

@Verifier Evaluation

Revision 1.2

Table of Contents

1	Introduction.....	5
2	References	5
3	Test Case & Conditions	5
4	@Verifier Inputs	6
4.1	Command line switches.....	7
4.1.1	Runtime Mode Switches.....	7
4.1.2	@Verifier Option Switches	9
4.1.3	Input files of @Verifier	12
4.1.3.1	Verilog Constraint File.....	12
4.1.3.2	atadaptive.tcl Script	13
4.1.3.3	Synchronizer Information File	13
4.1.4	Standard Assertion Languages.....	15
4.1.5	Random Simulation Feature.....	16
4.2	@Verifier Output	16
4.2.1	atverifier.log file.....	16
4.2.2	afv_checks.p file	16
5	AFV (Automatic Functional Verification)	17
5.1	FSM (Finite State Machine) Analysis	17
5.1.1	FSM Testcase Observations.....	17
5.2	Multiple Clock Domain Analysis.....	19
5.2.1	Multi-clock domain synchronization verification.....	19
5.2.2	Synchronization Model Checking.....	22
5.2.2.1	Data Stability across clock domain	22
5.2.2.2	Stability of signals on the Bus.....	23
5.2.2.3	Overlap between read and write pointer in FIFOs	23
5.2.3	Invoking @Verifier clock analysis	23
5.3	Multi-Cycle Path (MCP) logic.....	25
5.4	False timing path logic.....	26
5.5	Parallel and full case statements.....	27
5.6	Redundant latch pairs & unintended latch.....	28
5.7	One hot drivers.....	29
5.8	Index out of range	29
5.9	FIFO Verification	29
5.10	Reachability.....	29
6	Result Viewing and Debugging with @Designer	30
7	Conclusion	31
8	Appendix A: FSM Testcase Source Verilog	33

List of Figures

Figure 4-1: @Verifier flow overview	6
Figure 4-2: Constraint module example	12
Figure 4-3: CLOCK_INFO synchronizer pragma	13
Figure 4-4: SYNCHRONIZER_CHECK_FILTER_INFO synchronizer pragma	13
Figure 4-5: SYNCHRONIZER_RANK_INFO synchronizer pragma	14
Figure 4-6: SYNCHRONIZER_CONFIG_REG_INFO synchronizer pragma	14
Figure 4-7: Example of user-defined properties file (design_name.p)	15
Figure 4-8: Enabling and Disabling property checks using inline pragmas	15
Figure 5-1: Verifier Window invoked by the <i>+gui</i> command line option	18
Figure 5-2: Synchronizer with logic after the source register	19
Figure 5-3: Synchronizer without logic after the source register	20
Figure 5-4: Synchronization with source clock domains muxing	21
Figure 5-5: Rate change FIFO synchronization	22
Figure 5-6: MCP example from @Verifier Training v.2.5	26
Figure 5-7: False timing path example.	27
Figure 5-8: Full case example	28
Figure 5-9: Redundant latch example	28

Note: Some figures were gleaned from @HDL reference documents.

List of Tables

Table 4-1: Testcase runtimes with and without +size_reduction	9
Table 4-2: Hierarchical model checking using the DUT-Module2 testcase.	10
Table 4-3: DUT-Module2 testcase ran with the +data_path switch	10
Table 4-4: BMC using the DUT-Module1 testcase	11
Table 4-5: No reachability analysis using the DUT-Module1 testcase	12
Table 4-6: DUT-Module2 testcase with and without random simulation	16

1 Introduction

@HDL's @Verifier is a functional verification tool that combines formal verification techniques with the advantages of several verification methods. It incorporates DRC (Design Rule Checker) methods, formal methods using automatically generated and user-generated properties, and random simulation. Formal property checkers are most useful for finding bugs early in the design cycle, and @Verifier is most effective when combined with the @Designer gui debug cockpit. @Designer was the focus of separate evaluation writeup, which can be obtained from Paradigm Works. This evaluation therefore focuses mostly on @Verifier features. @Verifier has several features to efficiently verify a design. With the exception of random simulation, all of the features statically verify designs:

- AFV (Automatic Functional Verification)
- Property and constraint language
- “Bounded” and “un-bounded” model checking
- Random Simulation
- DRCs (Design Rule Checks)
- Result Viewing and Debugging using @Designer

We evaluated each of these features and the results are shown in the sections to follow.

2 References

@HDL's @Verifier Application Note: Managing Runtime

@Verifier Reference Manual v. 2.5.1

@Verifier Quick Start Guide v. 2.6

@Verifier Training v. 2.5

Paradigm Works' @Designer Evaluation

3 Test Case & Conditions

For this evaluation, we ran @Verifier v2.8 with Red Hat Linux version 2.2.19 running on a HW platform with dual Pentium Pro 1 GHz processors and 2GB of memory. We used a real design, hereafter referred to as the DUT, to test most of the features, but we also created some testcase designs for particular features of @Verifier that the DUT did not address or create interesting failures for (a couple of testcases are from the @HDL reference documents). During our evaluation we mostly ran @Verifier at the module level instead of the full chip (functional core only with RAMs black-boxed) because of long (day+), full-chip runtimes (Note: these full chip runs were without any runtime speedup enhancements that are also available in @Verifier like distributed processing across a server farm, the use of incremental model checking, bounded model checking, no reachability analysis, etc., that could have shortened the long run times).

The DUT has the following characteristics:

32-bit network microprocessor

~250K gates

Asynchronous multi-clock domains

DUT-Module1 characteristics:

- ~40K gates
- 1 clock domain
- 1 RAM (64x128)
- 254 properties extracted with AFV only

DUT-Module2 characteristics:

- ~83K gates
- 3 asynchronous clock domains
- 3 RAMs (512x32, 32x33, 128x40)
- 247 properties extracted with AFV only

4 @Verifier Inputs

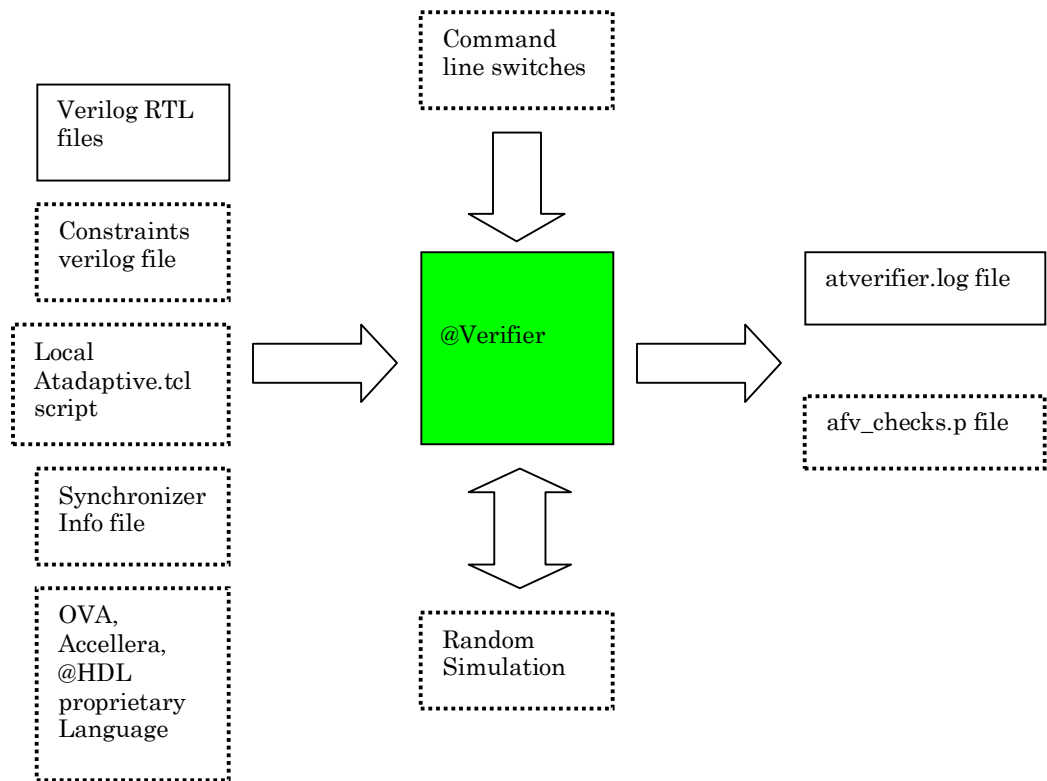


Figure 4-1: @Verifier flow overview

Figure 4-1 shows some of the possible inputs, interactions, and outputs from an @Verifier run. We explored in great detail how these inputs affect @Verifier's performance and how to interpret outputs generated.

4.1 Command line switches

The following command invokes the tool:

@verifier <verilog arguments> <switches option> file

In addition to its own switches, @Verifier supports all of the regular Verilog command switches (ie. -y, -v, etc.).

4.1.1 Runtime Mode Switches

[+afv] [+inline_prop] [+prop_file+<property file name>]
[+synchronizer_checks]
[+drc] [+gui] [+view] [+print_afv_checks]

- *+afv*: Invokes AFV (Automatic Functional Verification) (Refer to Sec. 5)
- *+inline_prop*: Tells the tool to look for @HDL assertions embedded in the RTL. Currently the only language supported for inline usage is @HDL's assertion language including @HDL's special extension (Refer to Sec. 4.1.3.1).
- *+prop_file*: If there are user-specified properties in file, @Verifier can do model checking on the properties in file using this switch. It will be explained more detail in Sec. 4.1.4.
- *+synchronizer_checks*: It invokes all checks related to synchronizer (Refer to Sec.6.2.2) only when there is no switch *+synchronizer_info*. When switch *+synchronizer_info* is turned on in command line already, no matter the switch *+synchronizer_checks* is used, @Verifier will do all synchronizer related checks.
- *+drc*: Invokes the DRC checker, which performs several design sanity checks:
 - Full language support for Verilog
 - Syntax and semantics checks
 - Coding checks for simulation, synthesis, and timing
 - Design practice checks for FSM (Finite State Machine), Flops, Latches, Clocking
 - Coding style checks

We found that in some cases like FSM unreachable state errors/failures (see Figure 5-1), DRC errors and AFV failures may flag the same issue. There is, however, flexibility for users to turn off or on each individual DRC check by editing the *design_checks.tcl* script, which is located in \$ATHDLROOT/apache/htdocs/athdl/tclscripts/.

