About Formal

# Formal Verification History

Our present day way of verifying chips all started when Cadence acquired Gateway DA back in 1989 -- and it started selling the Verilog-XL simulator. Prior to 1989, the closest thing to chip simulation was an old 1973 FORTRAN program from U. Berkeley called "SPICE1", but it only did nodal analysis of analog circuits. Cadence Verilog-XL was the world's first digital HW simulator that worked at the digital gate-level (or higher).

At first, Verilog simulation was used to do simple design assurance testing, making sure your chip's RTL code functionally matched the original design intent as described in your chip spec. "This is a simple adder. If I feed it 2 + 2 as inputs in my Verilog simulation, does it output 4?"

As design sizes grew, engineers began "bug-hunting" in Verilog by creating directed tests to uncover bugs for problematic corner cases.

Then Synopsys created Design Compiler, which automatically converted your chip's Verilog RTL source code directly into gates. And it had everyone asking:

"How do I know that my Design Compiler output in gates is functionally the same as the Verilog RTL source code that I fed into it?"

And this is where, for most chip designers, formal tools first appeared on the scene in 1999 as the Verplex Conformal LEC logic equivalence checker.

But for many chip designers, having any formal tool say "that chip is OK" didn't help at all. Why? Formal was considered to be rocket science that worked using peculiar mathematical interpretations. Engineers like to understand what's going on "under the hood" of a tool, and formal felt like a black art.

Instead of answering the "do I trust this Design Compiler output?" question, it just moved the problem into a "do I trust this formal tool" question.

But engineers are practical.

As time went on they learned to trust formal because it worked. If a tool makes their life easier, they will use it.

Over the past 20 years formal has grown in chip design verification due to:

- Assertions and properties are an accepted part of verification. Also since the early days of 12 assertion types , the chip verification community has de facto standardized on roughly 90% SVA use and 10% PSL use.

- Exhaustive state-space testing is something chip designers really like. Verilog/VHDL simulation plus debug tools plus linting is still useful for chasing bugs -- but they're not exhaustive. Formal is.

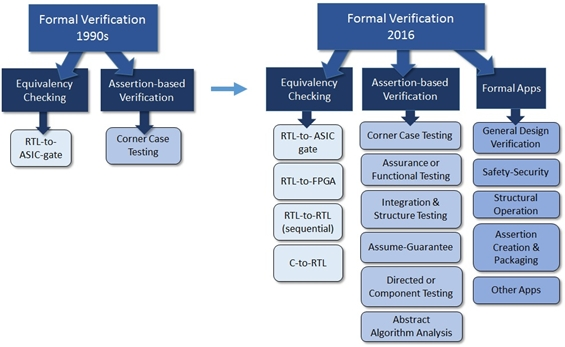
- Formal plays nice with simulation. Engineers can use formal output in VCS/QuestaSim/Incisive and vice versa. Formal complements sim.

- Formal Apps make using formal easy for non-formal users. Now you don't even need to write assertions because some app will do it for you.

- Faster CPUs and there are more new formal algorithms than before. Formal is now even used to detect unauthorized accesses from one part of your chip to another part of your chip -- something unheard of 20 years ago.

These factors have made formal common. In the old days Jasper shipped an AE with every product. Now you no longer need a guru on staff to use formal.

Here's how formal has grown from the mid 1990's to 2016:



To drive home my point again, this is not your father's formal.

## CHIP VERIFICATION SECTIONS

EC, F-ABV, AND APPS The 3 major sections where formal is used in chip verification today are:

- Equivalency Checking (EC)

- Formal Assertion Based Verification (F-ABV)

- Formal Apps

Below are quick engineering backgrounders for these 3 divisions of formal.

## EQUIVALENCY CHECKING (EC)

Equivalency checking (EC) involves formally comparing one representation of a design against another to verify that the functionality is the same. Some key ways equivalency checking is used today:

1、Comparing design representations for functional equivalency at different design abstraction levels. EC can further be distinguished by:

- RTL-to-ASIC Gate Level. Checking your RTL code against your post-synthesis gate level code of an ASIC design. Since neither Design Compiler nor Genus ASIC synthesis changes your register configurations -- your formal tool only focuses on the combinational logic between registers.

- RTL-to-FPGA Gate Level. Checking your RTL code against the post-synthesis gate level code of your FPGA design. Here, Synopsys Synplify, Xilinx Vivado, and Altera Quartus FPGA synthesis change the register configuration -- and your formal tool must also take the timing differences into account when checking functionality. This is known as "sequential" EC.

- C-to-RTL. Checking your SystemC/C/C++ code against your RTL code following high level synthesis (HLS) to see if your design intent is upheld. SystemC designs usually have very limited timing information -- they rely on Cadence Stratus or Mentor Catapult HLS to add timing detail to your final synthesized RTL code. As such, this is also a sequential operation. Calypto SLEC is commonly used here, but it's rumored that OneSpin is also working on SystemC sequential EC also.Throughout these changes, you need to make sure that your design's behavioral intent is maintained/honored.

2、Comparing different RTL code or gate level code of the same design to make sure they have the same functionality.

- RTL-to-RTL EC or Gate-to-Gate EC. Used after code optimizations.For example when you do power optimizations, make small ECOs, or add scan test logic. Cadence Conformal and Synopsys Formality are typically used to do this. In addition, sequential equivalency checking is often needed here, as your register configuration may have been altered along the way. This is where Mentor SLEC and OneSpin EC-RTL, JasperGold, and Synopsys Hector come into play.

EC was formal's first commercial "killer app", starting with the founding of Verplex in 1997.

## FORMAL ASSERTION BASED VERIFICATION (F-ABV)

The goal of Formal Assertion Based Verification (F-ABV) is to make sure:

- your design (or code segment) functions according to the spec, and

- your entire spec has been fully implemented.

