Formal Verification Experiences: Spiral Refinement Methodology for Silicon Bug Hunt

Ping Yeung, Mark Eslinger, Jin Hou
Siemens EDA, Fremont, CA

Abstract - Several companies have used formal verification to perform silicon bug hunting. That is one of the most advanced usages of formal verification. It is a complex process that includes incorporating multiple sources of information and managing numerous success factors concurrently. This paper will present a "spiral refinement" bug hunt methodology that captures the success factors and guides the deployment of various formal techniques. The objective is to identify the significant challenges and gradually improve each of the factors to "zero-in" on the critical bugs.

I. INTRODUCTION

Formal verification has been used successfully by a lot of companies to verify complex SOCs [1][2] and safety-critical designs [3]. As described in [1][5], it has been used extensively for A, B, C:

- Assurance, to prove and confirm the correctness of design behavior.
- Bug hunting, to find known or explore unknown bugs in the design.
- Coverage closure, to determine if a coverage statement/bin/element is reachable or unreachable.

Using formal verification to uncover new bugs is emerging as an efficient verification approach when functional simulation regression is stabilized and not finding as many bugs as before. In this paper, we will focus on the most advanced usage of formal verification: silicon bug hunt. Based on our experience, silicon bugs are incredibly tricky. Most of them happen deep in functional operation under unusual combinations of events and scenarios. We had tried various sophisticated approaches and gathered significant experiences in this area.

Experience formal verification users know how to refine the results of formal verification intuitively. However, new users have found this skill challenging to master, hence, lead to frustration and disappointment. In this paper, we capture the refinement process into a step-by-step methodology, formulate it graphically so that it is easy to understand and replicate.

II. SUCCESS FACTORS OF FORMAL VERIFICATION

It is essential to understand the factors that determine the success of formal verification on a project. Various formal teams from multiple companies have well documented the following six factors based on their experiences.

- The complexity of the design [3][8]
- The quality of the sub-goals and target assertions [6]
- The completeness of the interface constraints [5]
- The control and orchestration of the formal engines [8]
- The quality of the initial states for formal exploration [7]
- The formal expertise of the users [2]

Also, using formal verification for silicon bug hunt is different from traditional model checking in several ways:

- Over-abstracting – instead of verifying the whole design, we want to abstract or exclude the irrelevant modules so that formal verification can be as efficient as possible.
- Over-constraining – as we know which configuration, mode-of-operation, and interfaces caused the bug, we can constraint the design accordingly and disable the unused interfaces.
- Focused assertions – based on the bug scenarios observed in the lab, we write properties to capture the bug scenarios and the pre-conditions that lead to the bug formulation.
- Deep initial state – instead of starting formal verification from time 0, we can leverage simulation sequences to initialize the design amid functional operations or close to the bug formulation.

We will elaborate on these items in the following sections.

III. THE "SPIRAL REFINEMENT" BUG HUNT METHODOLOGY

If we want to be successful in hunting silicon bugs, we will inevitably need to wrestle with all of the success factors mentioned above. To help us manage them and (more importantly) allow managers to visualize them, we introduce the "spiral refinement" bug hunt radar, Figure 1. It captures the six factors in the axis of the radar chart. The objective is to identify the primary obstacle(s), such as availability of formal expertise or completeness of interface constraints. Then, we can gradually improve or refine the other factors to catch the bugs as observed in silicon.

