Formal Bug Hunting with “River Fishing” Techniques

Mark Eslinger — Mentor-A Siemens Business, Fremont
Ping Yeung — Mentor-A Siemens Business, Fremont

Abstract

Formal verification has been used successfully to verify today’s SOC designs. Traditional formal verification, which starts from time 0, is good for early design verification, but it is inefficient for hunting complex functional bugs. Based on our experience, complex bugs happen when there are multiple interactions of events happening under uncommon scenarios. Our methodology leverages functional simulation activity and starts formal verification from interesting “fishing spots” in the simulation traces. In this paper, we are going to share the interesting fishing spots and explain how formal engine health is used to prioritize and guide the bug hunting process.

Introduction

Formal verification has been used successfully by a lot of companies to verify complex SOCs [2] and safety-critical designs [3]. The ABCs of formal have been used extensively as described in [5]:

- Assurance: to prove and confirm the correctness of design behavior.
- Bug hunting: to find known bugs 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 is not finding as many bugs as before. Traditional formal verification, which starts to explore the design from time 0, is good for early design verification. As more blocks are being integrated, the state space of the design increases exponentially, making traditional formal verification approaches inefficient for bug hunting. As we have observed, complex bugs happen under combinations of events and scenarios.

Sophisticated approaches such as waypoints [4], coverpoints and goal-posting[5] have been proposed and used successfully to find bugs buried deep in the design, especially post-silicon bugs.

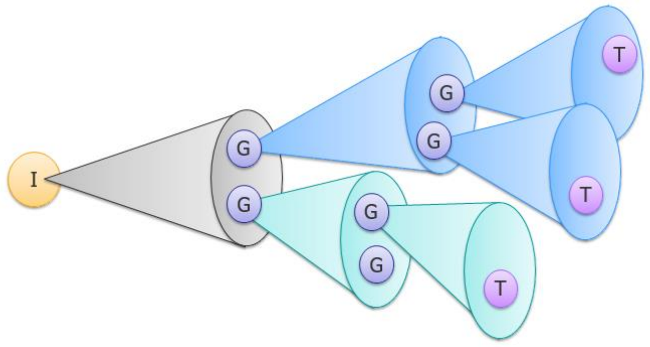


Figure 1. Waypoints, coverpoints and goal-posting methodology

The essence of these approaches is that instead of focusing on the targets that are hard to reach, formal verification targets coverpoints, goals, or waypoints that are found deep in formal analysis. Once those coverpoints are hit, they are used as initial states to explore further into the design, as illustrated in Figure 1. Using this approach, formal verification can leap from goal to goal, exploring deeper and deeper until the final targets are achieved.

River Fishing Approach

We have expanded these approaches further. “River fishing” [1] is a good metaphor for our methodology: by identifying some *good* “fishing spots,” we can significantly increase the number of fish we catch. Instead of using one initial state to start formal verification, we pick out interesting states from functional simulation traces, as depicted in Figure 2, and start formal verification from those fishing spots.

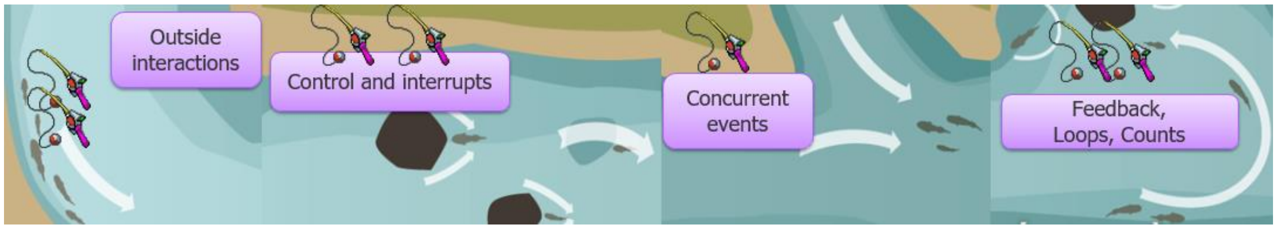


Figure 2. Interesting fishing spots

The major difference between this approach and goal-posting is that instead of targeting the coverpoints or goals to explore deep into the design, we are leveraging simulation traces, as in Figure 3, to explore interesting initial states close to the final targets. If functional simulations have exercised the coverpoints or goals already, we have an initial state as good as goal-posting. Also, if some targets require specific pre-conditions to happen ahead of time, we can also take advantage of the different functional tests and identify different fishing spots for those targets. Effectively, the river fishing approach leverages what has already been performed by functional simulations and starts formal verification as close to the targets as possible.