F-ABV verifies that the properties of your design source code functionality match those described in your spec. These properties of your design are described using assertions written in either SVA or PSL.

Formal ABV is also used to verify design intent under all implementations. That is, it tests to see if your original design functionality is still maintained regardless of any micro architecture changes.

Formal ABV is about asking design functionality questions, such as "Can my code do X?" or "Will my code ever do Y?" -- which the formal tool will answer by examining the entire state-space of the design.

For example:

Q: If block 1 sends a "request" to block 2, will block 2 always send an "acknowledgment" signal within X cycles?

A: The formal tool can tell you one of two possibilities:

- yes (or yes within the number of clock cycles that can be checked)

- no (and shows an example where this will not occur)

In its early days, formal assertion-based verification was mostly limited to verification engineers doing corner-case testing -- in situations where it was hard to create intricate scenarios using Verilog simulation.

Today F-ABV is used by all engineers for directed testing, functional block tests, IP integration, protocol compliance, safety critical designs, etc.

## FORMAL APPS

Writing assertions is a complicated and time consuming task for designers. To their delight, there are now formal apps which, for specific situations, automatically create assertions. Some formal app categories:

- General Design Issues like register checking, connectivity checking, structural property generating, unreachability, scoreboarding.

- Safety/Security like fault analysis, unauthorized accesses.

- Structural Operation like CDC, power management, X-propagation.

- Assertion Creation like sim-trace, protocol compliance, arithmetic precision, equivalence checking.

For an app example, let's look at a typical "Register Checking" formal app. Registers are sets of flip-flops buried all over your design code. For a large design, you may have 100s or 1000s of registers.

- Each register is uniquely identified using a unique address.

- All registers and their addresses are defined in a register (or memory) map - a file which is 100s of lines long.

- The actual register locations must be consistent with the map file to ensure consistency between hardware and software; it's common to make a mistake between the different versions of HW or SW.

The "Register Checking" formal app reads in your Verilog/VHDL design source code and your map file, then automatically:

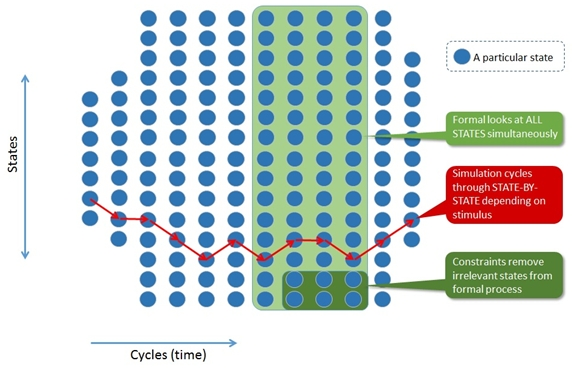
- Generates appropriate assertions (this may or may not be hidden)

- Uses a formal engine to test that each register is at its expected location and is operating as expected.

Careful note: RTL simulation is not even used here. It's purely formal.

# Where Formal ABV whomps HDL simulation for chip verification

The truth is Formal and Verilog/VHDL simulation go quite well together. While simulation is time-based, and formal is state-space based. Any verification plan that uses both simulation & formal will hit its coverage goals much faster because together they perform a broader range of tests in much less human time. Anyway, my true goal here is to show your readers how Formal Assertion-Based Verification (F-ABV) has expanded into chip verification -- and where it beats old school standalone Verilog/VHDL RTL simulation.



The general rule is:

- FORMAL operates on all states concurrently over short bursts of time. For example, if you have 27 scenarios which must be examined at the same time (e.g. 27 branch conditions), formal works well and will examine each of these branches to a typical depth of 30 clock cycles. And it can go up to 150 clock cycles in, depending on the code.

- SIMULATION can be thought of as operating on limited set of states over large timescales. For example, if you have one sequence going state-by-state over a 1000 clock cycles, simulation is far better.

## Fundamental differences between Verilog/VHDL simulation and F-ABV

### Simulation

Verilog/VHDL simulation requires time-based stimulus to transition your chip around a specific sequence of states. You write test vectors to get the simulator to cycle through a series of design states to reach a particular operational scenario of interest. With early simulation, you reach a reasonable level (~80%) of coverage quickly, because a few tests will run a design through most of its basic operating scenarios.

However, to get a high coverage level (close to 100%), a much larger number of tests is needed to get the design into the corner cases that early simulation doesn't touch. Writing testbenches to do this is a lot of work, and it's also humanly impossible to test all possible scenarios that a chip can get into.

### Formal Assertion-Based Verification (F-ABV)

Formal ABV operates in the state-space. It has a record of all states a design can get into, so it does not require stimulus to drive the design into a specific state. You use assertions to ask questions about different operational state scenarios, and receive an exhaustive reply as to every state related to the scenario in question.

As you increase the number of cycles, the number of state transitions for your design grows exponentially. This is why engineers like to use formal on the more obscure scenarios, so they can be tested.

To improve formal tool performance, engineers use constraints to focus it away from states which do not occur in real life. "Why waste formal tool CPU runtime testing states that the chip will never get into?"

F-ABV usually beats out Verilog/VHDL simulation when you use:

- Finite State Machine (FSM) logic and other control structures

- Communication or handshaking protocols where specific actions and reactions are expected over time

- Data transport mechanisms, such as pipelines and bus bridges

- Check logic that's wrapped around certain design structures, e.g. FIFO full/empty logic

- Bug hunting. Testing for corner-case scenarios in a large block -- especially where it is hard to drive the design into the state where this corner-case may be observed

- Testing stuck-at faults across various operating conditions

Formal ABV and simulation are already commonly used in the same design flow. The traditional approach is often to do simulation first, then look at ways to close the coverage gap using F-ABV. However, it's more efficient to look at the full design up front in F-ABV first to quickly identify the scenarios where F-ABV is more useful.

