Verification Continuum^{TM}

SpyGlass® Auto Verify Submethodology (for GuideWare 2019.06)

Version P-2019.06-SP1, September 2019

SYNOPSYS®

Contents

Preface

About This Book

The SpyGlass® Auto Verify sub-methodology guide describes the methodology of using the SpyGlass Auto Verify solution.

Contents of This Book

The SpyGlass Auto Verify sub-methodology guide has the following sections:

Section	Description
Introduction to SpyGlass Auto Verify Methodology	Provides introduction of SpyGlass Auto Verify solution.
RTL Issues Causing Functional Failures	Describes major problems related to RTL coding that are poorly covered by traditional verification solutions.
Using SpyGlass Auto Verify Methodology to Reduce Functional Failures	Describes step-by-step solution to reduce functional failures in a design by using the SpyGlass Auto Verify solution

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

Introduction to SpyGlass Auto Verify Methodology

Introduction

A typical RTL design contains complex control and data logic that is hard to be exhaustively verified. State machines, conditional control branches, and other design constructs perform complex functions needed in highly integrated heterogeneous systems. Ensuring the correctness of such systems is a difficult task out of the reach of a single verification tool.

Traditionally, design functionality is verified using basic lint, functional simulation, and formal verification. While lint can identify and report simple RTL issues, it is not intended to verify functionality of the design where multiple RTL constructs interact. On the other hand, simulation can identify functional bugs but is in no way exhaustive and can easily miss corner case issues. Assertion verification covers specific aspects of design functionality such as exhaustive protocol verification but requires in-depth formal expertise and significant time to write and verify assertions in a highly iterative process.

The need is to identify critical design intent automatically and exhaustively verifying it to catch corner case bugs that may not be identified by other tools in the verification flow.

This document introduces a methodology to perform automatic verification of critical design components such as finite-state machines (FSMs), case statements, and tri-state buses using the SpyGlass® tool suite. This document is useful to novice and advanced users of SpyGlass. Advanced users can proceed directly to the relevant sections of the document.

Tool and Methodology Version

- SpyGlass Version: Version P-2019.06-SP1
- GuideWare Version: 2019.06

References

- SpyGlass Auto Verify Rules Reference Guide
- SpyGlass Explorer User Guide

RTL Issues Causing Functional Failures

Designs described at the RTL level are subject to many functional failures due to incomplete specification, incorrect conceptual understanding of specification, and unintentional bugs introduced during RTL coding.

The following are the major problems related to RTL coding that are poorly covered by traditional verification solutions, such as functional simulation:

- *FSMs and Related Issues*
- *Redundant Logic*
- *X Generation*

FSMs and Related Issues

FSMs impact designs in two different ways:

- Correctness: State machines typically contain many states and a high number of transitions, inputs, and outputs. While writing RTL code, the user may not get a good picture of all states and their interconnections. This can lead to functional problems and consequently a chip failure. Although intent verification of the state machines requires knowledge of the design, many correctness aspects can be automatically verified. Common issues in state machines are:
    - ❒ Unreachable state: An unreachable state is a state in RTL code where the user has not created any transitions to reach it, or created transitions that cannot be exercised by the logic controlling it. An unreachable state indicates a functional problem or design redundancy.
    - ❒ Deadlock state: A deadlock state is a state from which no outgoing transitions exist or outgoing transitions are not exercisable due to control logic. When a state machine reaches such a state, it cannot transition to a different state.
    - ❒ Dead transition: A dead transition is a state machine transition that is present in RTL code but cannot be exercised. Dead transitions may cause deadlock or unreachable states.

    Designers must ensure that state machines present in their RTL are free of such bugs.

    An FSM illustrating the above issues is shown in *Figure 1*.

