With innovations in technologies and methodology, the benefits of formal functional verification apply in many more areas. If we understand the characteristics of areas with high formal applicability, we can identify not only which blocks are good candidates, but also what portions or functionalities of the blocks will give the greatest return on the time and effort invested. Today, formal verification can be more valuable applied partially within blocks by choosing the functions that have the highest return.

**Identify high-risk areas**

There are three objectives for use of formal tools: first, to accomplish what conventional methods cannot, such as proving that a particular property is always true; second, to achieve what other methods can do, but significantly faster; and, lastly, to enhance current tools and methods by providing another approach to achieve full coverage.

Regardless of objective, formal's return on effort must be compelling. Take the following bridge design as an example.

Figure 1 shows an L2/L3 Ethernet switch derived from a previous generation of L2 only switch. Some blocks such as the MAC, CSR, PCI, and the memory interface are legacy code. Other new blocks such as the L3 are verified extensively at the block-level because it is difficult to have good coverage on the system-level for all the different L3 traffic, and all new code is more vulnerable.
All blocks and functions are not equally critical. The ingress parses the L2 and L3 headers and performs the relevant checks (CRC for L2 and checksum for L3) to decide whether a packet should be discarded. The L2 parsing and checking are not new code, meaning little associated risk, but the L3 has higher risk because it is new code. Even within a block, some functions, if not behaving correctly, might have a minor performance impact, while others might cause major system problems.

Late-stage specification changes also introduce a major risk factor for certain parts of the design. Because the L3 block design can support many possible policies, it is possible there will be some late specification changes to add policy configurations. The risk is not just less verification, but the changes might also cause collateral damage to existing working functions. Applying formal here, to verify new functions and regress existing functions, becomes an attractive solution.

To apply formal, we should prioritize resources on the three factors mentioned above. If we have enough resources to apply formal to a certain number of blocks, we can prioritize our candidates and degree of verification.

In this example, the L3 block is a good candidate for complete formal. A good block for partial verification is the ingress block. The CSR block is the lowest priority.

The benefit of formal verification is greatest if we can fix all the block-level functional bugs prior to system-level verification, where bug-fixes are more time-consuming. Having the confidence in completeness for certain functions also allows the team to focus on other areas of concern during system-level simulation.

Blocks that are difficult to verify

In the Ethernet example, there are some blocks where it is difficult to get good simulation coverage. The bridge block must handle all different types of packets (lengths and types) and arrival time, flow-control from egress, and so forth. It is difficult for simulation to achieve high confidence with this many input possibilities.

A design that has many different configurations can be difficult to verify. For example, the configuration for this Ethernet switch can have any number of active ports, different types of policy threshold, and various external memory sizes. To verify all modes in their entirety is not practical. Instead, we usually choose a few representative modes for detailed verification. This example represents a case where formal has a great deal of potential value.

Exhaustive simulation is difficult with a standard protocol when the design needs to interface with another ASIC because a standard protocol tends to be more general; hence, it has many requirements associated with it. A typical verification environment does not exercise all the aspects of the protocol.

Applying formal verification

We hear about limitations associated with formal verification, yet many areas historically considered not applicable to formal have now become the main targets. The breakthrough comes both from engine and methodology improvements. Formal methodology has become a main enabler for formal verification. Consider the following example:
The block is a reordering buffer that buffers transactions sent to and from the memory. It relies on the following rules:

- If a newer entry is within the same burst as the oldest entry, it will be reordered to combine with the oldest entry.
- If a newer entry is within the same row as the oldest entry, it will be reordered to be next to transfer.

One obvious way of coding the expected behavior is to implement a reference model because it seems that we will need to search through the queue to find which entry should be the next to go out. To do that, we can keep a queue structure similar to the reference model and predict the output. Alternatively, we can use the internal signals and look into the content of the queue to make a decision.

There are drawbacks to these approaches. First, the reference model's complexity will be similar to the design's complexity making the problem twice as large. Second, the reference model can sometimes be too close to the implementation model if we create a cycle-accurate model.

Third, any modification on the specification results in a major rework on the reference model because the implementations of different functionalities tend to be interdependent. Finally, it is usually a tall order to ensure the accuracy of a reference model. For these reasons, engineers often say that it is almost as much work to build and maintain a reference model as the design itself.

Alternatively, we can specify properties to check that the queue reorders correctly. Here is one strategy:

1. Register the last entry entering the queue (E1) and the number of outstanding entries in the queue (N1).
2. If the entry exiting the queue is of the same row and burst, the next entry exiting the queue should be the E1.
3. If the entry exiting the queue is of the same row, the next entry exiting the queue should either be E1 or an entry with the same row and burst.
4. If the entry exiting the queue is not of the same row, the next entry exiting the queue should not be the E1 unless N1 is 1.
5. No entry should exit after E1 if N1 is 0.
Applying this approach breaks down the problem into several smaller properties, rather than one reference model. With specification changes, many of the properties are still applicable. We simply need to add and delete properties to the existing set. Therefore, proper planning and methodology can enable successful formal verification deployment.

**Applicable and non-applicable functions**

Formal verification applies to arbiters, although few apply it properly for complex arbitration schemes. For example, the arbitration priority of a port increases upon certain events and decreases upon other events. To consider all those events for all ports makes the property quite complicated.

We should instead be looking at the arbiter as two separate sets of functions: (1) the arbitration scheme is based on the current priorities of the ports and (2) certain events increase and decrease the priority. If the two functions are verified by separate properties, it becomes two rather simple problems.

Moving up one level of abstraction, an arbiter is actually one form of decoder that, when properly applied, has a relatively small sequential depth (referring to the number of cycles it takes to reach all the states within the arbiter).

