DOI 10.31114/2078-7707-2018-2-16-22

Practical Aspects of Formal Verification of Networking Chips

A.A. Sokhatski

Cisco Systems Inc., asokhats@cisco.com

Abstract — **Formal Verification (FV) is becoming now important part of Design Verification (DV) especially for areas which has higher requirements for quality of chips such as networking chips which should work 24x7 without failures while re-spin cost for huge chips is high. It is difficult to cover all possible scenarios by simulation having a lot of corner cases for packet alignments, sizes, combinations of values of configuration ports and registers. Formal Verification should be able to help to improve quality and reduce Time to Market but it requires:**

- **Selection of right scope, candidate and method for Formal Verification;**
- **Addressing Formal challenges, main of such is fighting with complexity and exponential grow of proof time with each proof cycle;**
- **Consistent Methodology to ensure verification coverage and to reduce effort.**

The paper goes through those aspects basing on experience at Cisco Systems Inc. with help from OSKI Technology [1], Formal Verification service provider and Formal sign-off company[2]. The paper covers:

1) **Brief review of Formal Applications while concentrating on End-to-End Formal sign-off for Design Modules along with criteria for selection of good candidates for Formal;**
2) **Structure of simple Formal Environment;**
3) **Methods helping to fight with complexity which author found especially useful for data transport modules of networking chips, such as floating pulse method, Wolper method, use of abstraction models;**
4) **Formal methodology aspects including:**

- **Document and test plan flow;**
- **Run Flow, Regression & Scripting;**
- **Coverage Flow;**
- **Reuse for Formal & Simulation.**

Keywords — **Design Verification, Formal Verification, SystemVerilog, SVA.**

I. Introduction

This paper is primarily based on the experience in Design Verification of networking chips at Cisco Systems Inc. There is good history of using Formal Verification at Cisco. However in our department we found that we need more consistent shared flow and infrastructure support. We started from training from OSKI Technology, recognized experts delivered Formal Verification services, and then worked on establishing Formal Methodology.

Formal Verification now is a good complimentary to simulation based Design Verification [3]. It has advantages such as:

- High quality: potentially exhaustive proof for implemented checkers for any legal input sequence and configuration combinations; is able to catch corner case bugs;
- Typically less time to setup and verify; could be started by designer at early stages.

On the other hand it has the following restrictions and challenges:

- Restrictions on design complexity and input sequence length;
- Might require application of special techniques to fight with complexity which needs expertise;
- Time to closure is hard to predict, sometimes it’s even difficult to tell if design if suitable for FV.

The key point is to select right FV application, level and design entity.

II. Formal Applications & design selection

A. Formal applications types

Formal applications [4], [5] could be classified by objective, automation and effort from user, in particular:

1) Automated Formal Linter which doesn’t require writing user code but allows to find such issues as array boundary violation, multiple active drivers for the signal; some tools consider check for dead code, etc. from the next section also as part of Automated Formal check;

2) Formal Apps for particular verification aspects including:

- Formal Coverage Analysis (FCA) [6]: allows to find dead code, unreachable FSM state, etc.; we made it as part of DV Flow; pretty useful for simulation coverage closure;
- Connectivity Check: allows to describe in simple way source, destination points and conditions for

connectivity checks; easy to apply, could be used at different levels including chip level;

- X-Propagation: allows to find RTL issues even before simulation started, especially critical for X-optimism when RTL simulation shows concrete value instead of 'X';

3) Assertion Based Design Exploration / Bug Hunting: assumes writing some number of checker assertions (along with assumptions and coverage assertions) and trying to prove them; it could be used from early stage of the project by designer to late stage by DV engineer, some applications are listed below:

- Designer or DV engineer describes assertions for DUT interfaces; for input interfaces assertions should be turned later to assumptions by Formal Tool commands; for standard interfaces Assertion VIP could be used if available; Formal Tool should be able to check compliance with interface protocols [7]; internal assertions for modules could be added by designer and verified as well;
- Designer could add to that coverage assertions exercising some simple or corner-case scenarios and get waveforms without running simulation;
- At later stage of the project along with simulation DV engineer could write specific assertions to hit and verify corner case scenarios and close coverage; advanced Bug Hunting techniques could use as initial state for Formal proof, state, where design comes after some simulation cycles;
- During top-level simulation or even while exercising manufactured chip there could be found issues which require reproducing them; Formal could help here and also ensure that fix is complete;

4) Equivalence Check: assumes comparison between reference design / model and design to be verified; historically it was one of the first widely used Formal applications to check equivalence between RTL and synthesized code; now dedicated tools are introduced which do comparison between two RTL designs or even with C-model; this application requires RTL / accurate reference model;

5) End-to-end sign-off check of design modules basing on SVA assertions which assumes exhaustive verification at some scope.