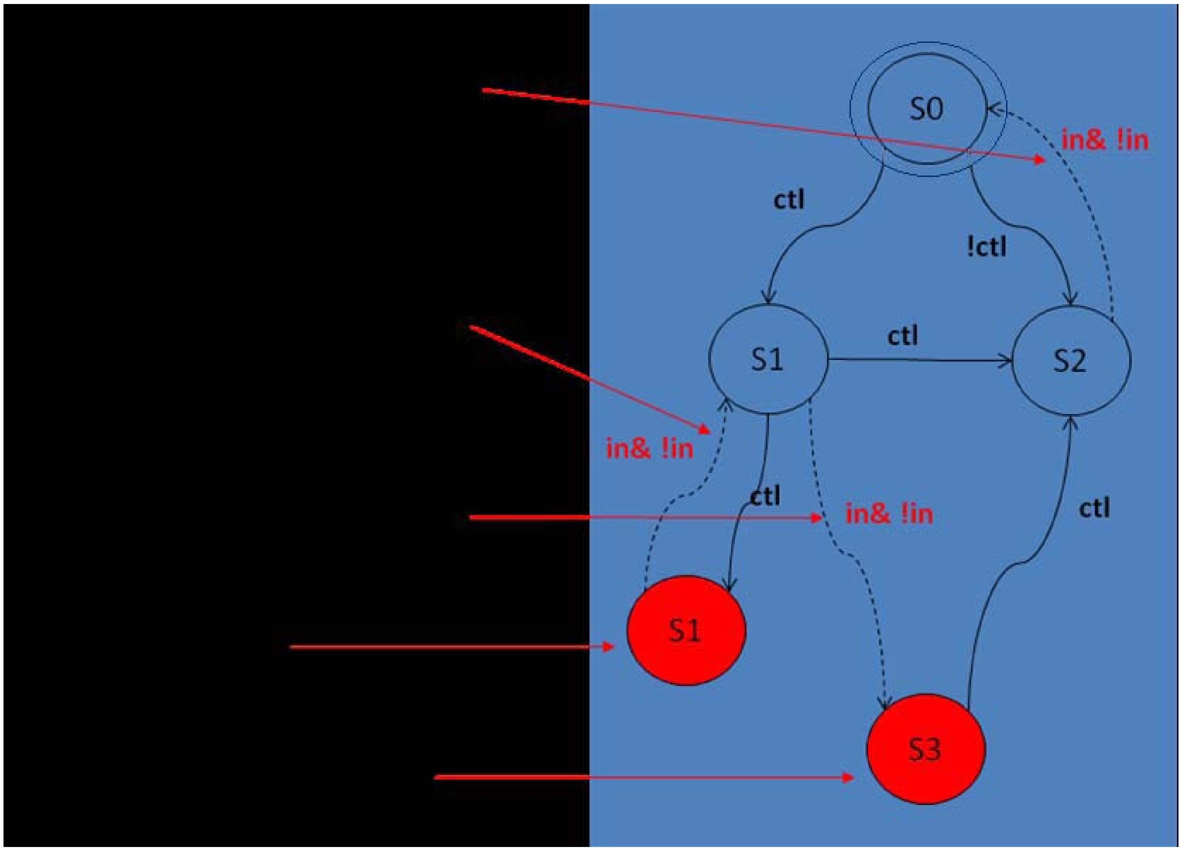

FIGURE 1. Functional issues with FSMs

- Implementation considerations:

  Several FSM attributes can be used to measure the quality of implementation in RTL. Examples of these attributes are:

  - the number of states, transitions, inputs and outputs
  - the depth
  - the encoding style
  - presence or absence of an initial state

  Designers must tune their design using these metrics to achieve the desired design objectives such as area, timing, and power goals.

For a detail discussion on the metrics impacting implementation and verification of state machines, refer to white paper “A Systematic Approach to Verifying FSMs”.

Since FSM are mostly implemented using case statements, many issues around the correct usage of these constructs. In fact, their usage entails pitfalls that can cause chip failures. We will discuss two problems that may arise in case statements: incompletely specified case statements and overlapping case statements.

A case statement lists all possible input values under which an operation is performed. What will be the behavior of a design if one possible input assignment is not specified in the case statement body? The following example illustrates an incompletely specified case statement.

```
always @(posedge clk2) begin
  case (bs)
    4'b0000: out <= 0;
    4'b1000: out <= 0;
    4'b1010: out <= 1;
    4'b0101: out <= 1;
    4'b1111: out <= 1;
  endcase
```

What would be the value of out when bs takes the value 1001? By default, synthesis tools preserve the previous value of out in this case. If synthesis tools are given the flexibility of assigning out to either 0 or 1 when bs = 1001, the circuit can be significantly optimized. When the user knows that bs never takes the value 1001, he can use the pragma, “synopsys full_case” to declare a case statement as “complete”; in this case any unspecified case value in the case statement body is a “don’t care”. When using this pragma, a designer can easily overlook design functionality and incorrectly declare case-statement to be completely specified. This is particularly true when a design is changed in the context of re-use.

Designers must verify that a case statement declared as “full_case” is completely specified by ensuring that any unspecified term is not produced in the design.

When multiple conditional branches are present and more than one condition is true at the same time, there is a contention of what branch should be executed. Synthesis tools introduce priority-decoding logic that will prioritize the first branch in the order of appearance in the RTL code. The following example illustrates overlapping case items and the issue described above. In this example, the item 1010 and 10x0 are overlapping. If bs takes the value 1010, out takes 0 or 1 depending on the branch chosen. In the presence of a priority encoder (introduced automatically by synthesis tools) the first case item, 10x0, will win and the output will be predictably set to 0.

```
// Case statement using bs assignments as case items

always @(posedge clk2) begin

casex (bs)

4'b0000: out <= 0;

4'b10x0: out <= 0;

4'b1010: out <= 1; // Overlaps with 10x0

4'b0101: out <= 1;

4'b1111: out <= 1;

endcase
```

If a case statement does not have overlapping items, the priority encoder inserted by synthesis tools is not required and the implementation can be optimized. Unfortunately, synthesis tools do not have the intelligence to determine whether case statements are overlapping or not. To overcome this shortcoming, the pragma, “synopsys parallel_case”, indicates that the item has no overlapping case statements. Using this pragma, synthesis tools can further optimize the implementation. Now, designers can easily mark a case statement with “parallel_case” while there are overlapping case items. This may particularly happen when the RTL is modified for re-use. When a case-statement has overlapping items and is marked as “parallel_case”, a functional failure can occur.

Designers must verify that case statements marked as “parallel_case” have no overlaps, or overlapping values are not reachable. For example if 10x0 and 1x00 appear in a test case but the case-statement variable cannot take 1000 then there will not be any issue in the design.

