Formal Verification Experiences: Silicon Bug Hunt with "Deep Sea Fishing"

Mark Handover, Mentor, a Siemens Business, UK
Abdelouahab Ayari, Mentor, a Siemens Business, Germany
Ping Yeung, Mentor, a Siemens Business, US

Abstract

Formal verification has been used successfully to verify today's SOC designs. A few companies also used formal verification to perform post-silicon bug hunting. It 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. These post-silicon bugs are complex. Most of them happen deep in functional operation under unusual combinations of events and scenarios. Sophisticated approaches have been tried-and-true, and experiences have been gathered in this area. In this paper, we will share these experiences by first introduce the "deep sea fishing" bug hunt radar. It captures the success factors and helps guide the deployment of various methodologies. The objective is to identify obstacle(s) and to gradually improve or refine each of the factors so that we can "zero-in" on these critical silicon bugs.

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 to be an efficient verification approach when functional simulation regression is stabilized and not finding as many bugs as before. In this paper, we are going to focus on the most advanced usage of formal verification: post-silicon bug hunt. Post-silicon bugs are unexpected problems. Finding them is an iterative process that requires successful management of numerous factors concurrently.

Success factors of formal verification

Formal verification enables verification to be done early in the design cycle and also late in the design cycle when functional simulation regression is not finding many bugs. In different usage models, it is essential to understand these factors that determine the success of formal verification:

- 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]

The "deep sea fishing" bug hunt radar

If we want to be successful 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 "deep sea fishing" bug hunt radar, Figure 1. It captures the six factors in the axis of the radar chart. The objective is to identify the major obstacle(s) such as availability of formal expertise or completeness of interface constraints first. Then, we can gradually improve or refine the other factors so that we can catch the bugs as observed in silicon.

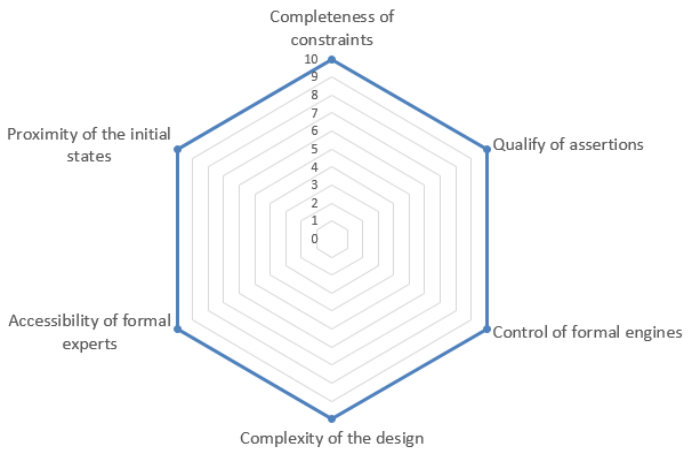

Figure 1: Radar chart of "Deep Sea Fishing" Bug Hunt

The "deep sea fishing" bug hunt process

Sophisticated approaches have been tried-and-true, and experiences have been gathered in this area. To summarize, the "deep sea fishing" bug hunt process consists of these significant steps:

1. Identify formal experts who can help with the project. It is most likely a management task
2. Use information from the lab, by divide-and-conquer, or process of elimination to identify the buggy module(s)
3. Write assertions to capture the pre-conditions, the 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 hunt is an iterative process with successive refinements, as depicted in Figure 2. To simplify the explanation and help with understanding, we classified them into three phases:

Phase 1: "Initial"
radius = 8 (red), gather all the 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.