While all Formal applications are pretty beneficial, the paper is focusing on the End-to-end sign-off Formal Verification which could replace simulation for certain design modules, release DV engineering resources and exhaustively check design at that level. Formal could be applied at different levels including architectural. The paper primarily talks about Formal Verification of RTL design code.

B. Design Block selection for Formal Verification

Due to Formal Verification complexity & potential exponential grow of Proof time for each next clock cycle because of exhaustive nature of Formal proof, special care needs to be taken when selecting modules for Formal proof.

The following factors need to be taken into account when identifying modules for formal proof:

- Type of the design, best fit is control or data transport design; data transform design having multipliers, wide adders inside is difficult for Formal model check, for this type of design equivalence check might work;
- Complexity of design: size of design should be decent, pipeline depth and input sequence for verification should not be too long; it is difficult to give certain numbers as it depends on functional complexity, symmetry of design, used Formal Tool, etc., just some example of design proven by FV has several thousands of flip-flops, several hundred inputs, summary of pipeline depth and input sequence to prove – around 20 cycles;
- How much benefits we are getting from Formal comparing to simulation: number of corner cases, configuration combinations, is it reuse block which should work in different modes and should be exhaustively verified;
- Status and history of verification: if previous version was verified via simulation, how different is the new one; what is simulation coverage; were any bugs found after simulation verification done.

After identifying candidates, evaluation and planning needs to be done including:

- Measuring design complexity: number of flip-flops, delay;
- Better understanding design, parameters;
- Matching skill set of available FV resources.

III. Structure of Simple Formal Environment

Formal Environment for end-to-end check targeting sign-off typically includes:

- Interface components implemented protocol checks;
- End-to-end checkers.

Structure presented in the figure below.

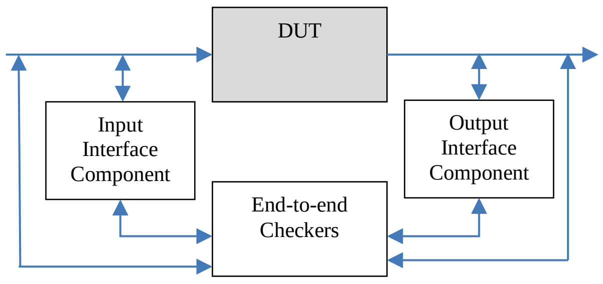

Figure 1. Simple Formal Environment Structure

Interface component features and applications:

- Located in separate modules;
- Contains SVA checker assertions, assistant code and coverage assertions, no assumptions;

- Assertion naming convention should reflect direction, for example source-destination: *src__dst_<name>;* it is critical when signals of both directions are integrated inside the same interface;
- Assertions changed to constraints (assumptions) from inside Formal environment script for input signals / Input Interface Components;
- Interface components instantiated inside simulation environment as well as in Formal which is especially important for Input Interface Components to check for over-constraints (failure of assertion in simulation is sign of over-constraint);
- Interface Components could contain intermediate results which could be used inside End-to-end Checkers

End-to-end checker features:

- Typically more complex than Interface Components and require special methods to fight with complexity and exponential time grow; the methods are described in the next section;
- Could use intermediate results from Interface Components.

As an example of data transport module from networking chip we'll take Delimiter Removal module which converts one packet protocol with Start-Of-Packet (SOP) – End-Of-Packet (EOP) delimiters located inside data flow to another protocol which has those signals located in separate signals. Next figure illustrates the simplified function. In real life the protocols are more complicated: there are several bytes passed each cycle; packet with low priority could be interrupted by packet with high priority.

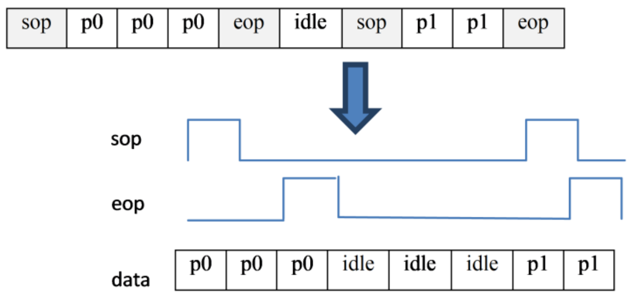

Figure 2. Simplified function of Delimiter Removal Module

Here Input Protocol is more complex and requires functions to extract SOP-EOP flags from data stream using assistant Verilog code. The results could be used by End-to-end checkers.