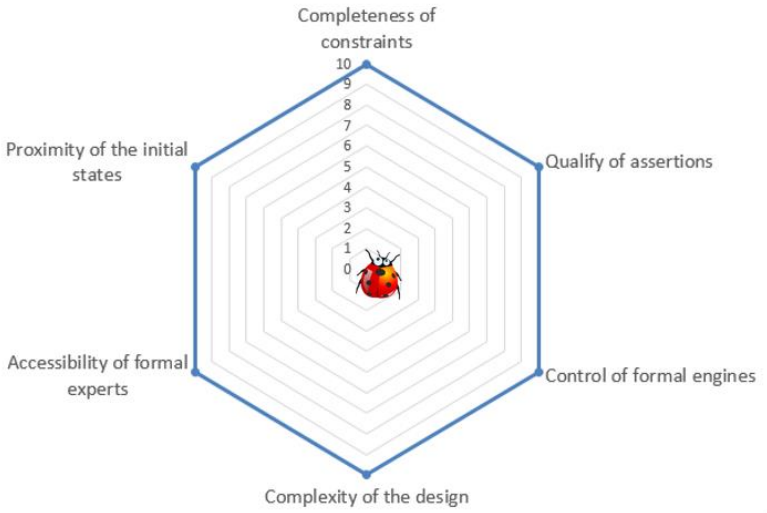

Figure 1: Radar Chart of "Spiral refinement" Bug Hunt Methodology

IV. THE "SPIRAL REFINEMENT" BUG HUNT PROCESS

Sophisticated approaches in this area have been tried and proven and experiences with them have been gathered. To summarize, the "spiral refinement" bug hunt process consists of these significant steps:

1. Identify formal experts who can help with a project (this is most likely a management task)
2. Use information from the lab, through divide-and-conquer, or process of elimination to identify the buggy module(s)
3. Write assertions to capture pre-conditions, bug scenarios, and if possible, the sequence of events
4. Identify and extract a set of good initial states from simulation traces [7]
5. Run formal verification with minimal constraints to ensure formal has good reachability into the design
6. Refine the constraints and assertions to "zero-in" on the actual bug(s)

Silicon bug hunting is an iterative process with successive refinements. To simplify the explanation, we classified the refinement steps into three phases:

Phase 1: "Initial"

radius = 8 (red), gather all information and set up the formal verification environment.

Phase 2: "Improved"

radius = 6 (green), improve each of the success factors to define the scenarios close to the bug(s).

Phase 3: "Final"

radius = 3 (purple), optimize the critical success factors to find the bug(s).

Using the spreadsheet in Table 1 and the charting function in Microsoft Excel, we can generate the concentric radar chart, as shown in Figure 2.

	All	Initial	Improved	Final
Completeness of constraints	10	8	6	3
Quality of assertions	10	8	6	3
Control of formal engines	10	8	6	3
Complexity of the design	10	8	6	3
Accessibility of formal experts	10	8	6	3
Proximity of the initial states	10	8	6	3

TABLE 1: THE TABLE TO GENERATE THE RADAR CHART OF "SPIRAL REFINEMENT" SILICON BUG HUNT

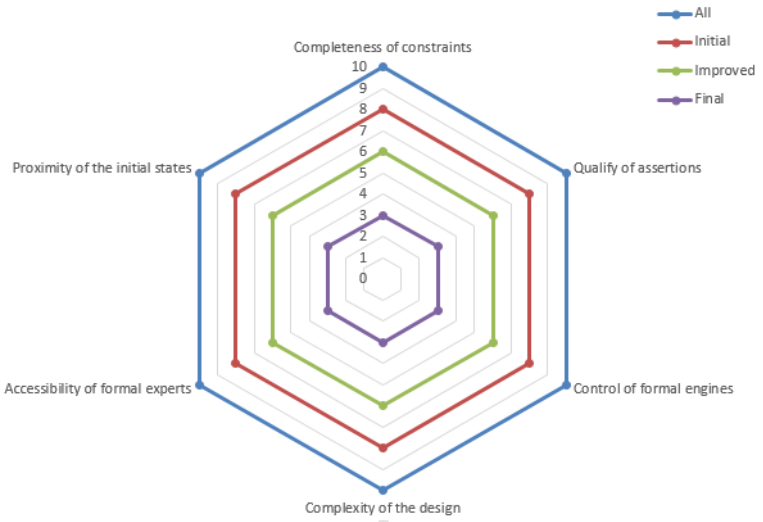

Figure 2: Radar chart of "spiral refinement" Silicon Bug Hunt

A. Phase 1: Initial

1. Complexity of the design:
    a. It is unrealistic to expect formal verification to handle the complete SoC design. It will be much more efficient to focus on the IP blocks with standard interfaces.
    b. Remove blocks or functionalities that are not relevant for the properties
    c. Turn memories and storage structures into black boxes
2. Accessibility of formal experts:
    a. Before deploying formal verification on a design, it is crucial to secure someone with formal expertise.
    b. Similar to simulation regression that will require a testbench environment, a formal regression will require a formal test environment as well. It will consist of assert and cover properties, constraints for standard and proprietary interfaces, models for complex and formal unfriendly modules.
    c. More importantly, with a formal expert in the team, he or she can work with designers to capture the design intents in properties and help designers run formal verification on the blocks early in the design cycle as the RTL is being developed.
    d. He or she can also set up the formal regression runs, maintain them, and triage issues when there are failures.
3. Control of formal engines:
    a. It is important to determine early on whether formal verification will be efficient in the design or not. Automatic formal checking of some pre-defined properties (such as is deadcode, FSM deadlock, etc) is an easy way to determine whether formal can control and observe the design.
    b. It is common for formal verification tools to have ten or more formal engines for handling difficult design structures, finding counter-examples in complex situations, or addressing special formal properties (such as liveness). It is useful to have a good understanding of the formal engines and the mechanism to control them.
4. Quality of the assertions:

   a. The quality of the formal results will only be as good as the formal properties. Imprecise or incorrect properties will lead to false firings that will take time to debug.
   b. Formal verification can examine multiple properties simultaneously. Hence, users can capture numerous scenarios in different properties or capture a single scenario in multiple properties with varying flavors of expression.
   c. It is common for new users to find property languages (such as SVA or PSL) a challenge to use or difficult to be precise. To ease the learning curve, we encourage designers to use RTL modeling code to simplify problematic assertions into simple ones.
5. Proximity of the initial states:
   a. Although toggling the reset signal is sufficient to initialize most design blocks, it is advantageous to have a good understanding of the design so that you can set up a meaningful initial state for formal runs.
6. Completeness of the constraints:
   a. Constraints help to limit the design space for formal verification. They enable formal to create realistic counter-examples and to obtain proofs. It is recommended to use unconstrained stimulus at the beginning of the formal process. They allow users to to see firings from properties so that they can be ensured formal can explore those properties. Also, counter-examples can help users understand the properties better to correct and refine the properties.
   b. To double-check the constraints, it is useful to include the constraints in the simulation environment. It will allow users to check whether the constraints are generally true or not.

B. Phase 2 and 3: Improved and Final

Bug hunting is a continuous refinement process. It is common for users to iterate between phases 2 and 3 to explore the different scenarios of the bug(s).

1. Complexity of the design:
   a. Besides turning unrelated modules into black boxes, formal verification cutpoints, counter, and memory remodeling are other fine-tuned approaches[6] to reduce the design complexity for formal verification. They give formal verification additional freedom to control the cutpoints directly.
2. Accessibility of formal experts: N/A
3. Control of formal engines:
   a. Different formal engines specialize in different classes of properties. As we focus on finding bugs, it will be more efficient to deploy only the counter-example finding engines instead of the proof engines.
   b. Each formal engine has a different profile. By monitoring engine health [7], we can identify the most effective engines and their configurations. The tool can remember this information in the formal knowledge database for subsequent formal runs.
4. Quality of the assertions:
   a. The assertions will be continuously updated. As we learn more about the design from the bug scenario(s) and/or the counter-examples from formal verification, we will refine the assertions. As formal verification can handle a lot of assertions concurrently, it is common to add assertions to cover different scenarios.
   b. Two golden rules are mentioned in [7]: 1) keep properties as simple as possible, and 2) keep properties as short as possible
   c. Some inconclusive assertions may be inefficient for formal verification. As highlighted in [6], we can reduce their complexity by reducing the depth, adding helper assertions, decomposition, and using an assume-guarantee approach.
5. Proximity of the initial states:
   a. In a bug hunt situation (especially silicon bug hunt) where the bug happens thousands of cycles into functional operation, formal verification could not recreate a long history of events. As explained in [7], traditional formal verification, which starts from time 0, is good for initial design verification, but it is inefficient for hunting complex functional bugs. The recommended methodology is to leverage functional simulation activity and start formal verification from interesting states in the simulation traces. Also, instead of using one initial state to start formal verification, multiple states from functional simulation can be used to launch multiple formal runs concurrently.
6. Completeness of the constraints:

a. In general, it is dangerous to over-constrain the formal verification environment. When over-constrained, formal verification may not be able to find any counter-example. However, there are advantages to use constraints to improve the efficiency of the formal runs.
b. Constraints can be used to set up a divide and conquer approach for formal verification. For instance, if a design can operate in different modes, we can set up multiple formal runs by constraining the mode of operation. Each of the formal runs will be more focused, and the results will be more straightforward to understand.
c. There are situations where we can over constrain the formal environment. If we are looking for a bug under a unique circumstance, it is advantageous to over constrain the environment to focus on that particular situation. At the same time, we may want to confirm the conditions under which a property is true. We can over constrain the environment concerning these conditions so that the property can be proven.

