Project Number: *BS2 - 0218*

Formal Verification of ASIC Design Equivalence

A Major Qualifying Project Report

Submitted to The Faculty

Of the

Worcester Polytechnic Institute

In partial fulfillment of the requirements for the

Degree of Bachelor of Science in

Electrical and Computer Engineering

By

_____

Jonathan Ariza

Approved:

_____

Prof. Berk Sunar

Abstract

The goal of this project is to use a formal verification tool that checks all functionality of the RTL and GTL code to prove they are equivalent to one another. This check covers 100% of all the functions inside the code which is known as formal equivalence check. The Mentor FormalPro tool allows for easy debugging with the use of a built in GUI and several types of reports that provide information on errors within the code. This project is sponsored by PLSense Ltd. which is based in Yokneam, Israel that provide IoT SoC design to achieve minimum energy operation for the targeted performance in a wide range of frequencies.

Table of Contents

Table of Figures

Introduction

This Major Qualifying Project (MQP) was accomplished in a partnership with PLSense Ltd. which based in Yokneam, Israel. PLSense works to provide IoT SoC design to achieve minimum energy operation for the targeted performance in a wide range of frequencies (up to 100MHz). This is done with a combination of variety of unique patented technologies at different abstraction levels, including physical, circuit, logic cells, architecture and software.

Design of any Application-Specific Integrated Circuit (ASIC) today is done using a high-level code called Register Transfer Level (RTL). This language describes the logic behavioral of the silicon and is easier to write, debug and understand when writing directly to the logic call.

The transformation from the RTL level to the logic gate level (GTL) is done using an automated synthesis tool that reads the RTL code and the target libraries to convert the high-level code into a structure of registers and random logic gates. The transformation is done according to how the synthesis understands the RTL code but if this code is not written clear enough the synthesis can misinterpret the code and create logic which is not comparable to the RTL source.

To make sure the GTL netlist describe correctly the RTL code there is a need to check the results of the Synthesis tool. There are two ways to do this check, one by running logic simulations to verify that what is working at the RTL will also work the same at the GTL netlist – this task is very long and usually does not gives full coverage for the code. The second option is to use a formal verification tool that check all functions inside the RTL and GTL code to verify they are equal – this check covers 100% of all the functions inside the code and is called formal equivalence check.

The scope of this project was to learn the background and methodology of the formal verification and in addition to learn the Mentor FormalPro tool. Using the acquired understanding of the tool to run verification on the PLSense new generation chip equivalence task to verify that the result of the chip synthesis is equal to the RTL description.

Verification Attributes

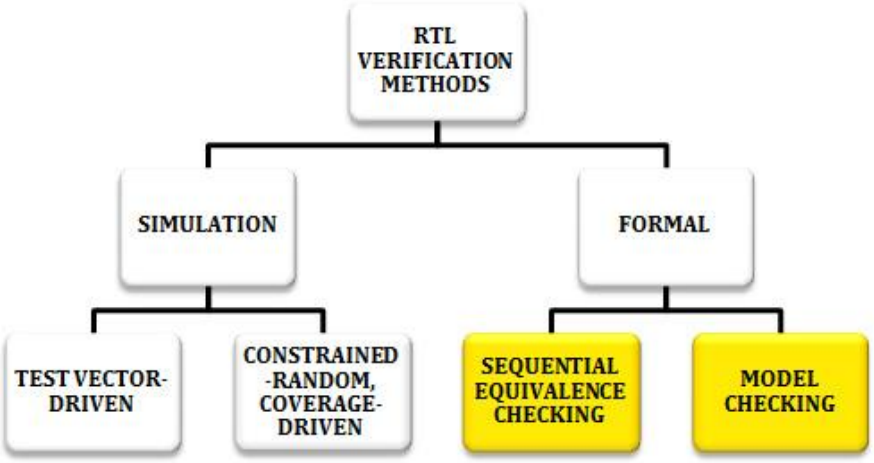

Figure 1: Comparison of Verification Methods between Simulation and Formal Verification [7].