Redundant Logic

Complex RTL coding styles can hide design redundancies that are not apparent to the naked eyes. Unintentional redundancies may indicate serious functional problems. If these redundancies are not optimized during implementation, they can impact a chip’s quality (timing, area, power). Redundancies are not always local to given block or area of RTL code. *Figure 2* illustrates design redundancy caused by convergence of logic across a sequential layer of flip-flops. Once the clock is active, the signal and_out can be simplified to a constant 0 since one of the flip-flops flop1out_reg or flop2out_reg will store the value 0.

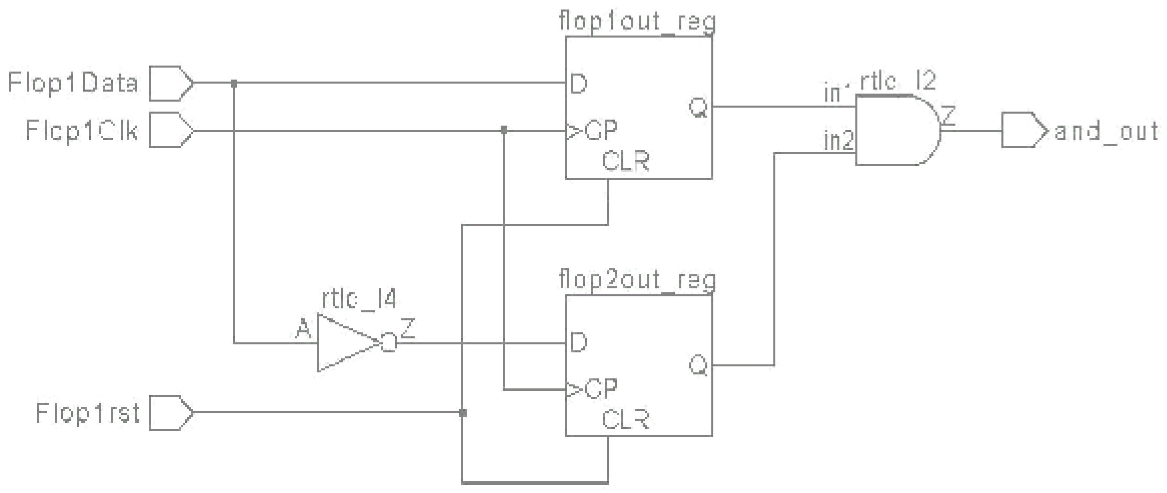

FIGURE 2. An example of redundant Logic driving the port and_out

This form of redundant logic is usually caused by branching constructs such as if-then-else and case-statements, which can break the data flow and result in dead code. Designers must ensure the absence of redundant RTL code of the form shown above.

X Generation

Arrays are often used in RTL code. When the array range is defined, the array must be always accessed within the given range. In a complex design, often arrays are accessed at a variable index, where the index is a complex expression. Improperly designed index logic can go out of bound causing chip failure.

The following RTL code illustrates a case of array bound violation when the counter, “count”, reaches the value 2.

```
module test (out1, out2, in1, in2, clk, reset);
output out1, out2;
input clk, reset;
input [2:0] in1, in2;
reg [2:0] out2;
reg [1:0] count;
assign out1=in1[count+1];
always @(posedge clk)
  begin
   if(reset == 0) count= 0;
   else if(count == 2'b10) count = 0;
   else count=count+1;
end
always
out2[count+1]=in2[count];
endmodule
```

Designers must verify that no arrays in the RTL are accessed out of defined range for the array.

Tristate buses can also result in Xs in the design. In tristate buses, multiple tristate gates with enables drive the same net. For the buses to function correctly, one and only one enable must be on at a time. Otherwise, either the bus will have no drivers or it will be multi-driven. Designers must verify that all buses in their design have one and only one driver at all times.

Using SpyGlass Auto Verify Methodology to Reduce Functional Failures

This section provides a step-by-step solution to reduce functional failures in a design by using the SpyGlass Auto Verify solution.

Using a systematic and step-by-step approach enables you to sign off automatic functional verification of important aspects of an early stage RTL design.

The following table shows the stages and their corresponding goals of this methodology:

Stage	**Summary**	**Goal**
Prerequisites for Functional Verification	Run design read, and specify design information, such as clocks, resets, design initialization information, and design modes.	None
Creating a Setup for Functional Verification	Generate clocks and resets for the design.	adv_lint_setup

Stage	Summary	Goal
Performing Design Audit	Check the quality of setup by checking the correctness and completeness of the design initial state and validating FSMs.	adv_lint_struct
Performing Functional Verification	Check problems related with FSMs, and identify dead code, static nets, and causes of x generation in the design.	adv_lint_verify

Prerequisites for Functional Verification

Before you start functional verification using SpyGlass Auto Verify solution, you need to ensure that design read is complete, and basic lint issues have been fixed.

You should also gather as much information about the design environment as possible. The following information about the design is important when you verify design functionality:

- Clocks: clock periods are important for functional verification and if the design specification has design clocks and their period then extract them for use during verification.
- Resets: resets and their active values are important for verification; gather any reset information from the design spec for use during verification
- Design initialization: design initial state, or how the design can be initialized is an important aspect of design functionality that need to be extracted for use during verification.
- Design modes: if you want to analyze the design for a given mode, then gather the information for verification setup.