For example:

```
@checks "$type.Flops.Resettable.Non-resetable" value off ;
@checks "$type.Flops.Asynchronous Reset.Asynchronous Reset" value on ;
@checks "$type.Coding.Bit Ordering.lsb -> msb" value off ;
@checks "$type.Coding.Bit Ordering.msb -> lsb" value on ;
```

These checks are well covered by the DRC checker and DRC errors do not stop the run. Users can disable specific DRCs by editing the *design_checks.tcl* script. At the end of the run, all DRC violations are sorted out by category (General, FSM, Flop, Latches,

Logic, and Synthesis) and/or on a module basis. Most error messages are hyper-linked to the source code and are generally informative and straightforward for quick debug.

Although you would expect the DRC checker to only get invoked by the `+drc` switch, it also runs when the `+gui` switch is used, even if the user did not specify the `+drc` switch.

- *+gui*: Invokes @Designer at end of the @Verifier run.
- *+print_afv_checks*: When this switch is used, @Verifier does not actually run a formal check for the properties. Instead it only generates the *afv_checks.p* file which contains a listing of the AFV properties checks and user-specified properties that would run against the design.

4.1.2 @Verifier Option Switches

- *+incr[(+timeout+fail+error+pass)]:*

This switch allows users to run incremental jobs using the results of a previous run. The user can specify to rerun only properties in specific results categories (ie. timeout, fail, error, or pass) or all of the categories if one is not specified.

- *+timeout*:

This corresponds to the amount of time that @Verifier will spend attempting to verify each property. It's easy to see that a design with thousands of properties can take a long time to run depending on the timeout value. Therefore, users can and should adjust the timeout value as appropriate. It's specified in seconds, and the default value is 45 seconds. We found that reachability properties are the biggest time synchs during a run. So, it's advisable to use the *+no_reachability* switch whenever possible (see below for a description of the *+no_reachability* switch).

- *+size_reduction*:

Using this switch, @Verifier does size reduction optimization when proving each property. This means that when attempting to verify a property @Verifier first chooses a variable and flip-flop set that yields a minimal size logic cone. If it cannot verify the property, it then expands the logic cone to include flops and other variables further upstream, and then reattempts to verify the property with this larger logic cone. It iterates through this process until it proves the property or the property times out. Size reduction option is off by default.

	Total Properties	Run time	Timeout	Failed
Default	254	1 hr 14 min	71	0
<i>+size_reduction</i>	254	2 hr 2 min	69	0

Table 4-1: Testcase runtimes with and without *+size_reduction*

As seen in Table 4-1, the runtime almost doubles with the *+size_reduction* switch, because of the iteration process, but @Verifier only proves 2 more out of 71 timeout properties.

- *+no_hierarchical_mc*:

By default @Verifier uses hierarchical model checking that starts at the lowest module level where the property is applicable. If the property can be proven at this level, verification for the property is complete. If the property fails, verification is performed at the next higher level of hierarchy and so on until the property is fails, passes, or times out. Using this, @Verifier stores the level of hierarchy at which a property is proven when the first instance of a module is encountered. Later, if the same property at the same level of hierarchy is encountered for another instance of the same module, @Verifier uses the stored information and makes sure that the property is already proven.

	Total Properties	Run time	Time Out	Failed
Default	247	30 min	5	3
No Hierarchical MC	247	30 min	24	3

Table 4-2: Hierarchical model checking using the DUT-Module2 testcase.

Users should be cautious using this switch because, as Table 4-2 shows, the `+no_hierarchical_mc` switch increases the number of timed-out properties, while having no effect on the runtime.

- `+data_path+<bitwidth>`:

This switch tells @Verifier to find all signals that are wider than the specified bitwidth and hold constant the bits above the specified bitwidth. For example, specifying `+data_path+16` tells @Verifier to hold constant bits [...]16] on every signal that is wider than 16 bits. This data path option is off by default.

	Total Properties	Run time	Time Out	Failed
Default	247	30 min	5	3
<code>+data_path+15</code>	247	56 min	1	142

Table 4-3: DUT-Module2 testcase ran with the `+data_path` switch

As shown in Table 4-3, using the `+data_path` switch caused timed-out properties to decrease, but the number of failed properties increased exponentially from 3 to 142. Also, the runtime almost doubled. Users should not use this switch as a blanket method for constraining signals. A constraint verilog file is best suited for this since the `+data_path+` switch gives misleading properties failures.

- `+model_check+bmc` and `+bmc_bound+<bmc bound>`:

By default, @Verifier attempts to prove properties over an infinite number of clock cycles. Bounded Model Checking (BMC) searches for failures within a specified number of clock cycles. BMC can significantly reduce runtime, but has a profound effect on results. The `+model_check+bmc` and `+bmc_bound+<bmc bound>` switches must be used together (the former turns on the BMC engine and the latter specifies the clock cycle bound).

The default behavior with BMC is to first try the bounded engine, and if the property passes, it will then try the default engine. While this can speed things up for properties that fail in the specified BMC boundary, it won't help for those that don't. To turn off this default BMC behavior and run solely with the bounded engine without the default engine, use the switch `+no_atfv`:

`+no_atfv+model_check+bmc+bmc_bound+20`

According to @HDL's @Verifier Runtime Application Note, the formal engine used for this proof is only useful at the module level because it has a limit of 5000 flops or 50000 gates for each property. If a property hits either one of the limits, it will be marked as "Timed Out".

	Total Properties	Run time	Time Out	Failed
Default	254	1 hr 15 min	70	0
+model_check+bmc +bmc_bound+20	254	1 hr 18 min	71	0
+no_atfv +model_check+bmc +bmc_bound+20	254	1 hr 30 min	66	2

Table 4-4: BMC using the DUT-Module1 testcase

As shown in Table 4-4, our testcase proved hardly any difference with bmc turned on. However, we still expect this feature to cut runtime depending on how high the bmc_bound is set, the characteristics of the properties being proven on the design, and the number of failing properties when running with the default formal engine (ie. the more properties that fail with the default formal engine, the more opportunity the bmc engine has to reduce runtime).

- *+mclk:*

When verifying clock domain crossings, clocks are specified using the *+mclk* switch (eg. *+mclk+top.dut.A_Clk+top.dut.B_Clk*). Clock analysis is discussed more in depth in Section 5.2 through 5.4),

In a multi-clock domain design, the fastest clock gets listed first as the reference clock, and slower clocks are listed with an integer multiple for the speed ratio to the reference clock (eg. *+mclk+fastClk:1+slowClka:2+slowClkb:4* could be used to specify 133 MHz (fastClk), 100 MHz (slowClka), and 40 MHz (slowClkb) clocks).

- *+synchronizer_info*+<file name> and *+synchronizer_rank*+<number>:

Users can specify clock analysis information on the command line or by adding this same information to a file (Refer to Sec. 4.1.3.3) and using the *+synchronizer_info* switch. The *+synchronizer_rank* switch tells @Verifier to check that all synchronizers have at least the specified number of flip-flops in series before the data signal is used in the target clock domain (See Section 5.2.3 for testcases and feedback).

- *+no_reachability:*

	Run time	Total Properties	Time Out	Failed
With Reachability (Default)	1 hr 15 min	254	70	0
No Reachability	2 min	6	6	0

Table 4-5: No reachability analysis using the DUT-Module1 testcase

The *+no_reachability* switch tells @Verifier to not check reachability properties during AFV (Refer to Sec. 5.10). As seen in Table 4-5, not checking reachability properties can significantly reduce runtime, and this is very useful for getting quick feedback on a design in progress.

4.1.3 Input files of @Verifier

In addition to design files, users may input other supporting files during a run:

- Constraint file
- atadaptive.tcl script
- Synchronizer information file

4.1.3.1 Verilog Constraint File

@Verifier can incorporate user-generated constraints that are used to avoid unwanted inputs combinations that could cause false failures. These input combinations may correspond to states that are impossible to reach in the real system or states that the user simply has no interest in testing.

The constraint file may be specified on the command line and uses standard Verilog syntax in addition to three @HDL system calls. The constraint module is defined in it's own verilog file, must be named *athdl_constraint* (the file name must match the module name), and has no ports. The clock signal must be declared as a register and named *clk*. This is a dummy clock, which will map to the main clock in the design. The three special @HDL system calls are \$random, \$legal, and \$onehot. Figure 4-2 shows an example constraint module using the \$legal @HDL system call.

```
module athdl_constraint;
reg clk;

assign {top.a, top.b, top.c} = $legal (3'b001, 3'b101, 3'b110);
endmodule
```

Figure 4-2: Constraint module example

4.1.3.2 atadaptive.tcl Script

Users may enable or disable most property checks by editing a tcl file called atadaptive.tcl, which is located in \$ATHDLROOT/apache/htdocs/athdl/tclscripts by default. @Verifier automatically checks the default location at the beginning of a run. It's wise to use the default file for global settings, and users can override the settings in the default atadaptive.tcl file by copying the file to the current working directory and editing the local file. Again, @Verifier automatically checks the current working directory for the atadaptive.tcl file, and the local file settings override the global/default file settings. For example, users can deactivate the "index out of range" property checking that is on by default in script:

```
@adaptive index_out_of_range off;
```

4.1.3.3 Synchronizer Information File

The synchronizer information file is used with the `+synchronizer_info` command line switch. There are four pragmas available to customize the synchronizer checks:

1. Specify synonym clocks. **CLOCK_INFO**

This pragma allows users to set synonym clocks in design and no checking will be performed for interactions between these clock domains. Synonym clocks are clocks in the design which may appear as separate or unrecognized clocks, but should be treated as if they are the same clock. Note that "SYNONYM" command is case-sensitive and it SHOULD be upper case of that command. (In @Verifier Manual v.2.5.1, it is defined as lower case) and otherwise, @Verifier does not notice this pragma.

