2021 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE OCTOBER 26-27, 2021

Netlist Paths

A tool for front-end netlist analysis

Jamie Hanlon, Graphcore Ltd, Bristol, United Kingdom (jamie@graphcore.ai)

Samuel Kong, Graphcore Ltd, Bristol United Kingdom (samuelk@graphcore.ai)

***Abstract*—Netlist Paths is an open-source library and tool for inspecting RTL designs in the form of a source-level netlist model, to provide rapid feedback to designers. The library provides a Python interface that aims to address the lack of lightweight EDA tools by making it easy to create custom tooling. In this paper we discuss the motivation and concept behind Netlist Paths, compare it to alternative approaches, and present three real-world applications of Netlist Paths we have found at Graphcore that have improved designer productivity.**

Keywords—Python; SystemVerilog; Netlist; RTL; Front-End design; Design automation

I. Introduction

There is an abundance of EDA tools available to RTL design engineers that assist in the development of ASIC designs, including for example, simulation, linting, power analysis and synthesis. Typically, these EDA tools read in designs in the form of a hardware description language (HDL) and elaborate them into internal, proprietary netlist models, representing the components and connectivity of the design. The internal representation is often made accessible to users for the purposes of the tool via a TCL Application Programming Interface (API) and/or graphical user interface (GUI). There is, however, an absence of a simple, generic tool that provides RTL design engineers programmatic access to a source-level netlist model, allowing them to inspect types, constants, variables and structure of a design.

In this paper, we shall discuss Netlist Paths, an open-source library and tool that enables the inspection of a netlist model of an RTL design using a Python API or command-line interface (CLI). We shall explore the benefits of using Netlist Paths and the applications it has in the chip design cycle, based on our experience of ASIC chip design at Graphcore. Applications include investigation and mitigation of critical timing paths, quality-of-result (QoR) analysis and unit tests for checking the connectivity of RTL. First, we shall review existing, related works and how Netlist Paths builds upon, complements, or differs from them. We will then provide an overview of how the core library works, and then describe three different ways in which Netlist Paths has been deployed at Graphcore to great effect.

II. Related Work

To analyse an RTL design, an engineer's first choice of tool will likely be one of the standard EDA simulators such as Synopsys VCS, Mentor Questa or Cadence Xcelium. These simulators provide powerful facilities to inspect the design for debugging, conventionally, with a GUI which does not easily allow integration into custom tooling to perform a repetitive or specialized task. Beyond these simulation-based tools, an engineer may turn to synthesis tools such as Synopsys Fusion Compiler or Cadence Genus to analyse a design. But taking this approach incurs longer run times and machine resources since synthesis of a large design is a complex task, as well as requiring often scarce licenses.

Addressing the issue of custom tooling, Synopsys provide a Native Programming Interface (NPI) to their Verdi platform, which provides TCL and C APIs to access the database produced during simulation [1]. NPI has interfaces to different aspects of the database, but in particular, a netlist model, which is the result of technology-independent, non-optimized synthesis. With this model, the structure of the design is accessible as low-level objects such as gates and memories connected via ports and nets. However, the NPI Netlist Model for source-level tooling does not maintain correspondence to the original RTL source code, and synthesis of RTL into combinatorial logic cells further complicates that correspondence.

There are several notable open-source tools that provide facilities for parsing and elaborating SystemVerilog: Verible (a project from Google) [2], Slang (SystemVerilog language services, taking inspiration from Clang) [3] and Surelog [4][5] are all libraries intended to be used as a front end for EDA tooling. Currently, their respective projects only include simple tools, e.g. for linting. Perhaps the best-known open-source tools are Verilator [6] and Yosys [7][8]. Verilator is a mature Verilog simulator, that is becoming increasingly widely used, particularly in the burgeoning open-source hardware community. Yosys is a Verilog synthesis framework geared towards FPGA development, but only supports Verilog 2005[1]. Netlists Paths builds upon Verilator, chosen primarily because of its maturity as a simulation platform and known compatibility with Graphcore's SystemVerilog codebase, and uses it to parse and elaborate the design. However, the alternative open-source language front ends listed may have equally been used by Netlist Paths, or could be in the future.