Creating a Setup for Functional Verification

Run the *adv_lint_setup* goal to create a setup.

During setup, clocks and resets are identified in a design. This step is required if the clock and reset information is not available. If the design has been analyzed using SpyGlass-CDC, the setup information used during CDC verification can be reused for SpyGlass-Auto Verify analysis.

Performing Design Audit

Run the *adv_lint_struct* goal to validate the quality of setup, such as initialization details of registers and summary of properties in the design. It is a preparation step for identifying FSMs in the design and for getting an audit of all the properties that will be verified next.

In this stage, you can analyze the following about the design:

- Initialization: During power-on, a design is brought to an “initial state”. The correctness and completeness of initial state is important for verification. If the state computed is not a valid initial state then the verification result is not reliable. Correct initialization and its impact are further described in later sections in this document.
- Candidate checks for verification: During audit, all structures such as FSMs are extracted but they are not verified. Reviewing the total number of checks allows you to estimate the complexity of the design and estimate the run-time needed for verification.

Rerun the goal if RTL or constraints file are updated for missing clocks, resets, black-boxes description and low initialization. User can use following spyglass parameters for the design setup:

- Increase the value of ieffort parameter to improve quality of design initialization
- Clocks and resets must be reviewed and frequencies should be provided for proper verification. User can run SpyGlass Auto Verify solution by using automatic clock and reset detection using use_inferred_clocks and use_inferred_resets or the user can run the goal adv_lint_setup to identify clocks and resets in the design. The results of this step must be reviewed by the user.

Rules to Cover the Aspects of Verification Audit

The following rules of the SpyGlass Auto Verify solution cover the above aspects of verification audit.

Rule	Description
Av_Info_Case_Analysis	Highlights case-analysis settings
Av_initstate01*	Reports initial state of the design

Rule	Description
Av_report01*	Reports statistics of properties and functional constraints.
Av_fsminf01	Reports all the FSMs in the design
Av_fsminf02	Reports all interacting FSMs in a design.
Av_multitop01*	Abort in case of multiple top level design unit
Av_init01*	Reports initialization issues for the run
Av_sanity01*	Reports an error if there is any issue in the property file
Av_sanity02*	Reports all the nets in the design which have multiple drivers
Av_fsm02	Reports state transitions of an FSM which cannot be activated
Av_case01	Reports all the sensitizable case items, which are not specified.
Av_case02	Reports all the case statements which have overlapping case items
Av_deadcode01	Reports redundant logic in the design
Av_bus01	Reports all the bus contentions in the design
Av_bus02	Reports all the floating buses in the design
Av_dontcare01	Reports sensitizable X-assignments in a design
Av_range01	Reports array bound violation
Av_complexity01	Reports design characteristics and complexity for RTL modules and FSMs in the design

The rules marked with “*” are always on, which indicates that they will be run during future verification steps too. In case the setup or the design changes, and the change adversely impact the reset for verification then you will get setup violations by the above rules. Such violations should be fixed with higher priority.

Performing Functional Verification

Run the *adv_lint_verify* goal verify critical design components, such as finite-state machines (FSMs), case statements, and tristate buses using the SpyGlass tool suite.

This stage enables you to accomplish the following:

- *Verifying FSM and Case Statements*
- *Identifying Deadcode and Redundancy*
- *Identifying the Cause for X Generation*
- *Analyzing Complexity of RTL Modules and FSMs*

Verifying FSM and Case Statements

The rules in this category help in identifying problems related to FSMs in the design. It reports dead states and transitions that cannot be sensitized in an FSM.

1. Choose the proper rule parameters to run the validation.

   The most common parameters are:

   - Parameter detect_assign_fsm=no (default; changing to “yes” will also detect assign style FSM in Verilog).
   - Parameter detect_ifelse_fsm = no (default; changing to “yes” will also detect if-else style FSMs).
   - Parameter detect_nested_fsm = no (default; changing to “yes” will also detect nested if-else style FSMs).

2. Resolve the Av_fsm02 violations for state transitions, which cannot be activated. Refer to Assertion details section of auto-verify.rpt

   - When status of Av_fsm02 is PASSED, it is an information message for the user and requires no action.
   - When status of Av_fsm02 is FAILED
     - Activate violation to bring up FSM viewer.