The following shows the way how two clocks, top.derived_clk and top.master_clk are specified as synonym in the synchronizer_info file.

```
--BEGIN CLOCK_INFO
// SYNONYM <derived clock> <master clock>
SYNONYM top.derived_clk top.master_clk
--END CLOCK_INFO
```

Figure 4-3: CLOCK_INFO synchronizer pragma

2. Filter out checks : **SYNCHRONIZER_CHECK_FILTER_INFO**

Synchronizer checks on any module or instance between BEGIN and END statements using this pragma will not be performed. If the module name is specified, the filter is applied to all instances of that module. The syntax is following.

```
--BEGIN SYNCHRONIZER_CHECK_FILTER_INFO
//<module or instance name>
top.cpu
--END SYNCHRONIZER_CHECK_FILTER_INFO
```

Figure 4-4: SYNCHRONIZER_CHECK_FILTER_INFO synchronizer pragma

3. Synchronizer rank check : SYNCHRONIZER_RANK_INFO

This pragma allows users to specify rank (number of levels of synchronizer, refer to Figure 4-5) requirements per module. @Verifier verifies that the synchronizers in the design have a specified rank. The module or instance specified with the rank number between BEGIN and the END statement will be checked to see if the synchronizer rank matches the rank number specified. If the module name is specified, the check is applied to all instances of that module. The syntax is following.

```
--BEGIN SYNCHRONIZER_RANK_INFO
//<rank number> <module or instance name>
2 top.c1.il
1 cpu
--END SYNCHRONIZER_RANK_INFO
```

Figure 4-5: SYNCHRONIZER_RANK_INFO synchronizer pragma

At the end of a run, @Verifier displays the actual and required FF depth/rank for the synchronizers in the design. If users don't specify any rank information, @Verifier will not check the design's synchronizer depths no matter how many FF levels the synchronizers use. Even if a synchronizer has only a single register, without the SYNCHRONIZER_RANK_INFO pragma, @Verifier will not flag a synchronizer error.

4. Configuration registers treated as constant : SYNCHRONIZER_CONFIG_REG_INFO

Although we did not explore this feature, users should be able to specify configuration registers as constants to @Verifier so that the multiple clock domain checks are not performed on these registers or the logic fed by these registers.

```
--BEGIN SYNCHRONIZER_CONFIG_REG_INFO
// <full hierarchical path name for var>
top.c1.cfg_reg1
top.c1.cfg_reg2[3:0]
--END SYNCHRONIZER_CONFIG_REG_INFO
```

Figure 4-6: SYNCHRONIZER_CONFIG_REG_INFO synchronizer pragma

4.1.4 Standard Assertion Languages

@Verifier provides a property library and users can write their own custom properties and incorporate them into a run using the `+prop_file` runtime mode (Refer to Sec. 4.1.1) and @Verifier supports property languages like Open Vera Assertions (OVA), Sugar, or @HDL's proprietary language. In here, we evaluate only @HDL's proprietary language.

Observation :

The user-defined properties are written in @HDL proprietary language similar to Verilog system calls and can also be simulated with the design using the Verilog PLI (we did not explore @Verifier with VHDL, but the tool does support it). For example:

```
// When load is active, the count is loaded from input cin in the next cycle
load -> ($next(count) == cin);
// If load is not true, and updown is true, counter counts up
(!load & updown) -> ($next(count) == count + 1);
// If load is not true, and updown is not true, the counter counts down
(!load & !updown) -> ($next(count) == count - 1);
```

Figure 4-7: Example of user-defined properties file (design_name.p)

Users specifies custom properties file and using switch `+prop_prop+[design_name.p]` to apply this assertion checking to simulation.

Additional switches for property checks

- `+print_afv_checks` (Refer to Sec. 4.1.1)
- `+inline_prop` (Refer to Sec. 4.1.1)

Using some examples already in @Verifier Manual v. 2.5.1 for testcases, @HDL proprietary language is fairly straightforward for users to write the properties for the design and if the user-specified properties fail, Verifier Window shows that what line of property file causes that. Also failed message has hyper-linked to waveform viewer. When we clicked on the message, @Verifier pops up message like “WaveForm has not been loaded. Certain operations require waveform to be loaded”. Without waveform viewer and limitation of information given by @Verifier, it is hard for users to figure out whether users' assertion checking or actual design is wrong.

@Verifier can enable or disable the property checks in the design and it is accomplished by using “// @hdl ichecks_off/on” pragma. This pragma applies to only the file contains it. This pragma can disable specific checking by using specific error number for AFV property checks. For example:

```
// @hdl ichecks_off [error number [, error number]..]
.....relevant verilog code ....
// @hdl ichecks_on
```

Figure 4-8: Enabling and Disabling property checks using inline pragmas

Note : This pragma enables or disables the error checking for only AFV, not user-specified property checking (assertion checking). (i.e. The custom assertion checking by users is NOT affected by this pragma).

4.1.5 Random Simulation Feature

The main purpose of the random simulation feature is to reduce run time by proving as many reachability properties as possible without invoking the time consuming formal engine. This feature is on by default but can be turned off with the switch `+no_random_reach`. To run the random simulation, the `$ATHDL_SIM` environment variable must be set to the name of the simulator to invoke (VCS, Verilog XL, NC, and Modelsim are supported).

@Verifier will generate a testbench that runs a random simulation on the design. The testbench applies the resets and generates clocks and random stimulus for the remaining inputs. It runs the testbench and monitors the block of code for each of the reachability properties. If the block of code is executed during random simulation, then it isn't necessary to prove the reachability property.

Observation :

	Reachability/Total Properties	Run time	Timed Out Properties	Failed Properties
Random Sim	52/238	32 min	5	3
No Random Sim	52/238	41 min	5	3

Table 4-6: DUT-Module2 testcase with and without random simulation

We ran the DUT-Module2 testcase with and without random simulation. We had issues getting the random simulation to run consistently. After some investigation, we noticed that if the simulator license (VCS in our case) is not available, @Verifier quits out of the random simulation step. A script could get around this issue, but it would be nice if @Verifier entered into a loop to wait for the license instead of just quitting out of and skipping the random simulation step.

We expected the random simulation to reduce the runtime. Roughly 22% of the total AFV properties extracted from the testcase were reachability properties. With random simulation the runtime was reduced by about 22%, as seen above in Table 4-6. Random simulation can at best reduce the runtime by the percentage of reachability properties out of the total properties. Using the DUT-Module1 testcase, random simulation had a minimal effect on the runtime because only 6% of the total properties were reachability properties. Additionally, random simulation will not significantly reduce the runtime if a high percentage of the reachability properties do not pass during the random simulation step and have to get verified with the formal engine. In an extreme case, the runtime will not get reduced at all if all of the reachability properties do not pass during the random simulation step.

4.2 @Verifier Output

The definition of the output of @Verifier is what @Verifier generates as a result of the run.

4.2.1 atverifier.log file

After the run is completed, @Verifier generates output log file, which is called "atverifier.log" in current work directory and it lists what command line switches users used, verilog files, the phases of run, and each property and constraint check and status as well.

4.2.2 afv_checks.p file

This file is generated only when command line switch `+print_afv_checks` is on (Refer to Sec. 4.1.1).

5 AFV (Automatic Functional Verification)

The AFV feature extracts design characteristics/elements from the RTL and applies design property checks on the design using formal techniques and/or random simulation. The design properties that are applied during AFV are from @Verifier's canned property library. This feature present in most if not all formal property checkers. Though, @HDL uses the term "AFV" in an attempt to distinguish this feature. To add AFV to an @Verifier run, you only need to add the +afv switch to your atverifier command. AFV can check a design against the following properties:

- FSM Deadlock
- Unreachable FSM States
- Terminal FSM States
- Clock synchronization errors
- Multicycle timing path errors
- False timing path errors
- Parallel and Full case synthesis directives
- Redundant latch pair errors
- One Hot coding style
- One Cold coding style
- Index out of range
- Memory/FIFO read or write errors
- Clock Gating
- Unintended latches
- Unreachable RTL code
- Undriven variables/signals
- Stuck at 1/0 errors
- Reset logic errors
- Combinatorial loops

Refer to the latest @Verifier Manual for a complete listing of properties available through AFV. We highlight and tested some of these AFV properties.