III. NETLIST PATHS

Netlist Paths is a library written in C++ with two front ends: a Python API for integration into new tooling and a Python command-line tool that exposes all the main features of the library. It is open source and available on GitHub [9][10]. Having core functionality implemented in C++ is important for performance and the usability of the library because real-world designs are complex, so their representation in XML or as a graph can be large. For example, the RTL for Nvidia's open-source Deep Learning Accelerator (NVDLA) [11] produces a 2.3 GB Extensible Markup Language (XML) netlist file. Providing a high-level Python interface makes it simple to use the library and/or to integrate it with existing Python code, particularly since Python is becoming an increasingly common part of the infrastructure for ASIC design and verification flows. This split between C++ and Python therefore provides a good trade-off between performance and ease of use/integration.

Netlist Paths works by reading an XML representation of a Verilog design produced by Verilator, which is a single flattened module with all tasks, functions, and sub module instances inlined (Verilator is run using the `--xml-only` and `--flatten` options). The XML is traversed to build up a directed graph data structure representing the design netlist, where nodes of the graph are logic statements or variables, and edges are data dependencies between them. The data-type information of the design is also read from the XML, and a corresponding data-type table is constructed, that each variable references. To see how the graph corresponds to the design, consider a module that implements an adder:

```
module adder
  #(parameter p_width = 32)(
    input  logic [p_width-1:0] i_a,
    input  logic [p_width-1:0] i_b,
    output logic [p_width-1:0] o_sum,
    output logic               o_co
  );
  assign {o_co, o_sum} = i_a + i_b;
endmodule
```

This design produces the graph in Figure 1 representing its netlist.

[1] Yosys also supports some SystemVerilog features. A paid-for licensed version of Yosys provides a different front end that supports SystemVerilog 2012, see https://www.yosyshq.com/tabby-cad-datasheet.

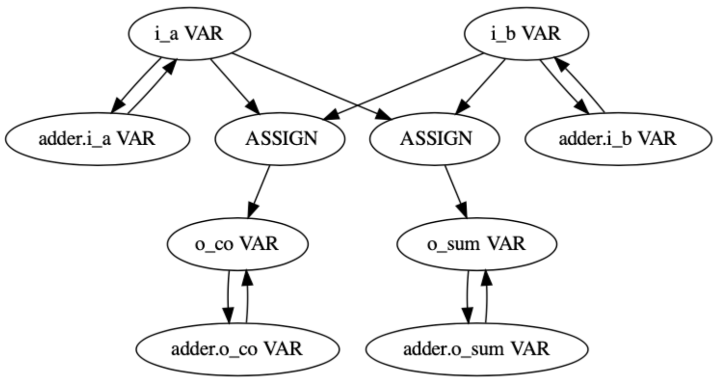

Figure 1: Netlist graph for an adder module

Nodes in this graph are labelled with their type (in uppercase) and if they are variables, their name. You can see that the outputs `adder.o_sum` and `adder.o_co` are dependent on the two input variables `adder.i_a` and `adder.i_b`, as is specified in the source code. A path between these points can be queried using the command-line tool, showing that `o_sum` is dependent on `i_a` via an assignment statement on line 11, along with the data-type information of the variables showing they are 32-bit logic vectors:

```
$ netlist-paths adder.sv --compile --from i_a --to o_sum
Name  Type DType        Statement Location
----- ---- ------------ --------- -----------
i_a   VAR  [31:0] logic ASSIGN    adder.sv:11
o_sum VAR  [31:0] logic
```

To support querying of paths that start and/or end on registers, as is the case with physical timing reports, registers are identified during parsing of the XML so they can be split into two components: a source and a destination, with out-edges and in-edges of the original node respectively. Register variables are identified when they appear on the left-hand side of a non-blocking assignment[2]. To see how this affects the construction of the graph, consider the following module that provides a bank of registers:

```
module pipestage #(parameter p_width=32) (
    input  logic                 i_clk,
    input  logic                 i_rst,
    input  logic [p_width-1:0] in,
    output logic [p_width-1:0] out
  );
  logic [p_width-1:0] data_q;
  always_ff @(posedge i_clk or posedge i_rst)
    if (i_rst)
      data_q <= '0;
    else
      data_q <= in;
  assign out = data_q;
endmodule
```

The netlist graph in Figure 2 has two distinct components: the fan in-cone to `data_q` including the sensitivity list of the always block and the input port, and `data_q` driving the output port.

[2] Note that this condition is simplistic and will be extended to check the always block is sensitive to a rising or falling edge of a clock. It is however sufficient to identify registers in most RTL code, which generally does not mix blocking and non-blocking assignments.

