Innovative Uses of SystemVerilog Bind Statements within Formal Verification

Xiushan Feng, Christopher Starr
Samsung Austin R&D Center, Austin, Texas
(s.feng; c1.starr)@samsung.com

***Abstract*- Bind statements inside SystemVerilog are frequently used by simulation-based verification benches to add verification code to RTL design modules, interfaces, or compilation-unit scopes. With bind statements, verification code is separated from RTL design, so design and verification teams can maintain their own files. Under the context of formal verification, bind statements are also heavily used. In addition to common usages, due to the unique nature of formal verification, the bind statements become more powerful. For example, inside formal verification, an assumption can be used to constrain inputs; a snip point can be used to cut the driver logic of a signal; a floating net or un-driven signal can be controlled by formal tools the same way as primary inputs. In this paper, the authors will use real-life examples to demonstrate a few innovative uses of bind statements within formal verification.**

I. Introduction

Let's first have a quick review for the syntax of SystemVerilog bind statements and basic usages within the SystemVerilog language.

1.1. Basic Usages of Bind Statements

SystemVerilog IEEE1800 standard defines the following two formats of a bind statement [1].

```
bind_directive ::=
     bind bind_target_scope [: bind_target_instance_list] bind_instantiation ;
   | bind bind_target_instance bind_instantiation ;
```

The first format binds to a module or interface. Without the optional [: bind_target_instance_list], the bind will be applied to all target instances of the module or interface instead of a selected list of instances. The second format binds to one single instance. The resolution of bind_target_instance follows SystemVerilog scope resolution rules. The bind_instantiation is defined as a program/module/interface/checker instantiation. For example, if we need to bind a checker module to an RTL design module, the bind_instantiation is how we instantiate the checker module within the RTL module.

```
bind_instantiation ::=
                        program_instantiation
                      | module_instantiation
                      | interface_instantiation
                      | checker_instantiation
```

When using a bind statement, port directions should be consistent with the Design Under Test (DUT). E.g., if in the RTL of the DUT, port *foo* is defined as an input and port *bar* is defined as an output.

```
module DUT (
  input foo,
  output bar,
…
```

In a checker module, we very often need to define both *foo* and *bar* as inputs to passively read signals from the design instead of driving them.

```
module checker(
 input foo,
 input bar,
 …
);
…
endmodule: checker
bind DUT checker checker_inst (.foo(foo), .bar(bar));
```

1.2. Formal Verification and SystemVerilog Assertions

Within formal verification, the most common usage of bind statements is to bind a module (or interface) with SystemVerilog Assertions (SVAs) to a DUT. Formal verification is assertion-based design verification. Properties are written as SVAs to define assumptions as constraints and assertions as verification targets. Guided by assertions, formal tools exhaustively explore the design state space and verify design features defined by assertions. There are some different opinions on where to add SVAs [2][3]. For example, you can embed assertions within the RTL design files or you can add assertions through bind statements outside of the RTL files. For our projects, we have a mixed solution. We encourage RTL designers to add in-line assertions during the development (RTL bring-up) stage, and require all assertions written by verification engineers to use separate assertion files together with bind statements. This guideline serves us very well. Formal verification and emulation tools can pick up SVAs directly from the RTL design files without fighting the non-synthesizable structures from the verification environment and still selectively pick up assertions from the verification environment using bind statements/files. In addition to the existing assertions, formal verification engineers can create SVAs outside of the RTL files using bind statements and properly categorize them for other tools/flows to consume. Some assertions are written for formal verification only, while the others can be used by simulation and/or emulation tools.

When doing formal verification, the formal tool reads the RTL design files, which have design logic and possibly embedded assertions. Then, the tool will process assertions outside of the RTL design files using bind statements. When all these System Verilog files are read in, the formal tool will use a TCL control file which defines basic tool settings, such as clocks and resets for formal search. With this information, the formal tool applies all possible inputs to drive the DUT and searches the design state space. These "all possible inputs" are finite sequences of input values that satisfy the current set of formal assumptions. In order to accurately and concisely implement SVAs, formal verification engineers may write RTL helper logic, similar to what RTL designers create, in order to build signals for the assertions to use. During formal search, the formal tool proves or disproves assertions. If an assertion is disproved, a counter example (CEX) trace will be provided for debug. The proof result can be inconclusive if no counter example is found within a certain cycle bound after circuit reset. Figure 1.1 shows a simple but very common formal property verification test bench. A bind statement is used to connect assertions with the DUT.