5.1 FSM (Finite State Machine) Analysis

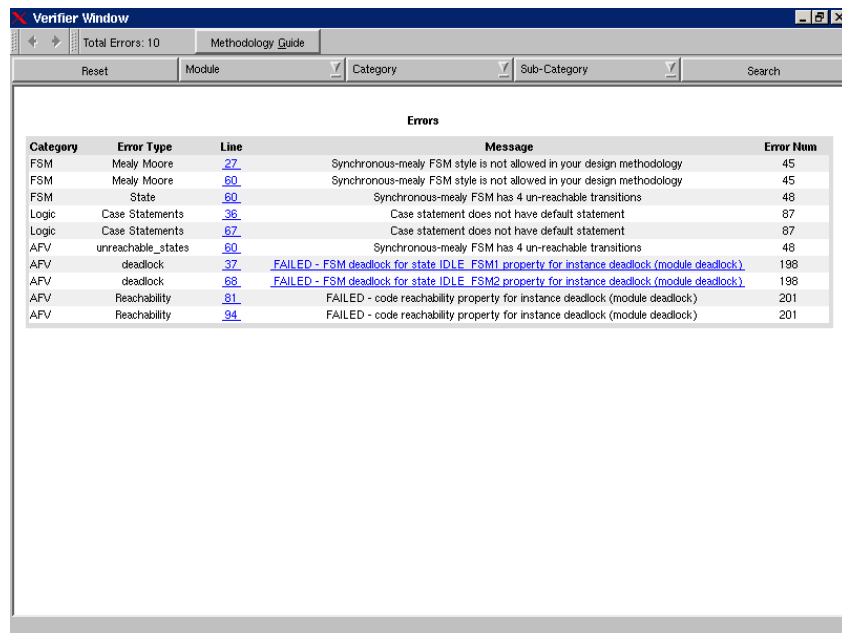
@Verifier's AFV feature contains three properties that can uncover bugs within a state machine. For the Deadlock and Unreachable_state properties, our test cases yield slightly different definitions of the property than that given in the @Verifier Manual:

- **FSM Deadlock:** when two FSMs are both expecting input from the other, but this input will never come because both FSMs are waiting for the other to generate the stimulus first.
- **Terminal State:** A state from which a transition to any other state cannot occur without applying reset. This differs from FSM deadlock because regardless of the input combinations given to the FSM, except for reset, a transition to any other state is not possible.
- **Unreachable State:** A state that is unreachable from any other state

5.1.1 FSM Testcase Observations

Our test design did not reveal any interesting FSM property failures, so we created the Verilog testcase shown in Appendix A: FSM Testcase Source Verilog and created a list of observations, in no particular order:

1. The @Verifier Manual gives the above definition for FSM Deadlock (ie. two FSMs waiting for input from the other), but our testcase shows that the tool treats FSM Deadlock in a more general sense; As a condition where the design is not capable of generating any input combination to the FSM to allow it to transition to another state.
2. Deadlock property failures point to the exact state that can get deadlocked. To track down the exact cause of the deadlock, you can click on the property failure message and it pulls up the waveform for the failing case. It automatically adds signals that it thinks are relevant to the failure, but all variables are saved so that the user can add any additional variables to the waveform. This was extremely helpful in pinpointing the root cause of deadlock property failures and without having to resimulate. However, we could not consistently get the waveforms to open up. We were unable to root cause this problem.
3. The @Verifier Manual defines the Unreachable_state property as a check for unreachable states, which seems very straight forward. However, this property actually checks for unreachable state transitions and not unreachable states. This means that a particular state may be reachable, but it may have certain transitions to it that are not reachable. In this case an Unreachable_state property failure will be flagged for each FSM with unreachable transitions, and these failures are not always easy to track down (see Figure 5-1). If there are multiple failures in a single FSM defined within an *always* block, all of the failures are lumped together into one failure statement. The failure statement only gives the first line of the *always* block to facilitate debug, as opposed to the specific lines inside of the *always* block that the failures occur on. It's easy to see that for complicated FSMs it can be quite hard or time-consuming to even figure out exactly where the failures occur within the FSM, even if there's only one failure.



Category	Error Type	Line	Message	Error Num
FSM	Mealy Moore	27	Synchronous-mealy FSM style is not allowed in your design methodology	45
FSM	Mealy Moore	60	Synchronous-mealy FSM style is not allowed in your design methodology	45
FSM	State	60	Synchronous-mealy FSM has 4 un-reachable transitions	48
Logic	Case Statements	36	Case statement does not have default statement	87
Logic	Case Statements	67	Case statement does not have default statement	87
AFV	unreachable_states	60	Synchronous-mealy FSM has 4 un-reachable transitions	48
AFV	deadlock	37	FAILED - FSM deadlock for state IDLE FSM1 property for instance deadlock (module deadlock)	198
AFV	deadlock	68	FAILED - FSM deadlock for state IDLE FSM2 property for instance deadlock (module deadlock)	198
AFV	Reachability	81	FAILED - code reachability property for instance deadlock (module deadlock)	201
AFV	Reachability	94	FAILED - code reachability property for instance deadlock (module deadlock)	201

Figure 5-1: Verifier Window invoked by the +gui command line option

4. We noticed that AFV and DRC give redundant errors for unreachable state transitions and terminal states. On a large design with many errors, this could make the already overwhelming list of errors that much more intimidating. Though in some cases code reachability property failures helped track down those nondescriptive unreachable state transition failures.

5.2 Multiple Clock Domain Analysis

@Verifier identifies clock derivation tree (like buffered, inverted, gated divide-down clocks), clock domain synchronizers, domain crossover, and data stability. Also users can view domain in schematic viewer.

@Verifier is supposed to identify :

- The clock tree and partition the design into various clock domains
- Clock derivation tree (like buffered, inverted, gated, divide-down clocks)
- All signals that cross clock domains
- Valid synchronizer elements among these signals. It recognizes 4 synchronization styles. These include synchronization based on FFs, domain enabled multiplexor logic, FSM's and FIFO's.
- All signals that cross clock domains without proper synchronization.

5.2.1 Multi-clock domain synchronization verification

@Verifier can identify four kinds of synchronizer schemes inferred from the RTL code and check to determine if they conform to synchronization rules. This check also identifies any signals that cross over from one clock domain to another without proper synchronization.

Synchronizer @Verifier supports the following :

A. Synchronizer with a series of FFs

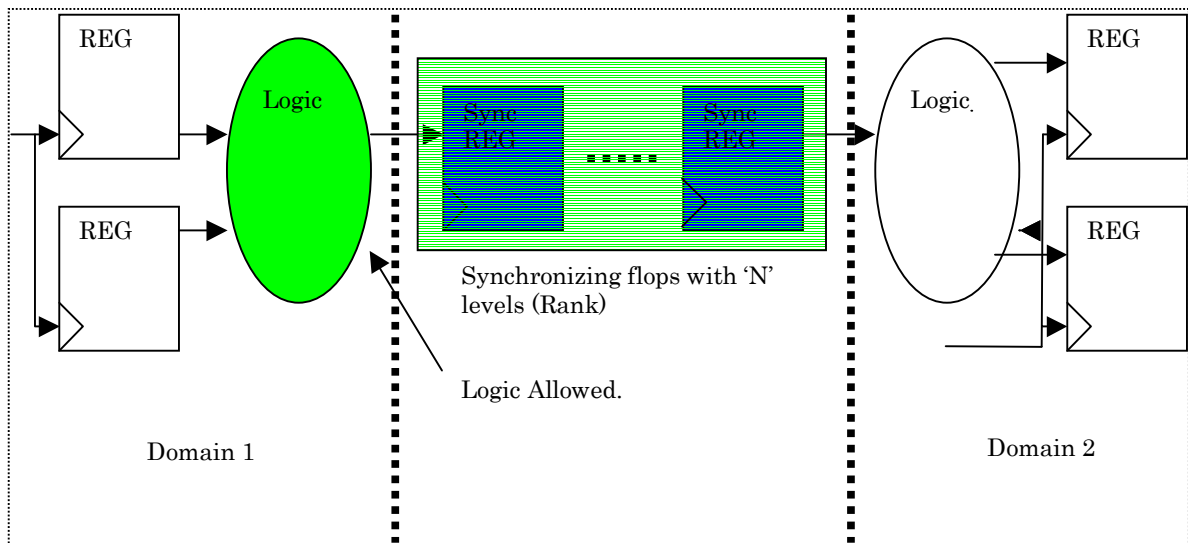


Figure 5-2: Synchronizer with logic after the source register

The assertions generated by @Verifier checks whether the specified number of FFs (rank) is used in the synchronizer defined in synchronizer info file using SYNCHRONIZER_RANK_INFO pragma.

In “atadaptive.tcl” script, the number of FFs required in a path that crosses clock domains is defined and default is 1. Users may customize this check to change the number of rank. For example:

@adaptive synchronizer 2;

Rank information in synchronizer info file sets the default; the one in synchronizer info file overrides the default and even new modified rank info in local “atadaptive.tcl” script. (Refer to Sec. 4.1.3.3)

B. Synchronizer with a series of FFs that are not allowed to use any combinational logic between the registers in the final stage of domain and the synchronizers.

