Never too late with formal: Stepwise guide for applying formal verification in post-silicon phase to avoid re-spins

Anshul Jain, Aarti Gupta, Achutha KiranKumar V M, Bindumadhava Ss, Shivakumar S Kolar, Siva Gadey NV
Intel Corporation
{anshul.jain, aarti.gupta, achutha.kirankumar.v.m, bindumadhava.ss, shivakumar.s.kolar, siva.gadey.nv}@intel.com

***Abstract*- Formal Verification (FV) is a proven efficient technique to exhaustively verify complex hardware designs. While FV has been a preferred pre silicon verification tool, it also is highly effective in assisting post-silicon debug and bringing in additional confidence for the bug fixes. Enabling formal verification in post-silicon phase poses a different set of challenges than the traditional pre-silicon mode. In this paper, we present a stepwise guide to apply formal verification in post-silicon phase to reproduce post silicon failures in pre-silicon, validate the bug fixes, build a comprehensive solution to uncover additional vulnerabilities, thereby guaranteeing higher confidence in the design. We also discuss two case studies that illustrate the effectiveness of the methodology on a successful re-spin.**

I. Introduction

The ever-increasing complexity of modern-day IPs and SOCs has increased the challenges of verification engineers working on these designs. Among various pre-silicon verification technologies, formal verification (FV) has gained wide acceptance as a reliable, fast, and efficient mechanism to achieve design verification left shift.

Though the industry is embracing formal verification across breadth of designs, its usage is limited by multiple factors such as lack of expertise, complexity of the design etc. Hence majority of the designs are still dependent on traditional dynamic simulations for functional sign-off. Since dynamic simulation depends on the test-stimuli, there may be cases where some design issues and inconsistencies fall through the cracks and make their way into the into the silicon due to inexhaustive coverage, resulting in post-silicon bugs. Post-silicon bugs usually require a quick and accurate root cause analysis for a high-quality signoff on the bug fixes. Traditionally, dynamic simulation has been the go-to methodology for post-silicon debugs using existing environments with the belief that it would offer a quick turnaround for the resolution. Formal verification is usually not considered for post-silicon bug reproductions because of following common presumptions:

Facts		Myths
Post-silicon issues are isolated at huge block-boundaries due to limited observation points available for debug in silicon	1	Formal property verification will not be manageable at such big design scopes because of formal complexity
Formal property verification is usually not run on large boundaries during pre-silicon, stable environment is not available	2	It will take unreasonably long time to bring-up formal property verification environment from scratch
Not all signals can be probed during post-silicon verification. Only a general high-level behavior leading to failure is observed	3	Formal property verification will need intricate signal-level details for bug reproduction

These beliefs are not totally unfounded, and admittedly, like any other verification methodology formal verification has its own limitations. Despite the limitations, meticulous application of formal property verification in post-silicon debug is crucial to shorten the post-silicon debug cycle and thereby time-to-market (TTM), thereby providing 100% confidence on the fix quality [1][2][9]

In this paper, we aim to share a comprehensive methodology for application of formal property verification on post-silicon bugs, ranging from bug reproduction, bug fix validation, to extending scope of formal checks beyond the spatial locality to find remaining bugs. We show the data and results of using our methodology through a case study busting the myth that formal verification is inapplicable to post-silicon debug, thereby promoting inclusion of formal methods in post-silicon root-cause analysis, and potentially saving precious debugging time.

II. Post-Silicon Formal Property Verification Methodology – **UNEARTH**

We propose the following stepwise approach to enable quick and accurate debug of post-silicon design issues as well as build a verification environment that is capable enough to evaluate the effectiveness of workarounds and robustness of bug-fixes. We name our stepwise approach UNEARTH, where each letter represents a step in the methodology.

*A. **U**nderstand the problem & collect all the collaterals*

This step involves collecting all the essential details about the post-silicon design issue. Such details are instrumental in getting a head-start in the right direction. These details will also be helpful during each step of the execution in the overall quest of finding the root-cause of post-silicon design issue.

1. *Description of the failure/problem seen in post-silicon testing.* Each post-silicon issue comes with its own panic and paranoia. Hence it is common to have multiple descriptions of the issue from various stakeholders, both design and validation side. It is important to document all such descriptions and extract vital information such as effect of the failure (e.g., data corruption, deadlock, livelock) sequence of events at different design interfaces (e. g., dependent read request for an address on which a partial write request is already in progress, CRC error detection/correction in retry message causing multiple iterations, non-posted message interleaved with a posted message when buffer was almost full).
2. *Design documentation.* Collect relevant specification and implementation documents to understand micro-architectural details of the design such as primary/auxiliary interfaces, modes of operation, flow of execution/transactions, etc. Such documents help climb the steep learning-curve through block-diagrams, reduce dependencies among stakeholders and provide a fresh perspective to the details.
3. *Design files.* Source the design files from that version of the repository which made it to the silicon. It is a common pitfall to start reproducing the post-silicon issue using design files available on the top of trunk.
4. *Waveform and register dumps.* Access the information dump from post-silicon test failure such as the values of configuration registers, error-logging registers, debug hooks, etc. Sometimes, a particular manifestation of the post-silicon issue might be reproduced in dynamic simulations while formal verification environment is still in works. The simulation waveform will help visualizing the issue better. Finding a bug manifestation in simulation should not call off the formal verification effort because there could be more scenarios of bug manifestation that simulation has not found yet.
5. *Sample waveforms from pre-silicon dynamic simulations.* Though the design documentation would generally include timing diagrams, however sample simulation waveforms from directed tests prove to be helpful while debugging failures seen in formal environments. Sample waveforms can be used to brainstorm some over-constraints and bypass formal complexity.
6. *Existing formal verification environments.* Locate all the existing formal verification collateral available. It could be a basic environment to bring-up design files in a formal tool, an assertion-based verification IP for standard interfaces or full-fledged formal sign-off environment.

