Automatic diagram creation for design and testbenches

Paul O'Keeffe, CreVinn Teoranta, Dublin, Ireland (paul_okeeffe@crevinn.com)
Jamie Beattie, CreVinn Teoranta, Dublin, Ireland (jamie_beattie@crevinn.com)
Gian Lorenzo, CreVinn Teoranta, Dublin, Ireland (gian_lorenzo@crevinn.com)

***Abstract*—This paper is to present some findings in an investigation into processes that can be used to automatically create diagrams for Verilog RTL and testbench code.**

Keywords — diagram; generation; XML; VPI; Verilator; graphviz; pydot

I. Related Work

The purpose of this paper is to present a flow for the integration of open source drawing/diagram tools into a design and testbench. The integration of these tools will allow for a number of different types of diagrams to be automatically generated for both specification and training purposes. As these diagrams are linked to the RTL/TB code, they can be re-generated at various points in the development cycle thereby keeping the design and testbench documentation continuously up to date. The novelty of this paper lies in the fact that the diagrams are automatically generated and can track the design and testbench development cycles. Another use case for this flow is for highly configurable designs where hierarchy changes based on configuration.

The types of diagrams that were generated include:

- Design and testbench hierarchical diagrams
- Clock and reset tree diagrams
- Design connectivity diagrams

Design and testbench documentation are critical to maintaining projects for the long term. CréVinn Teoranta have developed a Structured Design Methodology for hardware design which we call the Fletcher Design Methodology. The Fletcher Design Methodology (FDM) is based on methodologies presented by Bill Fletcher and Yourdon-DeMarco. The Fletcher Design Methodology is particularly effective when implemented in the initial design phase and leads to good design practices. The methodology generates a well partitioned design and corresponding design documentation and diagrams. The design diagrams can in turn be used for specifications documents and for reviews. The purpose of this paper is to present a methodology for design and testbench documentation that can be used in conjunction with the FDM for situations where the documented design is being used as IP in a larger System On a Chip (SOC) or to document the associated testbench.

II. Application

Five tools were investigated and used as part of the flow developed as part of this project. One comercial tool (Xcelium) and 4 open-source tools (Verilator, Graphviz, Verilog-Perl, Diagrams.net).

VPI (Cadence Xcelium)

The SystemVerilog Verification Procedural Interface (VPI) provides a library of C language functions and mechanisms for associating foreign language functions with SystemVerilog user-defined system task and system function names. For this project VPI programmes were developed and run on the Cadence Xcelium simulator. These programmes used VPI functions vpi_iterate() to traverse the design hierarchy and generate the diagram files .dot (graphviz) and .csv (diagrams.net) files.

The VPI interface to the Cadence simulator is a powerful way of interacting with a design or a testbench, the VPI routines can be integrated with an existing design flow to run with an existing xrun script.

```
xrun -clean -licq -sv +define+CADENCE_XRUN +define+WAVES=1 -timescale 1ns/1ps -access +rwc -
incdir ../rtl/tnk_corrib_r1127/rtl -F ../filelist/rtl_list.f ../tb/TB_tnk.v ../../../../scripts/VPI/createagram.c -
loadvpi :register_my_systfs -xmlibdirpath $RUN_DIR +define+WAVES=1
+define+WAVES_PATH=directory=$RUN_DIR
```

Next requirement is for the VPI function to be called from inside an initial block in the testbench, as can be seen below arguments can be added to the VPI function call and these are processed using the vpiArgument VPI operator.

```
    initial
      begin
……..
    $cg_display_hierarchy("tnk_design_hierarchy", // filename start
                          "TB_tnk", // top module
                             "tnk", // start module
                             "0", // how many levels
                             "3" // subgraph level
                          );
```

Vpi_scan and vpi_interate functions can be used to traverse the hierarchy [1], [2] and capture all design and testbench information for display.

```
 /* If the module has sub-modules, we have to print the sub-module details and execute walk_mod_hierarchy()
on each */
  if (ModuleI) {
    while ((SubModuleH = vpi_scan(ModuleI))) {
      _num_scope = 0;

      // NOTE: This includes tasks.
      objI = vpi_iterate(vpiInternalScope, SubModuleH);
      if (objI) {
        while (objH = vpi_scan(objI)) {
          vpi_mcd_printf( debug_file, "%s Internal Scope - %s\n",vpi_get_str(vpiName,SubModuleH),
vpi_get_str(vpiName,objH));
          _num_scope++;
        }
      }
```

Verilator

Verilator is an open source simulator tool that also parses a design and produces an XML representation of a design. A vl_hier_graph [6] python script that comes with the Verilator distribution was used as a starting point for a python script that parses a design and generates a graphviz output. An extra python library was integrated into the python script pydot [7] to facilitate the collection and verification of design/xml data.