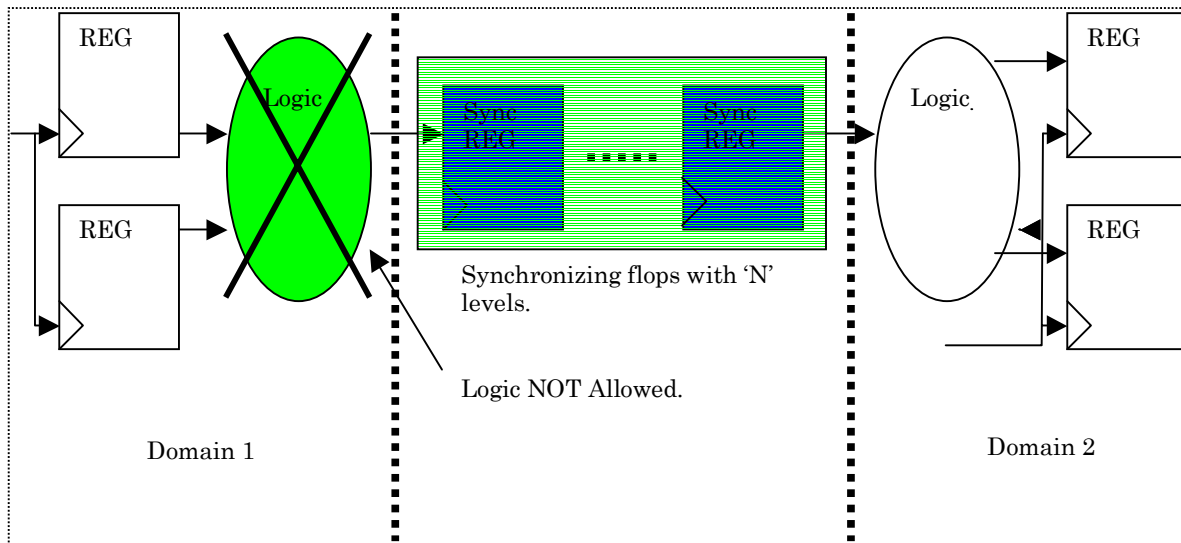


Figure 5-3: Synchronizer without logic after the source register

This scheme B is same as scheme A except no combinational logic before synchronizer and the assertions checks same thing as scheme A.

C. Synchronizer with mux enable.

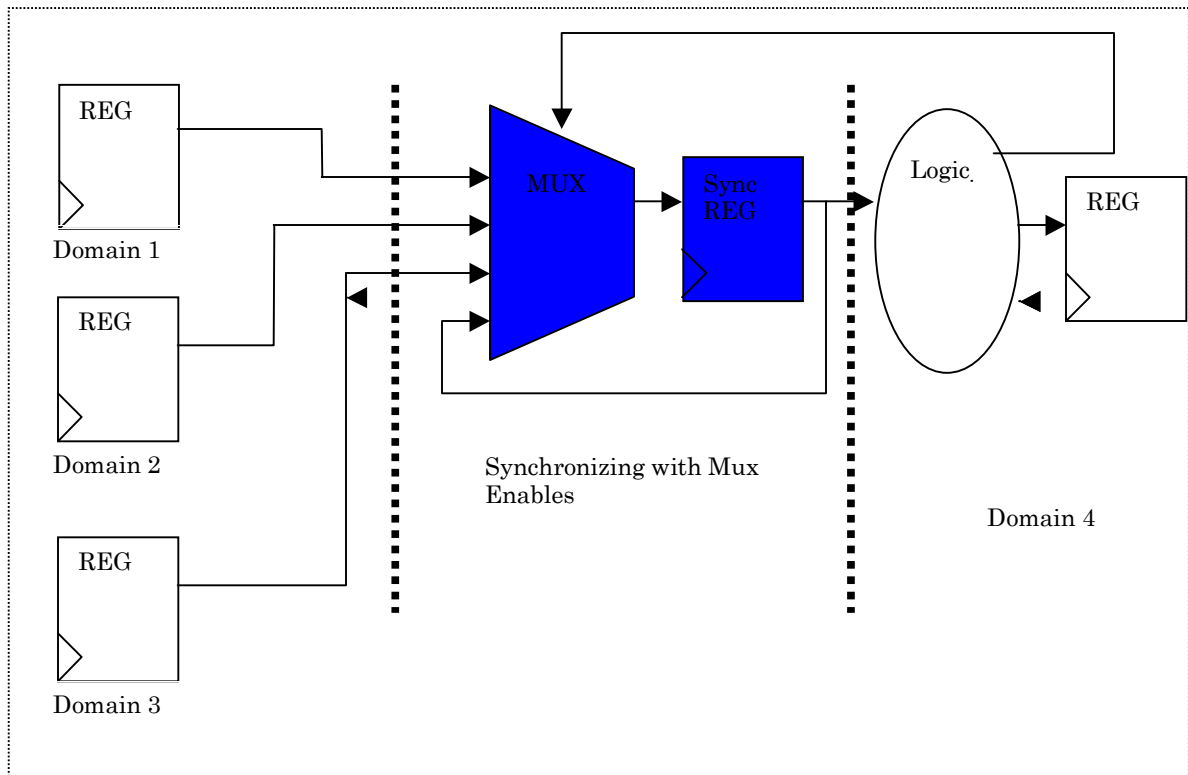


Figure 5-4: Synchronization with source clock domains muxing

This requires a hand-shaking scheme. The data generating domains 1,2, or 3 send a signal to the signal to the data-receiving domain 4, informing it that the data is ready. The data receiving domain 4, will control the mux whenever it is ready to accept the data. @Verifier identifies a mux to be part of the synchronizer and will not flag it as an asynchronous multi-domain crossover (Refer to 6.2.4) or combinatorial input to synchronizer.

Also @Verifier generates the properties that the data signals are stable while they are enabled by the mux and verifies it.

Using a custom testcase, we could notice that @Verifier identifies this scheme and verifies data stability property for input data in mux well. If the design violates this rule, @Verifier flags it as a Domain Data Stability Error, and the error message is hyperlinked to the waveform.

D. Synchronizer with FIFO's.

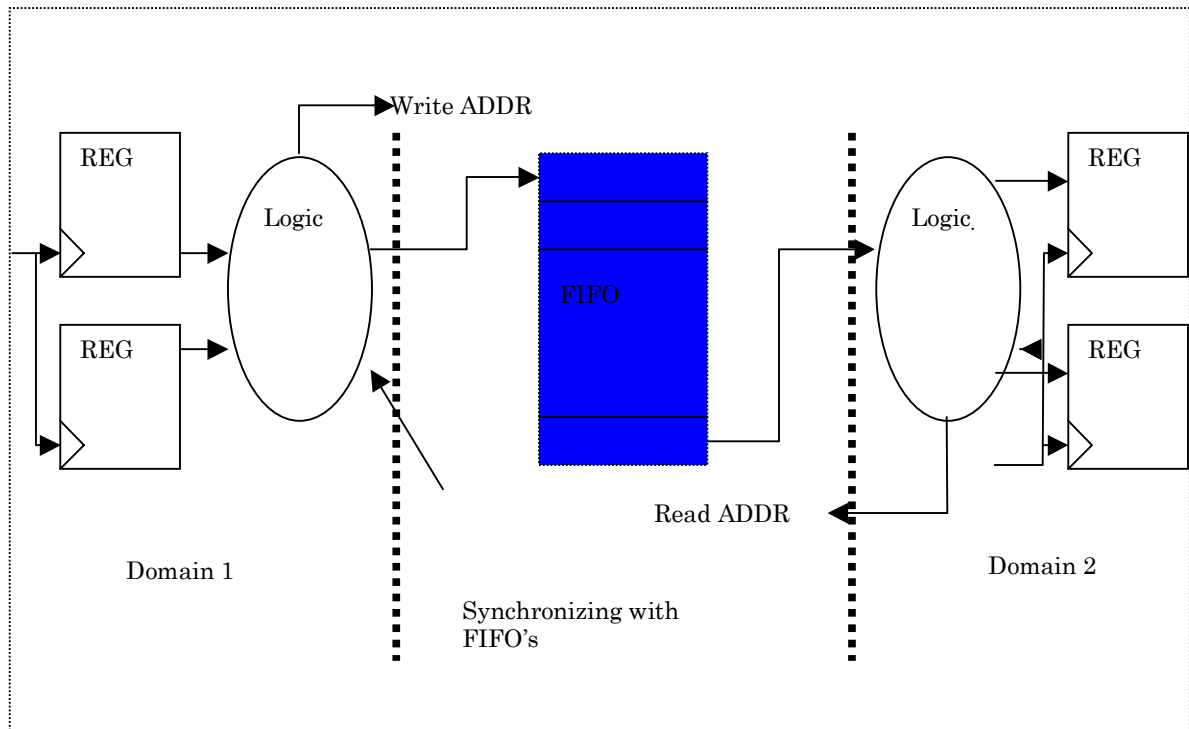


Figure 5-5: Rate change FIFO synchronization

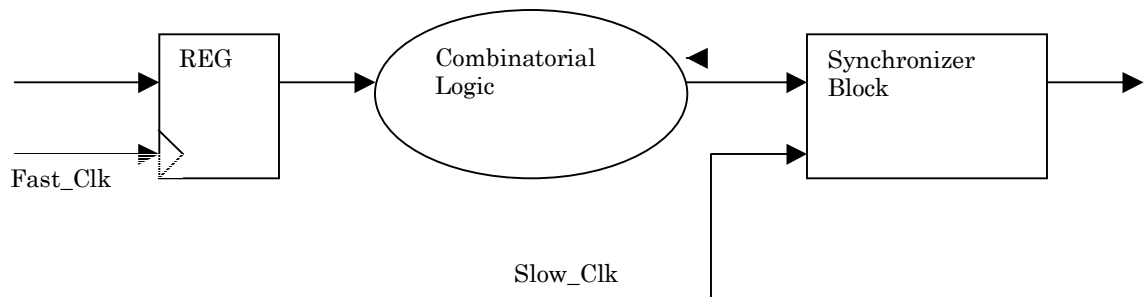
The data is written from the data generating part of the logic, triggered by Domain 1 clock while the data is read from the FIFO by the receiving logic using Domain 2 clock. The FIFO is used as a buffer and the size is determined by the ratio of the clock frequency given by users and the bandwidth of the data to be transmitted. The assertion generated by @Verifier makes sure that the read pointer (driven by destination clock domain) does not coincide with the write pointer (driven by the destination clock domain). If they did, it would lead to the destination domain reading the data that the source domain was writing, which would break the synchronization scheme. (Refer to Sec. 5.2.2.3). We don't exercise this scheme in this documentation.