Congratulation, you have found the chip-killing bug! Besides finding the bug, now you also have the formal environment to verify any potential fix for the bug. Once the RTL code has been updated, it can be verified quickly within the formal environment. It has happened multiple times that the first RTL fix made by the design team did not solve the bug completely. The first fix may expose other vulnerabilities in the design that will need to be handled as well. From our experience, the formal environment can validate the bug fix(es) a lot more effectively than the simulation-based environment.

V. RESULTS

A. *Case #1: DDR3 controller [5]*

The post-silicon bug: a sequence of write commands to the specific memory bank and row combinations had caused the DDR3 memory to fail. It was caused by DDR3 protocol violations related to pre-charge timing.

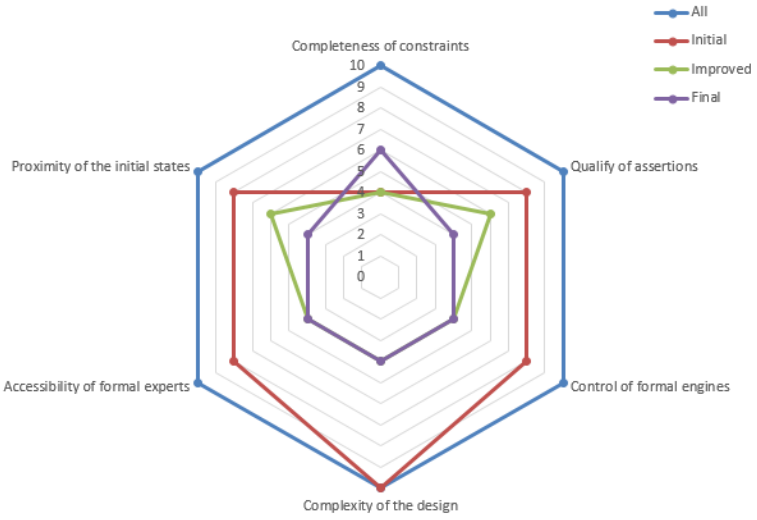

Figure 3: The Bug Hunt Process for finding the DDR Controller Bug in Silicon

Figure 3 captures how the project team optimized the formal verification success factors and found the silicon bug.

1. Complexity of the design: picking the DUT at the right level of hierarchy reduced the design complexity for formal verification. At the same time, the DUT has standard interfaces that helped constrain the design.
2. Accessibility of formal experts: the project team did not have in-house formal expertise. They contacted the tool vendor to set up a pilot project to get help with the tool and receive training on formal know-how.
3. Control of formal engines: memories and unessential parts of the design had turned into black boxes. Preliminary formal runs were done to confirm that the formal engines have adequate control of the design.
4. Quality of the assertions: Assertions were written to capture the bug scenario and the sequence of events leading up to the bug. Using the sequence of assertions as sub-goals, we used formal goal-posting [5] to recreate the sequence of pre-conditions in formal. It helped guide the formal engines towards the bug scenario.

5. Proximity of the initial states: it is essential to configure the design for proper operation. The serial nature of the design made it challenging to apply formal techniques. We were fortunate that the initialization sequence employed in the design had an "init_ok" signal, which asserted once the initialization of the design had been completed. Also, using the initial states from the sub-goals significantly improved the proximity of the initial states that led to the bug scenario.
6. Completeness of the constraints: an assertion protocol library was used to constrain the AXI interfaces. Although the DUT has 5 AXI interfaces, we initially disabled four of them to reduce complexity and focus all transactions on one interface. As depicted in Figure 3, the design was over-constrained initially. Later, other interfaces were enabled to study the interactions between the different interfaces.

B. Case #2 Memory Controller

The post-silicon bug was a read/write re-ordering defect in the memory controller. When the read/write transactions were re-ordered by control logic inside the memory controller, old data was read before the location had been updated. Three different sub-blocks were involved: the interface control unit, the buffer unit, and the memory controller. The mean time between failure (MTBF) in silicon was two to eight hours. Dynamic simulation was not able to hit the failure condition.