*B. **N**ail-down the formal property*

Once the manifestation and the scenario of the failure are well understood, shift the focus on thinking about the properties (from the specification document) of the design that could be violated in the post-silicon failure. Start the process by brainstorming general properties such as “each request from the host should be acknowledged by the device eventually”, “device should send the responses in the same order as requests were received from the host” and move towards more specific properties such as “buffer should never become full and should have one entry always free for pointer management”, “dispatcher should not preempt a request more than n times”. Depending on the type of failure and the level of details we have gathered so far, we can come-up with multiple such properties that are worth checking. Write all such properties in plain, simple, human-readable language without worrying about the implementation details. Once we have succinct definition of any property, we can engage with the micro-architects to identify the observation points for the property and implement it using light-weight instrumentation code and SVA [3][9].

*C. **E**tch the design boundary for formal search*

Post-silicon issues are mostly isolated at sub-system boundaries such as memory controllers, physical layer, L2 cache etc. Bringing-up even a basic formal property verification environment at such sub-systems would typically need weeks of efforts and many complexity resolution techniques to reach formal sign-off. Therefore, we need a better approach in selecting the design under test (DUT) for locating the source of bug than selecting the entire sub-system as our DUT. Apart from general recommendations of selecting a design scope with minimal area (for minimizing formal complexity), minimal perimeter (for minimizing constraint modeling), and standard interfaces (for minimizing environment modeling by using assertion-based verification IPs), we recommend picking a block boundary at which we have all the interface signals to observe the validity of the properties brainstormed in previous step. Assuming we have identified correct properties, this approach would yield counterexamples under three categories shown in figure 1. Feedback from these counterexamples would guide us in selecting/expanding our design scope to root-cause the post-silicon issue. Starting with a smaller design scope is safe against false proofs and manageable as we can increase the design scope by integrating the neighboring blocks in the DUT without re-modeling the whole environment.

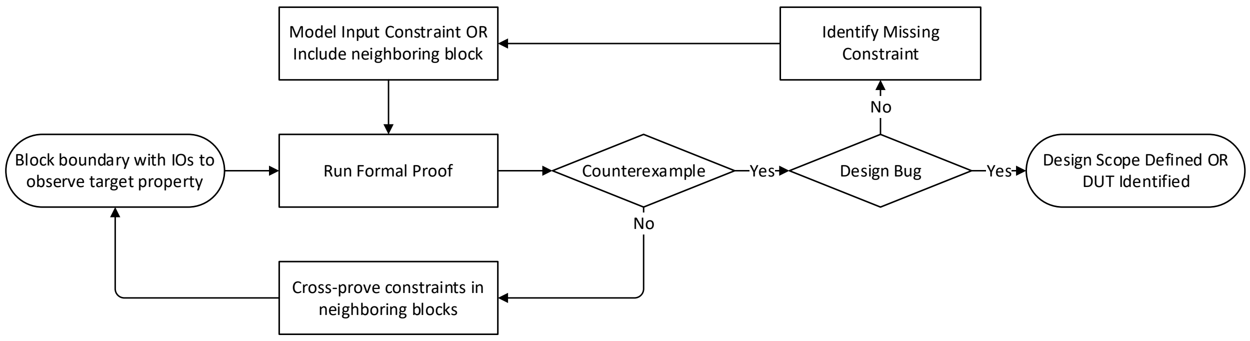

Model Input Constraint OR Include neighboring block
Identify Missing Constraint
No
Block boundary with IOs to observe target property
Run Formal Proof
Counterexample
Yes
Design Bug
Yes
Design Scope Defined OR DUT Identified
No
Cross-prove constraints in neighboring blocks

Figure 1: Design Scope Definition

D. Assess reachability by covering your way to the source of bug

Most of the post-silicon issues are extreme corner cases that require some highly unlikely sequence of events happening in a specific order and proximity. Though formal technology is highly effective in finding such corner-cases because of its breadth-first, exhaustive search of the state space, these corner cases could be hard to reach due to the formal complexity. Depending on the size (number of gates, flops, latches) of the design and the cone-of-influence (COI) of the formal property, we often run into situations where the formal tool gives-up before reaching those states of the design where the property would not hold true. Such "bad" states could be deep in the state transition graph. One of the most effective way to guide bug-hunting formal engines to reach such deep design states is to implement spot covers for the intermediate events for the manifestation of post-silicon issue and functional covers to witness the exact sequence of events that drove the design into "bad" state. We recommend using a simple FSM to model the sequence of events instead of complex SVA sequences. Another class of effective functional covers are the ones that generates witness for a particular "warm" state of the design at the time of failure. For example, an exact state of the linked list (head pointer, tail pointer, node links) or an exact combination of multiple state machines. Covers not only helps the formal engines to work more effectively, but they also help build confidence in the direction of our efforts and provide an estimate of depth at which the failure should be found. Suppose we know that the bug manifests mostly when an internal FSM transitions to certain states (say "STOMPED" & "DRAIN") after an internal linked list has been populated with a certain number of nodes (say N). We should use this information to our advantage by embedding covers such as C1 (length of linked list = N), C2 (FSM state = STOMPED) and C3 (FSM state = DRAIN). Such covers enable the tool to use its SAT-based constraint random engines [10] to generate witness traces for deep states quickly and reposition the locus of its bug-hunting bounded model checking engines [11] from reset states to interesting "warm" states.

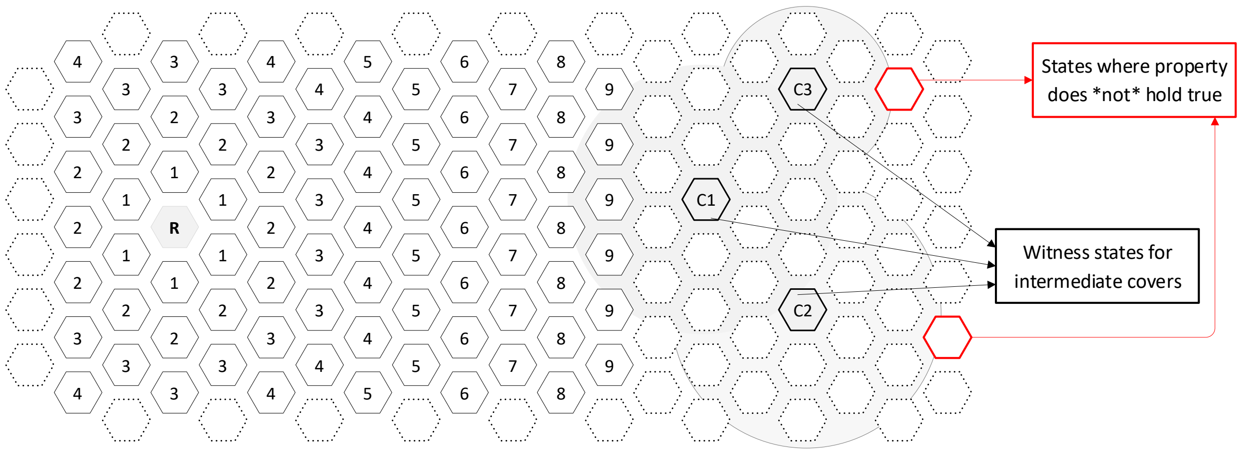

R
C1
C2
C3
States where property does *not* hold true
Witness states for intermediate covers