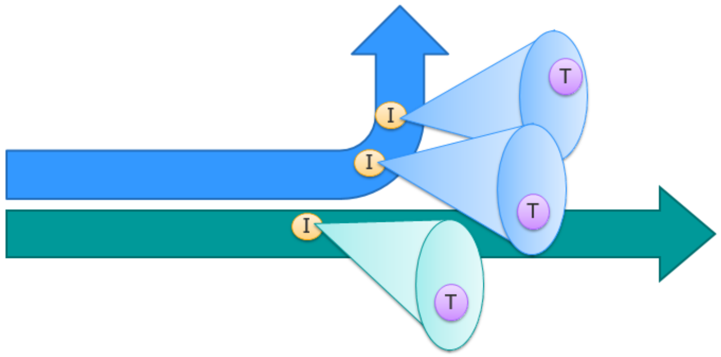


Figure 3. Launching formal verification from interesting fishing spots

We encountered one problem with this approach, however. In a simulation regression environment, there were just too many fishing spots, and there was no reliable way to tell whether a spot would be fruitful or not. As a result, there was not enough time to try many of these fishing spots, and servers were stalled on spots that did not yield any result. We learned that it is insufficient to find just *good* fishing spots; it is also important to make sure it is easy to catch fish at those spots.

As formal verification users become more mature, they are requesting more transparency to understand how formal verification is doing on their designs. Based on our experience, formal engine health is one of the best indicators. It can be used to measure the progress made by formal verification. Also, formal engine health is a matrix that can be used to estimate, prioritize, and monitor formal verification runs. Using formal engine health, we evaluate the fruit-fulness of fishing spots, screen out the low-quality ones, and prioritize the rest for formal verification. By continuing to monitor formal engine health during these subsequent formal verification runs, we can terminate the unfruitful ones early to save resources for new runs.

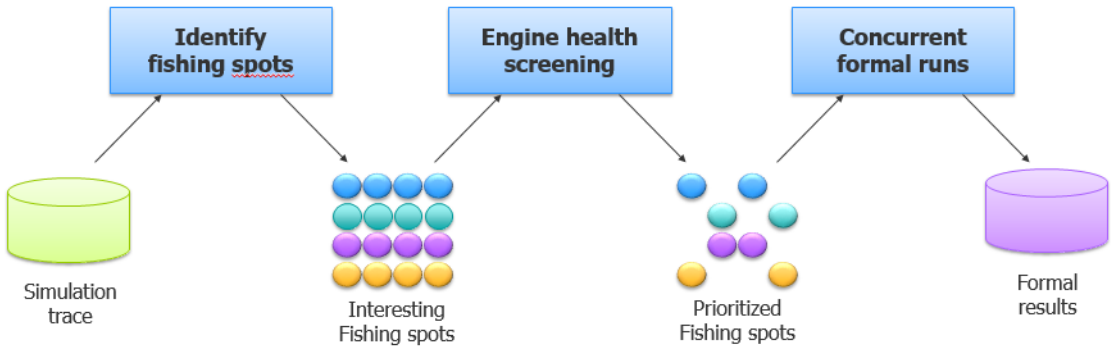


Figure 4. Formal verification using "river fishing" techniques

To summarize, the "river fishing" formal bug hunting methodology consists of three major steps:

1. Identify and extract a set of good fishing spots from the simulation traces
2. Screen and prioritize the fruitfulness of these fishing spots using formal engine health
3. Launch multiple formal engines concurrently on a server farm environment

In the following sections, we describe each of these steps in more detail.

Identifying "Fishing Spots"

River fishing experts [1] have suggested that the interesting fishing spots are special sections of the river where there are unusual activities. Picking interesting spots from simulation traces for formal verification is similar. Our approach is based on the patent, Selection of Initial States for Formal Verification [6], with some variations.