The use of logic simulations is becoming less practical to use as a method of verification between the RTL code and GTL netlist. The coverage this method provides is also not entirely accurate and can leave hidden problems that can cause serious complications down the line. The required aspects for formal verification to overcome the limitations the logic simulations manifest is to be fast, useable, and a reliable to provide optimal coverage. Formal verification is potentially very fast because it does not have to evaluate every possible state to demonstrate that a given piece of logic meets a set of properties under all conditions. However, its performance depends greatly on the type of logic on which it is deployed and the way it is applied [5]. Depending on the way the application is designed along with the size of the code formal verification can take as little as

5 seconds to complete, but with more complex designs the time required can shoot up to 24 hours. Although the amount of time it can take to do verification can become significantly larger the reason for this is simple and depends on the code size and complexity. With formal verification all possible behaviors of the design are analyzed to detect any reachable error states. This exhaustive analysis ensures that critical control blocks work correctly in all cases and locates design errors that may be missed in simulation [6]. For these reasons formal verification has several advantages compared to other forms of verification that can be found on the market today. Therefore, formal verification was the optimal choice to use for verifying the latest project of PLSense to ramp-up this process.

Background

This project was completed in several phases over a span of four months were two months are done off site and the other two are done on site.

Phase I

The first phase consisted of studying the Verilog language to establish a firm understanding on the format and design style. This was done to understand potential code one could encounter on site. The Mentor FormalPro tool was also learned from reading the reference and user manual to understand how the tool operates.

Phase II

The next phase is to apply what was learned in the previous phase and create a simple Verilog code. This was done to gain a more profound understanding on how to write Verilog code and interpret possible code encountered on site. The code was then taken and was run through a

synthesizer and then run through the verification tool to get a look at the different types of reports FormalPro generates. This also gave a preview of what errors one could expect to see and the process to take for debugging errors in the code.

Phase III

The third phase was accomplished on site in Israel, using all the acquired skills from the previous stages. The goal of this phase is to run the equivalence check on all the PLSense PLS15 synthesis blocks to verify that each one of them is identical to the source RTL.

Chip Components

The components of the chip are made of three main blocks being the Real Time Clock (RTC), Always On (AON), and main CPU subsystem functionality. Each of these blocks contain code that had to be run through the Mentor FormalPro tool separately and then altogether. The RTC as the name suggests contains code for a Real-Time Clock for the chip to determine what the current time and date is as well as other functionality pertaining to a clock. The AON is the component of the chip that is always on to allow the chip to wake up when it enters its sleeping state. Lastly the CPU subsystem contains all the functionality of the chip and therefore contains a majority of the code.

Study Phase

The first two phases were considered the study phase of the project with the last phase being the work phase. To complete this project, the first task was to become familiar with the Verilog language. This was done through an online tutorial that covered several aspects of the Verilog language [3]. Once the tutorial was completed a good grasp of Verilog was achieved and

further enforced by summarizing key points of the tutorial. The next part was to implement different design levels of Verilog code. This was done to get a feel of how the Verilog language is a hardware description language and therefore completely different from other languages such as C/C++. The practice code that was implemented was a microwave oven timer and a four-bit calculator. The steps taken to write the code started with creating diagrams of necessary components and functions that a timer and calculator would need to operate. The simplest code of the two was then run through a synthesis tool and then through the FormalPro tool. The tool generated several reports and a short summary report that gave the number of plain differences or unmatched inputs encountered during the run. This gave a general idea of what information was important to view when an error was encountered and when the report looked like when no errors where found. To gain a better understand the functionality of the tool the user manual was provided along with the reference manual. Key points of the manual were summarized to be used in understanding what information the reports provided and what is important within the report. The manual also gave the commands needed to launch the GUI and commands to debug different types of errors in the code. The most important thing from the manual was the necessary files and commands one would need to run the FormalPro tool.

Once on site a version of the RTC code was given that had a known error inserted into the code to test my understanding of using the tool. The generated summary report showed several mismatched registers that were being affected by only a few top-level registers that were being passed on through the code. The approach to find the error was by looking at the unmatched objects report to find the specific registers that were not matching. The next step was to view the netlist and RTL code and compare the two. This showed that two registers were missing from the netlist

but are present in the RTL. Once the registers were added back into the netlist the verification passed without any errors.

RTC & AON

Upon completeing the practice code the next phase began starting with the verification of the RTC block. The amount of code in this block was fairly small with simple code and was not expected to have many issues. The verification report showed that their was only one issue and by looking in the unmatched objects report it was easily found to be an input in the Black Box A design was unmatched to that of the B design. When the code was compared in the netlist and RTL it was noticeable that the input had been defined as the wrong size within the black box and was given a size of five when the actaul value needed to be seven. After the size was changed in the black box and the code was resynthesized the RTC code passed the verification.