## SIX WAYS FORMAL ABV IS BETTER THAN SIMULATION

Let's dive into more detail on where design teams generally find F-ABV is more efficient than simulation.

1. General Assurance or Functional Testing

F-ABV is particularly useful for verifying state-intensive blocks such as FSMs and control logic. Here assertions are created to get full overall coverage on a design block -- and is particularly effective when there's a complex state machine involved which must be exhaustively tested.

1. Corner Case Testing or Bug Hunting

F-ABV is often used to test certain scenarios that are hard to recreate during Verilog / VHDL simulation, due to the complex stimulus required to get the block into the problem state. This is especially true when many parts of the design must be manipulated to reach the target problem state. Formal is then used to augment HDL simulation because you can specify a particular problem corner case using constraints and assertions.

This is so thorough that it's used to test nuclear power plant controls.

1. Assume-Guarantee

If you have Verilog/VHDL code that's too big for Formal ABV to digest, you can partition that source code into smaller, more manageable blocks.

For example:

- You break your design into 3 blocks that are chained together in sequence, such that the 1st block drives the 2nd block which drives the 3rd block.

- You then verify the 1st block with F-ABV using an initial set of assertions and constraints specific only to that 1st block. This verification task then creates a new set of assertions called "1st block output" assertions.

- You then convert those new "1st block output" assertions into constraints which are fed as "2nd block input constraints". The assertions for the 2nd block are now much simpler for F-ABV to create because they have been constrained from the 1st block.

- Next, the 2nd block's output assertions are turned into constraints which are fed as inputs constraints to the 3rd block.

- And so on.

This is easy to do because the SVA assertion format and SVA constraint formats are very similar. This "Assume-Guarantee" approach makes it easier to develop assertions for large design elements. Assume-Guarantee comes from the idea that you are testing assumptions about blocks, then propagating them downstream through the design.

1. Integration or Structure Testing

This is using Formal ABV to test if your IP blocks are all wired up to each other correctly, plus it goes beyond that to check for functional inconsistencies during operation, based on the chip's architecture. An example is: before RTL synthesis, you have a 4-bit register array. A 4-bit array can only be addressed from 0 to 15. F-ABV checks to see if bad addresses 16, 17, 18... come in. These are the cheap seats for formal, because it's a replacement for simple linting. However, the additional checks for operational errors due to connectivity mistakes are something that linting doesn't do. It's heavily used to do protocol compliance checking with established interface standards like ARM AXI, PCIe, AHD, and APB.

1. Directed or Component Testing

Most of the time, it's verification engineers who are the ones who use F-ABV. Directed testing is used by design engineers. In this case, the designers use formal ABV to pre-verify small IP components for specific functionality, prior to integrating them into a larger block. With this directed test method, the designers then either provide the assertions to the verification team with the IP, or use the assertions to demonstrate a sign-off criteria. A recent twist is F-ABV runs examples of chip operation with minimal input, and then use these same examples to generate simulation tests. You create your assertions, run the formal tool, verify it's correct, then you feed those assertions into your RTL simulation. And yet another new tweak to this twist is that your formal tool also now generates a Verilog/VHDL simulation testbench. Designers love this.

1. Abstract Algorithm Analysis (C/C++ Property Checking)

This is where F-ABV is used on SystemC/C++ algorithm code targeted for High Level Synthesis (HLS). The assertions here only have limited sequential detail which corresponds to the high level specification of algorithm behavior. This use of F-ABV is not limited to SystemC/C++. It can be applied to any code written at a high level abstraction. Engineers are working on using these same assertions post-synthesis on fully-timed RTL code to test that the algorithm's behavior has been preserved. The problem here is that they must write assertions that are relevant to both the untimed pre-synthesized code and the cycle-based HLS output -- not an easy task to do.

## FIVE WAYS TO CREATE ASSERTIONS WITHOUT MANUALLY WRITING THEM

System Verilog Assertions (SVA) and PSL both have a rich syntax which lets you create full temporal assertions ranging from simple operation checks to extended transcriptions of specification elements. One major roadblock to using assertions was that it took a lot of engineering man-hours to write them. Engineers hated this extra work. This section is on the available options engineers have to save time in creating SVA or PSL assertions automatically instead of by hand.

1. Generating from RTL design code

There are assertion (or property) synthesis tools such as Synopsys NextOp and Cadence JasperGold which attempt to synthesize assertions directly from your chip's RTL source code and simulation tests. There are also formal apps which do the same thing. Other vendors sell tools that automatically generate assertions to test for specific common design issues (e.g. "array out of bounds" access). They extend automated testing beyond plain old HDL linting by checking the operation of your chip -- in addition to code syntax and semantics. These tools include OneSpin Inspect, Real Intent Ascent, and Mentor AutoCheck.

1. Abstract/Graphical Representations

A number of schemes exist which allow assertions to be written in an abstract or graphical fashion. These include IBM Diver and Cadence Jasper Visualize for graphical entry. For abstract entry, OneSpin Operational Assertions is a SV class library to provide transactional assertions. You write abstract SV functions which are then automatically converted into lower level assertions. The Accellera Portable Stimulus Group is currently focused on simulation stimulus, but is also rumored to be discussing higher level assertions.

1. Alternative Specification Models

In #2 above (Abstract/Graphical Representations), the engineers create assertions directly -- ans they are thinking in terms of the assertions themselves.

With an "Alternative Specification Model" the engineers are instead thinking in terms of their design spec. They write a separate model in a special proprietary language that is then converted into testbenches and assertions.