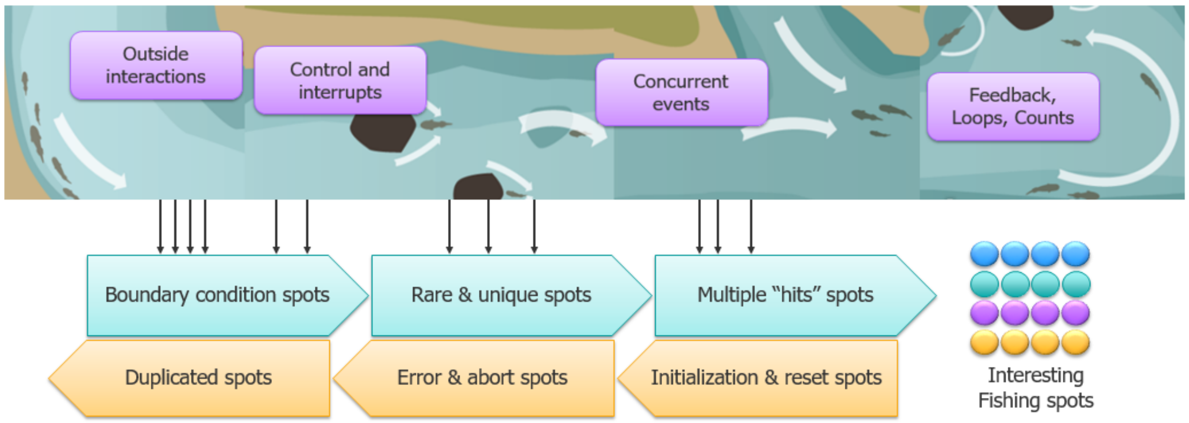


Figure 5. Identifying interesting fishing spots

A quick and comprehensive two-tier approach is used to pick out the interesting fishing spots. The primary criteria are:

1. Interface interactions
   Inter-module communication and standard protocol interfaces are infamous for design issues. As protocol monitors are commonly used to instrument these on-chip buses and common interfaces, we can capture interesting fishing spots based on activities on these interfaces. Functional simulation is already generating tons of bus transactions; formal verification can leverage them to explore new corner cases.

2. Control and interrupts
   These are FSMs, bus controllers, memory controllers, flow charts, algorithmic controls, and so on representing critical control logic in a design. It may not be trivial for formal to traverse all the possible or even some of the deep state sequences. Hence, every new state or transition exercised by simulation are potentially interesting fishing spots for formal verification.

3. Concurrent events
   These are the arbiters, schedulers, switches, multiplexing logics, etc in a design. The timing and sequence of events are important. Functional simulation may exercise the same sequence of events repetitively. Instead, formal verification can leverage the setup, but change the ordering of the events to ensure the design is robust.

4. Feedback, loops, and counts
   These are the FIFOs, timers, counters, and so forth handling the data transfers, bursting, and computations in a design. Functional simulation exercises the non-stressful situations well. By leveraging the simulation activities, formal verification can explore the stressful corner case scenarios; such as starting, stopping, overflow, underflow, and stalling.

5. User-defined assertions and coverage properties
   The fan-in cones of the user-defined properties are great coverpoints and sub-goals for formal verification. If they have already been covered by simulation, they will be good fishing spots for formal verification to verify the target properties.

Secondary criteria are related to inclusion or exclusion of the primary ones as described in the patent [6].

- Inclusion criteria
  When gathering the primary fishing spots, we want to mark the high-value ones; such as states that satisfy multiple criteria, states with low frequency of change, and states with new activities.

- Exclusion criteria
  At the same time we want to filter out poor-value fishing spots; such as duplicate states, error states that lead to misleading results, and states during the power-up, reset, initialization, or configuration sequence.

Screening with Engine Health

Formal verification is different from simulation. Simulation runs are predictable and will finish. When running formal verification on a large set of assertions, it is difficult to predict when and if it will finish. Hence, it is important to understand formal engine health and leverage it to measure the progress made by formal verification. Based on our experience, formal engine health is a good matrix to estimate, prioritize, and monitor formal verification runs.