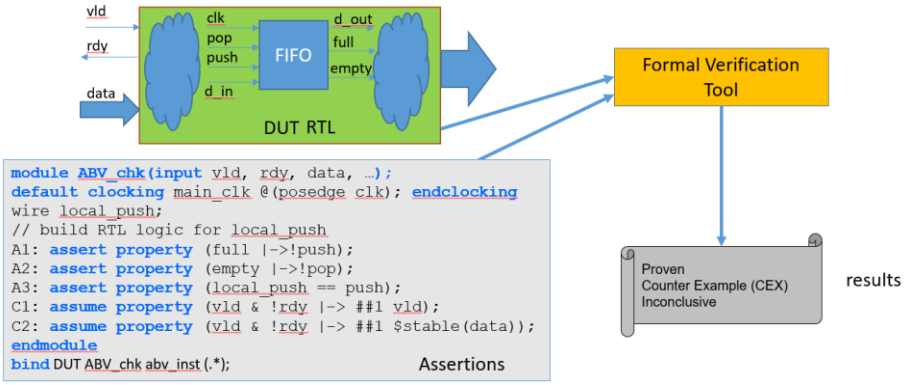


Figure 1.1 A simple formal property verification test bench

When using bind statements within formal verification, formal tools obey all rules defined by SystemVerilog IEEE1800 standard, but will also have its own special usages.

First, the primary inputs of the DUT are control points for formal tools to drive using random values as long as no constraint is violated. For the *checker* module example from 1.1, we should define *foo* as an input to avoid conflicts. Un-driven signals are control points too. An X-assignment can be treated as un-driven because formal tools can use either 0 or 1 to replace an X randomly. With formal verification tools, there is a way to remove the driver logic of a signal. We call it a "snip" or "cut" point. Within TCL control files processed by formal tools, we can force any signal to be un-driven and let formal tools, instead of design, control it.

For example, in Figure 1.2, we snip signal *FOO*. After applying the snip_driver command (from a TCL control file), *FOO* becomes un-driven. The original driver of *FOO* can still be referred to as $driver(*FOO*), which is the logic that drives *FOO* before applying the snip_driver command. In this paper, we use Synopsys VC Formal TCL commands [4] to describe the examples. Most other formal verification tools have similar interfaces with equivalent commands or system function calls.

Figure 1.2 Snipping internal signals

Second, we can use assumptions to control "physical" connections. Almost all formal tools have the capability to support concurrent assertions and immediate assertions. Immediate assertions cannot be bound through a bind statement directly. We can use an assume-type concurrent assertion to tell formal tools that some signals are tied together after reset or even within the reset stage (through a TCL assumption).

For example, in Figure 1.1, we can ignore port connections by removing ".*" from the bind statement. Now, all input ports are un-driven. Assumptions can be used inside the *ABV_chk* module to connect ports, such as:

```
ASM_conn_vld: assume property (@(posedge clk) vld == DUT.vld);
```

With the above assumption, the *vld* port inside the *ABV_chk* module is driven by the *vld* from the DUT through the formal assumption *ASM_conn_vld*.

In the next sections, the authors will present some real-life examples of how to leverage features from most formal tools and combine them with bind statements to make formal verification tasks much easier. We not only want to spend less time to build the bench, but also want to spend less effort to maintain the bench in the future. These examples are based on hundreds of formal verification test benches that the authors have built. We have compiled a list of usages that are not obvious to most new users of formal verification tools.