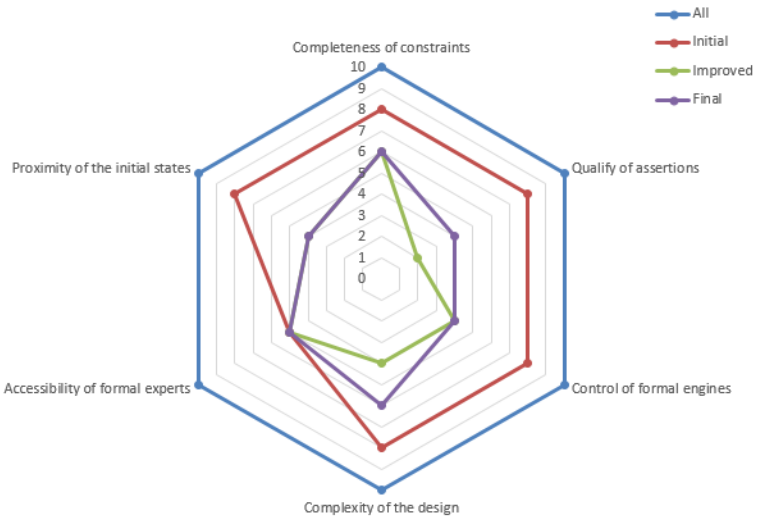

Figure 4: The Bug Hunt Process for finding the Memory Controller Bug in Silicon

Figure 4 depicts the silicon bug hunt process performed by the project team

Phase 1: Initial: understand and capture

1. Complexity of the design: the failing scenario was first observed in the lab during post-silicon testing. Based on the bug triage in the lab, the memory controller was identified as the source of the problem.
2. Accessibility of formal experts: it was fortunate to have formal experts in the team. They were involved early to guide the formal verification process.
3. Control of formal engines: memories and unessential parts of the design had turned into black boxes. Preliminary formal runs were done to confirm that the formal engines have adequate control of the design.
4. Quality of the assertions: Assertions were written to capture the failing scenario. Also, a lot of cover properties were also added to monitor the combinations of events that lead to the bug. The functional simulation was able to exercise the individual contributor of the bug (by triggering the cover properties). However, it was not able to re-create the combination of events that was observed in the lab.
5. Proximity of the initial states: Although the MTBF in silicon was at least 2 hours, we were able to understand the essential sequence of events that set up the design for failure. We had captured it as the initial state for formal verification.
6. Completeness of the constraints: we intentionally left all the interfaces unconstrained to explore all the scenarios and ensure formal verification could find the combinations of events that lead to the bug.

Phase 2: Improved: abstraction and assertions

1. Complexity of the design: We homed in on the functionalities of the memory controller and its sub-blocks. We continued to eliminate sub-blocks that were not relevant.
2. Accessibility of formal experts: N/A
3. Control of formal engines: Cutpoints were added to enable formal algorithms to control various configuration registers and counters. We had added cutpoints to some state machines as well, but as it turned out, they were not necessary.
4. Quality of the assertions: This was the most important factor of this bug hunt. As simulation was able to hit only a small subset of events that lead to the bug, a large number of assertions with different combinations and subsets of events were written to understand the pre-requisites for the bug. Formal verification was able to hit 85 percent of the pre-requisites conditions. After some refinements, it was able to find the bug with a few well-timed transactions and external triggers. However, the sequence of events did not exactly match what had been observed in silicon. It led us to the question: were there multiple causes of the bug?
5. Proximity of the initial states: As the project team continued to test the silicon in the lab and explored possible software fixes, we were able to setup the initial states to be closer to the scenario in silicon.
6. Completeness of the constraints: After formal verification had found the silicon bug (in the unconstrained environment), interface constraints, setup, and configuration constraints were added into the formal runs to refine the counter-example. It was set up cautiously to be under constrained.