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

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

	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 "Deep Sea Fishing" Silicon Bug Hunt

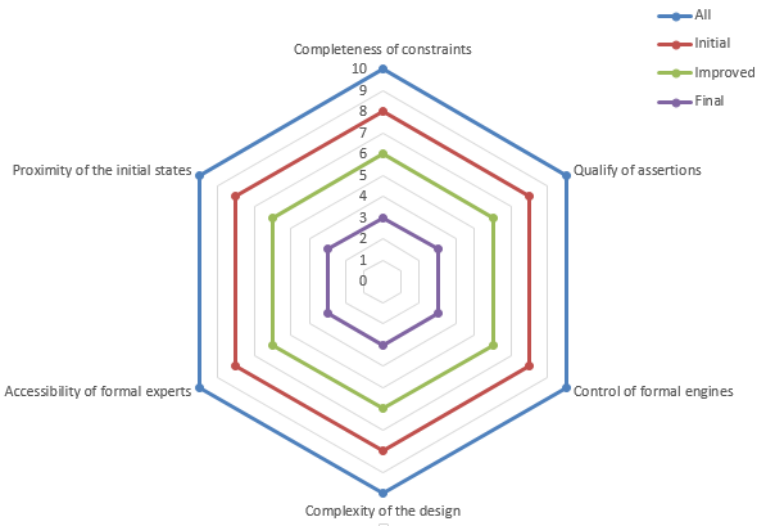

Figure 2: Radar chart of "Deep Sea Fishing" Silicon Bug Hunt

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. In addition, he or she can 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 whether formal verification will be efficient in the design or not. Automatic formal checking of some pre-defined properties (such as is deadcode) is an easy way to determine whether formal can control and observe the design.
    b. It is common for formal verification tools to have 10 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.
    b. Different from a simulation with a single testbench, formal verification can work on multiple properties at the same time. Hence, users can capture multiple scenarios in different properties or capture a single scenario in multiple properties with different favor of expression.
    c. It is not uncommon for new users to find property languages (such as SVA or PSL) challenging to use or difficult to be precise. To ease the learning curve, we often encourage designers to use a mix of RTL modeling code and SVA/PSL together.
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 we 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 be unconstrained at the beginning of the formal process. It enables users to see firings from properties so that they can be ensured formal can explore those properties. In addition, counter-examples can help users understand the properties better so that they can 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.

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 engine specializes in a different class(es) of properties. As we are focusing 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 its configurations. The tool can remember this information in the formal knowledge database that can be used in the subsequent formal runs.
4. Quality of the assertions:
    a. The assertions will be updated constantly. 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 efficient of the formal runs, such as:
    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.

Results

Result Case 1: DDR3 controller [5] with company F

The post-silicon bug: a sequence of write commands to the specific memory bank and row combinations would cause a DDR3 protocol violation related to pre-charge timing.