5.2.2 Synchronization Model Checking

@Verifier extracts the following properties from RTL and verifies the corresponding properties. If this condition is NOT met, @Verifier flags failure. This checks the following.

5.2.2.1 Data Stability across clock domain

When data crosses from a fast clock domain into a slow clock domain, this check verifies if data is held for until the rising edge on the slower clock using users' clock domain information based on ratios that specified in +mclk switch.



Using a testcase that contained 2 different clock domains (using switch +mclk with ratio of master clocks) and simple combinatorial logic is used to examine this property.

Domain Data Stability failures only give the beginning of *always* statement as the line that the errors occurs on instead of the specific line inside of the *always* block that the error occurs. The error message is like “FAILED – Clock domain data stability property for instance <instance name>” and it is hyper-linked with waveform viewer not evaluated because of the limitation of random simulation.

If there are multiple FSM property failures in a single FSM defined within an *always* block, each of the failures will give the beginning of the *always* statement as the line that the errors occurs on instead of the-specific line inside of the *always* block that the error occurs (this can also be quite a nuisance even if there is only one failure in a complicated state machine).

5.2.2.2 Stability of signals on the Bus

This property identifies whether all the data on a bus are stable while the bus is enabled being latched. This property is NOT verified in here.

5.2.2.3 Overlap between read and write pointer in FIFOs

This property identifies whether the read occurs while the write is in progress. This property is NOT verified in here since our DUT does not have FIFOs.

5.2.3 Invoking @Verifier clock analysis

@Verifier is invoked with the master clock option on the command line and appropriate options turned on in the “atadaptive.tcl” file (default file located in \$ATHDLROOT/apache/htdocs/athdl/tclscripts) to do the clock analysis. For example:

```
atverifier -f <filelist> +mclk+<master clk1>+<master clk2>
```

The clock analysis can be opened when users click on @Designer GUI’s “Clock Analysis” button. Clock analysis window contains 6 categories that users can click on for corresponding information:

- Clock tree
 - It shows all master clocks and line numbers that it is specified in design for each one. Line numbers are hyper-linked to source code.
- Synchronizers

Users can view the synchronizers in design and hyper-linked line number in here highlighted portion of source code to represent synchronizers that is nice for users to debug. It reports the status synchronizers used in design. Also it has a capability to save synchronizers result as text file when users click on hyper-linked “Save As Text” in upper right corner.

The Synchronizers category shows whether the synchronizer depths are sufficient according to the rank specified with the `+synchronizer_rank` command line switch. We created three testcases that explore synchronizer checks:

- Testcase1

This testcase uses the `SYNCHRONIZER_RANK_INFO` pragma within a synchronizer information file to specify the rank information. The testcase specifies a rank of 4 and the synchronizer in the RTL use a rank of 2. @Verifier shows that the status of the synchronizer is GOOD in this testcase, even though the synchronizer depth implemented is less than the rank specified via the pragma.

- Testcase2

This testcase uses the `+synchronizer_rank` switch to specify the required rank. Unlike the `SYNCHRONIZER_RANK_INFO` pragma, @Verifier correctly caught insufficient synchronizer depth errors with the `+synchronizer_rank` switch. However, when we added the `+dir` switch to specify a directory path for incremental results, the synchronizer rank checks were ALWAYS GOOD no matter what rank value we specified with the `+synchronizer_rank` switch.

- Testcase3

This testcase uses both the `SYNCHRONIZER_RANK_INFO` pragma within a synchronizer info file and the `+synchronizer_rank` switch to specify the rank. We did this to see if one method had priority over the other if both are used. We found that the `+synchronizer_rank` switch was dominant. We still saw the same anomaly with the `+dir` switch as with Testcase2.

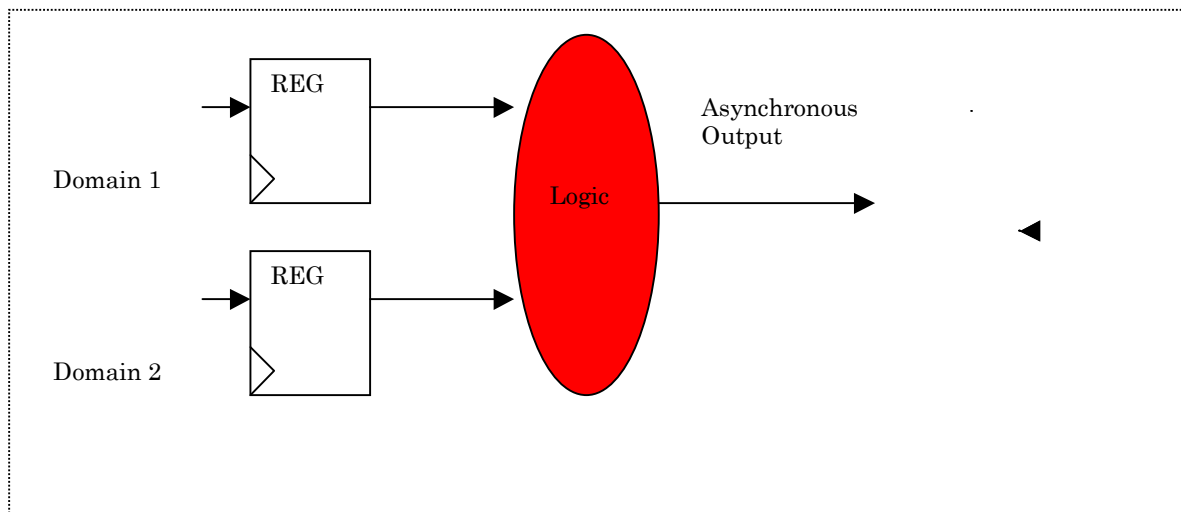
- Incorrect Synchronizers

@Verifier identifies whether there is any combinational logic at the input of any synchronizer. By default this check is on; however users can turn this check off by using `+allow_comb_drivers` command line switch.

- Insufficient Synchronization

Any synchronizers that does not have specified number of FF are listed in this category. For example, when users specify a rank of 2, @Verifier verifies that the signals go through at least 2 flops in a row before being used. If they don't, it flags it as “insufficient synchronization”.

- Asynchronous crossover



@Verifier checks whether combinatorial logic in design has inputs directly from multiple clock domains and list them in this category.

This property is examined by using DUT's module that contains ~80k gates and 3 clock domains (40MHz, and 2 different clock phase 25MHz). @Verifier detects Asynchronous Cross failures when combinatorial logic in module has inputs from multiple master clock domains without synchronization. Whole lines related to the corresponding logic are highlighted in the source code by clicking line number in this category. Messages are detail enough for users to debug failure and like Synchronizers category, it also contains the capability to save result to files.

- Unrecognized Clocks

Unrecognized clock category includes the following cases: clocks that cannot be derived by @Verifier and clocks that are found without any relationship with any of the specified master clocks.

Users may fix this by either specifying these as master clocks or define relationships to an existing master clock using the synchronizer info file. (Refer to Sec.4.1.3.3).

Using a testcase that contains a dummy clock does not have any relations with master clocks, @Verifier lists this clock in this category. A hyper-linked line number highlights line of unrecognized clock definition instead of lines of unrecognized clock used.

5.3 Multi-Cycle Path (MCP) logic

For this check, users need to provide the MCP file containing the list of multi cycle path using "atadaptive.tcl" script. @Verifier reads the file and generates several formal properties to make sure whether the output of the source FF is not read by the input of the destination FF, until the number of cycles specified is reached. If MCP file is NOT given by users, even if users turn on this check in "atadaptive.tcl" script, @Verifier flags an error for this.

Since this check is off by the default in "atadaptive.tcl" script, users may modify this script to turn on and provide the MCP file. Users need to give the information in MCP file. For example:

@adaptive multi_cycle_path "MCP file name" on; (in the atadaptive.tcl script)

number_of_cycle signal_1 signal_2 : (in MCP file)

In this case, we use the same testcase as Figure 5-6. A testcase set the MCP path from Mult_Out to Out and Read_Mult takes 2 clock cycles to be enable the mux. Using this information, the syntax of MCP file will be like “2 Mult_Out Out”. @Verifier identifies MCP path as failure whenever MCP path is longer than it is supposed to be. Line of errors is hyper-linked to source code. The error message is hyper-linked to waveform viewer and waveform viewer shows inputs of design and some inputs are redundant. Since the signal name is hierarchical, it is a nuisance to display signal name itself if there are multiple hierarchy.