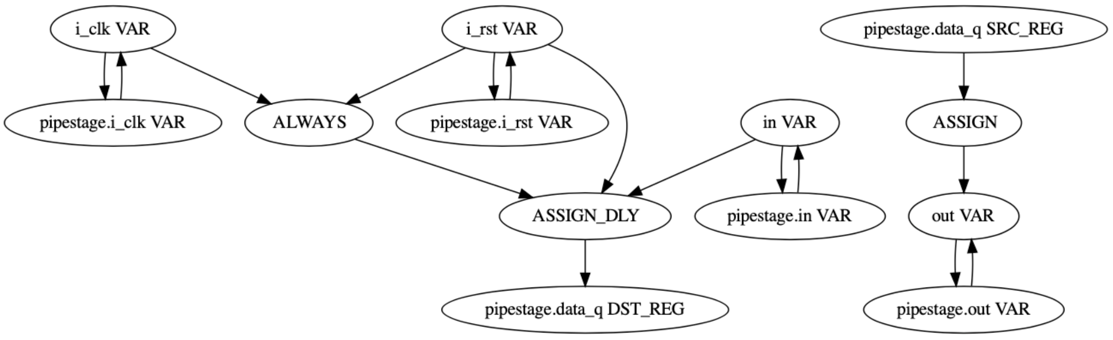

Figure 2: Netlist graph for a module providing a bank of registers.

The corresponding paths can be queried as:

```
$ netlist-paths pipestage.sv --compile --from in --to pipestage.data_q
Name             Type DType        Statement Location
---------------- ---- ------------ --------- ---------------
in               VAR  [31:0] logic ASSIGN    pipestage.sv:10
pipestage.data_q REG  [31:0] logic
$ netlist-paths pipestage.sv --compile --from pipestage.data_q --to out
Name             Type DType        Statement Location
---------------- ---- ------------ --------- ---------------
pipestage.data_q REG  [31:0] logic ASSIGN    pipestage.sv:13
out              VAR  [31:0] logic
```

There are several types of path queries supported by Netlist Paths: between specific start and end points (point-to-point), where any or all matching path will be returned, from a particular start point to return all the fan-out paths, and from a particular end point to return all the fan-in paths. Paths in the netlist graph are determined using a depth-first traversal from a specific start point to produce a tree sub-graph with the leaves being all the nodes that are reachable. A point-to-point path query will check if the end point exists in the leaf nodes, and if so, trace a path back to the source node. A fan-out query will return a path between each leaf node and the source node. A fan-in query reverses all edges in the graph and uses the same algorithm for fan-out paths. Since in general it is not practical to enumerate all paths matching a start and end point[3], path queries can be refined by specifying 'through' variables that must be visited, or 'avoid' variables that must not be. Various options are also provided to facilitate matching of start and end points, such as using wildcard expressions or regular expressions, and to select any variable matching a pattern, rather than requiring an exact match. These options are useful in establishing correspondences with the output of other tools as we will see in the next section.

It should be noted at this point that Netlist Paths does not currently perform elaboration of the design at the bit-level, meaning that dependencies are between named variables, rather than components of variables, such as subscripts, slices, or fields. As such, the dependencies it does infer are at a coarse level of granularity, but which are sufficient for the applications it is applied to within Graphcore. Full bit-level elaboration will substantially impact the size of the netlist graph, and the corresponding runtimes, but it is something we may investigate supporting in the future.

Applications and Experience

A. *Critical path investigation*

The original motivation for developing Netlist Paths was to tackle the problem of corresponding critical paths identified during timing analysis in the Physical Design flow back to the design's RTL. This is a problem for Front-End design for two reasons: firstly, because the time between committing an RTL change to getting feedback from timing analysis is around 12 to 24 hours for particular Graphcore blocks that have many timing-critical data and

[3] This is because the number of paths between a start and an end point grows exponentially with respect to the number of nodes.