Pydot library has a number of functions for adding graphviz subgraphs, nodes, edges etc.

```
      graph_dot = pydot.Dot(graph_type='graph')
……..
     graph_dot.add_subgraph(self.graph_clusters_d[_cluster_id])
```

```
……..
    graph_dot.add_node(pydot.Node(_cell_hier,
                                  shape='circle',
                                  color=_hier_colour,
                                  penwidth='5',
                                  label=_node_label) )
……..
    graph_dot.add_edge(pydot.Edge(_cell_hier,
                                  _inst_cell_hier,
                                  label=_inst_name))
```

After all of the graph elements have been added, a pydot "self-check" function can be written to check the numbers of each element versus maximum allowed values and also versus golden values generated in previous runs. This allows the creation of a self-checking regression for checking graph output when tool versions have changed etc.

```
    for _subg in graph_dot.get_subgraphs():
        _subg_cnt += 1
……..
        for _node_s in _subg.get_nodes():
            _subg_node_cnt += 1
……....
        for _edge_s in _subg.get_edges():
            _subg_edge_cnt += 1
……....
```

Pydot also allows for upto 36 different picture formats to be generated.

```
    elif _out_format == "dot": graph_dot.write_dot(_out_format_file_name)
……..
    elif _out_format == "jpeg": graph_dot.write_jpeg(_out_format_file_name)
……..
    elif _out_format == "pdf": graph_dot.write_pdf(_out_format_file_name)
```

Verilog-Perl.

The Verilog-Perl library is an open source building point for Verilog support in the Perl language, part of this library is Verilog::Netlist which builds netlists out of Verilog files. This allows scripts to determine things such as the hierarchy of modules. So the functionality of Verilog-Perl is very similar to what is available in the VPI language and one of the aims of this paper was to compare the functionality offered by Verilog-Perl and VPI. A script Verilog-to-graphviz [10] was used as the starting point for the work with the Verilog-Perl tool.

Graphviz.

Graphviz [11] is an open source graph visualisation software. It takes an input file in the dot format and outputs a graphical diagram. The main option used here is "layout". Graphviz has a number of different layout options. The main ones used in our case were "dot", "circo", "twopi" and "sfdp". The tool produces images (.png, . jpg, .pdf) which are not editable. The main appeal of Graphviz for the applications are its swift output and readability of the dot input file format.

The "dot" layout is designed to create hierarchical diagrams of directed graphs. It is a great option for showing the hierarchy of a verilog RTL project. The diagrams created here are well designed and easily readable. An alternative

option for the use of the "dot" layout would be to draw the design of a state machine. This layout should be ideal for this purpose.

In order to graph the signal layout of an RTL module, three different layout options were investigated: "circo", "twopi" and "sfdp". The "circo" layout places each node in a circular layout. This can be a single circle or multiple, connected circles. Each node can then communicate with the nodes beside it on the circle or across the circle from it. This is a very good layout for diagrams that have a lot of communication throughout all modules on that level.

The "twopi" layout places nodes on concentric circles surrounding a root node. This is a great layout for diagrams that have a central module that other modules or ports communicate with, with little communication between other modules and ports.

The "sfdp" layout uses a spring-model layout for large graphs. It seems to be less planned than the previously discussed layouts and so the resulting diagrams are less predictable. This layout is quite effective for diagrams with few modules but can become unreadable when there are many modules and signals involved. It could be a good option when there is no obvious choice of layout between twopi and circo.

One of the main issues with the layouts discussed above is that if there are too many edges (signals) drawn between nodes (modules and ports) then the diagrams do get unstructured and edges and nodes can overlap other nodes. This can be fixed by just having an input, output and inout edge to show communication between modules. Similarly, adding text to edges to show signal names can make the diagrams too detailed and unreadable. However, small, simple labels, such as edge numbering do work ok. Many options were explored to try and improve both of these issues, but it seems that edges and edge labels are not optimised.

Another issue that was evident was that nodes do not space-out very well when node sizes are increased. This results in nodes becoming too close (or, in sfdp, overlapping) and edges become very short. There are many options to space nodes apart, however, these options do not seem to work as intended.