II. BIND STATEMENT IN FORMAL CLOCK GATING VERIFICATION

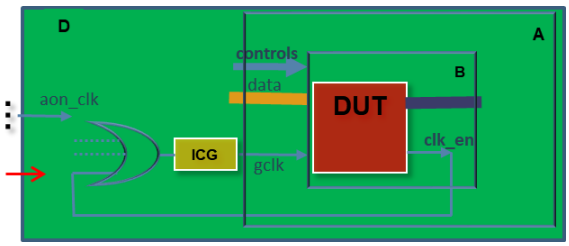
2.1. Problem Statement

Our designs for mobile devices have very aggressive clock gating to save power. Formal sequential equivalence checking tools are used to prove the equivalence of the design with and without clock gating. If a heavily clock gated design (implementation) is functionally equivalent to the same design that has clock gating disabled by setting a clock gating disable chicken bit (specification), then we have proved the correctness of the clock gating implementation. For some large designs, a divide-and-conquer approach has to be done at the sub-unit level to have a conclusive proof. Later, at the unit level, we can carry over proof information from the sub-units to simplify the the unit level formal verification. At sub-unit level, most modules have clock enable logic without instantiations of clock gating cells. The gated clocks are primary inputs of these sub units, and we need to build test benches for the sub units quickly. Also, the RTL designers may be busy implementing new functionality and put off instantiations of the clock gating cells until later. If the clock enable logic exists, we can get an early start on clock gating verification.

2.2. Bind Clock Gating Cells to Any Design Module with Clock Enable Defined.

Our solution is to use a bind statement to connect an ICG (Integrated Clock Gating) cell to the DUT, which is a sub-unit of the implementation design that has clock gating enabled, and then use a TCL control file to connect signals.

Remember, our goal is to prove the sequential equivalence of the design with and without clock gating. We call the design instance without clock gating the specification design where the clock enable logic within the ICG is always true due to setting a chicken bit. All gated clocks in the specification design are active. Whether a gated clock is active or not is controlled by the clock enable from the implementation design where the chicken bit is not set. Figure 2.1, we have a unit D. A sub-unit, DUT, is the module that we want to verify. Our goal is to have a conclusive answer, or full proof for the sub-unit DUT.




```
bind DUT:impl ICG fv_icg (
 .oclk(), // output
 .iclk(),  // input
 .enable(clk_en), // input
…);
```

Figure 2.1 A simple clock gating formal verification test bench for a sub-unit

At the module boundary of the DUT shown above, there is a *clk_en* signal output. It will be used to turn on the gated clock, *gclk*, outside the DUT. The clock gating cell, ICG, is outside the DUT. Our goal is to prove the equivalence with and without clock gating for the DUT module.

The most common approach will be to create an RTL wrapper module for the DUT that instantiates the ICG cells and uses the wrapper as the top module for formal tools. This will require port definitions and connections. If there are RTL updates for the ports, extra effort will be needed to maintain the bench.

For this problem, we have a very simple solution using a bind statement with a minimum number of lines for a test bench. All we need to do is bind the missing ICG cell into the design. We don't need to connect all the ports of the ICG cell. We can use formal assumptions build the rest of the connections.

The right-side figure of Figure 2.1 shows the bind statement. A signal, *clk_en*, is an output of the DUT that enables the gated clock. We can leave the input and output clocks floating. Within the sequential equivalence checking tool, we can map all inputs excluding the gated clock input, create a formal verification clock, and then connect the gated clock from the bind statement to the implementation design.

```
map_by_name -exclude gclk
create_clock {spec.gclk impl.fv_icg.iclk}
fvassume impl.gclk == impl.fv_icg.oclk -env
```

For this clock gating formal sequential equivalence checking problem, we have two instances of the same design, spec and impl. The first TCL command is used to map all IOs excluding *gclk* between spec and impl. For a pair of mapped inputs, the tool will drive them with the same input values (i.e., implicitly doing *fvassume spec.in == impl.in*). For a pair of mapped outputs, the tool will create assertions to make sure the two outputs are equivalent every cycle (i.e., implicitly doing *fvassert spec.out == impl.out*).

In formal verification, we need an always-on clock to drive the design and the assertions. This always-on clock will drive the input clock for the spec instance and the input clock for the ICG cell bound to the impl instance (with a hierarchy name *impl.fv_icg.iclk*). The formal tool has a create_clock command for this purpose.