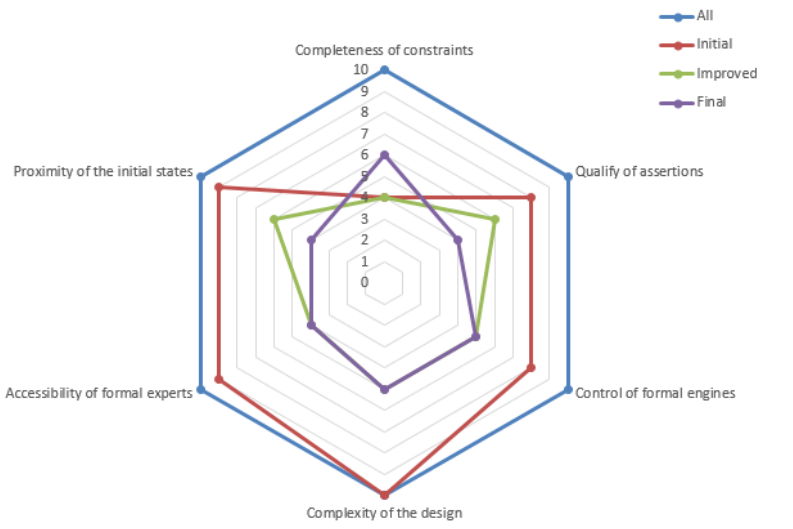

Figure 3: The Bug Hunt Process for finding the DDR Controller Bug in Silicon

Here is 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 formal expertise in the house. They contacted the tool vendor to set up a pilot project so that they can get help with the tool and formal knowledge.
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. This is important. By using the sequence of assertions as sub-goals, we were able to deploy formal goal-posting [5] to re-create the sequence of the events 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 for applying 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 completed. Also, by using the initial states from the sub-goals, it significantly improved the proximity of the initial states that lead 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, initially, we disabled 4 of them to reduce complexity and to focus all transactions on one interface. As depicted in Figure 3, the design was over-constrained initially. Later, more interfaces were enabled to study the interactions between the different interfaces.

Result Case 2: Memory Controller with Company Q

The post-silicon bug: it was a read/write re-ordering defect in the memory controller. When the read/write transactions were re-ordered by control logics inside the memory controller, old data was being read before the location has 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 2 to 8 hours. A dynamic simulation was not able to hit the failure condition.

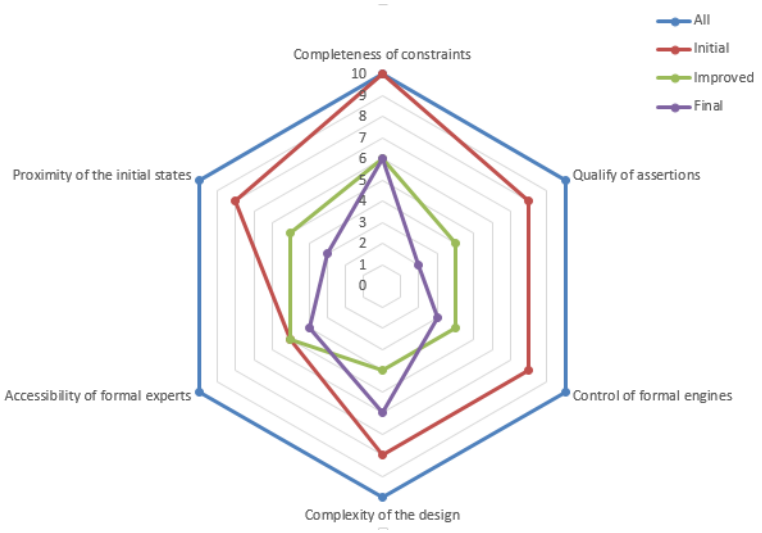

Figure 4: The Bug Hunt Process for finding the Memory Controller Bug in Silicon

Here was the silicon bug hunt process performed by the project team

Phase 1: Initial

1. Complexity of the design: the failing scenario was first observed in the lab during post-silicon testing. Based on this observation, the memory controller was quickly 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 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. In addition, a lot of cover properties were also added to monitor the combinations of events that lead to the bug. Simluation 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 had been 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 setup the design for failure. We had captured it as the initial state for formal verification.
6. Completeness of the constraints: we left all the interfaces unconstrained purposely to explore all the scenarios and to ensure formal verification could find the combinations of events that lead to the bug.

Phase 2: Improved

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% of the pre-requisites conditions and 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 match exactly what had been observed in silicon. This leads 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 setup cautiously to be under constrained.

Phase 3: Final

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 for formal verification, we were not concerned as we already had a good handle of the other success factors.
2. Accessibility of formal experts: N/A

3. Control of formal engines: formal knowledge for 72% of the assertions had been cached. By leveraging the formal engines that were known to perform well, subsequent formal runs were significantly faster than before.
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 exact 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 had identified the configurations that will enable the bug and the configurations that will disable it. Using this information, we had 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.

Summary

It has been proven that formal verification is essential for silicon bug hunts. The "deep sea fishing" radar can provide the necessary guidance to this stressful and interactive process. It captures the success factors and helps guide the deployment of various methodologies. As described and demonstrated in the results, the process is to identify any obstacle, gradually improve or refine each of the success factors so that we can "zero-in" on the bug that happened in silicon. Also, 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 help prevent the introduction of new issues.

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