Figure 2: State Space Search with Covers

*E. **R**egulate constraints to strike a balance between over-constraints and under-constraints*

Keeping the formal environment under-constrained is a recommended approach to reach pre-silicon verification sign-off [9]. However, both over-constraints and under-constraints play an important role in post-silicon formal verification effort. Since we would be running only a small set of properties to root-cause the post-silicon issue, we do not need accurate constrained interfaces of the design. For example, if we suspect the design to be sending messages out of order, then we may not need to perfectly constrain the initialization interface, as the formal tool may figure out the correct initialization sequence to bring the design to a state where it can accept the messages and send them on the output out-of-order. On the other hand, we can narrow-down the formal search by Constraining the configuration registers to values observed in the failing silicon test. This technique can be improvised by carefully scaling down some of the configuration register values to bring deep states closer to reset states. For example, reducing the preemption threshold value to trigger preemption more often or block entries of a linked list to reach full state with lesser transactions. Formal search can be made more directed by disabling traffic on interfaces that is not expected to in the post-silicon scenario.

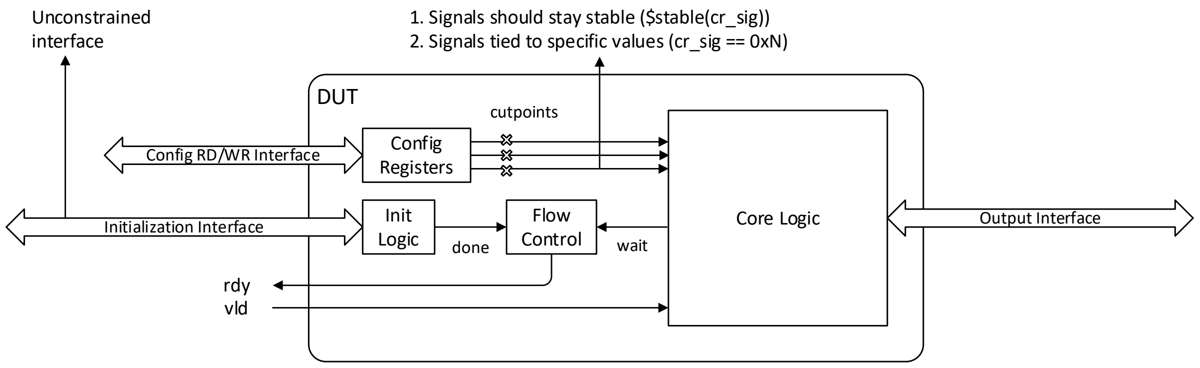


Figure 3: Over-constraints and under-constraints

We recommend the constraint strategy explained in figure 4 to reap the benefits of both under-constraints and over-constraints for a faster reproduction of the post-silicon issue. Over-constraints will help narrow the scope of the formal tool search and allows the tool to spend focused efforts on failing scenario. Under-constraints will eliminate the chances of accidental masking of real design issues due to lack of stimuli. Bring-up the DUT without any constraints, setup the clocks and reset, and witness the logic is alive by running basic sanity covers at the outputs of interest. Then over-constraint non-participating interfaces such as scan, train, auxiliary channel etc. and ensure the design is still able to toggle the outputs of interest. Switch-back to under-constrained approach of adding or refining the constraints based on the counterexample produced by formal for the target property.

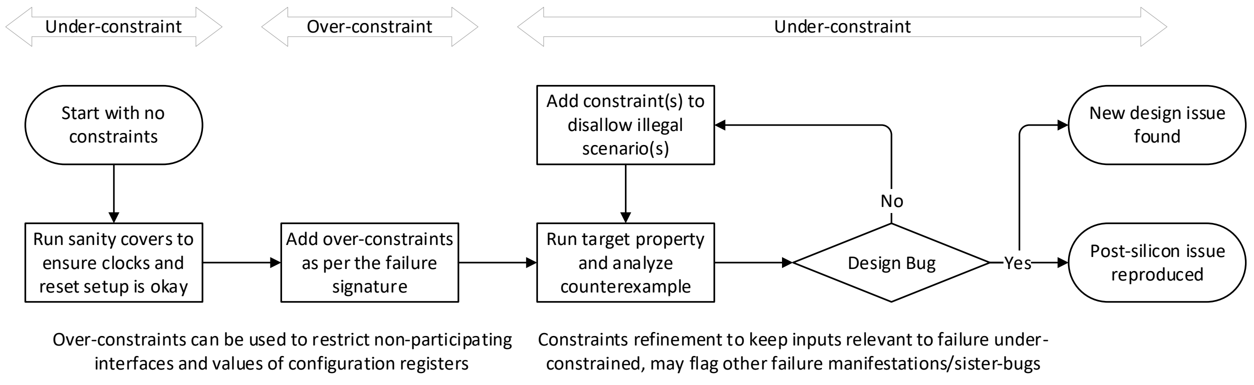

Under-constraint
Over-constraint
Under-constraint
Start with no constraints
Run sanity covers to ensure clocks and reset setup is okay
Add over-constraints as per the failure signature
Add constraint(s) to disallow illegal scenario(s)
Run target property and analyze counterexample
No
Design Bug
Yes
New design issue found
Post-silicon issue reproduced
Over-constraints can be used to restrict non-participating interfaces and values of configuration registers
Constraints refinement to keep inputs relevant to failure under-constrained, may flag other failure manifestations/sister-bugs

Figure 4: Constraints Strategy

*F. **T**ap the details from sample simulation waves to start the formal search from a "warm" state*

Some of the large design scopes may require long and specific sequence of standard events before it could process the functional traffic. Common examples of such sequences are cold boot sequence, warm reset sequence, credit initialization sequence, frequency sync sequence, etc. In pre-silicon, reset abstractions prove to be helpful in abstracting out such time-consuming, un-interesting sequences to start from a design state that can consume functional traffic. Reset abstraction may require certain refinements and can become an iterative process spanning over days and weeks of effort. In post-silicon phase, it is better to bypass long standard sequences more deterministically than reset abstraction. Simulation waves serve as excellent starting point for formal search from a "warm" state. All we need is to generate a simulation wave, identify the timestamp where the design comes to a stable state after all the boot, reset, initialization sequences, load the simulation wave in formal tool, specify the timestamp of interest and start formal proofs.

