A Reusable, Scalable Formal App for Verifying Any Configuration of 3D IC Connectivity

Daniel Han
Xilinx
2100 Logic Drive
San Jose, CA 95124
daniel.han@xilinx.com

Walter Sze
Xilinx
2100 Logic Drive
San Jose, CA 95124
walter.sze@xilinx.com

Benjamin Ting
Xilinx
2100 Logic Drive
San Jose, CA 95124

Darrow Chu
Cadence
2655 Seely Ave.
San Jose, CA 95134
darrow@cadence.com

ABSTRACT
In this paper we will show how we developed a formal verification "app" – i.e. a completely automated flow that even engineers without formal knowledge can run – to tackle the unique requirements of "3D IC" design. Specifically, we will describe how to maintain the design hierarchy required by backend physical tools, yet automatically create "bundled" assertions to rapidly and exhaustively verify 10's of thousands of connections between logic slices – something that proved impossible with our prior simulation-based flow. With this app we were able to find 13 bugs in a matter of hours, in a design that was thought to be "correct by construction".

Keywords:
Assertions, SVA, ABV, Formal Verification, Apps, 3D IC

1. INTRODUCTION
"3D IC" design – where multiple die are mounted on an interposer substrate, and then packaged as a single unit – enables our company to rapidly introduce high capacity and high performance devices with high yield and a minimum of overhead. However, this design style introduces a whole new verification challenge vs. conventional ASICs; namely, the need to verify a massive number of connections between the different die and the corresponding signal paths that transit in and out of the interposer. Initially we applied our existing, simulation-based flow that we have used for all our prior parts to these new 3D devices. However, it was quickly apparent that this flow would only be effective for a few representative cases and/or not be able to exhaustively cover all connections as required given the drastic increase in the scale of the requisite testbenches and the subsequent run time. (It would a minimum of a man-month before simulations would begin to produce meaningful results.)

Note that we do strive to make the design "correct by construction", and automate the IP assembly process to avoid connectivity errors. However, the cost of even one conceptually trivial connectivity bug from a swapped wire – in both re-spin cost and (much more significantly) impact on our market positioning -- is unacceptable, so thorough verification is a must. To give an idea of the problem scope: even small parts comprised of identical die can have over 10,000 connections between the dice and interposer – for heterogeneous parts that intermix FPGA slices with various IPs (processors, transceivers, etc.) the number of connections can be significantly larger.

An added, complicating factor is that the demand of 3D physical design requires anywhere from 4 to 8 layers of design hierarchy. Of course, from a digital verification point-of-view all this extra hierarchy is redundant an unnecessary, yet we must preserve it to ensure the fidelity of our verification as well as preserve all the hooks needed for the downstream tool chain.

Finally, we are constantly creating new configurations and/or parts to be responsive to market and customer needs. Hence, to prevent having to reinvent a new flow for each new product, our connectivity verification flow has to be

very flexible, parameterizible, “future proof”, and as automated as possible.

2. OVERVIEW OF FORMAL

The primary role of verification is to provide a means to check that a design under test (DUT) matches its specification. Assertions are a way to capture design intent, and in affect they become an executable specification that forms the basis for evaluation of the DUT’s construction and behaviors.

While assertions in a simulation flow act as watch dogs on the lookout for correct behavior, they can only execute their checks on the limited state space that is traversed by a given simulation. Even the most well written, loosely constrained testbenches can only traverse a fraction of the total number of states that are mathematically possible.

In contrast, formal analysis tools use assertions as the basis of a mathematical proof that shows that for all possible inputs the DUT will behave as specified by the assertion. If the DUT can be shown to behave contrary to the assertion, the tool displays a counter example showing the user exactly how the circuit can misbehave. Of course, the exhaustive nature of this approach combined with the ability to automatically pinpoint the given error is perfect for our connectivity verification application (where it bears repeating that even one misplaced connection would be painfully expensive to rework later).

Finally, one additional note specific to connectivity checking is that the assertions required for this connectivity checking do not need to capture temporal behavior i.e., behavior spanning multiple clock cycles, a/k/a link with concurrent assertions. As we will discuss below, this enabled us to “bundle” assertions to improve the throughput of the app without losing fidelity to the DUT netlist.

3. CREATING THE APP

3.1 Requirements

Below are our requirements we defined for this app:

* The app must perform an exhaustive analysis

* The app must accept a Verilog netlist which is itself derived from a schematic.

* The app must retain and respect the hierarchical signal information to arbitrarily depths

* Since few engineers on our team have any experience with formal or ABV, the operation of the formal tool had to be completely automated and/or essentially hidden from the app user

* Creation of the connectivity assertions also had to be completely automated

* Capturing the connectivity spec. had to be straightforward, and be human and machine readable in an open, non-proprietary format

* Running the app should be like executing any common command line utility

* The app’s output should be easy to interpret -- leading the user directly to the specific connection(s) that are in error.