Finally, we need to connect the output clock from the ICG cell to the impl instance as its input clock. Whether *impl.gclk* is active or not is controlled by *clk_en*. The bound ICG instance creates the gated clock logic. We simply connect it to *impl.gclk* by a formal assumption. The option, *"-env"*, is used in the TCL command to tell the formal tool that this assumption should apply to the formal reset simulation stage also. This allows the sub-unit, DUT, to be properly reset for formal verification.

We have had a big success with this approach for sub-unit clock gating. We achieved a lot more full proofs by using assume-guarantee at the unit level. The basic idea is to snip and force the gated clocks of some sub-units to be a static clock (assume) after we have full proofs (guarantee) for the sub-units. In the end, more than 30% of the units for the project have full proofs for clock gating.

2.3. Apply Temporary RTL Fixes without Touching Any RTL files.

We found another way to use bind statements when we were working on clock gating formal verification. For some newly developed units, we caught several clock gating bugs in a short period of time and the RTL designers provided several local RTL fixes. These RTL changes need to be formally verified before pushing them into the release repository. In our environment, the RTL files are generated by an RTL build system. If possible, we don't want to modify the local RTL files directly. We also don't want to change the source files and re-generate the RTL files, which can be a very time-consuming process. Using bind statements with some tool TCL control commands can help us simplify this process.

Our solution is to create a new module with fixed RTL logic and bind it to the original RTL module. Then we use a snip point to remove the driver logic from the buggy RTL and let the modified RTL logic from the bound module drive the buggy RTL with the fixed logic.

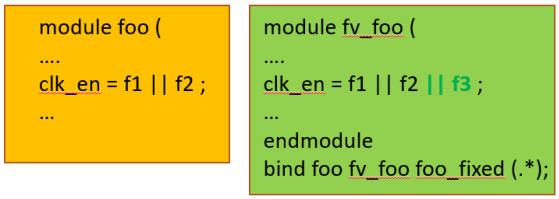


Figure 2.2 Apply an RTL fix without touching any RTL files

For example, in Figure 2.2, a bug was found within module *foo* and the RTL designer proposed to add a missing term, *f3*, to the clock enable logic (*clk_en*). Instead of modifying the local RTL file (e.g., *foo.sv*), we can create a very simple *fv_foo* module that has only the bug fix. A few lines of RTL as shown in Figure 2.2 will do this with the following TCL commands.

```
snip_driver -env -seqmap -exact spec.foo.clk_en
fvassert -expr {$drive(spec.foo.clk_en) == $drive(impl.foo.clk_en)}
fvassume BUG_xxx_fix -expr {impl.foo.clk_en == impl.foo_fixed.clk_en} -env
```

The first TCL command will snip the driver of *foo.clk_en* for both the spec and impl instances, then map the two snipped points – i.e., assume they are driven by the formal tool using the same values. At this point, *clk_en* from both spec and impl are un-driven but mapped. The second command is used for a safety check (the guarantee part of an assume-guarantee approach). We need to make sure that when we snip and map a pair of internal points, the two signals should have the equivalent drivers. Otherwise, the pair is not safe to map (assume equivalent). The third TCL command uses the modified RTL logic to drive the snip point with the fixed logic for the impl instance. Because both the impl and the spec instances are mapped for this snip point, they are driven with equivalent values by the formal tool. This means the assumption is implicitly applied to spec instance as well.

For this case, we can avoid using an extra module and bind statement, and purely rely on a TCL command if the missing term (*f3*) exists at the original design. E.g.,

```
fvassume BUG_xxx_fix -expr {impl.foo.clk_en == ($driver(impl.foo.clk_en) || impl.foo.f3)}
```

However, if the RTL fix is complicated and requires extra logics (such as flops), it will be much easier to use a module with a bind statement. Also, it is easy to document different bug fixes using different System Verilog code with the same generic TCL commands.

Once the RTL fixes have been verified, the modified RTL files will be submitted into a repository. Then we will remove our binds and TCL commands as soon as the RTL is released.