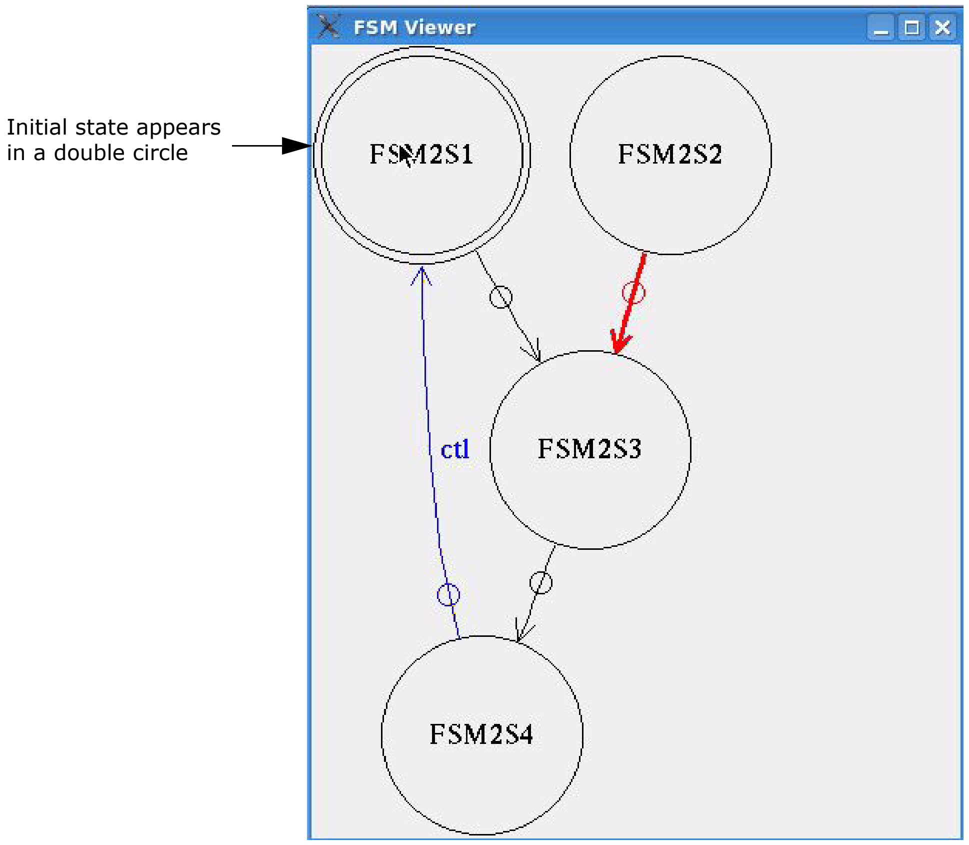

FIGURE 1. The FSM viewer

- Fix or remove the transition and states in FSM.

❒ When status of Av_fsm02 is Partially Analyzed (SpyGlass is not able to conclude in the given amount of time), rerun the design with following options:

- Increase assertion run-time by using the atime parameter.
- Use incremental analysis by providing the propfile parameter.
- Use the abstract parameter which applies abstraction techniques to reduce complex verification problem into simpler and solvable problem

3. Resolve the violation reported by Av_case01 and Av_case02. Refer to Assertion details section of auto-verify.rpt
   - ❒ When status of rule is PASSED, it is an information message for the user and requires no action.
   - ❒ When status of rule is FAILED:
     - Activate the violation to bring up incremental schematic and waveform viewer.
     - Navigate incremental schematic and waveform viewer to find the cause of failure.
     - Fix the issue.
   - ❒ When status of a rule is Partially Analyzed (SpyGlass is not able to conclude in the given amount of time), rerun the design with following options:
     - Increase assertion run-time by using the atime parameter.
     - Use incremental analysis approach by providing the propfile parameter.
     - Use the abstract parameter which applies abstraction techniques to reduce complex verification problem into simpler and solvable problem

Rules for FMS Analysis and Verification

The following rules cover FSM analysis and verification.

Rule	**Description**
Av_fsminf01	Reports all the FSMs in the design
Av_fsminf02	Reports all the interacting FSMs in a design.
Av_fsm02	Reports state transitions of an FSM which cannot be activated
Av_case01	Reports all the sensitizable case items, which are not specified.
Av_case02	Reports all the case statements which have overlapping case items

Av_fsminf01 and Av_fsminf02 reports all FSMs identified in a design and statistics about their complexity and implementation details. These rules do not indicate any design issues but can be used to better optimize them for verification and implementation. Av_fsm02 reports potential bugs in the design. Av_case01 and Av_case02 reports issues around the usage of the pragmas: full_case and parallel_case. The goal fsm includes the above six rules and should be run after running the goal redundancy

Identifying Deadcode and Redundancy

The rules in this category identify dead code and static nets in the design. By eliminating redundant logic, the design size is reduced. It also helps in correcting bugs due to an un-executable code.

To identify the dead code and redundancy issues, perform the following actions:

1. Set appropriate parameters to run SpyGlass CDC validation.

   The commonly used parameter is `dead_code_scope`. Set it to `if` to check only for the `if` conditional blocks. By default, its value is `if_case`.

2. Resolve the *Av_deadcode01* issues in the design. Refer to the *Assertion details* section of auto-verify.rpt to know the status reported by the Av_deadcode01 rule.

   Based on the status, perform appropriate actions, as described below:

   ❒ When status of Av_deadcode01 is **PASSED** in report, it is an information message for the user and requires no action.

   ❒ When status of Av_deadcode01 is **FAILED**, then the violation is also reported. Activate and analyze the violation. Fix or remove the dead-code block.

   ❒ When status of Av_deadcode01 is **Partially Analyzed** (This happens when SpyGlass is not able to conclude in the given amount of time). User needs to rerun the design with following options:

   - Increase assertion run-time by using the atime parameter.
   - Use incremental analysis by providing the propfile parameter.

- Use the abstract parameter which applies abstraction technique to reduce complex verification problem into simpler and solvable problem.