The concept of engine health helps us determine if, and to what degree, formal engines are contributing to the progress made on a given set of targets. We define formal engine health with a set of parameters:

1. Formal targets concluded (proven/fired/covered/uncoverable)
2. Sequential depth explored or *cone of influence* analyzed
3. Formal knowledge and engine setting acquired

Targets concluded and *sequential depth explored* are two well-established metrics that have been used regularly to determine the overall progress of a formal run.

On the other hand, if we want to understand how formal is doing on an individual property or target, we have to go deeper to examine the cone of influence and the formal knowledge acquired on those targets. A formal setup may work extremely well on a few targets, but not so well on the others. Cone of influence provides more information about the depth explored per target. Cone of influence provides detailed information on the fan-in logic and their dependence concerning the targets. It highlights the “rocks” in the fan-in logic where formal gets hung up, spending a lot of time analyzing. These difficult design elements can be identified, and users can determine whether it is necessary to intervene. Adding cutpoints, abstracting the design elements, or enabling some special engines are a few approaches to enable the “rock” to be analyzed more efficiently.

As mentioned before, it is insufficient to have just *good* fishing spots. It is also important to make sure it is easy to catch fish at those spots. From our experience, a good simulation environment will yield a lot of interesting fishing spots.

Traditionally, formal engine health was used to monitor computational intensive formal runs. Until a recent project, it did not occur to us that it can also be a good approach to screen fishing spots. With a lot of formal runs to perform on limited compute resources (and licenses), we found ourselves constantly terminating runs with potentially unfruitful fishing spots. The formal engine health parameters (described above) were used to determine whether compute servers were allocated to explore the spots thoroughly.

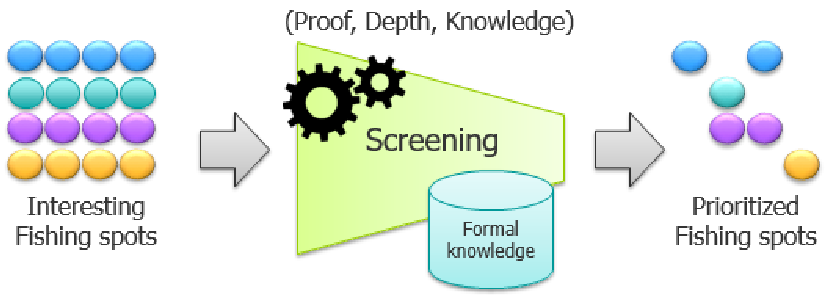

(Proof, Depth, Knowledge)
Interesting Fishing spots
Screening
Formal knowledge
Prioritized Fishing spots

Figure 6. Screening fishing spots with engine health

The process goes like this. As the goal is to screen and prioritize the effectiveness of the fishing spots, a quick run with a set of explorative formal engines is performed. An initial fishing spot is used to establish the initial values of the formal engine health parameters, as in (Proof, Depth, Knowledge):

$$\begin{aligned} F(Spot_0) &= (P_0, D_0, K_0) \\ \Delta F(Spot_n) &= (P_n, D_n, K_n) - (P_0, D_0, K_0) \\ &= \Delta(P_n, D_n, K_n) \end{aligned}$$

Then the subsequent fishing spot is compared with the initial one. If the delta, ΔF, is low, we drop the spot from further investigation. By doing this we can screen the fishing spots accordingly and launch formal runs with only the differentiating and fruitful spots. At the same time, we also cache the formal knowledge during the screening process, so CPU resources are not wasted. Subsequent formal runs will be faster.

Monitoring Engine Health

With a prioritized list of fishing spots, extensive formal runs can be launched concurrently on a server farm environment. A master server will manage and monitor all the formal processes. It will collect the formal engine health parameters from each of the formal runs continuously.