III. BIND STATEMENT FOR C2RTL DATA-PATH FORMAL VERIFICATION

3.1. Problem Statement

When working on data-path verification using C vs. RTL (or C2RTL) formal equivalence checking tools, sometimes, we found it is very hard to map IOs between the RTL and C models due to different implementations and data formats. Instead of creating complicated TCL scripts to build the mapping logic, we prefer using bind statements with RTL helper logic. We will use a C2RTL formal bench for the GPU color blender as an example to demonstrate this.

For the GPU color blender design, we have two implementations. One is a C model and the other is an RTL design. The C and RTL models have different encodings for the RGBA color format of the source input data. Using TCL commands inside the C2RTL tool to map each input bit is a tedious task because it is difficult to create constraints for valid inputs. Several text macros for color formats are used inside both the RTL and C models. These text macros can change over time and the values of the text macros don't necessary match between the C and RTL models. Text macro resolution for both the C and RTL models within a TCL programming environment will be very complicated. In our case, a bind statement is created for the RTL input interface to align the RTL inputs with the C model. Also, helper RTL logic can define valid input combinations also.

3.2. Bind Statement to Map and Constrain Inputs with Data-path Verification

With a bind statement and a map_bits module, we can re-assemble the RTL input source color data (src_color) from 38b (ALU_PIXEL_WIDTH) to 32b (PIXEL_WIDTH) per pixel in a way that can be mapped back to the integer types within the C model based on the destination's color format (color_fmt). In addition, we can also create an input valid check vector (valid_pixel). If any bit of a pixel is invalid, valid_pixel will be cleared for that pixel. If

all input pixels are valid, inDataValid flag will be set. This signal can be used to constrain the input data as shown below.

```
module map_bits #(`include "cb_params.vh")( …);
…
always_comb begin
   for (pix_num=0; pix_num<NUM_PIXELS; pix_num=pix_num+1) begin: pixel
    valid_pixel[num_pix] = 1'b1
    case (color_fmt)
      FORMAT_1: begin
          …
       newData[(pix_num * PIXEL_WIDTH)      +: 8] = inData[(pix_num*ALU_PIXEL_WIDTH)       +: 8];
       newData[(pix_num * PIXEL_WIDTH + 8)  +: 8] = inData[(pix_num*ALU_PIXEL_WIDTH + 10)  +: 8];
       newData[(pix_num * PIXEL_WIDTH + 16) +: 8] = inData[(pix_num*ALU_PIXEL_WIDTH + 20)  +: 8];
       newData[(pix_num * PIXEL_WIDTH + 24) +: 8] = inData[(pix_num*ALU_PIXEL_WIDTH + 30)  +: 8];
           ….
      end
      FORMAT_2:…
     …
     // many other formats
      default: begin
            valid_pixel[pix_num] = 1'b0;
            newData = 'X;
      end
    endcase
 end : pixel
 assign inDataValid = &valid_pixel;
…
endmoule

bind color_blender map_bits src0 (.inData(src0_color),.color_fmt(src0_color_fmt), …);
```

Within the C2RTL TCL control file, source data 0 can be mapped between the C model (spec) and RTL (impl) at phase 1 using two simple assume commands. The second assume will make sure that only valid pixels with a valid color format can be used to drive both the C and RTL models.

```
assume spec.src0Data (1) == impl.src0.newData(1)
assume impl.color_blend.src0.inDataValid (1)
```

The advantage of this approach is that within the RTL helper logic, all text macros from the RTL design are available. We don't need to use any hard-code text macro values to align the C and RTL. If the values of the RTL text the macros change in the future, we don't need to update the test bench. Besides this benefit, it is much easier to implement the helper logic using the power of the SystemVerilog language. The RTL helper logic can be easily understood and supported by other engineers.

For a few cases, when we use RTL helper logic with bind statements within C2RTL benches, we also found that it makes the convergence effort much easier. We can do case-splitting or assume-guarantee on the RTL helper logic that can significantly simplify formal proofs. You can find more discussions on C2RTL convergence techniques from [5].