The next block of code was to be run was the AON, using the RTC as a template to create necessary files to run the verification. The first run of verification showed several different complications within the code with a majority of them being removed comparison points. Looking further into the complications it was discovered that there was a major problem pertaining with the black box. To fix these issues the whole black box need to be rewritten from scratch which took some time to complete. Once the black box was fixed the FormalPro run came back with a few problems as shown in figure 1 and 2. The thing to note from the figure 1 is the unmatched objects, namely the real inputs, outputs, and Black Box inputs/outputs in both designs. As one can see there are a total of eight real unmatched ports in the a design and a total of sixty six in the B design. At this point of the report one can tell that the verification will fail due to these issues shown in the unmatched objects. The second figure gives the comparison summary with the most

important thing to notice are the removed comparison points, different comparison points if there are any, and lastly the unmatched ouputs similar to the report in figure 1. All the removed comparison points that are being shown in the report where not completely relevent when it came to finding the cause as they were usually a result from the unmatched ports. The causes of these unmatched outputs were easily found when looking at the RTL and netlist code. A few of the ports were defined as inout, but needed to be inputs and others as outputs. There was also an input that did not have the correct size like in the previous block, once all these changes were made to the black box and resysnthesized the verification reported back clean.

Matched objects

	in	out	BBout	BBin	Uin	Uout	dff	latch	total
Design A	85	203	61	274	0	0	1001	11	1635
Design B	85	203	61	274	0	0	1001	11	1635

Unmatched objects

	in	out	BBout	BBin	Uin	Uout	dff	latch	total
Design A									
real	3	1	1	2	0	0	1	0	8
benign	6	0	0	2	0	0	4	0	12
Design B									
real	0	2	1	0	0	0	0	63	66
benign	6	0	2	0	0	0	0	0	8

Target Generation

	out	BBin	Uout	dff	latch	total
Targets Generated	203	274	0	1001	11	1489
Suppressed Targets:						
No-Path	0	0	0	0	0	0
Ignorable	0	0	0	0	0	0

Solve

Engine run	Targets fed	Equiv. targets	Diff. targets	Unsolv. targets	Removed/ Deferred	Elapsed hh:mm:ss	% Compl.
Totals	1489	810	0	0	679	0:00:04	100.00

WARNING: Some targets have a cycle or a multiply driven,
floating, or unmatched net in their fan-in.
For more details please refer to the report:
./formalpro.cache/reports/detailedComparison.report

Figure 2: Unmatched Objects is the most important section in this part of the report as one can see the real unmatched inputs and outputs in the design that are one of the reasons the verification can fail. The run can then be aborted to solve these issues to get a cleaner run.

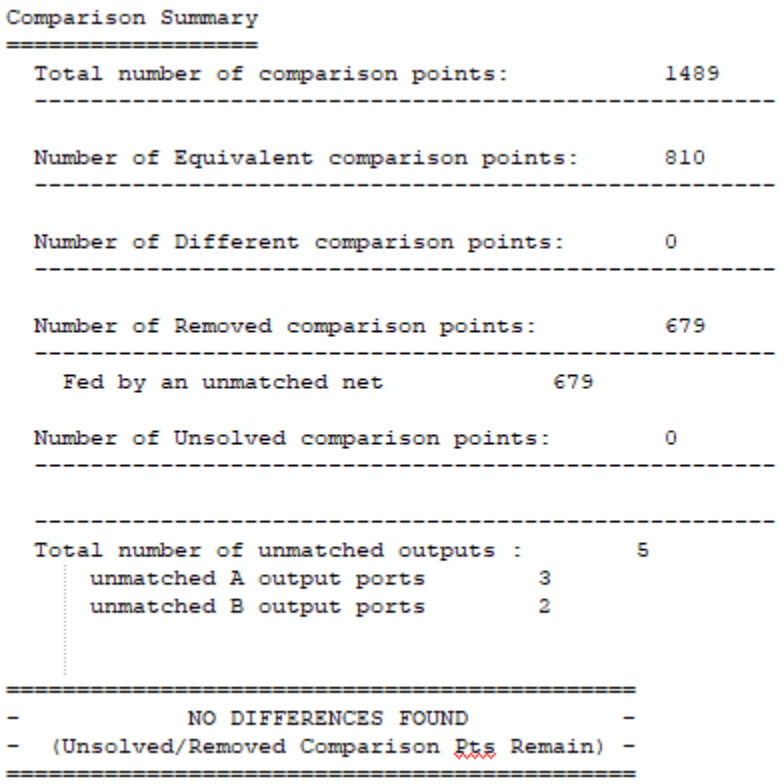

Figure 3: AON Summary Report note the removed comparison points and the unmatched outputs give the number of errors that the verification found in the run. Although the last statement the verification gives is not differences being found it cannot be considered a clean run as there are removed/unsolved comparison points remaining.