Examples of this approach are Logic Refinery RuleSmith, and arguably Cadence Verisity Specman 'e' (although there's no 'e'-to-SVA translator).

1. Automated Tools & Formal Apps

In another section of this report, we discuss how 16 types of formal apps generate assertions based on a specific design and/or for a specific complex verification task. Some companies allow these assertions to be visible and editable, whereas others do not. Not all companies offering these formal apps are formal verification vendors (e.g. security verification company Tortuga Logic).

1. Assertion Verification IP (VIP)

Most formal verification vendors, along with a number of consultants and IP providers, sell assertions packaged into Verification IP (VIP) to test specific design scenarios, busses, and protocols. "Why should I have to write PCIe assertions when we can just buy them?"

Cadence, Mentor, OneSpin, Synopsys all offer a wide range of VIP models.

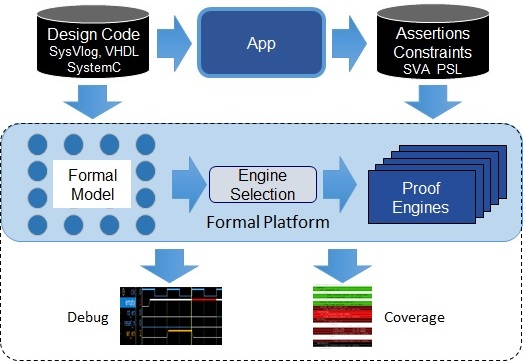
What follows next is a detailed technical discussion of the metrics and gotchas of F-ABV tools.

# 9 major and 23 minor Formal ABV tool metrics - plus their gotchas

Here are the 9 major metrics that I feel an engineering team must consider when choosing what specific Formal ABV to use on their project (if any):

1. Performance/Capacity/Convergence
2. Debug
3. Set-Up and Control
4. Property Types
5. Standard Language Support
6. Coverage Mechanisms
7. Formal/Simulator Interoperability
8. Formal Engine Access
9. Customer Tech Support

Here's a generalized F-ABV flow. Start at "design code" for input:



Notice that "debug" and "coverage" are the final output as well as info on whether the assertions have passed. Using this framework, below I explain in detail the impact and pitfalls of 9 major and 44 minor metrics will have on your F-AVB tool choice.

## PERFORMANCE/CAPACITY/CONVERGENCE

Measuring a formal tool's capacity and performance is tricky, as the tools are highly dependent on your individual input design code and assertions. The rate of convergence (the number of assertions that actually complete) is also a factor, because some tools never converge for some checks. The convergence rate is based on effectiveness of the 15 to 20 possible formal engines used in your formal tool -- as well as your specific design and the quality of its assertions themselves. Convergence can range from 70% right up to 98% (i.e. only 1 in 50 assertions not completing).

Companies often do a lot of benchmarking to get a good handle on these three factors. However, formal is very sensitive to benchmark bias as assertions are often written with a specific formal tool in mind. For example, what works in Questa may not work in JasperGold. Using those same assertions on a different tool can lead to poor results if the characteristics of the new tool are not considered.

Characteristics which impact capacity, performance, and convergence:

- Underlying Formal Proof Engines

Unlike Verilog/VHDL simulation, where you have one core engine for each simulator, with formal tools you have a collection of proof engines; each leveraging different algorithms. These engines are applied based on the specific design and assertion characteristics for best results. Some common public algorithms inside formal engines are:

- Binary Decision Diagrams or BDD

- "SATisfiability" Proofs or SAT

- Bounded Model Checking or BMC

- Craig Interpolation, usually shortened to "Craig"

- Property Directed Reachability or PDR

Also, proprietary algorithms for special situations are common. Often the more mature the tool, the more engines with different algorithms it will contain. For example, Cadence JasperGold and OneSpin Verify both have approximately 15-20 formal engines/algorithms.

- Formal Engine Selection

How a specific formal engine is selected is a key factor. It used to be engineers did this by hand choosing from 4 to 8 different algorithms with limited information about the best one to use. Today, some formal tools have an automated engine selector which logically picks from ~20 choices -- and some vendors are working on adding heuristic or experience-based mechanisms which use both your Verilog/VHDL source code and assertions as their basis of formal engine selection.

- Formal Model, or Data Structure

The formal model is the proprietary data structure used to store your chip design in memory. They contain your design's state and transition information. These models (or data structures) are often optimized to minimize the storage requirements -- while still allowing for rapid access to specific info. It can easily take 100's of engineering R&D man-years to build an optimal combination... with the right combination being the "secret sauce" for consistent performance, capacity, and convergence.

## DEBUG

Debug was historically a major limitation of formal tools. For a specific assertion, the formal tool would cryptically say "pass/fail/issue" with limited (or no) info on exactly why the design failed. Today formal vendors are making strides to have their formal tool debug more closely mirror the debug-friendly Verilog/VHDL simulation environments. What is important look for?

- Graphical Debugger

Many Formal ABV tools today now offer some sort of GUI with features such as: waveform display, source code display with annotation, driver analysis, and code navigation. Examples are Mentor Questa debugger, OneSpin Debugger, and the Synopsys Verdi environment.

- Debug Trace Generation

A trace is a sequence of events that make up a particular source code operation leading to an assertion failure. Most formal tools can generate a trace, or a counter-example, to demonstrate why an assertion fails. It can then use a simulation-style debugger to show the signals that relate to the failed assertion. The debugger can be proprietary to the formal tool provider or a third party solution like Synopsys Verdi. The debugger may also include some form of root cause analysis to allow the original source of the issue to be discovered quickly. Some formal tools can also auto-generate simulation stimulus from the trace, so that the engineer can further debug the problem in simulation.