Phase 3: Final: bug causes, effects, and fixes

1. Complexity of the design: As we wanted to explore different scenarios of the bug, multiple sub-blocks were added back to the formal runs. Although this approach had increased the complexity of formal verification, we were not concerned as we already had a good handle on the other success factors.
2. Accessibility of formal experts: N/A
3. Control of formal engines: Formal knowledge for 72 percent of the assertions had been cached. By leveraging the formal engines that were known to perform well, subsequent formal runs were significantly faster.
4. Quality of the assertions: We had to answer the question: were there multiple causes of the bug? To do so, we had refined the assertions to find the exact combination and sequence of events that lead to the precise bug scenario as observed in silicon. This exercise helped us understand the shortcoming of the design, where the silicon bug was just one of the observable scenarios.
5. Proximity of the initial states: After running formal verification with a number of initial states, we identified the configurations that will enable the bug and the configurations that will disable it. Using this information, we identified the software fix for the bug. Formal verification was used again to confirm that the software fix was sufficient to prevent the bug from happening again in silicon.
6. Completeness of the constraints: N/A

As the RTL fixes for the bug include internal and external IPs, it was a long process. Fortunately, with the software fix already available, the design team had sufficient time to work closely with the internal and external IP teams to identify robust fixes for the multiple scenarios identified by the formal bug hunt.

C. Case #3: Packet Scheduler

The spiral refinement methodology was initially developed for hunting silicon bugs. However, as some users gained experience, they have also adopted it for verification of IP blocks. It is especially useful when the simulation environment is not ready or not available for the IPs. Formal verification can improve the quality of the IPs significantly before check-ins. The application of the spiral refinement process consists of three phases.

- Phase 1: Initial: abstraction and reachability
- Phase 2: Improve: proofs and bug exploring
- Phase 3: Final: coverage closure

The design is a packet scheduler that prioritizes and transfers incoming packets based on the pre-programmed arbitration scheme. Figure 5 shows the 3 phases of the spiral refinement methodology.

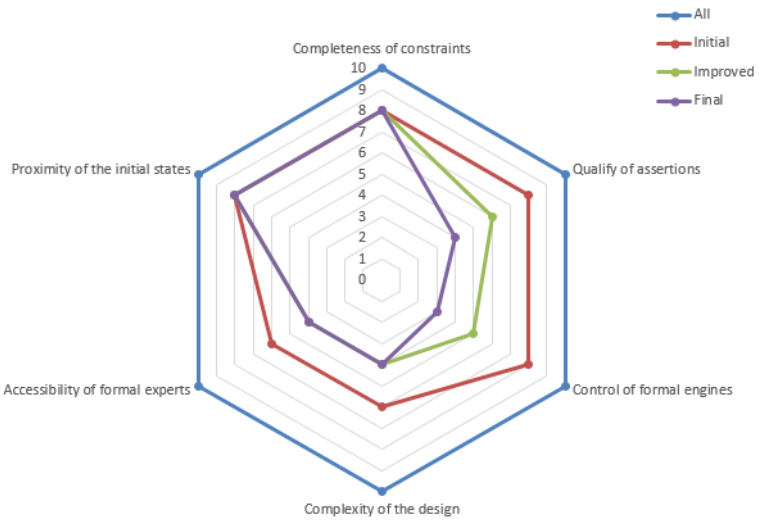

Figure 5: The Formal Coverage Closure Process for the Packet Scheduler Design

Phase 1: Initial: Abstraction

1. The complexity of the design: the depth and breadth of the design make it inefficient for formal verification to handle directly. To improve efficiency, we reduced
    - the number and sizes of channels,
    - the packets' length, and
    - the amount of buffering for outstanding transactions.

    It was not difficult as the design is very configurable. The design team helped us generate multiple simple-and-true representations.
2. Accessibility of formal experts: In-house formal experts were involved early to help with the design abstraction process. We had recognized their SVA knowledge and formal coding technique as one of the determinate factors for the project's success.
3. Control of formal engines: memories and unessential parts of the design had turned into black boxes. Preliminary formal runs were done to confirm that the formal engines have adequate control of the design.
4. Quality of the assertions: formal experts worked closely with designers to create assertions and capture failure scenarios at critical parts of the design. Complex assertions were written for the packets' data integrity and credit-based scheduling schemes. We measured the assertion density to ensure sufficient assertion coverage.
5. Proximity of the initial states: we used a known-good sequence to initialize the design. As shown in Figure 5, it was kept constant for all phases.
6. Completeness of the constraints: formal assertion IPs had been developed early on for the packet transfer protocol, AXI bus interfaces, configuration, and memory interfaces. They were kept constant for all phases.