* The run time of the whole app and any supporting scripts shall be at least 100x less than the equivalent testbench simulations preparation and run time.

3.2 Defining Connectivity

We used the standard connectivity definition template MS Excel file as a starting point.

Assertion	Source path	Signal	Destination path	Signal
check_1	slave_beg.slice	LVB_K2[7]	master.slice	LVB_K10[6]
check_2	slave_beg.slice	LVB_K2[8]	master.slice	LVB_K10[7]
check_3	slave_beg.slice	LVB_K2[9]	master.slice	LVB_K10[8]
check_4	slave_beg.slice	LVB_K2[10]	master.slice	LVB_K10[9]
check_5	slave_beg.slice	LVB_K2[11]	master.slice	LVB_K10[10]
check_6	slave_beg.slice	LVB_K2[12]	master.slice	LVB_K10[11]
check_7	slave_beg.slice	LVB_K2[13]	master.slice	LVB_K10[12]
check_8	slave_beg.slice	LVB_K2[14]	master.slice	LVB_K10[13]
check_9	slave_beg.slice	LVB_K2[15]	master.slice	LVB_K10[14]

The spec file includes the source and the destination of each connection. It also includes information regarding repeating patterns of each connection and how many times it repeats.

3.3 Black Boxing Support

The tool vendor also provides a utility called "bbgen" that takes as input a Verilog netlist and strips out any interior logic, only leaving the pinouts of the given block behind. In our application, where we are dealing with a relative handful of logic blocks (compared to an SoC with 100's of IPs, which might require some human intervention to manually cleave some entities), we are able to completely embed the execution of this utility inside our app so it's invocation is transparent to the end user.

3.4 Assertion Creation

As introduced above, formal engines (in this case, Cadence's Incisive Formal Verifier) are employed under-the-hood of the app to analyze the design in a meaningful and sufficient way. Therefore it's essential to create a complete set of assertions derived from the connectivity spec for the formal engines to prove.

Our solution was to completely automate the assertion generation process from the device spec with a Perl script. In a nutshell, the script reads the spec file describing souce/destination path and signal names as well how many times each connections are repeated. The script generates both cvs files to be imported as Excel spreadsheet and assertions files.

Here is an example assertions:

```
CHECK_LVL_B2_10_to_LVL_B10_9_m0_to_s2: assert property (dut.m0.slice.LVL_B2[10] == dut.s2.slice.LVL_B10[9]);
CHECK_LVL_B2_11_to_LVL_B10_10_m0_to_s2: assert property (dut.m0.slice.LVL_B2[11] == dut.s2.slice.LVL_B10[10]);
```

Here is an example of bundled assertions:

```
CHECK_LVL_B2_s1_to_m0: assert property ((dut.s1.slice.LVL_B2[10] == dut.m0.slice.LVL_B10[9]) &&
(dut.s1.slice.LVL_B2[11]==dut.m0.slice.LVL_B10[10]) &&
(dut.s1.slice.LVL_B2[12]==dut.m0.slice.LVL_B10[11]) &&
(dut.s1.slice.LVL_B2[13]==dut.m0.slice.LVL_B10[12]) &&
(dut.s1.slice.LVL_B2[14]==dut.m0.slice.LVL_B10[13]) &&
(dut.s1.slice.LVL_B2[15]==dut.m0.slice.LVL_B10[14]) &&
(dut.s1.slice.LVL_B2[16]==dut.m0.slice.LVL_B10[15]) &&
(dut.s1.slice.LVL_B2[17]==dut.m0.slice.LVL_B10[16]) &&
(dut.s1.slice.LVL_B2[18]==dut.m0.slice.LVL_B10[17]) &&
(dut.s1.slice.LVL_B2[19]==dut.m0.slice.LVL_B10[18]) &&
(dut.s1.slice.LVL_B2[20]==dut.m0.slice.LVL_B10[19]) &&
(dut.s1.slice.LVL_B2[21]==dut.m0.slice.LVL_B10[20]) &&
(dut.s1.slice.LVL_B2[22]==dut.m0.slice.LVL_B10[21]) &&
(dut.s1.slice.LVL_B2[23]==dut.m0.slice.LVL_B10[22]) &&
(dut.s1.slice.LVL_B2[24]==dut.m0.slice.LVL_B10[23]) &&
(dut.s1.slice.LVL_B2[25]==dut.m0.slice.LVL_B10[24]) &&
(dut.s1.slice.LVL_B2[26]==dut.m0.slice.LVL_B10[25]) &&
(dut.s1.slice.LVL_B2[27]==dut.m0.slice.LVL_B10[26]) &&
(dut.s1.slice.LVL_B2[28]==dut.m0.slice.LVL_B10[27]) &&
(dut.s1.slice.LVL_B2[29]==dut.m0.slice.LVL_B10[28]) &&
(dut.s1.slice.LVL_B2[30]==dut.m0.slice.LVL_B10[29]) &&
(dut.s1.slice.LVL_B2[7]==dut.m0.slice.LVL_B10[6]) &&
(dut.s1.slice.LVL_B2[8]==dut.m0.slice.LVL_B10[7]) &&
(dut.s1.slice.LVL_B2[9]==dut.m0.slice.LVL_B10[8]));
```