We used a VPI programme to generate dot programmes which in turn can generate hierarchical diagrams through graphviz with a number of options for the diagram layout. The top-level module and depth of the diagram could be chosen to create smaller, neater diagrams where necessary. Frames could be placed around designated modules and their submodules to denote specific functionalities of the design. Clocks and reset signals can be traced throughout the hierarchy of the modules. The resulting diagrams were tidy and easily readable. The options available allowed for smaller, neater diagrams to be created whenever diagrams were too large and expansive to be effectively viewed.

Current work in progress and future work involves using VPI to output the signal layout of each RTL module in dot format to generate the signal diagrams using Graphviz. There is a lot of data to deal with when working with module signals and mapping signal connections between modules has proven difficult, though there has been good progress made on this thus far. Once completed, Graphviz layouts will have to be tested to find the optimum settings and options to produce the neatest diagrams. Ease of use and speed of diagram generation are essential here and are the main draw of using Graphviz. For that reason, diagram settings will be the same across each diagram which may produce some diagrams that are not optimally designed. Personalisation options should be available here for the user in the case where an output diagram is poorly designed.

Diagrams.net (formerly Draw.io).

Diagrams.net is a free online diagram editor that enables you to create flowcharts and more. It is open source and has two mechanisms for importing diagrams namely these are through .csv and .xml files. Diagrams.net displays an editable image (XML) which can then be exported to standard image formats .png, .jpg, .pdf. Another option for generating hierarchical diagrams (and hopefully full fletcher diagrams in the future) that was explored was using diagrams.net and its csv import function. The strengths of diagrams.net over graphviz is that it generates an editable image. Diagrams.net also has a lot of customization such as more advanced layouts and manual placing of objects in the diagram. One other option that was explored was by generating the image using XML. Diagrams.net files are XML format so this would in essence give us the most control of the output. The problem with XML is that while it

gives more control, it takes away the “auto layout” feature (available in csv) and forces the user to place every object via coordinates.

The following diagrams are possible additions to this flow, but have yet to be implemented.

- Diagrams tracing an interface through the hierarchy to the block in which it is first processed.
- Testbench and interface diagrams
- FSM diagrams

Pydot

Pydot is a python interface to Graphviz, can parse and dump into the DOT language used by GraphViz. Some of the functions that were used as part of this project were described in the Verilator section above.

III. RESULTS

Diagrams were generated for three separate designs; two designs were under development by CreVinn and an additional design was an open source RISC-V processor by lowRISC.

1. VPI (Cadence Xcelium), the design hierarchy was easy to generate as there were examples of how to do this with the iterate and scan function. Slower progress was made in getting the design net/reg connections. Overall though once the method to establish the design connections was worked out the VPI code worked very well. The loadvpi switch in Xcelium made importing VPI code into the simulation very easy. A further advantage of this flow is that the VPI code can easily be integrated into an existing design/test flow and diagrams could be generated on an ongoing basis. One possible future extension to the VPI scripting would be for it to generate IPXACT XML code instead of dot/csv code. This way the VPI and verilator flow could be combined as ultimately they would both be processing XML code to generate dot/cvs diagram output.
2. Verilator was not initially considered for this investigation but once the XML export and associated starter script was discovered this tool warranted investigation. Pydot was also a major benefit to the python script as it allows the script output to be easily verified prior to generating the diagrams. No real documentation of the Verilator XML code could be found so it was a process of trial and error to work out the parse syntax. Another challenge with Verilator is that it would probably exist as a separate flow to the normal design/test flow. A separate flow would add to the maintenance of the diagram generation environment.
3. Verilog-Perl, Once the latest version of the Verilog-Perl tool was installed, the Verilog-to-graphviz script provided a fast way to get started with Verilog-Perl. In order to get the module connections, the pinselects from the Netlist were used. Ultimately some limitations were found when trying to traverse the connectivity of the a design with Verilog-Perl, Verilator is a better option in this respect.
4. Graphviz, The "dot" layout is designed to create hierarchical diagrams of directed graphs. It is a great option for showing the hierarchy of a verilog RTL project. The diagrams created here are well designed and easily readable. However, when more details were to be added to the diagram, the output may not be so clean and readable. Luckily, this is not necessary for the hierarchical diagrams that we are designing. In order to graph the signal layout (connections) of an RTL module, three different layout options were evaluated; "circo", "twopi" and "sfdp".
5. Diagrams.net, two basic options in diagrams.net for adding diagrams using scripts, these were CSV and XML inputs to the tool. Initially the XML input was preferred as this was ultimately the language in which the diagrams were stored. However the CSV input had some key advantages over the XML input in that the CSV input allowed better placement of components and arrows. The diagrams that were generated with this method has the potential to capture all of the design connections of the hierarchy, but it would have required some manual intervention to make it readable, so this method would not have been fully automatic.