Rules to Identify Deadcode and Redundancy

The following table lists all rules in this category. Depending on the project needs the methodology can be customized to use these rules. Note that these rules cover flip-flops, conditional statements, and simple assignments. Therefore, the number of such checks can be very high and consequently the run time for these rules can be high. Refer to the performance and quality of results section later in this document for tips on how to close verification effectively in reasonable time.

Rule	**Description**
Av_deadcode01	Reports redundant logic in the design
Av_staticnet01	Reports globally stuck-at-0 or stuck-at-1 nets in a design.

Identifying the Cause for X Generation

The rules in this category help in identifying various causes of x generation in a design.

1. Resolve Av_bus01, Av_bus02, Av_dontcare01, and Av_range01 violations. Refer to Assertion details section of auto-verify.rpt
   - ❒ When status is PASSED, it is an information message for the user and requires no action.
   - ❒ When status is FAILED:
     - Activate violation to bring up waveform viewer.
     - Navigate waveform viewer to find the cause of failure.
     - Fix or remove the undesired array access.
   - ❒ When status is Partially Analyzed (SpyGlass is not able to conclude in the given amount of time), rerun the design with following options:
     - Increase assertion run-time by using the atime parameter.

- Use incremental analysis by providing the propfile parameter.
- Use the abstract parameter which applies abstraction techniques to reduce complex verification problem into simpler and solvable problem

Rules to Identify the Cause for X Generation

The rules that provide this functionality are described in the table below.

Rule	Description
Av_bus01	Reports all the bus contentions in the design
Av_bus02	Reports all the floating buses in the design
Av_dontcare01	Reports sensitizable X-assignments in a design.
Av_range01	Reports array bound violation.

Analyzing Complexity of RTL Modules and FSMs

SpyGlass Auto Verify solution provides complexity analysis of RTL modules and FSMs to enable the user to better understand the design, repartition the modules and FSMs if needed, and estimate the effort required for verification. The rule for getting complexity measures is Av_complexity01.

After running this rule:

1. Look at the spreadsheets Av_complexity01_module.csv and Av_complexity01_fsm.csv.
2. Analyze design statistics on FSM (number of states, number of transitions, depth, ...)
3. Review the values in the column Cyclomatic Complexity. If the number is too large for a module, consider decomposing it into smaller modules. Typically, cyclomatic complexity greater than 100 is considered high. Otherwise, make sure that enough test cases are written to cover all branches as part of simulation/dynamic verification step. The number of test cases should be greater than the cyclomatic complexity for the module.

The main complexity measure is called cyclomatic complexity; it measures the number of branches in a module or an FSM. Typically a cyclomatic complexity of 100 is considered high. If the user encounters a large number, he should consider decomposing the module into smaller modules. Alternatively, he can use this number as an estimate for the number of test cases that would be needed in a dynamic verification flow.

Other complexity measures for FSMs include the number of states, the number of state transitions, state encoding, and depth. Other complexity measures for modules include the number of inputs and outputs, the number of case statements, and depth of nesting in if/else statements.

The above information is provided in two spreadsheets: Av_complexity01_module.cvs and Av_complexity01_fsm.csv. The various columns represent different complexity measures while the rows show the FSMs and modules in consideration.

Tips for Functional Verification

The verification rules described in previous sections perform exhaustive functional verification that may be run time intensive and some exhaustive checks may not complete. SpyGlass Auto Verify solution performs two different types of verification automatically. Users do not need to take any step to trigger or control these verification approaches as they are done automatically. However, they need to know the following concepts to better understand the results reported by SpyGlass Auto Verify solution and further improve them by iterative runs:

- Bug detection: Once the design reaches a setup for all flip-flops, known as a state of the design, SpyGlass Auto Verify solution exhaustively cover all possible design inputs assignment to find if a bug can occur in this state. SpyGlass Auto Verify solution can therefore report results indicating the “sequential depth” analyzed exhaustively. A design verified by SpyGlass Auto Verify solution for 10 cycles, for example, can represent millions of simulation stimulus automatically covered.
- Proof: SpyGlass Auto Verify solution can also prove a check as correct for all possible values of flip-flops in the design (states). This analysis involves complex mathematical modeling and proving approaches that may or may not be possible depending on design complexity, property complexity, clocking complexity, etc.

The following sections provide the tips that cover various aspects of verification to improve the quality of verification and run time:

Dealing with Incomplete Results

The outcome of functional checks performed by SpyGlass Auto Verify solution is as follow:

Status	**Description**
FAILED	For a check that failed, SpyGlass provides a simulation trace that can be loaded in the waveform viewer by activating the violation and clicking on the waveform viewer icon (next to schematic viewer icon in the GUI).
PASSED	You can see which checks have passed in the report file, accessible from the GUI pull-down menu Report->auto-verify. You do not have to worry about these messages, as they do not indicate any problem in the design.
PA (Partially Analyzed)	These are instances of checks that are un-concluded. SpyGlass provides the number of cycles that have been explored during which no violation has been found.

Both failed and partially analyzed checks require user attention as they may represent real design bugs. When dealing with functional checks that do not complete, that is checks that are reported as partially analyzed, you can do the following:

- Increase the amount of time that SpyGlass spends on validating a single property. Currently, the default run time is set to 20 seconds per functional check. The parameter used to change the run time is called atime.