IV. METHODS TO REDUCE PROOF TIME

Formal Verification exhaustively verifies design for any legal combination of inputs and design states. It starts from initial state (typically after reset) and checks behavior for any combination of input signals. Then goes from the set of reached states and checks any combinations of inputs again, etc. That is why each next step (clock cycle) proof could take same amount of time as for all previous steps together leading to exponential grow of proof time depending on proof depth (number of clock cycles). It makes sometime non-realistic to get full proof or reach required proof depth in reasonable time. There are several approaches how to fight with that by reducing proof complexity. Some of them are described below.

A. Use of symbolic variables

Basic idea: we are tracking and checking only one arbitrary instance of design or sequence item selected by symbolic variable. It could significantly reduce proof time. Note that we are not extra restricting input space but allowing any legal input sequences.

For example, if we have several input ports and several output ports we could track only transactions which go from some arbitrary selected input to some arbitrary selected output port. Selected variable values should be unchanged through the proof.

Symbolic variables also could be used to select one arbitrary bit inside byte, for example:

```
wire [2:0] sym_data_bit;
sym_data_bit_stable: assert property (
    ##1 (sym_data_bit == $past(sym_data_bit)
);
```

B. Selection of sequence item with Floating Pulse method

Here we are tracking and checking only one sequence item, it could be byte, it could be packet or another transaction. Floating pulse is binary signal which is constrained to be active during any but only one cycle. It selects the sequence item.

Item which is going to be tracked is typically marked (colored) at the input – so we should be able to detect it at the output. One bit could be used for coloring data byte. That bit has inactive (typically 0) value for all bytes except colored byte when the bit has active (typically 1) value. It is done via constraints (SVA assumptions).

Here is illustration of application of floating pulse method for Delimiter Removal Module.

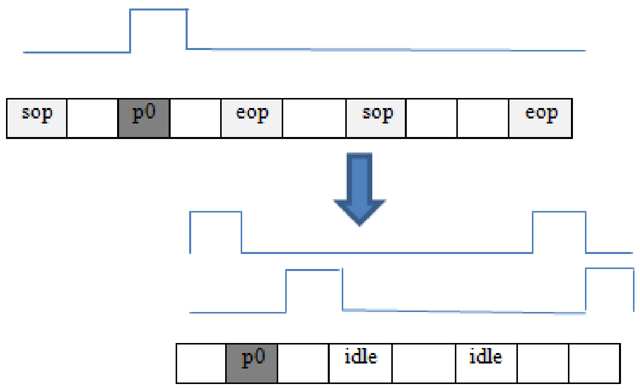

Figure 3. Floating Pulse method application

Example of code for generation of floating pulse:

```
wire floating_pulse;
reg floating_pulse_reg;
always @ (posedge clk) begin
   if (reset)
```

```
        floating_pulse_reg <= 1'b0;
    else if (floating_pulse)
        floating_pulse_reg <= 1'b1;
floating_pulse_model: assert property (
    floating_pulse_reg |-> !floating_pulse
);
```

Example of code for coloring bit 0 inside input byte:

```
color_bit: assert property(
    @(posedge clk) disable iff (reset)
    data_bus[0] == floating_pulse
);
```

In the discussed Formal environment Floating Pulse method used in the following checkers:

1) Check that any valid input byte after some number of cycles is detected at the output (Forward Progress Checker): additional counter is used which counts valid cycles after floating pulse before colored byte is detected at the output; counter value is compared with pipeline depth via assertion;

2) Check that packet boundary is preserved at the output: at the input we arbitrary color first byte of a packet and then check that colored byte is still the first at the output

Another method described below is used to check byte contents and bytes order.

C. *Using Wolper method to check data contents and order*

Wolper method colors two consecutive items at the input and checks that only two consecutive items are colored at the output. For completeness both 0 and 1 should be used for coloring. This method allows to prove that data contents and order is preserved, no drops, no injections.

It is used in the discussed environment for that purpose.