control paths; and secondly because timing-critical paths become heavily optimised and often only retain their start and end point names, it is very difficult to work out how these correspond to the RTL. This can be seen in the following (simplified) timing report taken from our tile processor, where only the start and end registers have source names:

```
Point                                     Ref name          Net
-----                                     --------          ---
cts_inv_15095706023/CK->X                 CKINV_Y2          ctsbuf_net_909743178
ctobgt_inst_716854/CK->X                  CKINV             ctobgt_76
cts_inv_7287698215/CK->X                  CKINV             ctsbuf_net_125742394
cts_inv_7283698211/CK->X                  CKINV_CB          ctsbuf_net_124742393
u_idecode_ibuf_rd_sdata_q_reg_9_/CK->Q2   FSDPRBQM4SS_V2Y   u_idecode_ibuf_rd_sdata_q_28_
HFSBUF_231_469368/A->X                    BUF_SM            HFSNET_21922
u_idecode_ctmi_18981/A1->X                ND2_CB            u_idecode_ctmn_16434
ctmi_293593/A->X                          ND2B_V1           ctmn_462686
ctmi_164819/A1->X                         NR2               ctmn_462752
ctmTdsLR_1_510627/B2->X                   NR3B_Y2           N625606
A434661/B2->X                             AN3B              ctmn_462803
ctmTdsLR_2_721789/A1->X                   NR3_Y2            copt_net_743354
...                                       ...               ...
ctmTdsLR_1_721788/A2->X                   AOI21             N624302
ctmTdsLR_2_645468/A2->X                   ND2               dec_mx_instr_instrn_3_
ctmTdsLR_2_721674/A1->X                   NR2               copt_net_743315
ctmTdsLR_1_721660/A->X                    AN3B              ZBUF_209_47
ctmTdsLR_2_729835/A->X                    OR3B_CB           copt_net_746628
ctmTdsLR_1_729834/B->X                    AO21B             u_lsu_i0_op2_mux_2_
ctmi_306787/A1->X                         AN2               ctmn_484205
ctmTdsLR_3_722448/A2->X                   OAI21_V1Y2        copt_net_743586
ctmTdsLR_3_635081/A->X                    INV               tmp_net720478
ctmTdsLR_4_635082/A1->X                   ND4               tmp_net720480
ctmTdsLR_1_722190/C->X                    AOAI211           u_lsu_i0_op0_mux_3_
u_lsu_e0_addr0_op1_op2_q_reg_16_/D        FSDPRBQM4SS_V2Y
```

Having the ability to rapidly explore paths between the matching start and end points bridges the information gap created by synthesis, making it easy to locate exactly how a critical path traverses through design RTL. Taking the above timing report as an example, the flattened names can be used directly with the Netlist Paths command-line tool using the option to ignore hierarchy markers (underscores, slashes etc) and to match with regular expressions. The path in the report can be found by specifying the start and end points and further constraining the path to match the known intermediate nets marked in bold (with the order of `--through` arguments being enforced), with the following summarised output:

```
$ netlist-paths xm_tilecpu.netlist.xml --ignore-hierarchy-markers --regex --from
u_idecode_ibuf_rd_sdata_q --to u_lsu_e0_op0_q --through  dec_mx_instr$ --through
i0_op2_mux
Name                                   Type DType         Statement Location
-------------------------------------- ---- ------------- --------- ------------------
xm_tilecpu.u_idecode.ibuf_rd_sdata_q   REG  [69:0] logic  ASSIGN    xm_idecode.sv:709
...                                    ...  ...    ...    ...       ...
xm_tilecpu.dec_mx_instr                VAR  packed struct ASSIGN    xm_tilecpu.sv:1045
...                                    ...  ...    ...    ...       ...
xm_tilecpu.u_mrf_rf.r1_data            VAR  [32:0] logic  ASSIGN    xm_mrf_rf.sv:1593
...                                    ...  ...    ...    ...       ...
xm_tilecpu.dec_instrn_stall            VAR  logic         ASSIGN    xm_idecode.sv:1290
...                                    ...  ...    ...    ...       ...
xm_tilecpu.u_lsu.i0_op0_mux            VAR  [31:0] logic  ASSIGN    xm_lsu.sv:602
xm_tilecpu.u_lsu.e0_op0_q              REG  [31:0] logic
```

This report gives a clear view of how the path traverses the RTL in the design, providing the designer with all the information they need to start investigating how to tackle the problem. Once a critical path has been located in this way, a second valuable use of Netlist Paths is to allow the designer to swiftly validate potential timing fixes by rerunning Verilator, then the `netlist-paths` query to see if the path still exists. This contrasts with the standard way of providing a trial patch to the Physical Design team and for them to re-running the full physical build and provide TA reports.