- Change the engine selection for functional checks. This changes the way verification is done. SpyGlass provides the `solvemethod` parameter to invoke various engines performing functional verification. This option takes 3 values (1, 2, 3) and depending on the design one or another may conclude the check.

Dealing with Long Run Times

Due to the complexity of functional analysis, you will often need to run SpyGlass iteratively with different options to sign-off the verification. The following flow diagram describes the incremental verification capability of SpyGlass Auto Verify solution that will enable effective iterative verification.

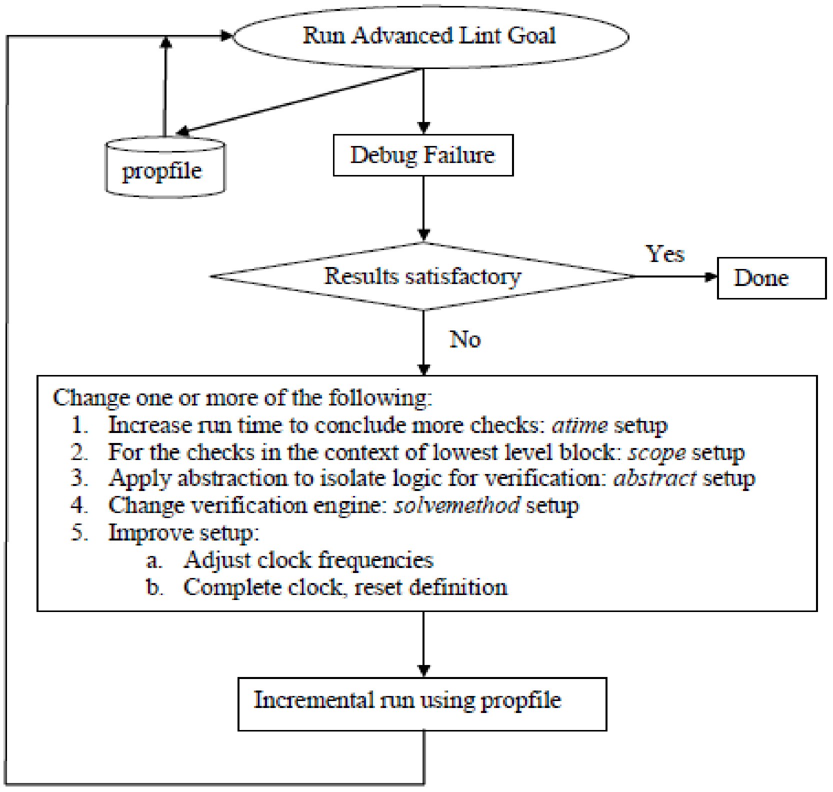
Run Advanced Lint Goal
propfile
Debug Failure
Results satisfactory
Yes
Done
No
Change one or more of the following:
1. Increase run time to conclude more checks: atime setup
2. For the checks in the context of lowest level block: scope setup
3. Apply abstraction to isolate logic for verification: abstract setup
4. Change verification engine: solvemethod setup
5. Improve setup:
a. Adjust clock frequencies
b. Complete clock, reset definition
Incremental run using propfile

FIGURE 2. Incremental verification flow

SpyGlass Auto Verify solution dumps auto_verify.prp in the $CWD/ spyglass_reports/auto_verify directory. This file contains the properties and the status, which is "off" for concluded properties and "on" for non-concluded properties. User has the option of updating the property status. When this property file is provided using the propfile option in incremental (i.e. subsequent) runs, SpyGlass Auto Verify solution will check only those properties whose status is "on".

For detailed description and impact of atime, scope, abstract, and solvemethod parameters and property file (propfile) for incremental verification refer to SpyGlass Auto Verify Rules Reference Guide.

Note that any change in the setup can impact previously verified checks and therefore they need to be run again. Once desired result is obtained in incremental flow, it is recommended to have a file run with complete setup to ensure changes in setup will not have adverse impact on checks completed in previous SpyGlass runs.

Functional analysis complexity increases with the number of asynchronous clocks in a design. Formal verification is exhaustive and involves complex functional analysis of a design. Clock frequencies may greatly affect the complexity of functional analysis. To understand how clock frequencies affect the functional analysis process, consider two clocks running with a 17ns period and a 13ns period, respectively. If the rising edges of the two clocks are aligned at time 0ns, then the next time the rising edges will again be aligned corresponds to 221ns (the least common multiple or LCM of two clock periods). This means that the design behaves asynchronously for 221ns. Any functional analysis process that would exploit the 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. A high design virtual cycle number makes it hard to verify the design functionality.

In some cases where the runs take long time, modify the clock periods to reduce the LCM. Let us take an example.

Device A has two asynchronous clocks: clk_33 is 33ns and clk_100 is 100ns. If you specify these clock periods in the SGDC file, the LCM of the two clock periods is (33x100) 3300ns, which is quite large. If you specify the 100ns clock in the SGDC as being 99ns, then the Design Virtual Cycle has been reduced to 99ns. Note that changing the clock frequency by this amount has impacted the behavior of the design and therefore the change should not be considered unless necessary. If such a change is introduced it should be documented.

SpyGlass reports the design virtual cycle in terms of the number of fastest clock cycles, as well as the number of non-overlapping edges of all clocks covered by the design virtual cycle.