Complication here though for Delimiter Removal Module is that it should maintain order of bytes within certain packet priority but packets of low priority could be interrupted by higher priority packet. That is why two consecutive bytes of low priority packets could be apart. It requires additional proof depth and extra care. So it makes difficult to reach required proof in reasonable time. Case splitting approach described below is used to resolve it.

D. *Splitting Checkers*

Idea behind that technique is to reduce complexity – Cone of Influence (COI) of each checker. It could be done different ways:

- By slitting property when complex expression is used on the right side of implication;
- By defining separate checkers for different functional aspects, for example, different operations;
- By defining separate checkers for different modes;
- Using Assume-Guarantee approach with selecting internal points and defining checkers to and from internal points

In the discussed Formal environment we had to fix values of some symbolic variables used along with Wolper method and run Formal proof for various values separately. For example position of bit inside byte was fixed to certain value, position of byte inside input bus was fixed. Note that in some cases after design analysis and discussion with designer, conclusion could be made that it is enough to make proof only for some corner case values of symbolic variables, for example first and last bit position.

In the following couple of sections we are discussing aspects of design abstraction models restricting them to few typical examples.

E. *Reset Abstraction Model*

It is difficult or sometimes not possible to reach large proof depth in reasonable time. On the other hand we need to ensure that design is working properly from any state even if it requires long sequence to reach that state.

For example, design has wide counter and action happened when counter reaches some big value. If we start at reset state then big number of cycles is required to exercise the action.

The idea behind Reset Abstraction is to make action state closer. To make it happened instead of getting initial value at reset we are allowing any value. So counter could get high value right at reset and action could be verified. Special care might need to be taken to sync other signals with arbitrary reset value.

Here is example from the discussed Formal environment. Here we have packet length counter which is reset for every new packet and if packet count reaches certain configurable value, packet should be marked with error and truncated. Reset abstraction leaves value at reset not driven but keep all other functionality not touched. This is illustrated in the next figure.

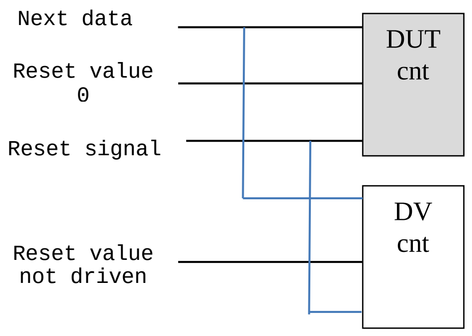

Figure 4. Example of Counter Reset Abstraction

F. *Memory & FIFO Abstraction Models*

Other typical examples of abstraction models which are used to reduce proof complexity are Memory & FIFO Abstraction models.

Example of Memory Abstraction Model:

- Maintains data only for one symbolic address : writes and reads data when actual address matches the symbolic one;

- Returns random data for all other addresses.

Examples of FIFO Abstraction Model:

- Use symbolic "any" depth of FIFO: it will allow to reach full condition earlier;
- If depth cannot be compromised and made symbolic, then use reset abstraction for read and write pointers and sync other signals with them;
- Keep track of only one data entry selected by floating pulse or by colored bit inside the data

V. Formal Methodology Aspects

A. Documents & Test Plan Flow

Formal Verification is less standardized than simulation. That is why having good set of documentation is especially important for knowledge transfer.

We developed some number of documents for the Formal flow support including:

- Formal Verification Process Description which describes phases and milestones from Formal Planning to Formal Complete, focus and exit criteria for each phase including reviews and approvals of checklists;
- Formal Milestone Checklist for different milestones to ensure that nothing is missed; examples of checklist items: "Reviewed Required Proof Depth with designer", "All tests in Formal regression passed";
- Formal Environment Template, including description of Interface Components; End-to-end Checkers, Test bench Configurations, Interesting scenarios, etc.
- Test Plan template.

Test Plan is supported by in-house tool and shared with simulation. In the Test Plan some intends could be covered by simulation and some intends could be covered by Formal verification methods. Test plan metrics contain references to actual code: for Formal proof metrics we require reference not just to check assertion but also to coverage assertion.

B. Run Flow, Regressions & Scripting

One could use just Formal Tool to run Formal tests especially at initial stages with GUI. However at some point typically you need to run Formal tool for different constraints, parameters, etc. Eventually you will have regression list which will be run in batch mode.

In-house script simplifies passing tests to Formal tool, setting signals and parameters, selecting assertions. The same script is used for running simulation as well.