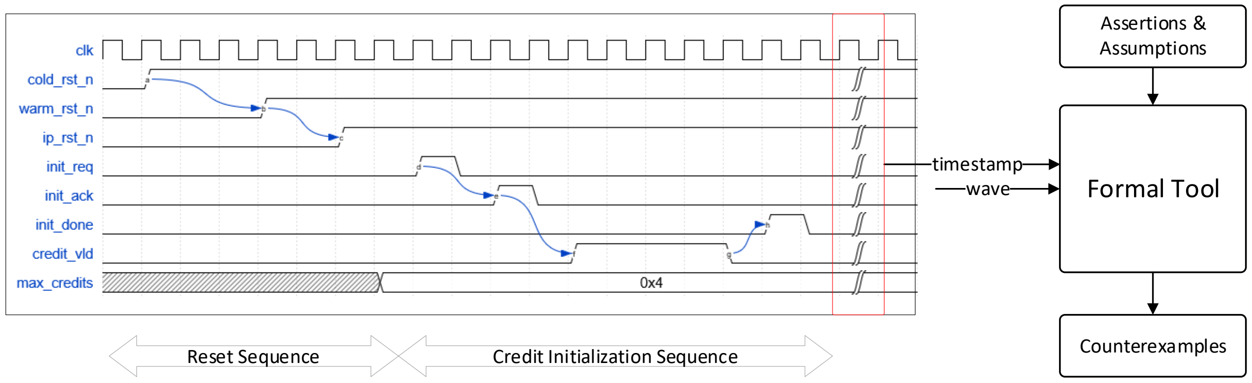


Figure 5: Loading reset state from simulation wave to formal tool

G. *Harness full potential of formal technology*

While all the aforementioned steps propose a faster path to root-cause the post-silicon issue, they should be executed with a greater goal in mind. After root-causing the post-silicon issue, same environment should be augmented to evaluate workarounds, verify the bug-fixes and achieve formal sign-off. After successful reproduction of post-silicon issue and all its possible manifestations, it is strongly advisable to employ bug-hunting, abstraction models, proof accelerating techniques to explore more bugs in neighboring logic. Formal verification environment developed in the post-silicon phase of current version of the IP/SOC should be the collateral to verify other features and updates in future versions of the IP/SOC.

III. Case Study – 1: Post-Silicon Bug in Bridge

A. *Problem Statement*

A configuration register data corruption bug was found in post-silicon validation. Though first observation point of the problem was at the interface between channel and mainband endpoint, post-silicon analysis indicated that the problem originated from the bridge. Bridge connects register-access endpoints to the mainband endpoint, and the sideband endpoint as shown in the Figure 6. The post-silicon failure also indicated that the bridge injected fake responses to the mainband endpoint while handling register write requests and register read responses with some magic numbers in the payload, eventually corrupting the register data in the response pipeline of mainband endpoint. Since similar register data corruption issues were found in pre-silicon verification as well, therefore, design team was concerned about the robustness of their proposed bug-fix on the post-silicon bug and needed absolute guarantee of the fix.

B. *Design Details*

Design implements the registers using register endpoints connected on a ring and are accessed by mainband endpoint and sideband endpoints. Bridge routes register access requests (and dummy responses) from the mainband endpoint and the sideband endpoint to register endpoint 0. Only one request each from the mainband endpoint and sideband endpoint can be outstanding at any point of time. Request and responses are configured as messages containing multiple flits with end-of-message (EOM) flag to indicate the completion of a message. Each message consists of header fields to decipher source, destination, opcode and data based on the message type. Register access request (and dummy responses) traverse the ring and come back to the bridge through register endpoint N. Request and responses always traverse as a pair inside the Ring. Bridge injects read request and a dummy read response for each register read access. While the read request comes back to the bridge as is, the dummy read response comes back as an actual read response with the data read from one of the registers in the ring. For every register write access, bridge injects write request and a dummy write response. Write request comes back to the bridge as is and the dummy write response comes back as an actual write response. Bridge discards the request received from ring and routes the response from the ring to the initiator of the corresponding initiator of the register access request. Routing of the response is done based on its source and destination information in the header of response message. If source and destination match to the mainband port id value, then it is sent to mainband endpoint. Otherwise, it is sent to sideband endpoint. If any double word (DW) of the packet matches to the MB source and destination, it is mistakenly identified as a completion and is sent as a fake completion (CMPL) to MB.