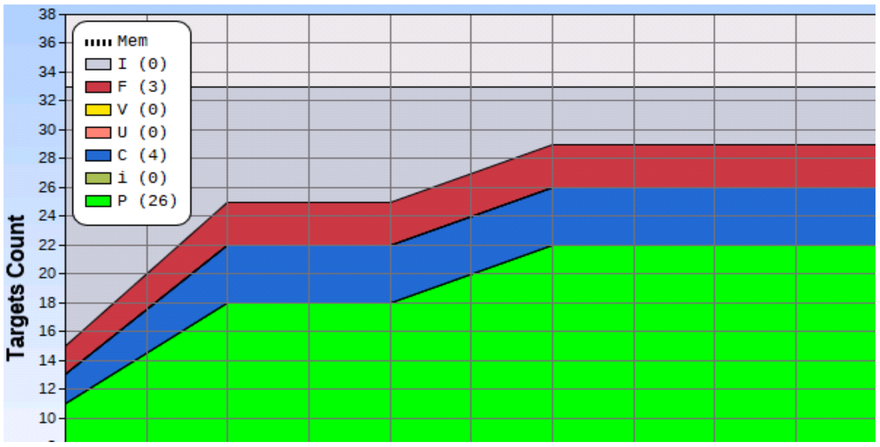
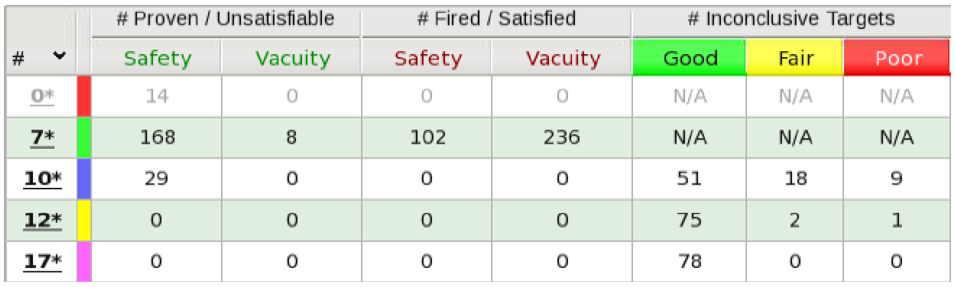


Figure 7. Formal targets proved or satisfied snapshot

Figure 7 shows the distribution of the formal targets at the beginning of a formal run. It lets users know how much progress has been made. Initially most of the 33 targets were inconclusive (I). With multi-cores running concurrently, formal verification gradually verified the targets into one of the following catalogs: firing (F), vacuous (V), uncoverable (U), covered (C), and proof (P).

A comprehensive formal tool, such as [7], has a set of formal engines to handle designs with different structures. As the formal run progresses, we can examine the status of the engines to understand which ones are finding results and which ones are not being productive. If an engine is not contributing to any of the current set of results, we can swap out that engine to save resources and focus the other engines on the task at hand. Figure 8 shows a snapshot of the engines in the middle of a distributed formal run.



		# Proven / Unsatisfiable		# Fired / Satisfied		# Inconclusive Targets		
#		Safety	Vacuity	Safety	Vacuity	Good	Fair	Poor
0*	🟥	14	0	0	0	N/A	N/A	N/A
7*	🟩	168	8	102	236	N/A	N/A	N/A
10*	🟦	29	0	0	0	51	18	9
12*	🟨	0	0	0	0	75	2	1
17*	🟪	0	0	0	0	78	0	0

Figure 8. Formal engine knowledge snapshot

In figure 8, the “Proven/Unsatisfiable” columns show which engines solve the safety/liveness/vacuity/cover type checks. The “Fired/Satisfied” columns show which engines generate the counterexamples. Engine 7 is very productive in finding a lot of proofs and firings. Engine 0 (the housekeeping engine) and Engine 10 have found some proofs. On the other hand, Engines 12 and 17 (designed for some difficult problems) haven’t been contributing to the results although they have been working on the problems. The “Inconclusive Targets” columns (Good, Fair, Poor) show the individual engine health for the work-in-process targets. Engine 12 is working on 75 targets that have good health; in other words, it is making good progress. There are three targets with fair and poor health. Similarly, Engine 10 is working well on 51 properties but is not effective on more targets.