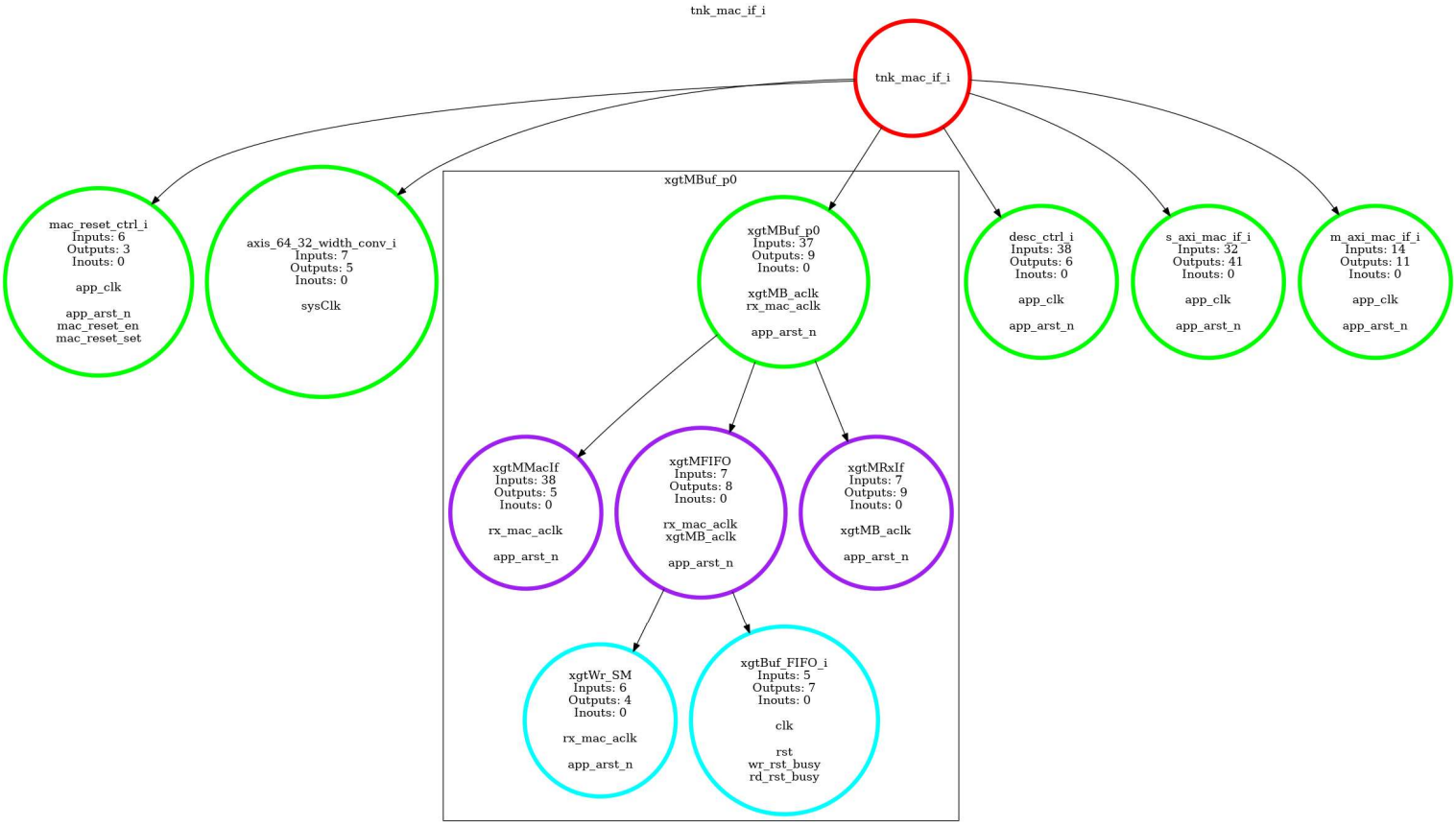

Diagram 1: tnk_mac_if block diagram (image generated using graphviz)