CPU Subsystem

The *cpu_subsystem* component of the chip is the most code heavy of the three parts of the chip. This fact led us to believe that the amount of time it would take to complete a verification run would be large amount of time to run. We came to this conclusion because the size of the code increased the probability of finding multiple errors and thus increasing the run time. Therefore, a small section of the block was taken and run on its own without the need of running the entire

netlist of the *cpu_subsystem*. This small section of code is known as archipelago and was run through the verification tool. The first run of the archipelago code ran the full specified run time of twenty hours. The result of the run came back with several problems such as unmatched inputs to removed comparison points due to combinational cycle and unmatched nets, along with a few plain differences. The first problem we needed to fix first was the unmatched inputs as from the little experience I gained working with the tool the unmatched inputs are the main problems that cause the verification to fail. From the general report it was seen that there was about the same number of unmatched inputs in both the A and B designs. When looking at the unmatched objects report and noticed that three unmatched flip flops were removed from the RTL when the code was ran through the synthesis tool, but FormalPro was reporting the registers to be necessary in the design as shown in figure 2. Using the information from the reports we came up with a hypothesis that the synthesis tool Oasys that was being used was either correct in optimizing these registers out or the synthesis tool was wrong in optimizing them out as FormalPro was reporting. To confirm which of the tools was correct we ran the verification on a version of the code before it was optimized to see if the problem persisted before the optimization. The results of the run where similar to that of the optimized code with the only difference being that the number of unmatched inputs in both designs were equal. After this we attempted a few other things such as forcing the registers to zero to a set value so that FormalPro could then ignore them. However, these other attempts did not solve the issue of the three unmatched registers.

We decided to shift our focus on the other errors we could solve and found that the unmatched inputs in one design were slightly different form the other design. The reason this was happening was a slight name change that occurred when running the Synthesis tool. As a result, FormalPro was having a problem with matching these names between the registers in the RTL and

in the netlist. To resolve this issue, we had to generate a file that told FormalPro what names matched between the RTL registers and the netlist. This was done manually and required a great deal of time to accomplish as there were over two hundred registers to match. Once this was done the verification was run once more for the full run period and reported back with all unmatched inputs being cleared up. This cleared up a large amount of removed comparison points but the three plain differences along with combinational cycles that were causing the verification to run the entire time. Looking into the plain differences the problem that were occurring had to do with the same three unmatched registers that were being appearing in the A design while not being in the B design. Once we reached this point we were not sure how to fix this problem, so we got into contact with Mentor support group. While we waited for them to get back to us and set a meeting we investigated the combinational cycles that was made up of mostly memory cycles that we could ignore. A few of the combinational cycles were an issue and could only be found by looking through the code and drawing block diagrams of what was taking place within the code. Once these combinational cycles were fixed the code was resynthesized and no longer being reported in the combinational cycles report.