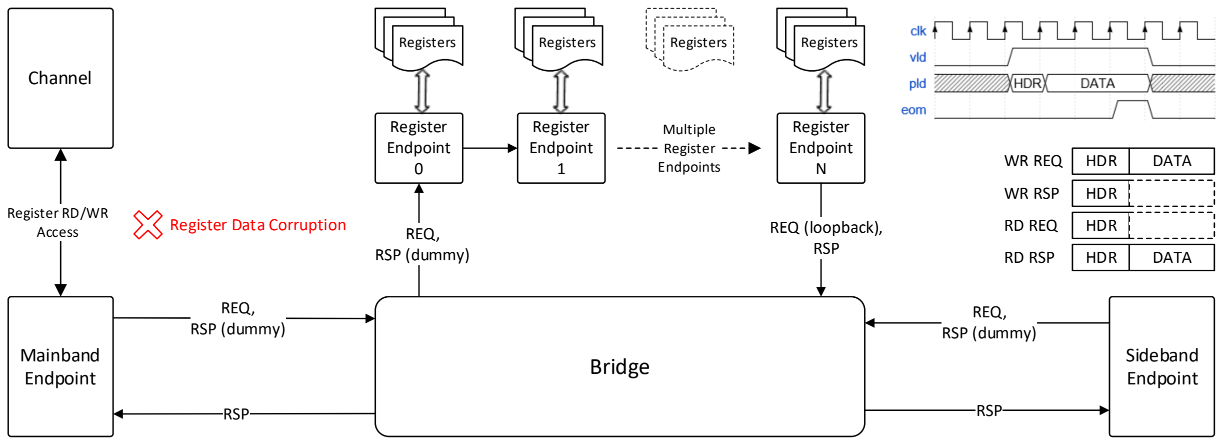


Figure 6: Bridge Block Diagram

C. Formal Verification Solution

Leveraging the debug analysis of the post-silicon validation team that the bridge seems to be sending fake response messages to mainband endpoint, we chose to implement the property on bridge and mainband endpoint interface to ensure "the bridge should send a response message to mainband endpoint only when both request and dummy response have been received from the mainband endpoint". The property was implemented using a simple FSM based reference model along with an SVA expression as shown in figure 7.

We chose the bridge and the ring of register endpoints to be the design under test. Formal interface models modelling the constraints for proper formation of request and response messages, credit management etc. were used to constrain bridge's interface with mainband endpoint and sideband endpoint. Entire ring was included in the DUT to reduce the modeling the constraints for the interface between the bridge and the ring. However, all the register endpoints combined increased the COI of the property and added long delays to process each request, hence posed convergence issues due to formal complexity. We chose to black-box all register endpoints except for the one directly interfacing with the bridge and used an abstraction model of the ring which introduced variable delays and randomized the data part of read response messages.

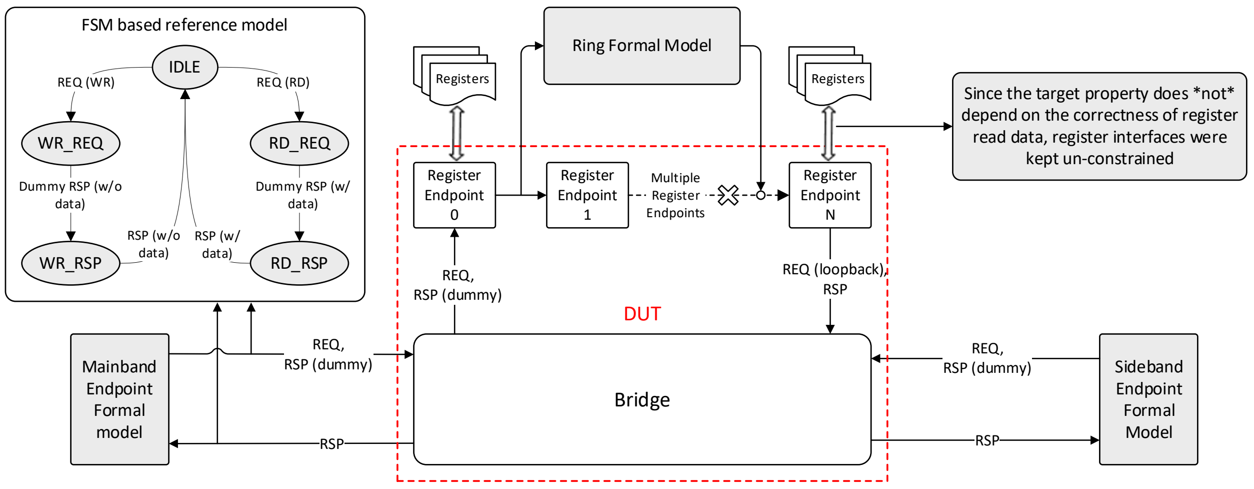

FSM based reference model
IDLE
REQ (WR)
REQ (RD)
WR_REQ
RD_REQ
Dummy RSP (w/o data)
Dummy RSP (w/ data)
RSP (w/o data)
RSP (w/ data)
WR_RSP
RD_RSP
Ring Formal Model
Registers
Register Endpoint 0
Register Endpoint 1
Multiple Register Endpoints
Register Endpoint N
Since the target property does *not* depend on the correctness of register read data, register interfaces were kept un-constrained
REQ, RSP (dummy)
REQ (loopback), RSP
DUT
Mainband Endpoint Formal model
Bridge
Sideband Endpoint Formal Model
RSP

Figure 7: Bridge Formal Verification Environment

```
SVA Property: (state==IDLE) || (state==WR_REQ) || (state==RD_REQ) |-> !b2m_vld
```

Both under-constraints and over-constraints were extensively used to save environment development as well as tool run time to reproduce the post-silicon issue fast. Apart from leaving register values completely un-constrained, many fields of the mainband endpoint and sideband endpoint requests were kept under-constrained. On the other hand, power optimization features such as block sleep/wake which would trigger clock gating, link re-initializations etc. were disabled (over-constrained).

Cover points added to visualize the flow of request and response message at various interfaces of the bridge gave additional confidence that we are treading the right track. Functional covers were implemented to witness request-response message pairs from mainband/sideband endpoints to bridge, bridge to register endpoint 0, register endpoint N to bridge and response messages from bridge to mainband/sideband endpoints. These cover points helped visualize the flow of messages through the DUT as well as helped estimate the minimum depth at which we should expect the SVA property to fail. After the bug fix for post-silicon issue was verified with the SVA property, more functional covers were added to witness multiple instances of the SVA property where it got triggered and passed in various scenarios.

D. Bug Reproduction & Root-cause

Response message routing mechanism in the bridge block had a logic bug. When bridge receives the request response message pair from the ring, it was supposed to discard the request message (opcode 0x01 or 0x02) and forward the response message (opcode 0x20) to the mainband endpoint if source and destination field of the header matches 0xQQQQ, else to the sideband endpoint. In other words, the bridge should forward the messages with their header matching 0xxx20_QQQQ to mainband endpoint and to the sideband endpoint otherwise. However, the bridge was matching 0xxx20_QQQQ for each DW irrespective of header/data DW. Therefore, whenever a data DW carried the magic pattern, bridge mistook it as the response message for mainband endpoint and caused the bridge to send fake response messages.

