*

### SpyGlass Handling

The *Av_ovl01* rule reports test expressions *test_expr* where an even number of bits are asserted at any time.

## assert_one_cold

### Declaration

```
assert_one_cold
  [#(severity_level, width, inactive, options, msg)]
  inst_name(clk, reset_n, test_expr);
```

Where:

- *width*: width of test expression *test_expr*
- *inactive*: If *inactive* is 0, then the *test_expr* must be one_cold or all 0. If *inactive* is 1, then the *test_expr* must be one_cold or all 1.

### SpyGlass Handling

The *Av_ovl01* rule reports test expressions *test_expr* that have been asserted to be one_cold encoded but are not modeled to be one_cold encoded.

The cause of this violation may be obvious if the signals are defined in a case statement with clear encoding. But the one_cold-encoded signals may be independent signals defined in different places in the RTL code. First, determine why you assumed the signals should be one_cold encoded. If the assumption is correct, then check why the signals are not one_cold encoded.

## assert_one_hot

### Declaration

```
assert_one_hot
  [#(severity_level, width, options, msg)]
  inst_name(clk, reset_n, test_expr);
```

Where:

- *width*: width of test expression *test_expr*

### SpyGlass Handling

The *Av_ovl01* rule reports test expressions *test_expr* that have been asserted to be one_hot encoded but are not modeled to be one_hot encoded.

The cause of this violation may be obvious if the signals are defined in a case statement with clear encoding. But the one_hot-encoded signals may be independent signals defined in different places in the RTL code. First, determine why you assumed the signals should be one_hot encoded. If the assumption is correct, then check why the signals are not one_hot encoded.

## assert_proposition

### Declaration

```
assert_proposition
  [#(severity_level, options, msg)]
  inst_name(reset_n, test_expr);
```

Where:

- *width*: width of test expression *test_expr*

### SpyGlass Handling

The *Av_ovl01* rule reports test expressions *test_expr* that are not always evaluated to be TRUE.

The `assert_proposition` assertion requires that the expression should always be TRUE whereas the *assert_always* assertion requires that the expression should be TRUE for all active edges of the clock.

## assert_quiescent_state

### Declaration

```
assert_quiescent_state
  [#(severity_level, width, options, msg)]
  inst_name(clk,reset_n, state_expr, check_value,
sample_event);
```

Where:

- *width*: width of state expression *state_expr*
- *state_expr*: The state signals
- *check_value*: The *state_expr* must be at this value at the edge of event
- *sample_event*: The event at which the check is performed

**NOTE:** *SpyGlass Auto Verify solution does not support Verilog macros specific to the* `assert_quiescent_state` *OVL assertion.*

### SpyGlass Handling

The *Av_ovl01* rule reports state expressions *state_expr* that are not in the state *check_value* at the edge of event *sample_event*.

## assert_range

### Declaration

```
assert_range
  [#(severity_level, width, min, max, options, msg)]
  inst_name(clk, reset_n, test_expr);
```

Where:

- *width*: width of test expression *test_expr*
- *min*: minimum value allowed for the test expression *test_expr*
- *max*: maximum value allowed for the test expression *test_expr*. Default is 2**width – 1.

### SpyGlass Handling

The *Av_ovl01* rule reports test expressions *test_expr* that do not always have a value in the *min*-*max* range.

## assert_time

### Declaration

```
assert_time
  [#(severity_level, num_cks, flag, options, msg)]
  inst_name(clk, reset_n, start_event, test_expr);
```

Where:

- *num_cks*: test expression must hold for that many cycles
- *flag*:
  - 0: Ignore any event once a first event has been started
  - 1: Restart the check whenever a new event is asserted
  - 2: Fail if a new event occurs after a first event has triggered the monitoring process
- *start_event*: Starting with this event the test expression must hold for the given number of cycles

### SpyGlass Handling

The *Av_ovl01* rule reports test expressions *test_expr* that are not asserted for *num_cks* cycles starting at the edge of event *start_event*.

## assert_transition

### Declaration

```
assert_transition
  [#(severity_level, width, options, msg)]
  inst_name(clk, reset_n, test_expr, start_state,
next_state);
```

Where:

- *width*: width of test expression *test_expr*
- *start_state*: Source state of the transition
- *next_state*: Destination state of the transition

### SpyGlass Handling

The *Av_ovl01* rule reports test expressions *test_expr* that have a transition out of state *start_state* to a state other than state *next_state*.

