

# Veriest Solutions

There are bugs you can find only if you know exactly where to look. And then there are bugs you would never think of looking for in the first place. That is exactly where **Formal Verification** comes in.

At Veriest, our Formal team approaches complex blocks and systems from a different angle. Instead of running scenario after scenario and hoping we happen to hit the right one, we ask a different question:

## **What can actually happen here?**

Not only in the scenarios we planned. Not only in the sequences written in the tests. And not only in the cases someone thought were worth checking.

For suitable blocks, Formal Verification can go significantly beyond simulation. It can provide a **full proof**, including corner cases that would be extremely difficult to reach through simulation alone. Formal can also begin finding bugs **before a complete simulation environment has been built**, enabling much earlier bug discovery.

Formal is not limited to testing **SVA properties** through Formal Property Verification (**FPV**). We apply it to a wide range of verification problems: checking **connectivity** across distant parts of a design; verifying **register behavior (CSR/FRV)** flows, including values, access rules, updates, resets, and dependencies; and proving complex datapaths and arithmetic (**C2RTL/DPV**), including multiplication and floating-point operations. We have also used applications such as **X-propagation** analysis, **protocol checkers**, and **Linting** tools.

These applications show the breadth of Formal Verification. Delivering results across them, however, depends on more than selecting an app: it requires solving the convergence, modeling, and environment challenges specific to each design.

A property may be valid and still fail to converge. The reachable state space may be too large; the selected proof engine may not match the problem structure, or part of the logic may need to be abstracted. The proof may also need to be decomposed differently, or the design may require a verification strategy that has not yet been developed.

## **This is where experience matters.**

We use **abstractions** to reduce the search space while preserving the behavior relevant to the proof. We **select proof engines** according to the structure of the problem rather than treating every property the same way. We also use **case**

# Veriest Solutions

**splitting**, parameter **reduction**, and carefully chosen configurations to turn large verification problems into tractable ones. When an existing methodology is not enough, we develop a new one. This includes new approaches to timing verification, such as the **P2S methodology** presented in our [DVCon 2025 paper](#).

The difference can be dramatic. In one of our DPV projects, a Floating-Point Multiplier proof for FP32 and FP64 failed to converge even after more than 12 hours. Simply giving the tool more time was not the answer. We identified the arithmetic structure inside the RTL, selected a dedicated engine, introduced the appropriate abstraction, and split the problem according to the relevant floating-point cases. The result? A problem that did not converge after **12 hours** was reduced to **roughly 1 minute** for FP32 and 7 minutes for FP64.

And that highlights one of the most important lessons in Formal:

**Good Formal is not just about knowing how to run the tool. It is about knowing how to shape the problem into something the tool can solve.**

For non-converging properties, another effective approach is targeted **bug hunting**. When the property doesn't converge, that means that there were no bugs found in the earlier cycles. Bug hunting enables you to find bugs in later cycles. Since the tool doesn't know on its own where to find these bugs, it works best when the verification engineer guides the tool where would be the possible places for bugs to hide.

**And finding bugs is only part of the job.**

We also verify the verification itself. Through **Formal Coverage checks** such as cone of influence (COI), proof core, and mutation analysis, we examine whether our properties observe the logic they are intended to verify. Did we cover the relevant design space? Which parts of the logic were necessary for the proof? If we deliberately introduce an incorrect change into the RTL, will the property suite detect it? A passing property is useful, but a property suite that has been challenged and shown to detect meaningful design errors provides something much stronger. **The goal is not merely to have passing properties. The goal is confidence.**

**What happens when there is no prepared environment?**

In such cases, **we create a customized environment** to enable Formal Verification. Knowledge of how to run Formal Verification alone is not sufficient for these challenges. They require deep environmental scripting expertise combined with strong formal verification knowledge.

# Veriest Solutions

Connectivity applications typically require data in CSV or IP-XACT format to set up the required tests. In one project, the required connectivity information was distributed across multiple unorganized Excel files. We organized and processed this information and **built custom environment scripts** to automatically generate the CSV or IP-XACT data required by the Formal flow.

Normally, a C2RTL project requires a C or C++ reference model. However, if the reference model is written in Python, existing Formal tools do not directly support it. To address this limitation, **we developed our own methodology** that enables us to use the existing Formal tools to verify RTL against a Python reference model instead of a C/C++ reference.

In another project, we needed to evaluate and verify new Formal proof engines. Existing Formal tools did not provide the required level of user access for this evaluation. Therefore, by combining deep formal verification expertise with AI techniques, we **developed our own formal tool** capable of generating CNF representations, allowing us to perform the required evaluation and verification.

**Ultimately, the distinctive value our Formal team at Veriest brings is the ability to build a verification methodology or environment tailored to the challenge at hand.** Experience with Formal tools, a deep understanding of the problem, and the ability to choose the right verification strategy are essential foundations. Our more advanced capability is turning those foundations into a complete, purpose-built approach for new or especially complex verification challenges.

Whether you already have a basic Formal verification setup in place or need to build one from scratch, our approach remains the same: understand the design, understand the verification challenge, and find the most effective path to proof.

**Because in Formal Verification, the real expertise begins where the straightforward flow ends.**