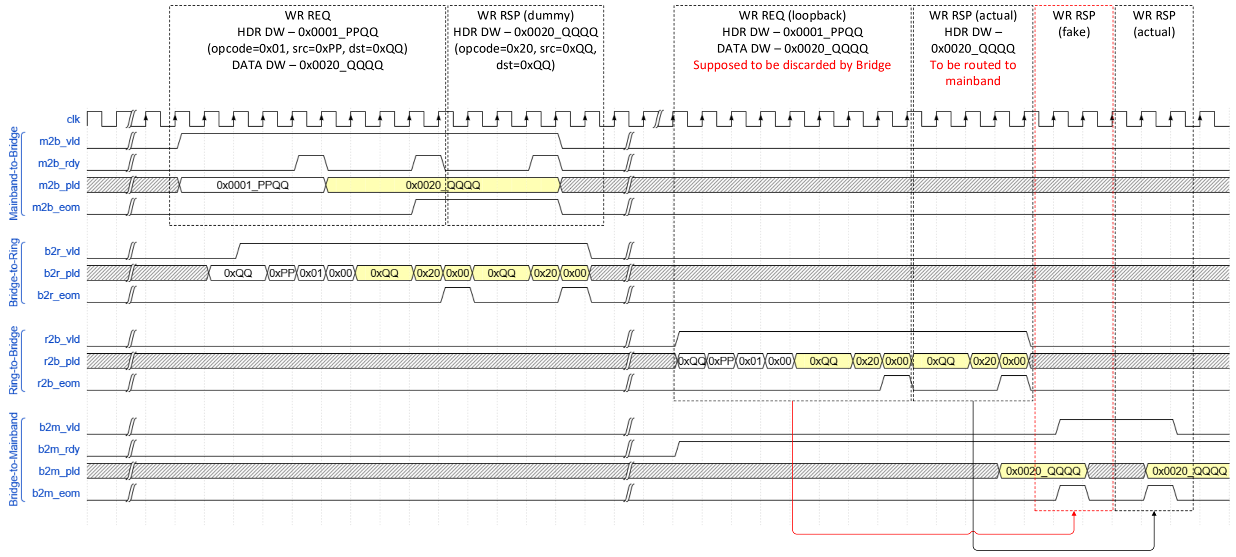


Figure 8: Bridge Post-Silicon Bug Scenario

E. Overall Results

Following the proposed methodology, we could reproduce the bug and root-cause the bug in a minimal 25K gate DUT. It took 3 person-days to setup the formal verification environment from scratch and reproduce the first manifestation of the bug. Design team requested to explore other manifestations of the bug that can aid in a robust fix implementation. Through some temporary over-constraints, formal verification environment produced five more bug scenarios involving traffic from both mainband and sideband endpoint. After bug reproduction, five person-days effort was spent to verify the bug-fix using the same property which reproduced the post-silicon issue. Formal verification environment was then enhanced to functionally sign-off the bridge block and a new issue was discovered in the response logic RTL implementation that evaded being discovered by all verification platforms. Furthermore, the same formal verification environment is being re-used to verify feature updates in two future SOCs and have so far found 8 functional bugs so far.

IV Case Study – 2: Post-Silicon Bug in Scheduler

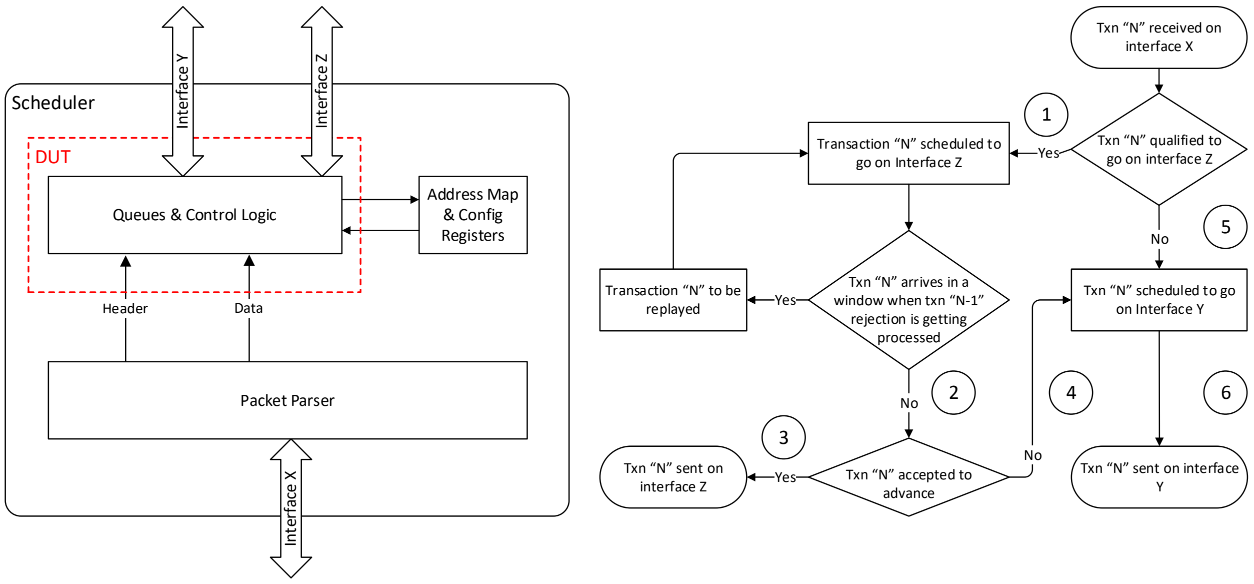
A. *Problem Statement*

A data corruption bug was found on the slicon and was speculated to be in one of the scheduler blocks. The scheduler had various pipelines to process transactions in parallel with a network of bypass and data forwarding logic to maintain consistency of metadata and the current state of a given transaction. The specific case of interest was a deep corner-case. A particular set of conditions had to happen to align the various stages of different pipelines in the scheduler to expose the bug. Bad data ended up getting forwarded causing an incorrect update of the metadata storage resulting in a data corruption

B. *Design Details*

The scheduler processes incoming transactions from interface X and sends it out on interface Y or interface Z depending on various conditions. The scheduler also does a protocol conversion when sending transactions from interface X to interface Y or Z as shown in figure 9. The control logic in the scheduler also had different pipelines to optimize transaction processing to meet the performance goals. Each transaction is pre-decoded and scheduled to go on different pipelines based on its target. There are different arbiters to ensure fairness across the various classes of traffic. Transaction is only processed on the availability of necessary resources in the consuming block.