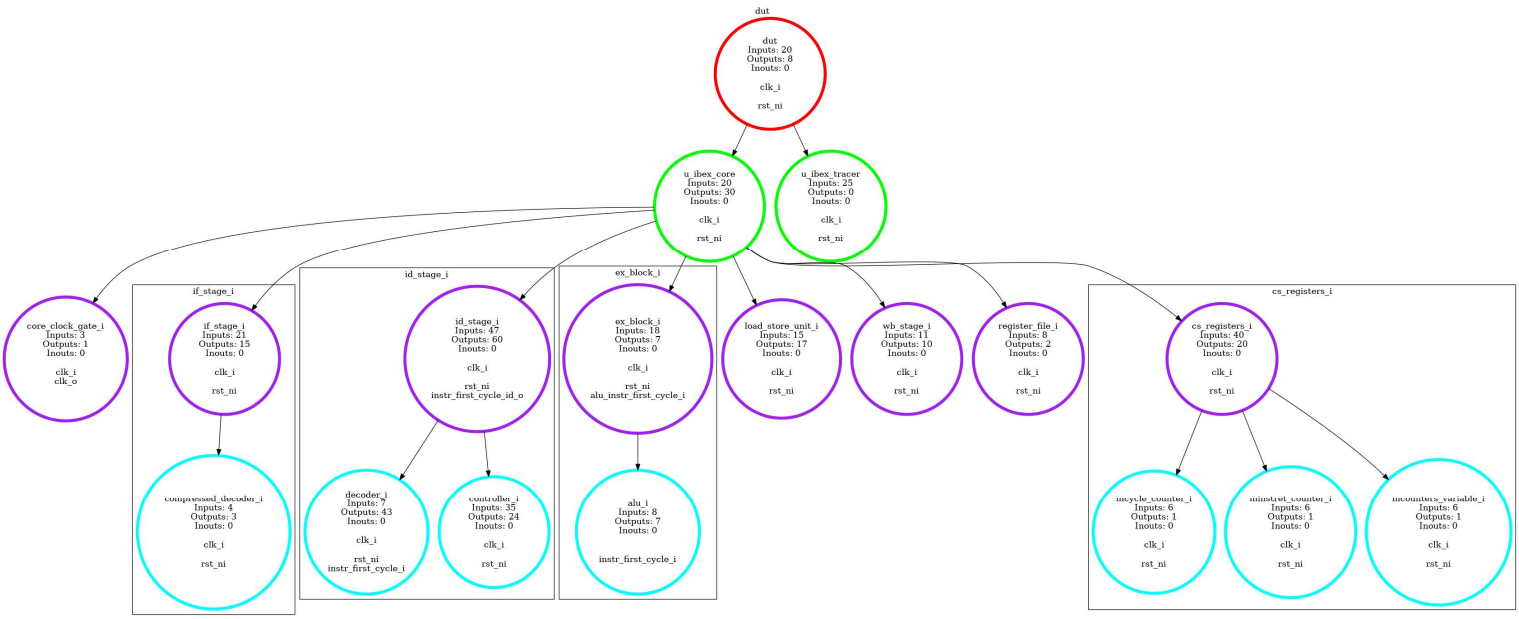

Diagram 2: lowRISC Ibex block diagram (image generated using graphviz)