Phase 2: Improve, Proofs and Bug hunting

1. Complexity of the design: We reduced the design complexity further by customizing the design abstraction w.r.t. the focus of the formal run.
2. Accessibility of formal experts: formal experts and tool vendors helped partition and launch the formal runs onto multiple cores in the server farm environment.
3. Control of formal engines: we ran formal verification with all engines to look for proofs and violations. We tried to be as efficient as possible by leveraging the four levels of multi-processing:
    a. Operation level: Multiple formal runs were setup for 14 configurations
        - The design team identified 14 representative configurations instead of 100s of possibilities.
    b. Design level: Formal runs with different abstractions. For instance
        - setups to focus on by-pass mode, a single channel/steam, etc.
        - reduce the buffering threshold to trigger the discard mechanism early
        - constraint inputs to distress test specific channel or functionality of the design.
        - turn unnecessary modules/instances into black boxes
    c. Session level: targets and constraints

- Setup multiple sessions with different combinations of related targets and constraints
- These sessions can be setup and monitored within the tool easily

d. Engine level: within the tool
   - The tool is fully automated to launch formal runs with an orchestrated set of engines on multiple cores and aggregate the results into a result database.

4. Quality of the assertions: the tool provided feedback on the formal runs [8] by showing the engine health dynamically and the active logic being analyzed by formal verification. We added data independence and nondeterminism to the assertions to further improve the design's controllability and formal efficiency.
5. Proximity of the initial states: same a phase 1
6. Completeness of the constraints: same a phase 1

Phase 3: Final (Coverage)

After reviewed the proven properties and debugged the violations in phase 2, we focused on the inconclusive properties and formal coverage in this final phase. It is part of the formal coverage closure methodology.

1. Complexity of the design: we used the same design abstractions as in phase 2
2. Accessibility of formal experts: same as in phase 2
3. Control of formal engines: we changed the formal orchestration to focus on formal coverages. As in phase 2, we were as efficient as possible by leveraging the four levels of multi-processing.
4. Quality of the assertions: Formal coverage was computed with various methods depending on the focus and accuracy requirements. There are different measurements of formal coverage:
   a. Functional coverage – assert, cover, and coverbin statements,
   b. Structural formal coverage
      - It computes from the structural cone-of-influence (COI) of the properties
      - It is a rough analysis that can be derived before extensive formal analysis
      - It can identify assertion holes in early-stage for adding more properties
      - It can target coverage elements such as statement, branch, FSM, etc
   c. Proof core formal coverage
      - It computes from the structures analyzed by formal, which is a subset of the structural COI.
      - It provides an engine-level view of the verification coverage
      - It reports verification metrics for late-stage coverage closure
      - It can target coverage elements such as statement, branch, FSM, etc
5. Proximity of the initial states: same a phase 1
6. Completeness of the constraints: same a phase 1

The formal coverage metrics gave the design team sufficient confidence that formal verification has performed throughout verification of all the functionalities of the design.

VI. SUMMARY

It has been proven that formal verification is essential for silicon bug hunts. The "spiral refinement" radar and methodology provide the necessary guidance to this stressful and interactive process. It captures the success factors and helps guide the deployment of various approaches. As described and demonstrated in the results, the process is to identify any obstacle then gradually improve or refine each of the success factors so that we can "zero-in" on the bug that happened in silicon. Once the bug has been identified, the formal verification environment can serve as a golden setup to verify any potential software and RTL fixes. It is very efficient in reducing the turn-around time and helps prevent the introduction of new issues.

VII. REFERENCE

[1] Ram Narayan, "The future of formal model checking is NOW!", DVCon 2014.
[2] M Achutha KiranKumar et al., "Making Formal Property Verification Mainstream: An Intel® Graphics Experience," DVCon India 2017
[3] Mandar Munishwar, Vigyan Singhal, et al., "Architectural Formal Verification of System-Level Deadlocks," DVCon 2018.
[4] Richard Ho, et al., "Post-Silicon Debug Using Formal Verification Waypoints," DVCon 2009
[5] Blaine Hsieh, et al., "Every Cloud - Post-Silicon Bug Spurs Formal Verification Adoption," DVCon 2015
[6] Jin Hou et al., "Handling Inconclusive Assertions in Formal Verification," DVCon China 2018
[7] Mark Eslinger, Ping Yeung., "Formal Bug Hunting with "River Fishing" Techniques," DVCon 2019
[8] Jeremy Levitt et al., "It's Been 24 Hours - Should I Kill My Formal Run?", Workshop, DVCon 2019
[9] Abhishek Anand et al., "Bounded Proof Sign-off with Formal Coverage," DVCon 2020