## assert_unchange

### Declaration

```
assert_unchange
  [#(severity_level, width, num_cks, flag, options, msg)]
  inst_name(clk, reset_n, start_event, test_expr);
```

Where:

- *width*: width of test expression *test_expr*
- *num_cks*: number of clock cycles after start event during which the test expression should remain unchanged
- *flag*:
    - 0: Ignore repetition of *start_event*
    - 1: Re-start with a new *start_event*
    - 2: Report violation if a new *start_event* is occurring while validating a previous sequence
- *start_event*: Event triggering observation of the test expression

### SpyGlass Handling

The *Av_ovl01* rule reports test expressions *test_expr* that change value within *num_cks* cycles after event *start_event*.

## assert_width

### Declaration

```
assert_width
  [#(severity_level, min_cks, max_cks, options, msg)]
  inst_name(clk, reset_n, test_expr);
```

Where:

- *min_cks*: The *test_expr* should remain TRUE for at least the specified minimum number of clock cycles. When *min_cks* is set to 0, then there is no minimum check (that is, test_expr may occur at start event).
- *max_cks*: The *test_expr* should not remain TRUE longer than the specified maximum number of clock cycles. When *max_cks* is set to 0, then there is no maximum check (any value is valid).

### SpyGlass Handling

The *Av_ovl01* rule reports test expressions *test_expr* that do not evaluate to TRUE for a specified minimum number of clock cycles and evaluate to TRUE for more than a maximum number of clock cycles.

## assert_win_change

### Declaration

```
assert_win_change
  [#(severity_level, width, options, msg)]
  inst_name(clk, reset_n, start_event, test_expr,
end_event);
```

Where:

- *width*: width of test expression *test_expr*
- *start_event*: starting event after which the test expression *test_expr* is supposed to change value
- *end_event*: End event before which the test expression *test_expr* is supposed to change value

### SpyGlass Handling

The *Av_ovl01* rule reports test expressions *test_expr* that changes value before the start event *start_event* or after the end event *end_event*.

## assert_win_unchange

### Declaration

```
assert_win_unchange
  [#(severity_level, width, options, msg)]
  inst_name(clk, reset_n, start_event, test_expr,
end_event);
```

Where:

- *width*: width of test expression *test_expr*
- *start_event*: starting event after which the test expression *test_expr* is not supposed to change value
- *end_event*: End event before which the test expression *test_expr* is not supposed to change value

### SpyGlass Handling

The *Av_ovl01* rule reports test expressions *test_expr* that changes value after the start event *start_event* or before the end event *end_event*.

## assert_window

### Declaration

```
assert_window
  [#(severity_level, options, msg)]
  inst_name(clk, reset_n, start_event, test_expr,
end_event);
```

Where:

- *start_event*: starting event after which (at the next clock tick) the test expression *test_expr* is supposed to hold true
- *end_event*: End event at the end of which the test expression *test_expr* is allowed to be false

### SpyGlass Handling

The *Av_ovl01* rule reports test expressions *test_expr* that do not always evaluate to TRUE from the start event *start_event* to the end event *end_event*.

## assert_zero_one_hot

### Declaration

```
assert_zero_one_hot
  [#(severity_level, width, options, msg)]
  inst_name(clk, reset_n, test_expr);
```

### SpyGlass Handling

The *Av_ovl01* rule reports test expressions *test_expr* that are neither one_hot encoded nor all “0”s.

# Appendix: SGDC Constraints

SpyGlass Design Constraints (SGDC) provides additional design information that is not apparent in an RTL.

In addition, you can restrict SpyGlass analysis to certain objects in a design by specifying these objects by using SGDC commands.

## SpyGlass Design Constraints

The following table lists the SGDC commands used by SpyGlass Auto Verify solution:

| **SpyGlass Auto Verify** | | |
|---|---|---|
| *breakpoint* | *clock* | *define_tag* |
| *formal_analysis_filter* | *ip_block* | *meta_design_hier* |
| *reset* | *set_case_analysis* | *special_module* |
| *simulation_data* | *watchpoint* | |

# List of Topics