- HDL Testbench Generation For RTL Simulation

Some F-ABV tools can generate another type of trace called a "witness" of a specific code operation scenario of interest -- and then create a standalone Verilog simulation testbench component of it. For example, OneSpin Verify can create a witness that may be fed directly into a testbench for Verilog simulators like Synopsys VCS, Mentor Questa, Cadence Incisive and Aldec Rivera. This is a fast way of identifying important operational details and automatically producing a simulation directed test for that detail.

## SETUP AND CONTROL

How do you measure how long it takes to get a Formal ABV tool up and running on a chip design? Some automated parts of tool set up are:

- Assertion Creation Templates : Templates that make it easier to write specific assertion types, such as FIFO checks, FSM Checks, etc. This can eliminate the 1-2 days it can otherwise take to write 10 good assertions for a FIFO test.

- Automatic Clock/Reset Recognition : Some F-ABV tools will recognize "reset" and clock signals, and make them be easily controlled. Although this often takes only 1-2 hours, they may have to be redone every time a significant block is added to the design.

- TCL : All tools today use TCL for scripting; however due to proprietary tool commands, TCL scripts are not interchangeable between F-ABV tools.

## PROPERTY TYPES

The design intent of a digital chip may be thought of as design properties.These are the fundamental characteristics of the design that the formal tool will verify. For formal verification, these are written as assertions. Some property types are:

- Safety Checks : Ensures your design behaves correctly and there are no circumstances where an error condition might occur. Example: Check FIFO control logic to make sure there can never be both a "full" and "empty" indication at the same time.

- Activation Checks : Ensure that specific code sections can be accessed and the verification problem is not over-constrained (related to the functional coverage below). Example: For a 30 state FSM, check that all 30 states may be reached as expected -- i.e. no mistakes have been made in coding the arcs between states.

- Functional Cover Checks : Makes sure that a specific operational scenario is actually tested during verification. Example: In a block-to-block handshake, make sure that an ACK signal is observed to happen during verification every time a REQ is generated and a legal data packet is transmitted.

- Liveness Checks : To check whether some event will eventually happen. Example: In an interrupt controller, make sure that every interrupt signal will be processed eventually -- even if it happens after many cycles.

- Structural Checks : Makes sure the structure of your design post-synthesis is correct (e.g. ensuring your "case" statements in your Verilog source code match your design intent), and that there are no structural errors due to incorrect RTL coding. This is not just wire connectivity checking. It's also functional checking. Example: You have a 4-bit register array. It can only be addressed by 0 to 15. Formal structural checks make sure that it's not indexed by 16, 17, 18... or any other invalid addresses.

If left alone a F-ABV tool will verify every possible state of your design with every possible input. However, there are some input possibilities the engineer knows in real life which can never occur -- for example, "reset" signal at the same time as an active input. Your F-ABV tool run will be a lot shorter if you use constraints to remove these impossible inputs. Constraints might include a specific reset sequence -- or even an area of design operation that you know does not require testing. The more clock cycles a proof must be tested across, the longer the formal run will take. Sometimes it may be too compute intensive to complete a particular proof. For example, a formal tool might report that "within 100 clock cycles, a specific signal combination will never occur". These are known as bounded proofs. It may be possible to verify that a specific check can be proven within, say, 10 clock cycles in 2 hours of tool execution; but to test for any time range might take weeks of tool execution.

For many projects this might do the job; and it can significantly cut down your overall verification time.

**Caveat**: Ask your F-ABV vendor which of the above property types they support. They may not have some of them.

## STANDARD LANGUAGE SUPPORT

Industry standard languages have highly evolved for formal verification. Having choices is still important.

- Chip Design Languages ：All the major Formal ABV tools support the Verilog and System Verilog design languages. Support for VHDL and SystemC/C++ is iffy.

- Assertion Languages ：

I System Verilog Assertions (SVA). System Verilog is now used by about 3/4th of the industry. It's part of the SV IEEE 1800 standard and is familiar to Verilog users. It fits in the Unified Verification Methodology (UVM) standard and is generally accepted as the testbench methodology of choice. The SVA format is considered the most powerful, and arguably the most complicated, of the HDL assertion formats.

II Property Specification Language (PSL) is an Accellera standard, commonly used with VHDL designs. PSL is older; mostly been replaced by SVA.

III C-Asserts are part of ANSI-C standard associated with SystemC/C++ design verification... Unlike the other assertion languages, C-asserts are combinatorial, so require extra code for sequential checks.

IV VHDL Assertions are a part of the VHDL language, which is still used on VHDL-centric product teams like the USA DoD, some FPGA users, and a number of design teams mostly in Europe and Japan.

V Proprietary assertion formats. There are companies that use internal assertions connected with certain tools, such as Intel ForSpec and IBM Sugar languages.

## COVERAGE MECHANISMS

Coverage is used measure your verification progress. It's usually in terms of percent. For example, for a specific 100 lines of Verilog code you hit 88% coverage during verification. Coverage is in two flavors:

**Code coverage**, the common coverage metric used for behavioral/RTL/gate-level Verilog/VDHL simulation. An example is 2,378 lines out of 5,800 code lines are covered -- 41% coverage. Code coverage inherently harder to measure during formal verification than RTL simulation, because the formal algorithm generally works in the state-space rather than on lines of Verilog.

**Functional coverage** is what percentage of your design intent that has been verified. It's easier to track during formal as it relates directly to the assertions that you're using. Caveat: functional coverage is only as good as the assertions you've written for your design. Getting 100% functional coverage on an incomplete (or badly written) set of assertions is classic Garbage In, Garbage Out.

Both coverage types are desirable metrics for formal. When considering formal coverage, it is important to differentiate between "controllability" and "observability".