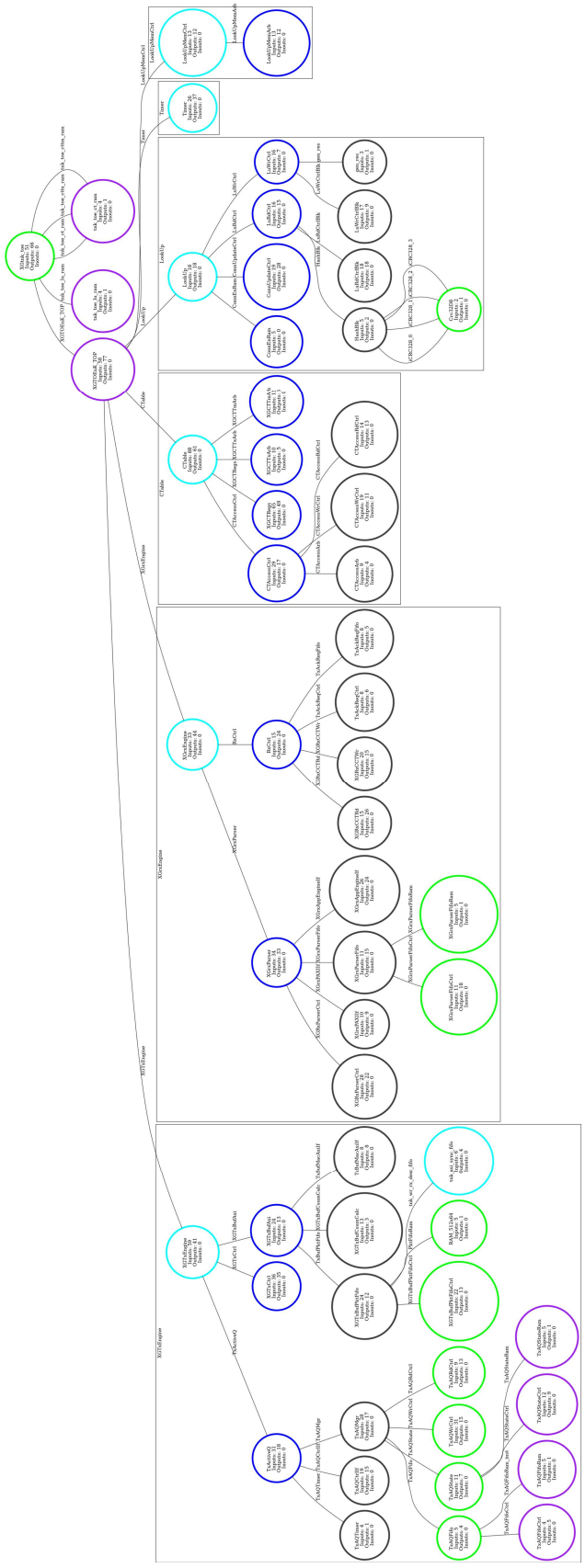

Diagram 3: XGtnk block diagram (image generated using graphviz)

I. Comparisons with other tools

By way of comparisons two comercial tools were investigated for generation of hierarchical and connectivity diagrams. The best tool was Xilinx Vivado (2018.02) which produces a good hierarchical diagram which captures the size of each block, this diagram is difficult to navigate when trying to find smaller blocks. Vivado also produces a good connectivity diagram showing all of the connections between the modules. The diagram is not editable and can end up being a very detailed diagram for larger blocks.

CONCLUSIONS

Most of the information required for automation of diagram generation is possible with the three packages VPI, Verilator and Verilog-Perl. Although some trial and error was required to work out some undocumented aspects to Xcelium VPI , Verilator XML and Verilog-Perl Netlist. The two diagram display tools have their advantages. Graphviz provides very quick output and a language that is readable. Diagrams.net has more options in terms of generating connectivity diagrams but the csv input is more difficult to generate. Overall the investigation was a success and the flow will be incorporated into future projects.

REFERENCES

[1] IEEE SystemVerilog_1800-2017 - LRM (Chapters 36-38 VPI)

[2] Cadence VPI examples hier_walker, count_args, dtran for tool version: XCELIUM_19.09.007 (tools.lnx86/inca/examples/vpi)

[3] S. Sutherland 2002 - SNUG - VCS PLI2.0/VPI

[4] W. Snyder - Verilator tool https://www.veripool.org/wiki/verilator

[5] W. Snyder 2019 - Chips Tools 2019 - Creative uses of Verilator

[6] W. Snyder 2019 Vl_hier_graph script https://github.com/verilator/verilator/blob/master/examples/xml_py/vl_hier_graph

[7] S. Kalinowski 2018 - Pydot https://github.com/pydot/pydot

[8] W. Snyder - Verilog-Perl tool - https://www.veripool.org/wiki/verilog-perl

[9] Diagrams.net diagram capture tool - https://www.diagrams.net

[10] S. Eldridge 2017 - Verilog-to-graphviz script - https://github.com/IBM/hdl-tools/blob/master/scripts/verilog-to-graphviz

[11] Graphviz - https://www.graphviz.org/

[12] E. Gansner, E. Koutsofios, S. North 2015 - Drawing graphs with dot, https://www.graphviz.org/pdf/dotguide.pdf

[13] Fletcher Design Methodology - Tadhg Creedon (CreVinn Teoranta)

ACKNOWLEDGMENT