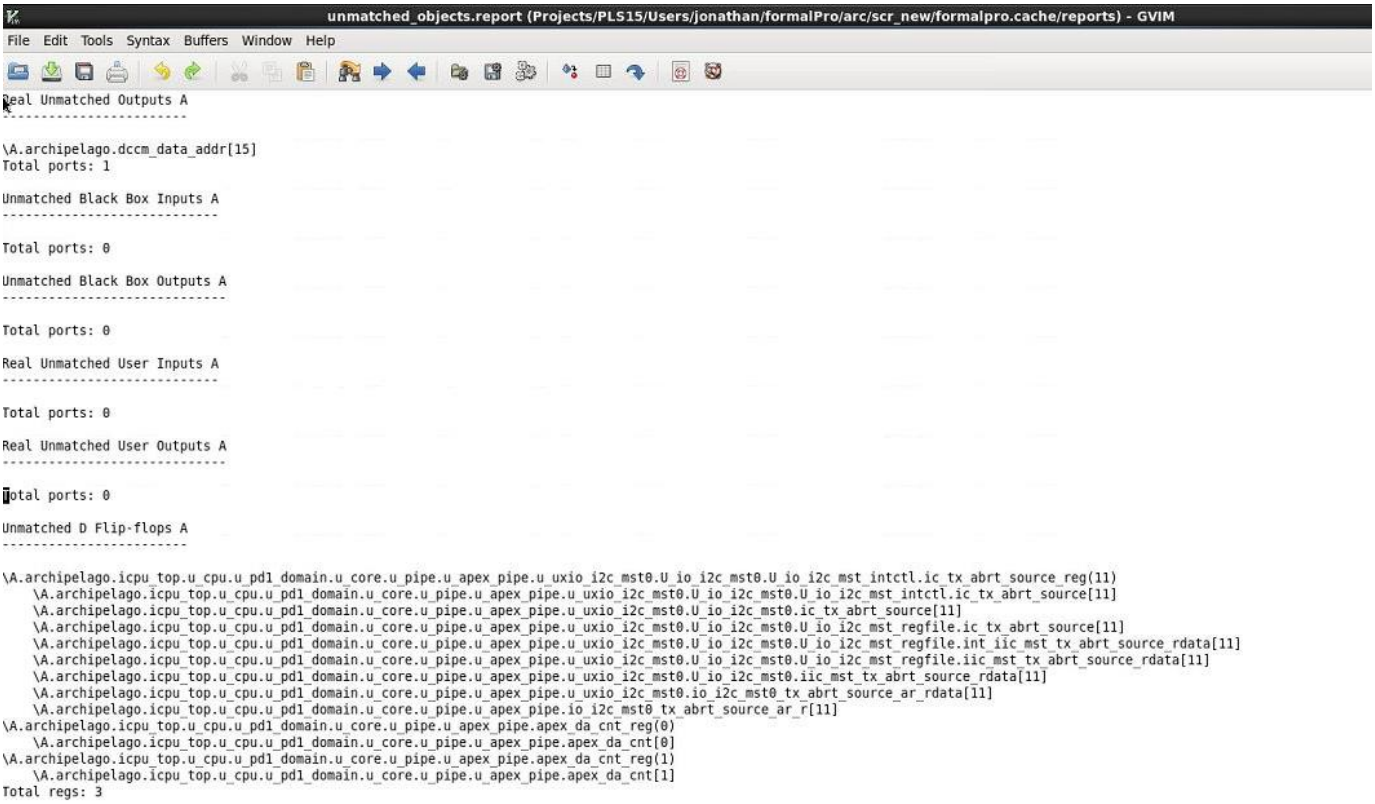

Figure 4: Archipelago Unmatched D Flip-flops in A Design shows three main registers that are appearing only in the A Design. Below these three registers are other registers that are affected by these registers being unmatched. These registers kept the verification from coming back clean and thus preventing the project from moving forward.

As the archipelago code was not getting us any further it was decided that we should go back and run the verification on the entire *cpu_subsystem* block. Using the same file, we had used to match registers in the archipelago block, we ran the verification on the *cpu_subsystem*. The run showed a few more registers that needed to be added to the file to clean up all the unmatched inputs in the A design. After going through another verification run with the updated match file, we now had twenty-six removed registers that had no matching registers. These registers gave us the same problem as we found in the archipelago run. After a few meetings with the Mentor support we learned the issue that was causing these unmatched registers. The reason these errors where occurring was because the Oasys synthesis tool was doing complex optimizations to the logic which the FormalPro tool could not do or fully understand and therefore caused issues during the