Debugging Functional Checks

In addition to RTL cross-probing and schematic highlights, failure of a functional check will generate a waveform indicating the circumstances of the failure. Once a violation is activated, click on the waveform-icon (close to the schematic icon) to activate the waveform viewer. Initially, a small set of signals are loaded in the waveform; these signals are a good starting point for debugging the waveform. You can right-click on a signal in the waveform viewer and select “fanin” from the pop-up menu to see the set of signals in the immediate vicinity of the selected signal for which a waveform is available. Select all or part of these signals and click on “OK” to load their waveform in the viewer. Note that you can cross-probe between the waveform viewer and the RTL-viewer.

Main Reasons of False Functional Checks Violations

Functional checks violations are always genuine under given constraints. To avoid false functional violations, it is important to properly constraint your design. Below are some of the most important constraints impacting the outcome of functional checks:

- Reset constraint: reset signals are used to initialize the design and they are generally disabled during functional checks. If not specified, resets can be randomly asserted and de-asserted to cause a functional failure. To avoid such situation provide all resets of the design.
- Initial state: SpyGlass identifies an initial state automatically and uses it as starting state for any functional checks. Functional checks may fail or pass depending on the initial state(s) used in functional verification. Always validate the initial state before investigating functional failures. Flip-flop values for initial state as well as how the initial state is obtained by spyglass are provided in rule Ac_initstate01.

Considerations for Chip Level Functional Verification

Typically, RTL code is written for blocks and only inter-blocks connectivity is present at the chip level with small glue logic. For many functional checks, once the block is verified, the same structure remains valid in the context of the chip. For some checks however if the verification is correct at the block level it still needs to be verified in the context of the full chip as the block environment can make them false. To understand this aspect of functional verification, we distinguish between the following types of checks:

- Safety checks that ensure that something bad cannot occur in a design. For example, bus contention should not occur in the design)

- Liveliness checks that ensures that something good must happen at least once in a design. For example, a give state of a state machine should be reached during the execution of the design).

The following table describes the validity of the block level checks in the context of the SoC based on whether the check is a safety or liveliness check:

Property type	**Block Result**	**Block result interpretation in the context of Top design**	**Action**
Safety	Pass	Pass	No Action
	Fail	Unknown	Run Top design
Liveliness	Pass	Unknown	Run Top design
	Fail	Fail	Debug block failure

Following are the list of safety checks in SpyGlass Auto Verify solution:

- Av_bus01: Reports all the bus contentions in the design
- Av_bus02: Reports all the floating buses in the design
- Av_case01: Reports all the sensitizable case items which are not specified
- Av_case02: Reports all the case statements which have overlapping case items
- Av_range01: Reports array bound violation
- Av_dontcare01: Reports sensitizable X-assignments in the design

The following checks are liveliness checks in SpyGlass Auto Verify solution:

- Av_fsm02: Reports state transitions of an FSM which cannot be activated
- Av_deadcode01: Reports redundant logic in the design
- Av_bitstuck01: Reports whether a net is stuck to a constant value or not

- Av_staticnet01: Reports globally stuck-at-0 or stuck-at-1 nets in the design
- Av_staticreg01: Reports all the static registers in the design which cannot toggle

Handling Duplicate Violations

There is some overlap between rules in Spyglass Auto Verify, SpyGlass Base, and Spyglass CDC. The following table lists rules that have the same functionality across the three policies.

Rules of SpyGlass Auto Verify Solution	**Rules of SpyGlass Base Products**	**Structural Rules of SpyGlass CDC Solution**	**Functionality**
Av_sanity02	W415	-	Reports non-tristated nets that have multiple drivers
Av_clkinf01	-	Clock_info01	Reports all clocks in a design
Av_rstinf01	-	Reset_info01	Reports all resets in a design

The following table lists the rules that have functionality overlap across SpyGlass Auto Verify and SpyGlass Base products.

Rules of SpyGlass Auto Verify Solution	Rules of SpyGlass Base Products	Overlapping Functionality
Av_case01	STARC05-2.8.3.3	STARC05-2.8.3.3 reports a violation when case labels of a case statement are not complete. The rule does not take into consideration whether the missing label can be functionally exercised. Av_case01 reports a violation when a missing case label can be functionally exercised. Therefore, the results of Av_case01 are more accurate.
Av_case02	DuplicateCaseLabel-ML	DuplicateCaseLabel-ML reports overlapping case labels. Av_case02 reports overlapping case labels that can be functionally exercised.
Av_dontcare01	NoAssignX-ML	The NoAssignX-ML rule reports a violation for all occurrences of X assignments. However, theAv_dontcare01 rule reports the case where the condition resulting in the X assignment can be functionally exercised.
Av_setreset01	SetResetConverge-ML	Like other functional checks, the Av_setreset01 rule checks functionally whether both set and reset can become active or not. However, the SetResetConverge-ML rule reports only structurally if set and reset are coming from a common source.

Av_staticnet01	FlopDataConstant, LatchDataConstant	The Av_staticnet01 rule reports all nets/flip-flops that have a constant value in all reachable states. This functionality overlaps with the specified rules of SpyGlass Base product.
Av_sanity03	CombLoop	Both rules reports the presence of combinational loops in a design