- Controllabilityis activation of code lines or design functionality. In Verilog/VHDL simulation this is a measure of how much of your stimulus in touches lines of code or functions. In formal this relates to sections of the design that can be "seen" by the tool in the state-space. If you "over-constrain" your F-ABV tool (to save on tool runtime), it's possible that it'll miss important design functionality in your chip.

- Observability is a measure of the Verilog/VHDL source code lines or functions that are being checked by the checkers. This is important for both simulation and formal, and measures how well possible bugs are detected by your testbench checks, or assertions. For example, if an assertion was checking function "ZZZ" described using 4 lines of code, then those 4 lines of code are "observed" by the assertion. Effective formal coverage is focused around observability metrics, which are applied to both source code (lines, expressions, etc.) and functionality. It's a direct measure of how effective your current set of assertions are. For example, you might receive a report that "one specific PSL assertion touched 27 lines in your source code", or that "95% of the lines in your source code were touched by one or more SVA assertions."

**Below are some of the primary formal coverage types.**

- Assertion Coverage：Assertion Coverage is a simple indication of assertion pass/failure, bounded proofs (e.g. "proven over 20 clock cycles only"), and other quality metrics. Its limitation is it only provides information on the status of your current set of assertions, and not the coverage of the design code or functionality.

- Cone of Influence (COI) Coverage：This is based on the activation design logic that drives a particular signal. COI coverage suggests that a particular code line is observed by the assertion, as well as all the lines of code that influence this observed line. This is a somewhat conservative metric, which is reasonably easy to compute; however, it suffers from the accuracy issue that some of the design logic may be optimized away in various formal engines.

In addition, to suggest a line of code is fully verified because it influences another line can be questionable. For this reason the COI method has been refined by some vendors to focus on the proof itself. For example, the Cadence "Proof Core" coverage metric.

- Mutation Analysis：If a line of design source code that is covered by an assertion is changed, then the assertion should be triggered. This is the principle behind mutation coverage, which tweaks lines of code one-by-one in your design to see if those changes trigger your testbench checkers or assertions. If all the lines in a design are changed, or mutated, one-by-one then you can arrive at a measure of coverage for the entire design. For example if 90% of the line changes triggered assertions, then your design is 90% covered. The mutation technique can be applied equally well to simulation and formal producing a consistent coverage model between the two. It's very accurate but requires a lot of CPU time as the verification must be rerun many, many times. Mutation analysis is used extensively in software testing, and was originally made popular for hardware design by Certess and their tool Certitude (now Synopsys).

- Fault Observation Coverage：Fault observation coverage takes mutation coverage a stage further. Instead of changing the lines of code, faults are applied to specific points in the design to see whether they trigger assertions. Your formal tool applies those faults to your formal model (the state-space of your design contained in computer memory) rather than your original Verilog/VHDL source code -- thus taking less CPU time because of a few formal runs in state-space, rather than many repeated simulation runs than what classic mutation coverage takes. This also avoids some of the false negatives than can occur with mutation analysis. Observation coverage only works in formal (not simulation), but it outputs a code coverage metric that can be related to simulation coverage very easily.

## FORMAL-SIMULATION INTEROPERABILITY

Because Formal ABV today is used alongside simulation, the interoperability between the two is important. A major issue is understanding which parts of your design that are covered by the simulation or formal tools. This has been a problem because the coverage models between the two are different. A number of simulators like Synopsys VCS, Cadence Incisive, Mentor Questa. Aldec Rivera, Silvaco Silos, Fintronic Finsim, Synapticad Verilogger, Winterlogic Z01X, Tachyon CVC and others all have links to formal -- mostly through their coverage mechanisms.

Caveat: your formal tool along with your simulator will output a boatload of coverage metric data. This coverage information needs to be loaded into a common database and viewed with a verification planning tool to assess what overall progress you're making on your chip. It is too easy to lose data between the two different methods if you don't do this! Some vendors allow for their databases to be read and written in a limited fashion from your formal and simulation tools, including third parties; but you must ask each vendor exactly which tools it does or does not support. The Unified Coverage Interoperability Standard (UCIS) is the industry's attempt to combine different sources of coverage into one cohesive database, using a common API. There are varying degrees of vendor support for UCIS. Most of the EDA companies have their own verification flows and coverage databases for their own tools only.

Caveat: You must look beyond a UCIS Support "check-box" to understand if a vendor allows UCIS data from 3rd party tools to be written in or read out of their databases. This lets you choose the best simulation and formal technology for your designs regardless of vendor.

## ENGINE ACCESS

Formal vendors offer different ways to access their formal engines. This is for acceleration through parallel execution, increasing CPU capacity, and/or remixing of licenses for access flexibility. The engine access mechanisms include:

- Parallel Execution：Running a single RTL simulation job across multiple machines is very difficult due to the serial nature of the simulation algorithm. However formal can do parallel execution to accelerate runtimes by running different assertions on different machines. For example, if 5 machines are available and you have a formal job with 5 assertions, each assertion may be run independently on a different machine; making the entire job run in the time it takes to test the slowest assertion of the 5. With a larger number of assertions they be grouped depending on the machine loading, making an almost linear speed-up occur.

- Cloud Access：Using the sea of machines in the cloud to do parallel execution lets you effectively turn your largest formal job runtime to be only as long as its slowest single assertion.

- Token or Standard Licensing ：Standard licensing means that each F-AVB tool has only one key and that the tool can only run one job on whatever machine you want the job to run on. (You effectively pay by # of licenses you use.) Token licensing lets multiple copies of the F-AVB tool run multiple different jobs on whatever mix of machines you want the jobs to run on. (You effectively pay by # of tokens your total run uses.) Token licensing lets you easily swap in and out different formal engines and different formal apps on whim. It's amazingly flexible. In addition, during project crunch time, it's trivial to get more F-AVB cycles on demand simply by buying more tokens -- not easy to do with standard licensed tools.