Caching the Formal Knowledge

Experienced fishermen remember the specifics about a river and the techniques used to catch fish. They learned the river, so to speak. The same applies to formal verification. Once formal has worked through a design element or solved a problem, it is marked as an acquired knowledge. This knowledge will be cached and stored for future use. Besides benefiting future formal runs, formal knowledge can help the current run immediately. Typically designs have a few common design structures and interfaces. Once formal has learned how to handle one of these efficiently, it can be applied to all of the similar ones right away. Hence, just-in-time knowledge sharing between different formal engines that run on different processors/servers is essential.

Results

Table 1 summarizes the results of applying the “river fishing” methodology on three IP blocks. Block De is a small design with 12 user-written properties. Block Pc and Cp have more user-written assert and cover properties. As Block De is small and formal verification was able to handle it comprehensively, the quality of the fishing spots did not make any difference in this case. Fishing spot *S0* was the design state after initialization and configuration. It was used to establish the initial formal engine health. Fishing spot ∑ *SP* was the centralized database that had been aggregating all the results together.

For Block Pc, the formal run from *S0* was able to conclude 66 targets PF/CU (proven/fired/covered/uncoverable) with 5 inconclusive targets after 24 hours. By providing more interesting initial states, the subsequent fishing spots were able to help formal verification complete all targets (0 inconclusive). The results for Block Cp was more significant. The fishing spots helped formal verification reduce the number of inconclusive targets from 21 down to 2 (with 19 more fired or covered targets).

Table 1. Block-Level Results

Block	Targets	Fishing Spot *S0*	Fishing Spot ∑ *SP*
Block De	12	12 PF, depth 23	12 PF, depth 23
Block Pc	71	66 PF/CU + 5 I, depth 134	71 PF/CU + 0 I, depth 256
Block Cp	81	60 PF/CU + 21 I, depth 65	79 PF/CU + 2 I, depth 161

PF/CU: proven/fired/covered/uncoverable. I: inconclusive targets

Table 2 summarizes the results from two sub-system level runs. Each sub-system has multiple IP blocks with interconnected buses (such as AXI). The properties are from various sources: some are user-written properties from the IPs; some are from assertion-based interface monitors; some are derived assertions from interconnectivity specification; some are extracted assertions (on FSMs, counters, FIFOs, etc.) via automatic formal verification. Each of these sub-systems is a good representation of a formal regression in which the targets have a wide range of complexity levels. Within the first 15 minutes of the formal run, the majority of the targets (98+%) had already been proven or fired. This was a pleasant surprise to us. As a result, the “river fishing” technique was deployed after the first 15 minutes when the tool had concluded and cached the easy targets. Again, fishing spots helped reduce the number of inconclusive targets significantly. For Design Ct, it was reduced from 37 inconclusive to 7 inconclusive targets. For Design Pb, it was reduced from 79 inconclusive to 44 inconclusive targets.

Table 2. Sub-System Level Results

Design	Targets	First stage	Fishing spot *S0*	Fishing spot $\sum$ *SP*
Design Ct	2356	2319 PF/CU + 37 I 15 min	2338 PF/CU + 18 I, 24 hours	2349 PF/CU + 7 I, 24 hours
Design Pb	15205	15126 PF/CU + 79 I 15 min	15154 PF/CU + 51 I 24 hours	15161 PF/CU + 44 I 24 hours

PF/CU: proven/fired/covered/uncoverable. I: inconclusive targets

We have captured two representative bug situations below. They were successful only after we started formal verification from fishing spots deep in the simulation traces. Formal engine health screening was also used to eliminate a lot of potential unsuccessful formal runs from the beginning or during the process.