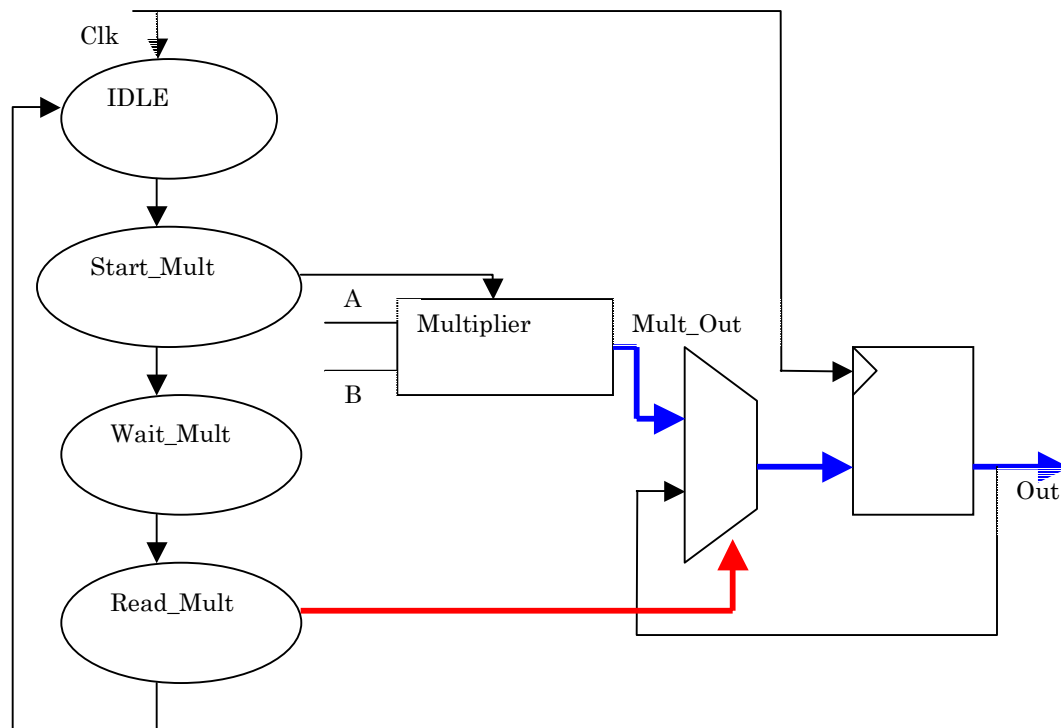


Figure 5-6: MCP example from @Verifier Training v.2.5

5.4 False timing path logic

Users need to provide @Verifier the file containing false timing path for this check like MCP and @Verifier read that file and generate corresponding properties to make sure whether false paths given by users will never be active in the design. (i.e. users identify the false timing path and when @Verifier sees only the false timing paths given by users active, it flags as False Path Error.). False timing path assertions can work at RTL or Gate level, but the normal restrictions on the constructs @Verifier support at gate level still apply. (i.e. UDPs, transistor logic, tristate, etc. are NOT supported).

Since this check is off by the default in “atadaptive.tcl” script, users may modify this script to turn on and provide the FTP (False Timing Path) file. Also users need to provide the information for false timing path in the design. For example:

@adaptive false_paths “false timing path file name” on; (in “atadaptive.tcl script)

signal_1 signal_2 (in false timing path file)

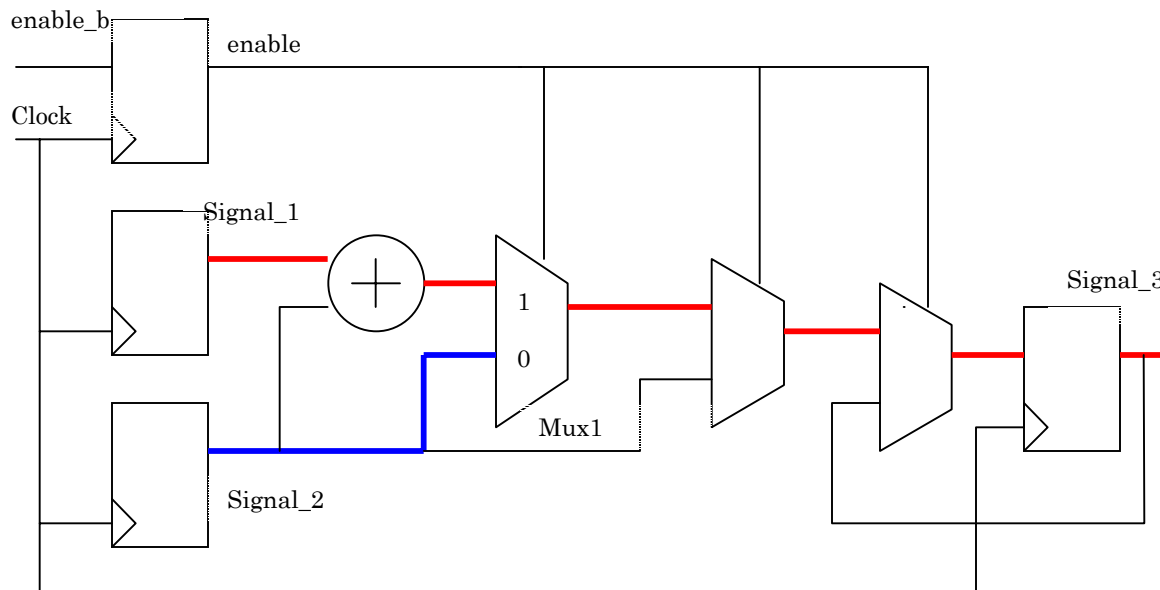


Figure 5-7: False timing path example.

In Figure 5-7, the false path from “Signal_1” to “Signal_3” (red line) is specified in a falsepath file using the *atadaptive.tcl* script. @Verifier uses the false path information from the falsepath file and verifies that the path from “Signal1” to “Signal_3” is active, in this testcase. When the “enable” signal hi, which makes the path “Signal_1” to “Signal_3” active, @Verifier flags a false path error. The error message is hyperlinked to the line in RTL source code. In this case, clicking the error message takes you right to the line in the source code that defines the register that outputs Signal_3. The false path error message is also hyperlinked to the waveform viewer, which brings up the appropriate waveforms that proves that the path in question is not a false path (the clock, enable_b, and enable signals for this example).

5.5 Parallel and full case statements

These checks verify that case statements specified as “parallel case” using synopsys directive (ie. *//synopsys parallel case*) during the synthesis truly have mutually exclusive conditions and case statements specified as “full case” using synopsys directive (ie. *//synopsys full case*) during synthesis don’t have unnecessary latches because of missing cases. Users need to specify these cases in RTL code in order to have these checks. The checks are on by default, and can be turned off using local “atadaptive.tcl” script. For example:

```
@adaptive parallel_case off;
```

```
@adaptive full_case off;
```

- Parallel case statement
- Full case statement observation

@Verifier detects full case in Verilog regardless of synopsys full_case directive when full_case check is on in *atadaptive.tcl* script. If the logic is not literally full case (i.e. there is a missing case) and

specified as full case, @Verifier verifies whether a missing case is used in design. If so, @Verifier flags it as Fullcase error. Like the Domain Data Stability Error, this error message is also hyperlinked to the waveform viewer. For example:

```
Case (full_case) //synopsys full_case
  2'b00 : fc = 2'b01;
  2'b01 : fc = 2'b10;
  2'b10 : fc = 2'b00;
  default : fc = 2'b00;
endcase
```

Figure 5-8: Full case example

In Figure 5-8, the case (full_case = 2'b11) is missing and when the same case is used in the design, @Verifier detects it and flags it as Fullcase error.

5.6 Redundant latch pairs & unintended latch

@Verifier identifies the latches in RTL code and verifies the conditions for redundant latch pairs and unintended latch.

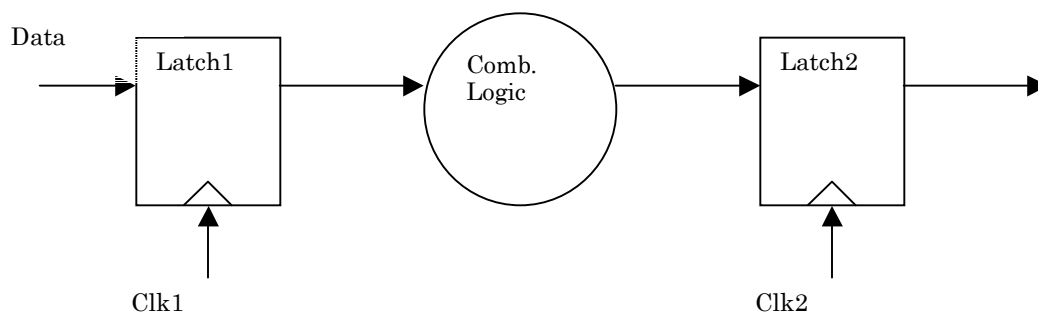


Figure 5-9: Redundant latch example

In Figure 5-9, when Clk1 and Clk2 are overlapping or same, this condition makes both Latch1 and Latch2 are transparent at same time and Data bypass Latch 1 and Latch2 at once. Only when Clk1 and Clk2 are non-overlapping, Latch1 and Latch2 are latching data independently at each clock phase acting like register. @Verifier checks that latches extracted from designs have this condition and if they do, @Verifier flags as “Redundant Latches”.

Moreover, default case or missing case statements exists in either case statement or if/else statement in RTL code, synthesis engine generates unintended latches instead of mux. @Verifier makes sure that unintended latches are generated by such a condition and if so, @Verifier will flag as “Unintended Latches”. We did not explore it, but users can avoid false errors by using constraints.

5.7 One hot drivers

This checks that the drivers for a tristate bus are one-hot, i.e. users doesn't get multiple drivers on a bus at one time. The check is on by default, and can be turned off using local "atadaptive.tcl" script. For example:

```
@adaptive one_hot_drivers off;
```

This section in the manual is NOT explained at all in the current version (v2.5.1), but it will be in manual v.3.0.

5.8 Index out of range

@Verifier checks that indexes specified for memories and buses are within specified range. For example, in the case of the vector "reg[31:0] a", @Verifier will flag if the variable a[i] with $i < 0$ & $i > 31$ is read or written. Also, in the case of memory, "reg[31:0] mem[1000:2000]", @Verifier will flag if the expression, mem[j] with $j < 1000$ & $j > 2000$ is accessed for read or write in the design.

Using a testcase that is a module (~83k gates) of our DUT for this check, @Verifier catches index out of range failure that is following.

Input A is 64-bit data and input B is 9-bit data. A[B] is used as a mux in a testcase and index B is out of range (supposed to be 6-bit) in this case and @Verifier flags.

5.9 FIFO Verification

There are two areas that @Verifier identifies for FIFOs. The first is clock domain analysis already covered in Sec. 5.2; it is one of the synchronizer types @Verifier recognizes.