Consider the following example.

![Diagram of arbiter and decoder](image)

This example brings us to a myth in formal functional verification:

**Myth 1: Decoders are not suitable for formal verification.**

Arbiters are generally considered one of the sweet spots for formal verification. And if we consider arbiters as a form of decoder, we contradict this generally accepted statement. The key to applying formal to decoders is to apply it in isolation.

Another consideration for decoder verification is how implementation-specific the decoder properties are. If the decoder is not an area of concern, in that the decoding function is very implementation specific, there might not be a need to apply formal. Some decoders simply decode the address space to determine a hit or miss, not requiring much verification. If the decoder is complicated and you can specify it in a different way (compared to the implementation), it might be worthwhile. Examples are scrambles and 8b10b decoders. However, if the decoder logic is overly complicated, such as some digital filters or floating-point algorithm, it is not the right place to apply formal, at least at the functional level. The properties of such logic are very difficult to specify, and the logic is also very complex for formal verification.
Now consider another classic sweet spot for formal: interfaces. Interfaces are considered good for formal because they have a large amount of concurrency, but are not sequentially deep. The properties are also straightforward to implement.

Many standard protocols also exist in a master/slave relationship, which has the constraint/assertion relationship. Therefore having one set of constraints/assertions, we can verify both the master and slave interfaces.

Many new interfaces use packet and cell type structures to transfer commands and data. The packet/cell-based protocols cause the sequential depth to be high and make it difficult to model properties. The handshake is no longer among signals, but among different packets or cells. Engineers often shy away from these types of interfaces, but they are not very different from verifying signal-handshake types of interfaces.

Consider the Ethernet design in Figure 1. It actually has three levels of interfaces.

![Figure 4](image)

Here the physical layer is no different from a traditional interface, while L2 and L3 are not as straightforward. To extract the controls for these "interfaces," we must parse the relevant packet headers to obtain the controls information. Consequently, one simple handshake that otherwise takes a few cycles now takes a few packets that can be 100s or 1000s of cycles. This might seem impractical for formal.

**Myth 2: Packet-based design is not suitable for formal due to its sequential depth.**

We can solve this problem using a good methodology and tool. We should avoid combining inter-layer protocols into one property set. For the Ethernet example, when we verify the L2 protocol, we have no doubt that we need the physical layer to capture the packet.

However, there is no reason to combine the analysis of L3 header and the rest of the payload within the L2 properties. Careful partitioning of the property in that manner can minimize the need for most of the states related to L3 or the payload portion of the logic. Sometimes this requires that the tool you use is smart enough to analyze the effective cone-of-influence (COI) beyond the traditional COI computation.

The modeling layer of the property is also important for capturing the relative layer of the protocol.
The modeling technique can make a significant difference in the outcome, but is beyond the scope of this article.

Now extend the interface characteristics to something more common, a cause-and-effect type relationship. Identifying and enumerating such behaviors can simplify the verification of an otherwise complex design. For example, interrupt handlings are actually a cumulative handling of individual interrupts; hence, they can be enumerated.

While instruction handling can often be complicated, we can verify some of the most important functions, such as exception handling and branch-prediction, which the processor architecture usually clearly describes.

**Myth 3: Avoid using formal verification with data-transport blocks.**

Data-transport is a property that exists in many designs, but engineers generally consider formal non-applicable because data-transport usually involves two characteristics: storage elements and large sequential depth.

The most straightforward way is to put local assertions along the datapath. This is a way to take baby steps, providing value. We should, however, verify the entire path without breaking it down whenever possible. It makes the number of properties a lot smaller and frees you from having to think of all possible implementation-specific properties along the entire path.

Recent innovations in technology and methodology have made end-to-end data-transport properties possible and desirable. We can overcome sequential depth issues by applying the methodology described earlier for packet-based designs. Storage elements, not usually the area of concern, can safely be abstracted.

One area of caution for data-transport verification properties usually focus on catching incorrect or corrupted data at the output. However, if the data never comes out at the output, there will be no violation. This involves some aspect of liveness such that the data should eventually come out.

Liveness properties are one area not recommended for formal as they have two main problems. First, usually some fairness constraints must accompany them. For example, there are usually flow controls along the datapath which, in theory, can be asserted forever.

We must include fairness properties such that they will be released from time to time. Identifying such areas through the design can prove to be a tedious task. Second, most formal tools do not support liveness properties, and for the few that do, the capacity is too limited (in liveness aspects) to be useful. You can model most liveness properties as safety properties, which can be effectively verified. With the proper methodology, you can ensure correctness.

**Conclusion**

To get the most out of formal verification, we need to assess applicability of formal on the type of functions within a block, rather than the design type. Determine applicability based on the following factors: difficulties posed for other means of verification, risks associated with the specific functions, and formal applicability to specific functions.

Another aspect to consider is that the methodology and tool are equally important in the success of applying formal verification. Finally, do not make a decision based on a particular design block in an all-or-nothing manner. Most blocks have at least some functionality that is applicable to formal. With proper planning of your verification approach (which includes test planning), proper methodology,
and tool selection, you can significantly benefit from formal functional verification.

Lawrence Loh is currently the director of applications engineering at Jasper Design Automation, responsible for worldwide applications engineering and methodology development. Prior to Jasper, Mr. Loh worked at MIPS Technologies where he held responsibility for verification and emulation of embedded processors. Previous to MIPS, he was a verification manager for the LAN Business Unit at Infineon. He has also held positions at Ikos, where he served as an applications engineer, and at Hughes Network Systems, where he served as an ASIC designer.