IV. BIND STATEMENT FOR DECOMPRESSOR FORMAL VERIFICATION

4.1. Problem Statement

We had difficulties using C2RTL to verify a graphic primitive decompressor because the C model was not formal-friendly. Modifying the C model would take a large amount of time. Meanwhile, we had formally verified the compressor logic using C2RTL formal verification. We decided to use the compressor RTL to verify the decompressor RTL. The challenging part of this problem is that the compressor and the decompressor are located in different GPU blocks. The module boundary and functionality of these two modules make it impossible (even for the RTL designers) to connect them directly using a wrapper RTL circuit without major modifications.

4.2. Snip points with Bind Statements

Our solution was to find a location within the decompressor logic that can take compressed data as an input. This can be an internal net/bus. We applied snip points inside the decompressor to remove the logic driving the compressed data. Then we used a bind statement to connect the compressor and the decompressor with assumptions to drive the snip points with the data coming from compressor directly. Our assertion inside the C2RTL tool is very simple. All we need to verify is that the data outputs of decompressor are equivalent to inputs of the compressor.

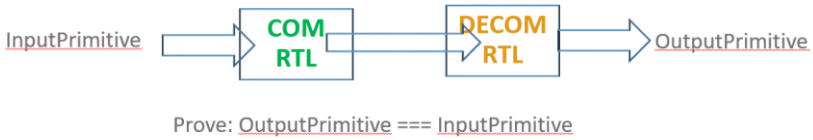


Figure 4.1 Verify decompressor using compressor RTL with bind statements

This is the bind statement.

```
bind COM DECOM decomp_inst (.sclk(sclk), .rst(rst));
```

With this bind statement, the decompressor module (DECOM) is instantiated under the scope name of the compressor (COM). Within the C2RTL tools, we can refer to signals inside the decompressor using COM.decomp_inst.<signal name>.

Our next step is to locate where the compressed primitive data is assembled and ready to be decompressed. In this design, it is an internal signal. We can use these TCL command to connect the compressed data to the decompressor.

```
set_cutpoint COM.decomp_inst.next_data_buffer_p0
//compile and elaborate design
…
cutpoint DECOM_indata_cut = COM.decomp_inst.next_data_buffer_p0 (1)
assume DECOM_dataInput = { DECOM_indata_cut[275:127] == COM.code_final[149:1](1) }
```

The above VC Formal DPV commands [6] will set a snip (cutpoint) point before the RTL compile and elaboration stage, then activate and drive the cutpoint at clock phase 1 only. At phase 1, we use an assume command to connect the compressed 149bit data from the compressor to drive the input of the decompressor. In our case, neither signal, *code_final*, inside the compressor nor *next_data_buffer_p0* inside the decompressor are defined as module IOs. Both of them are all internal wires.

Using "cutpoint" and "assume" TCL commands within the C2RTL tool, together with a simple bind statement, we can create wrapper circuitry to make any connection we want. Using this approach, we proved the decompressor design in a short period of time without fighting the complications of C models.

V. Bind Statements for Combinational Circuits with Concurrent Assertions

5.1. Problem Statement

Sometimes, we need to write assertions for a DUT with only combinational logic. There is no clock and reset port within the DUT scope. Most formal property verification tools require clock and reset signals. Even when a formal tool doesn't require a clock or reset port, we would like to avoid creating immediate assertions which cannot be bound directly without wrapping them inside a procedural block. In addition to allowing direct binding, concurrent SVAs are cycle based and won't have asynchronous behaviors between clock cycles that slow down evaluations and introduce potential contention from combinational glitches. Therefore, when creating assertions for formal verification, we always prefer concurrent assertions.

5.2. Bind Statement with Dummy Clock and Reset

Our solution is to use floating clock and reset ports at the DUT level for formal verification. If verification (formal, simulation, or emulation) is done a level higher than DUT level, we connect the real clock and reset ports from a module above the DUT.

```
module fv_checker (input clock, reset, ..);
// concurrent assertions
endmodule
`ifdef TOP_DUT
   bind DUT fv_checker fv_inst (.clock(), .reset(), .*);
`else
   bind DUT fv_checker fv_inst (.clock(FOO.clock), .reset(FOO.reset), .*);
`endif
```

For a formal bench at the DUT level (with `TOP_DUT defined), we just need to use these signals to define a clock and reset for formal verification as shown below.

```
create_clock {DUT.fv_inst.clock}
create_reset {DUT.fv_inst.reset}
```

With these commands, fake clock and reset are generated and formal tools will be happy with concurrent assertions within the design. When running the same formal bench at a higher level that includes the DUT, the same *fv_checker* file still works as long as *FOO* is the module where the clock and reset ports exist and *FOO* is on the top of the DUT within the hierarchy.