verification. The correct procedure to run FormalPro on a complex block like the *cpu_subsystem* is to use the Oasys Synthesis flow. This means adding a command to the Oasys tool to have it create a ready script for FormalPro that contains all the needed constraints which reflect the Oasys optimizations and assertions to check they are correct at the end of the synthesis. The approach of how we ran the *cpu_subsystem* was also changed to run in a hierarchical fashion where the tool runs from bottom-up verifying each block individually. Using this method, the first run of *cpu_subsystem* took less than 1 hours to complete and all the issues that we were encountering before were no longer a problem. The only issue with this approach was the scan ports that are inserted into the code by Oasys for testing purposes caused issues with the verification. This was confirmed by running a copy of the *cpu_subsystem* that contained no scan ports. To get the verification to pass with the scan ports we had to tell FormalPro to ignore or force the ports so that they do not cause any issues. The way to find all the scan ports was to look in the exception logs as shown in Figure 4 which gives which file contained scan ports. I went through this log file and created a global constraint file and once this was complete the verification was run and resulted with new issues. We then learned that this global file was somehow causing the inputs to be declared as benign in one design and therefore causing the ports to be labeled as unmatched. The only way to fix this was to add the constraints into each individual file that contains a scan port.

```
#   EXCEPTIONS REPORT   #
DIFF LIST
EX FILE D: formalpro/bitscan/bitscan.log
 EX FILE D: formalpro/core/core.log
 EX FILE D: formalpro/cpu/cpu.log
 EX FILE D: formalpro/dccm_ram/dccm_ram.log
 EX FILE D: formalpro/decode/decode.log
 EX FILE D: formalpro/dmp_load_aligner/dmp_load_aligner.log
 EX FILE D: formalpro/dsp_add/dsp_add.log
 EX FILE D: formalpro/dsp_aux/dsp_aux.log
 EX FILE D: formalpro/dsp_mac32x32/dsp_mac32x32.log
 EX FILE D: formalpro/execute/execute.log
```

Figure 5: Exception Log Report provides the log files that contained errors that needed to be addressed. In this case each file contained scan ports that needed to be addressed so that the verification could be clean.

Encountered Challenges

Throughout this project there were several challenges that were encountered while working with the FormalPro tool. The most challenging aspect of this project was the amount of time required in running the verification. With larger and complex code design the number of errors were higher and would thus cause FormalPro to take longer in doing the equivalence check. To acquire meaningful data from the verification run a maximum time limit of twenty or more hours were given. As a result, a lot of time was used waiting for the tool to either finish the run or time out. Over time we gained more experience with using the tool and determining what caused errors. When the tool is running it generates a quick number of matched and unmatched objects found in both designs along with the reports. It would then go into the solving stage where it could take a long to run. What we learned was that the unmatched objects were usually the cause of the errors we encountered and could thus stop the run once the unmatched object report was generated. The other option was to run the large block in the hierarchal mode where we run the design block by clock bottom up and by this reduce the run time significantly.

The other problems that were encountered dealt with using the tool as it was the first time using it as well as understanding the code. These problems were unavoidable as nothing could really be done about this. However, with the help of my sponsor I was able to become more familiar with dealing with the tool and code. The tech support was also very helpful in resolving the problems that we were unable to solve ourselves.

Outcome

The result of this project was the verification of the three main blocks of code that make up the chip. The last step to be completed was the verification of all the blocks together to ensure the chip functioned in equivalent. Although with the time constraint of this project this final step had to be done by someone else. Thus, on the final day of the project I passed down all the knowledge I obtained in working with this new tool to a co-worker. This was done so that the tool will be better understood as I devoted all my time working with the tool versus someone trying to figure out the tool on top of doing other important work. This allowed PLSense to get an understanding of using this new tool while also verifying code for their latest project. While on site I also used the tool to verify experimental code that was using a different optimization tool to see if the results would be better compared to the current one being used. Overall this project helped PLSense to ramp-up their equivalence check process in the fastest manner possible while using it to verify their newest chip.

References

[1] *FormalPro User's Manual*, Mentor Graphics Corporation, May 2017. Accessed on: Feb. 21, 2018. [Online]

[2] *FormalPro Reference Manual*, Mentor Graphics Corporation, May 2017. Accessed on: Feb. 21, 2018. [Online]

[3] Tala, Deepak Kumar. *Verilog Tutorial*, 9 Feb. 1970. [Online]. Available: www.asic-world.com/verilog/veritut.html

[4] Williams, Stephen. "Icarus Verilog." *Icarus Verilog*, Jan. 2010. [Online]. Available: iverilog.icarus.com/.

[5] Dempsey, Paul, et al. "Formal Verification." *Tech Design Forum Techniques*, 2012. Available: www.techdesignforums.com/practice/guides/formal-verification-guide/.

[6] Questa Formal Verification, Questa *Formal Verification*, 2017. Available: www.mentor.com/products/fv/questa-formal/.

[7] Oski Technology, *Oski Technology Formal Verification Methodology*, 2005. Available: www.oskitechnology.com/solutions.