## CUSTOMER TECH SUPPORT

Formal Verification has become much easier to use in the last few years. However, creating assertions and running the tools still requires a certain level of expertise. Depending on your internal expertise, you will need access to formal application engineering experts... Understanding the capability of the vendor's AE support is crucial. Caveat emptor.

Application Engineers (AEs) who are expert in formal verification will have spent years gaining the necessary experience to show others how to be productive with the tool. As such, it is important that your formal vendor maintains a specialized group of formal AEs on whom you may call for assistance.

Additionally some vendors offer consulting services to actually provide the verification itself using their F-ABV tools. Other vendors will contract outside 3rd party consultants to to ensure they are trained and certified on using their tools. Noted consultants include: Oski Technologies, TVS, and Paradigm Works. Formal Apps automate a specific formal task, so when using them, often a lower or minimal level of AE assistance is needed.

Caveat: it's crucial that you determine before you buy any formal tool exactly what type of customer support you will actually get, how they will be contacted, and what additional costs can and will be incurred from using customer support. Setting up a new formal tool can be very tricky, especially for newbies. Debugging formal problems can be very time and resource intensive. Be sure to know what your exact support system will be before you buy any F-AVB tool.

# 16 Formal Apps that make Formal easy for us non-Formal engineers

While standard F-ABV tools can solve a lot of verification problems for a verification engineer well versed in Formal -- there's a set of tools called "Formal Apps" that does design-specific assertion writing for you; and then it automatically runs these assertions in your Formal tool of choice for you. It's prepackaged Formal for the non-Formal user. This approach both saves engineering man-hours plus it ups the quality of the assertions used to verify your chip. (Remember assertions have a Garbage In, Garbage Out sensitivity. If you try to verify your design with bad assertions, your verification fails.)

Here's the 16 major categories of Formal Apps sold on the market today:

1. Register Checking App

2. Connectivity Checking App

3. Structural Property (or Assertion Synthesis) App

4. Activation Checks and Code Unreachability App

5. Scoreboarding App

6. Safety Fault Analysis App

7. Security Verification App

8. Clock Domain Crossing (CDC) App

9. Power Domain Management App

10. X-Propagation Checking App

11. Abstract Assertion Authoring App

12. Sim-Trace Property Synthesis App

13. Protocol Compliance Apps

14. Sequential Verilog/VHDL RTL Equivalency Checking App

15. Arithmetic Precision Analysis App

16. And the 3rd Party App Marketplace

Real Intent was the first formal vendor to offer a formal app. The company started off with property checking, then in 2007 they announced their Clock Domain Crossing (CDC) "checker" app.All the major verification players now have formal apps -- although the term "app" itself was made popular by Jasper.

## REGISTER CHECKING APP

Register checking involves verifying a register address map against your Verilog/VHDL RTL code. In your RTL code, a register consists of flip-flops buried within logic. It must be accessible from your bus interface.It's easy to make a mistake when wiring these flip-flops to the correct address decoder and "read"/"write" logic. This error won't be picked up until a software driver discovers it's not accessing the correct registers; which is quite late to be discovering HW problems in a chip project.

Without the app: Most chips can have 100s to 1000s of registers, so checking source RTL for addressing consistency is very painful to do by hand and it can take weeks. The process is error prone, and a common source of bugs.

With the app: Checking the register address map against your RTL source code is automatic, has no human errors, and only takes 1-2 hours. The register address map is typically in IP-XACT or Accellera RDL format. Depending on the app, other functional details -- such as correct "read and write" access over a specific bus protocol -- may also be checked. Often the same IP-XACT file is used in the chip's software spec, ensuring consistency between the hardware and software register addresses.

## CONNECTIVITY CHECKING APP

Are the parts of your design accessing each other correctly? This can be a real check to see if your chip wiring is correct -- or testing if a virtual connection path between two chip parts exists through on-chip networks, busses and other structures. "Is the interrupt output from peripheral 53 actually making it to the interrupt controller in CPU 3?"

Without the app: There are easily 1000's of real and virtual connections on a chip. Some are just direct wires, others may be complicated paths going through 10's of elements on your chip. Manual connectivity checking is an error prone, and irritating process, that can take days. No one wants to do this job. And it's not a value added task for engineers. It's perfect for automation.

With the app: Feed the app your Verilog/VHDL source code plus your list of key signal connections, and it automatically tests all the real and virtual connections. The entire check is just 1-2 hours for a 1000 connections; with no human errors.

## STRUCTURAL PROPERTY (OR ASSERTION SYNTHESIS) APP

The structural property, or assertion synthesis, app creates properties (or assertions) from your RTL code, to test for common HDL issues.

Without the app: Static linting tools like Spyglass use syntax and semantics to check your source code. It will then flag thousands of potential errors; of which only a few may occur during actual chip operation. This is the classic false positive problem. The chip designer must check them all out, which is an intensive process done by hand - and again, error prone.

With the app: Initially, the app appears to operate the same as a static linting tool; however this app generates formal assertions to check whether those errors will actually occur in the chip during operation.

Because this formal app only identifies actual operational problems, it narrows the human engineering follow-up needed and dramatically reduces the manual overhead needed -- by as much as 10X. It eliminates the time wasted having to sort through every "potential" error.For example, imagine an array of 47 elements, addressed with a variable that can take a value from 0-Y, where Y is greater than 47. That is, a classic "array-out-of-bounds" check.

- A linter will flag this as a possible error.