In addition to automating the "front end" of the process, the script also invokes the formal tool and does some post processing of the formal analysis such that the whole flow is completely automated. This automation allows us to use the "app" nickname to refer to this flow, and thus our colleagues – almost all of who have no experience with formal tools and SVA – aren't afraid to run it even though it uses technologies under-the-hood they don't normally use.

3.5 App Execution

Once the connectivity description spec is available, the user can run the app from the Linux command line. Invoking the app is

```
% bali_sll_conn.pl \
-top <top> -slice <slice>
-input <connection spec> \
-output <results file> \
-logfile <logfile name>
```

Upon execution the app reports intermediate results and any warning or error messages to the console. When the app is finished running, if any misconnections were found the user is directed to the counter example file where the errors are flagged.

3.6 Results Reporting

Here is an example output:

```
FormalVerifier> prove
Verification mode:
  CHECK_m0_T_NC_NONMIB_2_ : Pass
  CHECK_m0_T_B2M_0_TXDATA_90 : Pass
  CHECK_m0_LVR_BK2_8 : Pass
  CHECK_m0_LVL_X4_17 : Pass
  CHECK_m0_T_B2M_0_TXDATA_91 : Pass
  CHECK_m0_LVR_BK2_7 : Pass
```

```
  CHECK_m0_LVL_X4_18 : Pass
  CHECK_m0_T_B2M_0_TXDATA_92 : Pass
  CHECK_m0_LVR_BO4_29 : Pass
  CHECK_m0_LVL_X4_19 : Pass
  CHECK_m0_LVL_FM2_30 : Pass
  CHECK_m0_T_B2M_0_TXDATA_93 : Pass
  CHECK_m0_LVR_BO4_28 : Pass
  CHECK_m0_T_B2M_0_TXDATA_94 : Pass
  CHECK_m0_LVR_BO4_27 : Pass
  CHECK_m0_T_B2M_4_TXDATA_120 : Pass
  CHECK_m0_T_B2M_1_GTZTXRESET_ : Pass
  CHECK_m0_LVL_HF2_9 : Pass
FormalVerifier> assert -summary -time
Assertion Summary:
  Total            : 10464
  Pass             : 10464
  Not_Run          :   0
   Verification Mode  : CPU time = 4370.71s Real time = 5038.06s
```

Figure 1: Example App Output

In this case, the app ran through 10464 connections between 2 die through an interposer using the generated assertions.

3.7 Complete Work Flow

The complete work flow is diagrammed in Figure 2 below:

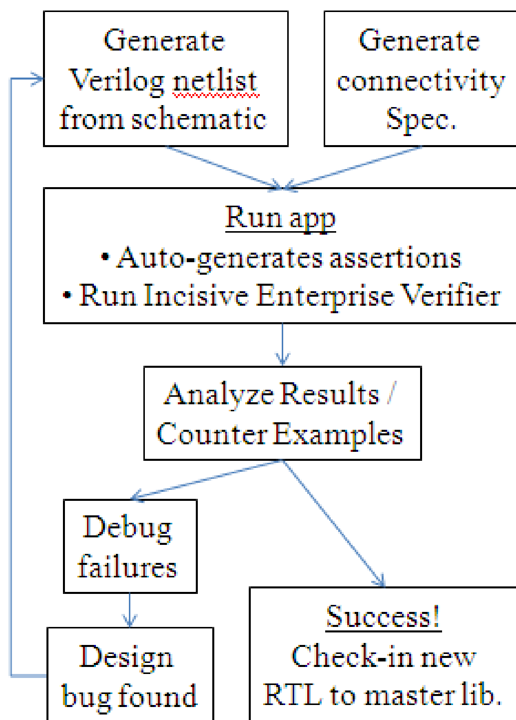

Figure 2: Flow chart of app usage

4. THE BASELINE PROJECT: VERIFYING CONNECTIVITY OF A HOMOGENOUS ASSEMBLY

4.1 DUT Description

The first project we engaged with this new app was a homogenous assembly of FPGA die on an interposer. Figure 3 below:

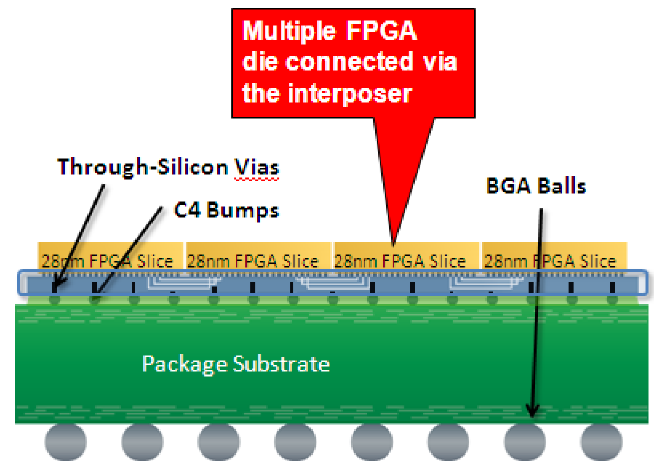

Figure 3: Side View of a Homogenous Part

The relative simplicity of the overall configuration allowed us to develop the broad outlines of the flow described above.

4.2 Applying the App – and Some Initial Surprises

The initial challenge was IEV to work with switch level Verilog netlist extracted from schematic. The Verilog netlist contained numerous inout ports and IEV did not handle inout ports properly.

With the IEV patched, first few assertions went through and found both spec and design bugs. However, it became clear that run time to prove each assertion was a problem. For example, one of the smaller dut with 24,863 assertions, we saw 37s to prove an assertion. Although this is order of magnitude better than simulation, we needed to find a better way.

4.3 Lesson Learned: "Bundling" Assertions

Given this "surprise", it became clear that the biggest challenge was to reduce the number of assertions. Since the each connection is point to point connection even though it travels through many layers of hierarchy, each assertions are independent. By combining assertions in reasonable chunk, we can reduce the total number of assertion statements in the app.

By combining multiple assertions, we lose ability to immediately identify the failing net. However, we can expand the failing bundled assertion in order to debug further.

The benefits of this bundling were immediate: as per the example cited above, we reduced the number of assertions to 91 from 24,864. The prove time per assertion increased for this bundled assertion by 14% to 42s per assertion. However, the overall run time significantly from 24 hours to 1 hour.

4.4 Tool Improvements

We saw significant runtime improvement by moving to the latest version of IEV from 9.20 to 10.20. On the smaller part, prove time per assertion improved from 42s per assertion to 0.77s per assertion, over 50x improvement. On the larger part, prove time per assertion improved from 854s per assertion to 0.59s per assertion, resulting significant reduction in verification time.

5. USING THE APP TO VERIFY HETERGENOUS PARTS

5.1 DUT Description

In contrast to the homogenous assembly, a heterogeneous part hosts a mix of FPGA slices and IP blocks on the interposer. Consider this example:

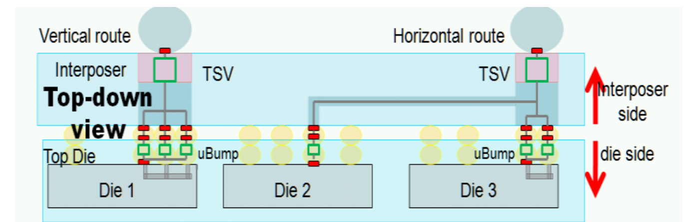

Figure 4: Side View of a Hetergenous Part

Clearly, the mix of upper level entities immediately makes for a more complex connectivity picture in general. As you may imagine, the composition of elements can be arbitrary, and be deliberately different to support different product lines and derivatives.

5.2 Applying the App These More Complex Parts

To our delight, the app and related work flow described above proved to be robust enough to handle these more complex parts. Since there are many unique connections made, the spec file listed these unique connections and it had to be reviewed carefully.

6. CONCLUSION

Compared to earlier simulation based techniques, utilizing formal techniques within the familiar "app" framework has enabled us to exhaustively verify the internal connectivity checking of extraordinarily complex 3D IC designs – something that proved impossible with our prior simulation-based flow. Specifically, in one of our pilot projects this app we was able to find 13 bugs in a matter of hours, in a design that was thought to be correct.

Since then we have taped-out multiple products successfully with this flow, and the silicon validation results have been perfect. This app has been a valuable contribution to the overall team's throughput, and a major cost and risk reduction for all of our products.

7. ACKNOWLEDGMENTS

Thanks to thanks to Joseph Hupcey III of Cadence Design Systems, Inc. for his feedback in reviewing the paper.

8. REFERENCES

[1] Cadence Incisive Enterprise Verifier User Guide.

[2] Archived Webinar, “Formal Apps to Automate Mainstream Verification Challenges”, http://www.cadence.com/cadence/events/Pages/event.aspx?eventid=680