VI. BIND STATEMENT FOR PARAMETERIZED RTL DESIGNS

6.1. Problem Statement

An Error Correction Code (ECC) RTL design is parameterized for the sizes of data and ecc bits. Throughout the whole project, there are a limited number of configurations (less than 10) that we need to verify without modifying the test bench.

6.2. Bind Statement with Parameters and "-pvalue" from VCS

Our solution uses bind statements to configure components of the ECC logic. By passing parameter values from environment variables to checker modules through bind statements, we can verify all configurations using the same bench. E.g., this is the bind statement.

```
bind ecc_wrapper ecc_checker  #(.DWIDTH(DWIDTH), .ECC_WIDTH(ECC_WIDTH) ecc_check_instance
```

The *ecc_checker* with SVAs is parameterized. Inside a TCL control file (*ecc_check.tcl*), we use environment variables to read parameters so we can create a shell script to verify all possible configurations.

We are also using Synopsys VC Formal as a reference. This tool uses VCS to elaborate designs. All the other formal tools have a similar feature that allows parameters to be overridden using control commands.

```
if {[info exists ::env(DWIDTH)] && [info exists ::env(ECC_WIDTH)] } {
   set   p_str   "-pvalue+ecc_wraper.DWIDTH=$::env(DWIDTH)-pvalue+ecc_wrap.ECC_WIDTH=$::env(ECC_
WIDTH)"
} else {
   set  p_str "-pvalue+ecc_wraper.DWIDTH=256 -pvalue+ecc_wrap.ECC_WIDTH=10"
}
elaborate  -top DUT  -vcs " … $p_str"
```

With this TCL control file, we can use a shell script to run the formal tool for all configurations without updating the formal bench. E.g.,

```
#!/bin/bash
…
export DWITH =128; export ECC_WIDTH=9
vcf -f ecc_check.tcl -out_dir ecc_${DWIDTH}_${ECC_WIDTH}
export DWITH =256; export ECC_WIDTH=10
vcf -f ecc_check.tcl -out_dir ecc_${DWIDTH}_${ECC_WIDTH}
…
```

These commands can be easily added into a regression system and with a simple result passing script, we can fully regress all valid configurations of the ECC logic within the whole project whenever an update is made to the RTL.

In this case, we have less than 10 runs to verify all valid configurations. If the number of possible configurations are big, then other approaches need to be considered. For example, if we can prove that a group of configurations preserve the same property without losing generality, then they belong to the same equivalence class under this property. With this information, we can partition the parameter state space (all possible values of parameters) into equivalence classes. Verifying a representative configuration of an equivalence class will verify all configurations within the same class [7]. With this technique, we can limit the number of formal runs that are needed.

ACKNOWLEDGMENT

The authors thank for great collaborations from Synopsys AE and R&D team, and valuable feedback from Srinivasan Venkataramanan – our paper shepherd from DVCon conference.

REFERENCES

[1] IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language IEEE 1800-2017
[2] SystemVerilog Assertions - Bind Files & Best Known Practices for Simple SVA Usage, Clifford E. Cummings, SNUG, 2016
[3] Who Put Assertions In My RTL Code? And Why? How RTL Design Engineers Can Benefit from the Use of SystemVerilog Assertions, Stuart Sutherland, SNUG, 2015
[4] VC Formal Verification User Guide, Synopsys, 2021
[5] Convergence Techniques for C vs. RTL Equivalence Checking, Xiushan Feng, Li You, Rachna Nambiar Jain, Yong Liu, DAC, 2018
[6] VC Formal Data Path Validation User Guide, Synopsys, 2021
[7] Equivalence classes over parameter state space. Xiushan Feng, Yinfang Lin, Jayanta Bhadra. US Patent number: 9069762. June 22, 2012