From our experience it is difficult to get full proof for end-to-end checkers but bounded proof for certain number of cycles could be enough. That is why it is important to analyze design and calculate Required Proof Depth (RPD) taking into account:

- Pipeline depth / Latency of the design;
- Results of Microarchitecture Analysis with and without designer;
- Length of input sequences which require proof including sequences for interesting corner case scenarios.

If Formal proof reaches RPD when proving checker or coverage property, we could consider that property is proven; if all properties are proven then the test passes. Support for defining and checking RPD is part of the flow.

We have regression support shared with simulation which allows to run set of tests with various combinations of configuration values when needed for case splitting.

C. Coverage Flow

Formal covers all possible combinations of input stimulus and states inside Cone Of Influence (COI) of particular checker assertion but we need to ensure that all design constructs are covered, there is no over-constraints, there is enough checkers, so they are able to catch DUT bugs.

We used the following techniques / rules to get confidence that DUT is formally verified:

1) Interesting scenarios

Coverage properties should be implemented to ensure that "interesting" corner case scenarios could be reached within RPD. It is recommended to include check for output results and ensure that design comes to idle state. Here is some example below:

```
intscen_interleave_cover: cover property (
// input sequence
(Srdy && S_intf.sop && (S_intf.pri==PRI_MC))##1
(Srdy && S_intf.sop && (S_intf.pri==PRI_UC))##1
 . . .
##[1:PIPE_DELAY]
// output results & state
((uc_cnt==2) && (mc_cnt==2)&& D_intf.idle)
);
```

2) Coverage for each group of checkers

Each group of checker assertions should be accompanied by coverage assertions to ensure hitting corner case scenarios

3) Controllability Input Stimulus coverage.

Main purpose here is to ensure that there is no over-constraints which do not allow to cover certain parts of design. The following flow support was implemented:

- Collect Formal coverage including line, FSM, toggle coverage:
  - For Formal environment with disabled constraints;
  - For regular Formal environment with constraints;
- Compare the coverage results to ensure there is no over-constraints.

4) Observability Checkers coverage.

This step is to ensure that all parts of design are actually checked by at least one checker. The step is more dependent than the other steps on Formal Tool which could help to collect such information.

Some tools could collect Proof-Core coverage for a checker which covers constructs really affecting particular checker results. The Proof-Core coverage per checker is merged for DUT.

Another approach which could be used along or without Proof-Core coverage and we applied is mutation coverage. It requires more time but gives more confidence. It could be implemented outside of Formal Tool. We use in-house mutation run script with the following base algorithm:

- Do the following for every flip-flop in the design (for data buses it could be restricted to only LSB and MSB bits):
    - Inject bug into design; We used constant 0 for the flip-flop;
    - Do Formal proof;
    - Ensure that some checker fails.

It could be easily implemented, fast enough, gives extra confidence for Formal and much more than regular code coverage for simulation.

D. Reuse

Reuse is important part of any flow. We have some reuse components in place and some to be implemented including:

- Components instantiated inside Formal Environment:
    - FIFO,
    - Pipeline / Delay component,
    - Components supporting Formal methods: Floating Pulse, Wolper;
- Deign abstraction models:
    - Memory,
    - FIFO.

VI. Conclusions

Formal Verification could be successfully used along with simulation in various applications.

We got the most benefits using end-to-end Formal Verification for design modules of decent size with big number of combinations of packet flow parameters and corner cases. It allowed to find corner case bugs & increase verification quality.

In order to reach results we had to establish consistent Formal Verification flow, including coverage flow and apply techniques which help to fight with FV complexity.

Acknowledgements

Author thanks OSKI Technology team who delivered training on Formal Verification and helped to establish FV flow and his colleagues from Cisco Systems who worked on development of the flow.

References

[1] Oski Technology. Formal Verification Methodology to enable Formal Sign-off. Available at http://www.oskitechnology.com (accessed 03.05.2018)

[2] Singhal V. The evolution of Formal Verification Sign-off. Available at https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/company/Events/jug/secured/jug-2017/wed-9-30am-fv-signoff-evolution-oski.pdf (accessed 03.05.2018)

[3] Sokhatski A.A. Practical Aspects of Design Verification of Complex Chips // Problems of Perspective Micro- and Nanoelectronic Systems Development - 2016. Proceedings / edited by A. Stempkovsky, Moscow, IPPM RAS, 2016. Part2. P. 16-23.