Any new transaction that comes in from interface X is scheduled to either go on interface Y or Z. Once a transaction is scheduled to go on either interface, address is decoded to determine the destination and appropriate resource checks are performed. Unavailability of the destination resources would mandate the transaction to arbitrate again for the selected interface. Certain scenarios would redirect the transaction destined on interface Z to be not accepted by the next block and instead scheduled on interface Y. A subsequent transaction on interface Z may end up getting replayed back if the prior transaction hasn't made it out (to the next block) on interface Y.



Scheduler
Interface Y
Interface Z
DUT
Queues & Control Logic
Address Map & Config Registers
Header
Data
Packet Parser
Interface X
Txn "N" received on interface X
Txn "N" qualified to go on interface Z
Yes
Transaction "N" scheduled to go on Interface Z
1
No
5
Txn "N" scheduled to go on Interface Y
Transaction "N" to be replayed
Yes
Txn "N" arrives in a window when txn "N-1" rejection is getting processed
No
2
4
6
3
Txn "N" sent on interface Z
Yes
Txn "N" accepted to advance
No
Txn "N" sent on interface Y

Figure 9: Scheduler Block diagram and Transaction Flow

C. *Formal Verification Solution*

To reproduce the post-silicon issue using FV methodology, we followed the UNEARTH approach. First, to understand the problem better, we collected all the essential information. There was a parallel-effort on bug-reproduction by the dynamic simulation team where they succeeded in a probable reproduction of the issue by artificial injection of transactions and credit scenarios. However, bug-reproduction by means of FV was essential to get a realistic failure scenario and then the ability to check the fix. FV team studied the simulation trace to understand the necessary conditions that could lead to the bug. Studying the flow of transaction traffic in the scheduler block, six important temporal events (depicted in Figure 9 as circled numbers ①, ②, ③, ④, ⑤ and ⑥) were identified. It was understood that for proper transactional flow, following properties must be true at these events:

1. Property 1: Transactions received at flow-point ②, should be in same order as they were received at ①.
2. Property 2: Transactions received at flow-point ③, should be in same order as they were received at ②.
3. Property 3: Transactions received at flow-point ⑥ that were qualified to go interface Z and, should be in same order as they were received at ④.
4. Property 4: Transactions received at flow-point ⑥ that were not qualified to go interface Z and, should be in same order as they were received at ⑤.

On further investigating the post-silicon failure, a hypothesis was made that property 1 should be adequate to reproduce the failure. Thus, property 1 was targeted first, and property 2-4 verification activity was taken-up later aiming for full verification after the successful bug-reproduction.

Based on the scoping considerations detailed in Table 1, Scope 2 was selected after experimenting with Scope 1 where basic covers were taking too long to hit due to huge design size.

TABLE 1: DESIGN SCOPE COMPARISON

Scope 1: Red rectangle in Figure 9		**Scope 2: Scheduler**	
Pros	**Cons**	**Pros**	**Cons**
Standard interface present at one end which can be modeled using pre-existing property set.	Huge design size [~4M gates]	Smaller design size [~1.5M gates]	Internal interfaces need to be modeled using properties.

Owing to the design size, over-constraints were needed to optimize the efforts by FV engines and to reach the states of interest faster. Some examples of over-constraints applied:

1. *Configuration Register Values*: In the given module, the working mode was dependent on thousands of bits of configuration values. To narrow down the search, this value was made same as the one fetched from post-silicon dumps, and later more legal combinations were checked.

```
setConfigVal: assume property (ConfigVal == `config_val_in_PostSiDump);
```

2. *Number of Physical Input Channels*: Transactions arrive at input interface of the DUT via multiple physical channels. To reduce state space complexity for FV tool, only 2 channels were allowed to send transactions.

```
setNumChnls: assume property (DataInValid |-> TransactionChannel <=0);
```

The signals of interest like transactional data and signals marking transaction validity were left unconstrained. Reset abstractions were made on critical resource counters to allow a wider reset state and help in faster path to the failure.

Covers were needed to check the sanity of the formal verification environment and understand the need of more design abstractions. In addition to the covers for the events 1 to 6, properties were written on internal signals to see the reachability of a state where a new transaction conflicts with a replayed instruction. This cover reachability gave proof of bug existence, which was later confirmed by end-to-end checker (property 1).

The DUT required a long and complex initialization sequence to set many multi-dimensional arrays to a specific state. We could have modeled this sequence exactly the way it happens in the design, but it will exhaust FV engines through many un-interesting states before proving meaningful checks. Other option was to carefully abstract reset state, but that needs a lot of iterations and carry the risk of incorrectly constraining the environment. To speed up the work in the post-silicon debug, a simulation trace was used to load the FV environment to a valid state after the initialization sequence. Later, this dependency was removed, and reset was modeled by formal abstractions to cover a larger initial state space. After the bug reproduction and checking of the RTL fixes, properties 2-4 were added for full verification.

D. Bug reproduction & Root-cause

Checkers for all properties 1-4 can be implemented using Formal Scoreboards [6]. Scoreboards in verification domain are essentially means of verifying data integrity across two interfaces. Formal scoreboards do this verification smartly by just using symbolic sequences instead of traversing all combinations. There are some off-the-shelf formal scoreboard solutions available from FV EDA tool vendors. Also, they can be coded using Wolper Coloring Technique [7], Floating Pulse Technique [8] etc.