B. Quality-of-result groups

A related issue with the interaction between the Logical and Physical Design teams is in the specification of parts of a design for QoR reporting. Conventionally, ad-hoc patterns are maintained in Physical Design scripts to typically identify groups of registers or timing paths. These groups are used to present metrics that indicate the quality of the build, such as total- and worst-negative timing slack, logical depth in gates, cell area, total power or cell types. This kind of QoR reporting provide a way to compare different builds and understand the effects of design or flow changes. As a design develops, patterns defined in this way inevitably become out of step with the design and cause the reporting to break, often in a non-obvious way. A better system is to maintain QoR group patterns alongside the design, so that they can be maintained with it. For example, the QoR patterns for a results pipeline module can be succinctly specified in Python:

```
# Define registers belonging to different pipeline stages.
results_regs = Collection('Results pipeline', [
  Entity('p0', []),
  Entity('p1', [
   "u_results/cpipe_uop_q",
   "u_results/cpipe_valid_q”]),
  Entity('p2', [
   "u_results/cpipe_uop_q",
   "u_results/cpipe_valid_q"]),
  ... ])

# Define paths between the different pipeline stages.
results_path_groups = Collection('Results paths', [
  Path('p1 fanin', ['.*'],                     results_regs.patterns('p1')),
  Path('p1',       results_regs.patterns('p1'), results_regs.patterns('p2')),
  ... ])
```

Which can then be exported for use with Physical Design tooling as TCL data:

```
...
set design_info(registers,results_pipe,p0) [list]
set design_info(registers,results_pipe,p1) [list \
  "*u_results?cpipe_uop_q*" \
  "*u_results?cpipe_valid_q*"]
set design_info(registers,results_pipeline,p2) [list \
  "*u_results?cpipe_uop_q*" \
  "*u_results?cpipe_valid_q*"]
...
set design_info(path_groups,results,p1_fanin)\
    {{".*"}\
     {"u_results/cpipe_uop_q" "u_results/cpipe_valid_q"}} \
set design_info(path_groups,results,p1)\
    {{"u_results/cpipe_uop_q" "u_results/cpipe_valid_q"} \
     {"u_results/cpipe_uop_q" "u_results/cpipe_valid_q"}} \
set design_info(path_groups,results,p2)\
    {{"u_results/cpipe_uop_q" "u_results/cpipe_valid_q"} \
     {"u_results/cpipe_uop_q" "u_results/cpipe_valid_q"}} \
...
```

Netlist Paths provides a simple way to enforce this consistency by using the Python API in a script that reads the Python QoR groups and the design’s netlist, then checks that each of the specified registers and timing paths exists. The design is additionally checked for registers that are not covered by the patterns, to ensure any new registers are covered by the groups. Creation and maintenance of these groups is aided by the interactive use of the Netlist Paths command-line tool to produce lists of variables contained in the design.

C. *Structural unit testing*

As chips are becoming more and more complex, there becomes an increasing reliance on design automation. A common automation practice within the semiconductor industry is to auto-generate top-level integration RTL code. Such auto-generated code could be hundreds-of-thousands of lines in length and attempting to visually verify that the connections are correct can prove tedious and error prone. During the early stages of development, verification testbenches may not exist or are immature to catch these types of functional integration bugs. When the test benches do exist, running top-level SoC simulations can be very expensive in terms of time and compute resources, and as such it can be time consuming to track down functional issues. Problems with design for test (DFT) signals may not be caught until a later stage still of the design process. Netlist Paths provides an opportunity for RTL designers to create simple unit tests in Python to ensure that the connectivity of the auto-generated code is correct. Figure 3 shows an example schematic diagram of an auto-generated top-level with an arbitrary number of SoC blocks that are interconnected together.

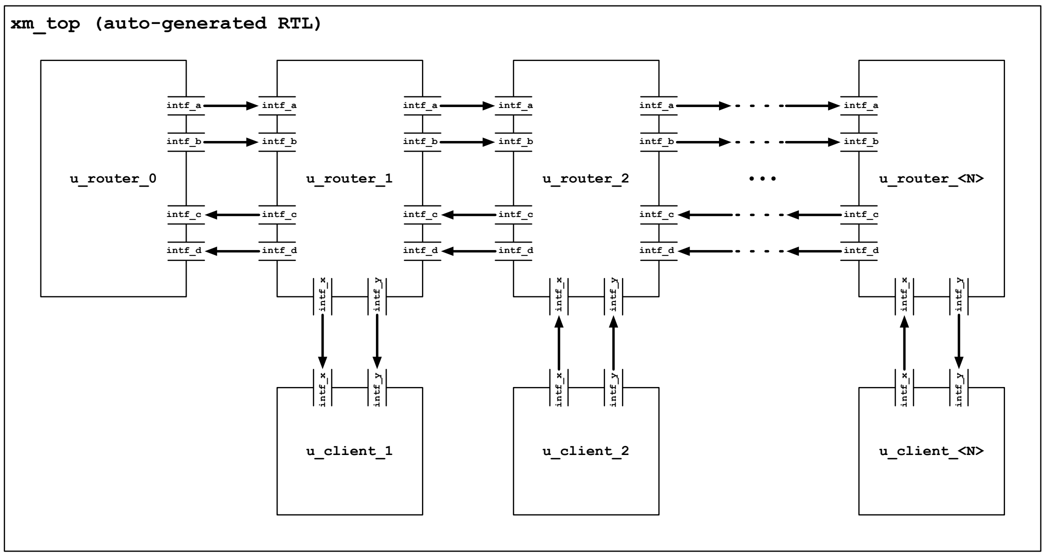

Figure 3. Schematic diagram of an auto-generated top-level.