[4] Seligman E., Schubert T., Kumar A.K. Formal Verification: An Essential Toolkit for Modern VLSI Design. Waltham, MA, USA: Elsevier, 2015, 352P.

[5] Murphy B., Pandey M., Safarpour S., Finding Your Way Through Formal Verification. Danville, CA, USA: SemiWiki LLC, 2018, 133P

[6] Tatarnikov Y., Labib K. Using Synopsys VC Formal Coverage Analyser (FCA) for Code Coverage Improvement. Available at https://www.synopsys.com/community/snug/snug-silicon-valley/location-proceedings-2016.html (accessed 03.05.2018)

[7] Tatarnikov Y., Labib K. Next step of Formal Verification utilization Available at https://www.synopsys.com/community/snug/snug-silicon-valley/location-proceedings-2018.html (accessed 03.05.2018)

УДК 519.714

Практические аспекты формальной верификации проектов блоков сетевых СБИС

А.А. Сохацкий

Сиско Системс Инк., asokhats@cisco.com

Аннотация — **Формальная верификация в наши дни становится важной составляющей верификации проектов цифровых блоков в особенности в областях, предъявляющих повышенные требования к качеству проверки. Так сетевые СБИС должны фунционировать безошибочно без перерывов в течении длительного времени, при том, что перепроектирование и повторное изготовление этих коплексных микросхем требует существенных затрат. Довольно сложно проверить функционирование на всех возможных входных последовательностях путем моделирования при наличии множества граничных условий связанных с различной длиной и выравниванием пакетов, комбинациями значений входных настроечных портов и регистров. Формальные методы должны помочь достичь полноту проверки и сократить время разработки, но это требует:**

- **правильной стратегии выбора блоков и модулей проекта для формальной верификации, а также метода формальной верификации;**
- **применения решений для ответа на проблему экспотенциального роста времени формального доказательства в зависимости от глубины доказательства;**
- **последовательной методологии для обеспечения верификационного покрытия и сокращения затрат.**

В статье рассматриваются вопросы формальной верификации на основе опыта работы автора в компании Сиско Системс Инк. и консультаций поставщика услуг формальной верификации компании OSKI Technology. Излагаются следующие аспекты:

1) **Краткий обзор применений формальной верификации. При этом статья фокусируется на задаче полной (sign-off) сквозной проверки (end-to-end check) отдельных модулей проекта. Рассатриваются вопросы выбора модулей для формальной веривикации;**
2) **Структура простого окружения для формальной верификации;**
3) **Методы, позволяющие увеличить глубину доказательства, в частности, использование символических переменных, символического выбора элемента последовательности с применением плавающего импульса (floating pulse), метод Волпера (Wolper), использование абстрактных моделей;**
4) **Аспекты методологии формальной верификации, включая технологии планирования, документирования, исполнения и регрессионных последовательностей, покрытия проекта, совместного использования для формальной верификации и моделирования**

Ключевые слова — **СБИС, формальная верификация, моделирование, RTL, SystemVerilog, SVA.**

ЛИТЕРАТУРА

[1] Oski Technology. Formal Verification Methodology to enable Formal Sign-off. Available at http://www.oskitechnology.com (accessed 03.05.2018)

[2] Singhal V. The evolution of Formal Verification Sign-off. Available at https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/company/Events/jug/secured/jug-2017/wed-9-30am-fv-signoff-evolution-oski.pdf (accessed 03.05.2018)

[3] Сохацкий А.А. Практические аспекты верификации проектов СБИС // Проблемы разработки перспективных микро- и наноэлектронных систем (МЭС). 2016. №2. С. 16-23.

[4] Seligman E., Schubert T., Kumar A.K. Formal Verification: An Essential Toolkit for Modern VLSI Design. Waltham, MA, USA: Elsevier, 2015, 352P.

[5] Murphy B., Pandey M., Safarpour S., Finding Your Way Through Formal Verification. Danville, CA, USA: SemiWiki LLC, 2018, 133P

[6] Tatarnikov Y., Labib K. Using Synopsys VC Formal Coverage Analyser (FCA) for Code Coverage Improvement. Available at https://www.synopsys.com/community/snug/snug-silicon-valley/location-proceedings-2016.html (accessed 03.05.2018)

[7] Tatarnikov Y., Labib K. Next step of Formal Verification utilization Available at https://www.synopsys.com/community/snug/snug-silicon-valley/location-proceedings-2018.html (accessed 03.05.2018).