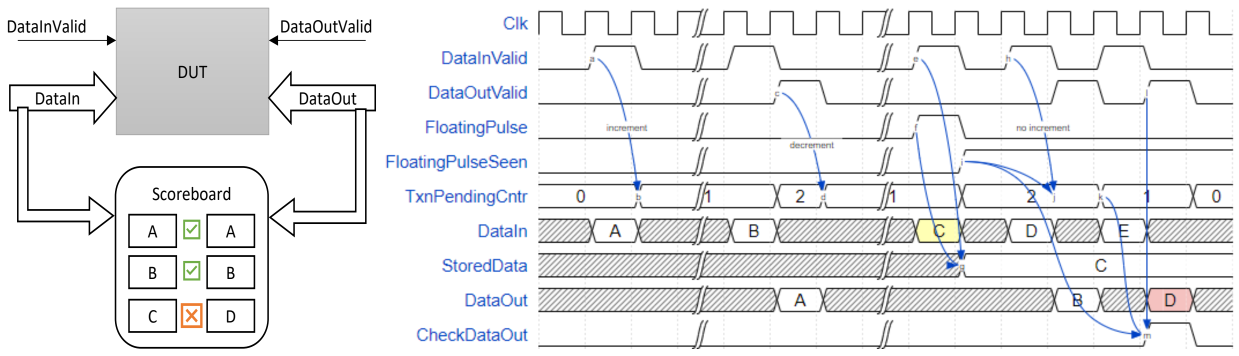


Figure 10: Floating Pulse Scoreboard

In this work, Floating Pulse Scoreboard was used. Its functionality is briefly explained in Figure 10. “FloatingPulse” is created by declaring an undriven signal that FV tool is free to drive to any value at any cycle with a constraint that it is a single cycle pulse. A pending transaction counter “TxnPendingCntr” is maintained that increments when a DataInValid is received and FloatingPulse is yet to come. This counter decrements when a DataOutValid is received. When a “FloatingPulse” is seen alongwith a DataInValid, the corresponding transaction data “DataIn” is stored in a register “StoredData” and increments on “TxnPendingCntr” are stopped. When “DataOutValid” comes while “TxnPendingCntr == 1”, check is initiated to compare “DataOut” with “StoredData”. Figure 10 shows a mismatching condition where this check fails.

```
wire FloatingPulse;
FloatingPulseOnce: assume property (FloatingPulseSeen |-> !FloatingPulse);
```

For verifying Property 1, the scoreboard’s primary signals were connected as following:

1. DataInValid: Combination of signals flagging the event ①
2. DataIn: Transaction data and parameters at event ①
3. DataOutValid: Combination of signals flagging the event ②
4. DataOut: Transaction data and parameters at event ②

Using this scoreboard, a data-corruption failure was observed. On comparing the internal conditions leading to the failure, it was confirmed that this was one of the manifestations of the post-silicon bug. In the failing scenario, a transaction “N” was getting replayed on interface Z (refer Figure 9) pending processing of a prior transaction “N-1”. At the same time, a new transaction “N+1” came in on interface X. The specific stages of various pipelines aligned in such a way that the data for transaction “N+1” was incorrectly forwarded to update the metadata storage, thereby overwriting the value of transaction “N”. This eventually caused a data corruption when transaction “N” was scheduled to go out on interface Z.

E. Overall Results

The post-silicon bug was reproduced in 4 weeks of starting the activity with approximately 300 person-hours of dedicated effort. Apart from the targeted bug, three other failing scenarios were detected. FV not only helped in determining the minimum conditions required to expose the failure set, but also arrive at the robust fix. Two fixes were tested and verified to work for all cases. The FV setup is getting reused in the next project to check the functionality of this complex logic. The key takeaway from this case study was that design size need not be a limiting factor in FV application for post-silicon debug. Many IPs smaller than this DUT have dismissed FV in past thinking the modules are too large or complex for the application.

V. SUMMARY

We presented a comprehensive guide to successfully apply formal property verification on post-silicon debugs to root-cause and debug failures that yielded impactful results as shared in the case studies. We could also show-case that issues embedded deep inside complex designs can be found through simple checks following the systematic methodology as described. In both the case studies, bug-fixes were tested for their soundness using formal property verification by rooting out incomplete bug-fixes in time and boosted the confidence in the final fixes that went into production. Through this work, we recommend that all post-silicon issues in control-logic should be reproduced in formal verification environment and their bug-fixes must be validated with formal technology before the design goes for re-spin. In past one year, we have reproduced, root-caused many post-silicon issues on a wide variety of designs before dynamic simulations could succeed. Furthermore, deploying formal property verification in post-silicon phase motivates transition from reactive to proactive approach as momentum built in post-silicon formal activities result in identifying ideal designs for formal sign-off and target bug prone designs for cleaner logic in the next generations of IPs and SOCs.

ACKNOWLEDGMENT

Thanks to our management for their support, post-silicon validation team for sharing failure details, dynamic simulation team for providing sample waves, Paras Gupta and Mayank Singhal for helping in development efforts.

REFERENCES

[1] J. Kasak, "Accelerating Post-Silicon Debug with Formal Verification," JUG, San Jose, 2018.
[2] C. R. Ho, et al., "Post-Silicon Debug Using Formal Verification Waypoints," DVCON, 2009.
[3] IEEE Std 1800™-2017, IEEE Standard of SystemVerilog – Unified Hardware Design, Specification, and Verification Language.
[4] Sakhatski A. A, “Practical Aspects of Formal Verification of Networking Chips”, DOI 2018.
[5] B. A. Krishna, et al, “Formal Verification of an ASIC Ethernet Switch Block”, FMCAD, 2010.
[6] A. Gupta, et.al., “Cross-Domain Datapath Validation Using Formal Proof Accelerators”, DVCON India, 2014.
[7] Pierre Wolper, “Expressing interesting properties of program in propositional temporal logic” 13th ACM SIGACTSIGPLAN symposium on Principles of programming languages, pages 184-193. ACM Press, 1986.
[8] T. Patel et al, “Using Formal Sign-Off to Deliver Bug-Free IPs”, Oski Decoding Formal Club, Dec 2019.
[9] M, Achutha KiranKumar, Erik Seligman & Tom Schubert, Book on “ Formal Verification – An Essential Toolkit for VLSI Design”, 2015
[10] Sabih Agbaria, Dan Carmi, Orly Cohen, Dmitry Korchemny, Michael Lifshits and Alexander Nadel, “SAT-based Semiformal Verification of Hardware”, FMCAD 2011
[11] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, Yunshan Zhu, “Bounded Model Checking”, in 2003