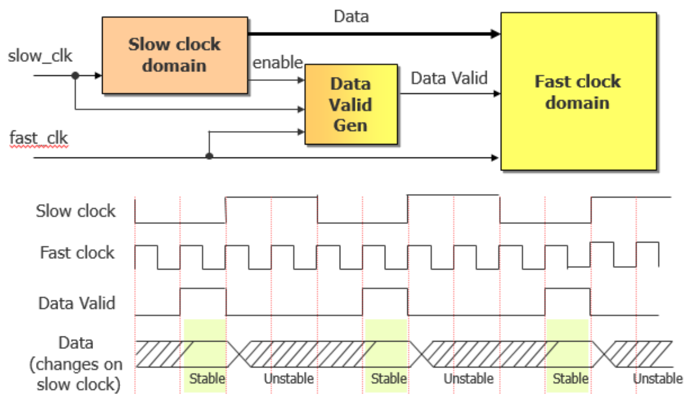


Figure 9. Ratio-synchronized data transfer interface

<u>Case1</u>: To save power, today's designs have a lot of ratio synchronous clocks that run each component as slowly as possible. As a result, ratio synchronous interfaces are common. At one of these interfaces, the fast clock domain was designed to sample the data at the end of a slow clock period when the data valid condition was asserted. However, under some corner case situations (unknown to the design team initially), the data was sampled even when the valid condition was not asserted. As a result, corrupted data was registered and passed on within the system. The fishing spots were determined based on activities on the two interfaces, counters, and control logic. Several fishing spots were picked after the design had been initialized and also after data had been flowing correctly for hundreds of cycles between these two interfaces. Formal verification exposed the incomplete handshake between these two interfaces which caused the data to be sampled invalidly.

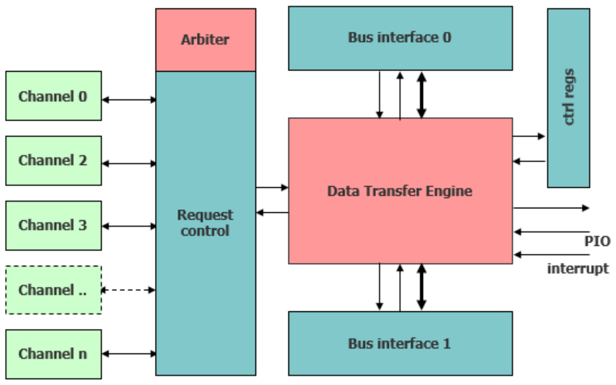


Figure 10. Data transfer controller

Case 2: In this data transfer controller, dynamic transfer channels were set up to handle data packets with different priorities. In this case, when more than one channel finished the transfer at the same time, one set of address pointers was de-allocated twice while the other set was not de-allocated, causing memory leaks and data corruption. The fishing spots were determined based on activities on the concurrent events, counters, and complex control logic. In this type of design, it is very difficult for formal verification to set up these data transfers. However, they are readily available in the simulation regression. By leveraging multiple fishing spots deep in the simulation runs and prioritizing them based on engine health, formal verification was able to expose the weakness in the channel de-allocated logic and synchronized two channels to complete the transfer at the same time.

Summary

We firmly believe that simulation and formal methodologies can be used together to accelerate the verification of intricate designs. Some companies have already made organizational changes to facilitate this approach. The idea is to leverage what has been learned or achieved in one methodology as stepping stones for the other methodology. In this paper, we have described a *river fishing* technique. It leverages the functional simulation activities and starts formal verification from interesting fishing spots in the simulation traces. As described, it consists of three major steps:

1. Identify and extract a set of good fishing spots from the simulation traces
2. Screen and prioritize the fishing spots using formal engine health
3. Launch and monitor multiple formal runs on the computing servers

To identify a set of fishing spots, we have highlighted several criteria, including interface interactions, control and interrupts, concurrent events, feedback loops and counts, user-defined assertions, and coverage properties. Then the fishing spots are screened and prioritized based on the formal engine health. We define formal engine health with parameters consisting of formal targets proven or fired, sequential depth explored, and formal knowledge acquired. Then the set of fishing spots are used to initialize multiple formal runs while a centralized database is aggregating all the results together. Based on the results presented and the complex bugs found, we can conclude that the river fishing technique does help improve the quality of the results in a formal regression environment.

References

[1] Takemefishing.org, www.takemefishing.org/freshwater-fishing/types-of-freshwater-fishing/river-fishing
[2] M. Achutha KiranKumar V, et al., "Making Formal Property Verification Mainstream: An Intel® Graphics Experience," DVCon 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] Andrew Seawright, Ramesh Sathianathan, Christophe Gauthron, Jeremy Levitt, Kalyana Mulam, Richard Ho, Ping Yeung, "Selection of initial states for formal verification," US7454324.
[7] Questa PropCheck, https://verificationacademy.com/courses/Formal-Assertion-Based-Verification