The following code snippet demonstrates how the Netlist Paths API can be leveraged to implement simple structural connectivity checks. The example checks that each of the router components shown in Figure 3 are connected in a chain.

```python
import py_netlist_paths

# Enforce signal names must match exactly.
py_netlist_paths.Options.get_instance().set_match_exact()
# Allow start and end points to be at nets rather than registers.
py_netlist_paths.Options.get_instance().set_restrict_start_points(False)
py_netlist_paths.Options.get_instance().set_restrict_end_points(False)
# Load the netlist XML.
netlist = py_netlist_paths.Netlist('xm_top.netlist.xml')

for i in range(N - 1):
    # Check the router is connected to the next router along.
    for signal in intf_a + intf_b + intf_c + intf_d:
        # Construct hierarchical paths to the signals.
        point_a = f'xm_top.u_router_{i}.{signal}'
        point_b = f'xm_top.u_router_{i+1}.{signal}'
        # Create a waypoint object for the start and end points.
        waypoint = py_netlist_paths.Waypoints(point_a, point_b)
        # Check that the two points are connected together as expected.
        assert netlist.path_exists(waypoint)
```

Structural connectivity checking is not a novel concept and there are alternative methods. One method is to use an EDA tool that provides access to an elaborated netlist, such as Synopsys Fusion Compiler. An RTL designer could make use of Fusion Compiler's TCL API to carry out connectivity checks. However, as shown in Table 1, Netlist Paths can complete such structural connectivity checks in a fraction of the time taken by the heavyweight synthesis tool.

Table 1. Time taken to run connectivity checks compared with Fusion Compiler

Complexity	**Run Time (seconds)**	
Number of router instances	**Fusion Compiler**	**Netlist Paths**
4	252	20
8	295	26
16	345	50
32	463	77
64	721	195

Because of Netlist Paths' lightweight approach to producing and analysing a netlist model, it is possible to run these tests as part of our Continuous Integration (CI) test-suite with minimal impact on the overall run time. By making use of the Python API, Netlist Paths has provided versatility and ease-of-use that cannot be found in existing EDA tools and as a result, has improved RTL code quality and engineering productivity.

IV. Conclusions

We have presented Netlist Paths, a lightweight, open-source tool that provides RTL design engineers easy access to a source-level netlist model of their design through a Python API and CLI. The combination of these features enables the straightforward creation of custom tooling for Front-End workflows, where previously the inertia of using licensed EDA tools with C or TCL interfaces prevented this from happening. We have described how three applications of Netlist Paths at Graphcore has increased productivity within the team by improving interactions and providing faster feedback to designers: critical timing paths reported by Physical Design can be identified rapidly in the RTL and fixes for them tested without having to wait for the design to be rebuilt; patterns specifying groups for QoR reporting can be owned by RTL design engineers by having tests to validate them against the design; and structural checks on top-level SoC code to catch bugs early before long-running Verification test benches will pick them up.

References

[1] Synopsys VC Apps: Native Programming Interface (NPI), Version Q-2020.03, March 2020.

[2] Google Verible, https://google.github.io/verible

[3] Slang (SystemVerilog langugage services), https://sv-lang.com

[4] Alain Dargelas, Henner Zeller. Universal Hardware Data Model, Workshop on Open-Source EDA Technology (WOSET) 2020.

[5] Surelog, https://github.com/alainmarcel/Surelog

[6] Verilator, https://www.veripool.org/verilator

[7] Clifford Wolf, Johann Glaser. Yosys - A Free Verilog Synthesis Suite. In *Proceedings of Austrochip 2013.*

[8] Yosys, http://www.clifford.at/yosys

[9] Netlist Paths Github page, https://github.com/jameshanlon/netlist-paths

[10] Netlist Paths documentation, https://jameshanlon.github.io/netlist-paths

[11] NVIDIA Deep Learning Accelerator (NVDLA), http://nvdla.org