The second area is following:

- Multiple writes before read in the FIFO
The location in the FIFO has multiple data written to it before any read is performed.
- Read before write
Read of a location is taking place before the data is written (i.e. the data read is old, stale data)
- Memory address out of range
The address decoded by the device is not in the range of the memory specified.
- Memory address unknown during write
During a write operation if the address has unknown contents (indicated by X's), @Verifier detects logic that generates an unknown address and notifies the user. Either the logic needs to be corrected or the unknown contents needs to be filtered out.

For the second area, @Verifier runs these properties, not with formal verification, but with random simulation. FIFOs implemented with Memory, which are non-synthesizable (or behavioral) logic can't be done in formal verification since one of the steps under the hood of @Verifier is a synthesis. @Verifier only runs these properties with random simulation and for random simulation, any sort of FIFOs is working with @Verifier. We did not explore this feature.

5.10 Reachability

AFV can check that all code is reachable based on some input combination, which may or may not be legal in the real system (constraints can be applied here). However, there is an issue with reachability

property analysis. While reachability property analysis finds design bugs, they are very time consuming to check and often the number of reachability assertions can be large. Even a small design can yield over 10000 reachability properties. Meaning that running this with the default timeout of 45 seconds on one CPU could take a day+. In the case of our DUT (~250K gates), the runtime takes an estimated 4.5 days (Note: this is an estimate because we killed the job after it ran for 2+ days). There are several ways of reducing this runtime issue with reachability analysis:

- Use the command line switch `+no_reachability` to run without checking reachability properties
- Adjust the timeout setting (default timeout limit is 45 seconds).
- Random Simulation (Refer to Sec. 4.1.5).
- Distributed Processing
- Bounded Model Checking (Refer to Sec. 4.1.2)

6 Result Viewing and Debugging with @Designer

@Designer is @HDL's graphical verification debug cockpit, which has a similar feature set to Debussy (ie. FSM diagramming, waveform viewer, source code window, and several buttons to interrogate and move around the design).

After the @Verifier run is finished, users can view the results and debug failing properties with @Designer by viewing the "atverifier.log" file or invoking the @Designer GUI with the `+gui` switch in the @Verifier run command. The GUI consists of the @Designer window and a *Verifier Window*, which shows failures and errors, if any. If there are no failures or errors, then Verifier Window gives the message "Empty reply from server".

As part of the @Designer debugging environment; users can also examine the waveform for failed properties to determine whether the condition can actually occur. Then, the designer can either correct the design or place constraints on the input condition to avoid "false" property failure.

The `+mc_status` switch invokes the Property Monitor window while the @Verifier run is executing. Among other things, this window shows how long a job has been running and gives an estimate of how much longer the job will take to complete. It is very useful for watching the status of long runs.

The Verifier Window shows all failures and errors by module and by category, or by both module and category DRC, Simulation and Formal. It provides a convenient debug interface for DRC errors and property failures by hyperlinking the failure/error's line number to the line in the source code that actually caused the error. Moreover, it has a nice feature that allows users to search for failures or errors by Module, Category, and Sub-category (General,FSM,Flops,Logic,Synthesis,AFV, etc).

However, there are the following issues:

- During simulation, there is no GUI that could control the simulation process like stop, interrupt, save, resume, and so on. Currently, @Verifier is running in batch mode and all GUI windows (@Designer, Verifier Window, Property Manager) are invoked at end of simulation even if the `+gui` switch is assigned.
- Some messages are NOT informative enough for debug and also give the false line number.

7 Conclusion

Our first attempt to run a full-chip simulation on our test design resulted in long runtimes, at day+ without any special switches or tricks. @HDL does a good job of reducing runtime by using random simulation, ignoring reachability properties, bounded model checking, incremental runs, and distributed processing. @Verifier didn't show any design size capacity issues. Again, our test chip was about 250k gates. A larger design would obviously push the tool's limits for design size capacity.

@Designer appears to seamlessly interact with @Verifier for tracking down bugs, whether by viewing the FSM diagram, waveforms, or highlighting the offending lines within the source code. However, we found that the error messages reported by @Verifier can sometimes be inadequate feedback for the individual doing the debug. We also ran into issues with the error message being hyperlinked to the waveform viewer and couldn't always get @Designer to properly open the waveform view. For most of our feedback, @HDL is either working a fix into a future rev of the tool or investigating the issue. Here's a list of caveats and gotchas:

- @Verifier can have trouble finding module definitions depending on the order the modules are defined. For example: 4 levels of hierarchy (a, b, c, and d from root to leaf) exist in a design; the top-module (a) is defined in one file (a.v) and the rest (b, c, and d) in a separate file (b.v); modules b, c, and d are defined within b.v in the order b, d, then c; the path for b.v is given with the `-y` command line option; a.v is given as the top-level Verilog file. Under these conditions, @Verifier will error out because it will not be able to find the definition for module "d". A workaround to this problem is to define the modules in the same order as they appear in the hierarchy, from root to leaf.
- Full chip runtimes can be extremely long (on the order of several days). @HDL now has a Runtime App Note which discusses several techniques to shorten runtime: turn off reachability analysis; invoking random reachability analysis simulations before invoking the formal engine; BMC (Bounded Model Checking); BMC with the `+no_atfv` flag; distributed processing using LSF or other utilities; and incremental model checking.
- The tool doesn't handle multiply embedded ternary (`Y = Sel ? A:B`) statements very well. For example: If an assign statement has 6 ternary statements embedded, @Verifier may flag reachability property failures but will not be able to distinguish between which part of the statement is not reachable and which part of the statement is redundant. When @Verifier sees redundancy in assign statement containing multiple ternary statements, it confused with between statement that is redundant and statement that is not reachable. Admittedly, this coding style can simply be uninviting to read, but it's valid Verilog and our test design used this in many places. We recoded these statements as `if, else if, and else` statements.
- @Verifier currently does not offer wildcard blackboxing of modules (eg. `ram*`, `*ip_module*`, etc.). They are investigating working this into a future rev.
- Using the `+synchronizer_rank` switch in conjunction with the `+dir` switch causes @Verifier to always pass synchronizer rank checks even if the implemented rank is less than the specified rank (Refer to Section 5.2.3).

The following is the list of items that could use further investigation to enhance this document:

- Simulation with PLI – This is NOT well documented and @HDL plans to address this in the next manual rev (v3.0).
- OVA assertions
- Distributed processing – This uses LSF or other distributed processing applications to run jobs on multiple machines and merge the results when the distributed jobs are completed. When LSF license is available, it needs to be investigated.

This evaluation was carried out using software from @HDL with their release from December 2002, @Verifier version 2.8, and has therefore undergone several iterations since the original evaluation. At the time of our evaluation the tool was undergoing a constant series of improvements. We also evaluated only a subset of its features, leaving out things like OVA assertions and distributed processing. Despite some of the issues raised herein, we feel that @Verifier warrants further investigation by DV teams embarking on new or existing verification efforts. It has some powerful features and has the potential to significantly accelerate and increase the efficiency of DV efforts.

8 Appendix A: FSM Testcase Source Verilog

```
module deadlock ( rst_n,
                  clk);

    input rst_n;
    input clk;

    reg [1:0] fsm1;
    reg [1:0] fsm2;

    reg  advnc_fsm1;
    reg  fsm2_rst;

    parameter [1:0] IDLE_FSM1  = 2'd0;
    parameter [1:0] STATE1_FSM1 = 2'd1;
    parameter [1:0] STATE2_FSM1 = 2'd2;
    parameter [1:0] STATE3_FSM1 = 2'd3;

    parameter [1:0] IDLE_FSM2  = 2'd0;
    parameter [1:0] STATE1_FSM2 = 2'd1;
    parameter [1:0] STATE2_FSM2 = 2'd2;
    parameter [1:0] STATE3_FSM2 = 2'd3;

    // FSM1
    always @(posedge clk or negedge rst_n)
        if (~rst_n)
            begin
                fsm1[1:0] <= 2'd0;
                fsm2_rst <= 1'b0;
            end
        else
```

```
begin
//      fsm2_rst <= 1'b0;
case (fsm1)
  IDLE_FSM1 :
    if (advnc_fsm1)
      fsm1[1:0] <= STATE1_FSM1;

  STATE1_FSM1 :
    if (advnc_fsm1)
      begin
        fsm1[1:0] <= STATE2_FSM1;
        fsm2_rst <= 1'b1;
      end

  STATE2_FSM1 :
    if (advnc_fsm1)
      fsm1[1:0] <= STATE3_FSM1;

  STATE3_FSM1 :
    if (advnc_fsm1)
      fsm1[1:0] <= IDLE_FSM1;
endcase // case(fsm1)
end // else: !if(~rst_n)

// FSM2
always @(posedge clk or negedge rst_n)
  if (~rst_n)
    begin
      fsm2[1:0] <= 2'd0;
      advnc_fsm1 <= 1'b0;
    end
  else
    case (fsm2)
      IDLE_FSM2 :
```

```
begin
    advnc_fsm1 <= 1'b0;
    if (~fsm2_rst)
        begin
            fsm2[1:0] <= STATE1_FSM2;
            advnc_fsm1 <= 1'b1;
        end
    end

STATE1_FSM2 :
begin
    if (fsm2_rst)
        fsm2[1:0] <= IDLE_FSM2;
    else
        begin
            fsm2[1:0] <= STATE2_FSM2;
            advnc_fsm1 <= 1'b1;
        end
    end

STATE2_FSM2 :
    if (fsm2_rst)
        fsm2[1:0] <= IDLE_FSM2;
    else
        fsm2[1:0] <= STATE2_FSM2;

STATE3_FSM2 : fsm2[1:0] <= IDLE_FSM2;
endcase // case(fsm2)

endmodule // deadlock
```