- This formal app inspects your RTL code \*operation\* to see if the addressing variable ever goes greater than 47, and only flags the actual code only if this occurs. It also generates counter-example waveforms, making these errors easier to fix.

## ACTIVATION CHECKS & CODE UNREACHABILITY APP

These formal apps check your RTL design source code for unreachable parts.

Without the app:

- Design error. One example of a design error could be if a code segment is not connected correctly, such as a bad arc in a state machine. This error could take a lot of time and effort to uncover through traditional Verilog/VHDL simulation.

- Over-constrained code. These are constraints added to narrow the formal verification process to only legal operations. Over-constraining is when important functionality is ignored during verification. You had constrained so much you fail to test functional parts of the design.

- Integration issues. Your IP was miswired during integration, resulting in dead parts of your chip. This leaves swaths of inactive and unreachable RTL code.

- Unremoved test code. In some cases unreachability is originally created on purpose -- for example, temporary test code -- but was accidently left in your design. It's an unreachable independent block within the chip, costing area and potentially more serious problems if it happens to get activated.

- Redundant code. Two identical blocks included for fail safe operation in the chip. Can both blocks be activated independently? This is hard to verify because you can get strange effects during RTL simulation if you are not expecting it. This formal app will pick out the two blocks and tell you if one, or the other, or both blocks are active.

With the app: You catch all these dead spots that RTL simulation failed to see. This formal app delivers a coverage metric for simulation by telling your coverage tool which exact parts of your design can't be simulated. The way it works is that you run the formal tool, get a list of dead spots, then re-simulate in Verilog/VHDL. This tells the simulator "don't bother trying to increase coverage on these parts of the design because they are unreachable".

The app has general activation checks, toggle checks (signal activation),FSM transition tests, etc.

## SCOREBOARDING APP

Verification engineers must often check for data transport issues through various on-chip structures -- such as does all the data going to a specific design block arrive as expected? For example, "did the read from memory 47 actually arrive at CPU 6?"

Without the app: There are 1000s of data transport permutations. Imagine a bus bridge with multiple data input and output ports. Using Verilog/VHDL simulation, you would need to check all the possible permutations to all the data inputs, and then check for the correct outputs. This requires a tremendous amount of design stimulus. It can take 1000s of man-hours to create and weeks of simulation to run. It's also very easy to miss a specific case due to the sheer volume of work required.

With the app: You come up with a set of rules based on the chip design spec on how the data will be transported. For example, "a 32 bit data packet that arrives at input port C will then be routed 30 cycles later to output port F without duplication." The formal app inspects your design source code to provide an exhaustive report on your expected data transport behavior. It confirms that the data transport in your RTL code is operating correctly without having to test every data permutation. This includes looking for missing data at an output, duplicated data, or data output at an incorrect time. This aspect of formal is brilliant because it saves 1000s of man-hours on a project. And you also don't miss any possible errors.

## SAFETY FAULT ANALYSIS APP

When a safety critical chip is in the field, it must be "fail-safe", meaning it can recover in real-time from an operating condition failure -- such as a memory bit flip due to electromagnetic interference. This app makes sure bad things won't happen if any part of your chip fails. That is, the chip detects and fixes the problem instantly and keeps working even with the failure. (Error correcting codes and redundant modules are two HW examples of this.)

Without the app: The initially obvious way to do safety fault analysis is to break some part of your Verilog/VHDL code (i.e. change your design) and to then re-simulate. However, ISO 26262 rules require that you run your test without changing your design. So instead you must inject faults one-by-one all over your design, and then see if your chip successfully recovers. Others use old-fashioned fault simulation tools. The trouble with fault simulation is that you must re-simulate the entire design for every single fault. For a million gate design, you have a fault on every output of every gate, i.e. 1,000,000 faults. A good fault simulator collapses the faults; however the number of faults can still be as high as 70,000. This takes months of CPU time.This ISO standard also has an Automotive Safety Integrity Level D (ASIL-D) requirement for diagnostic coverage; which sets the level of ability to recover and be failsafe at >90%. This means that the chip must still be able to recover even if >90% of all possible faults (injected one-by-one) are triggered.

With the app: Using formal, all the safety fault analysis problems described above are taken care of automatically. What happens is:

- Fault injection is done automatically (without changing your design).

- The follow-up analysis happens automatically.

- Fault injection is done in parallel, making a task which normally takes months take only a few hours.

- Segments of design source code are inspected as though many faults were simultaneously injected, then verifies the chip's recovery. That is, it goes even further than the inject faults one-by-one testing.

This type of app is used for safety critical designs such as automotive,aeronautical, satellites, medical, and nuclear power generation chips.

This app can also be used to meet high-reliability requirements which are not safety critical. That is, if you have good recovery mechanisms on your chip -- plus if you raise the "Mean Time Before Failure" (MTBF) of your chip's various parts, it'll nicely be able handle transient errors. This is important because with the life of cars now in the 10's of years, it's more likely transient errors will be the most common problem encountered during those decades instead of catastrophic failures.

## SECURITY VERIFICATION APP

This is the anti-hacker app. It sniffs around for unexpected data access paths internal to the chip that go from authorized areas to unauthorized areas. It also looks for any backdoors into the chip. "Who knew you could get to the president's bedroom if you took a path through the back kitchen cellar entrance?"

Without the app: Simulating every possible data access path permutation is not humanly feasible. There are always going to be paths that you miss. So companies will often just build the actual chip (or at least an FPGA prototype), give it to some hackers, and hope they find the flaws.

With the app: The verification engineer specifies the known possible attack methodologies to the app. For example, designs use encrypted keys for various purposes. These keys are stored in internal secure registers that should only be accessible through one specific "read" method. Hackers try to gain access by looking for a backdoor -- and the chip scan chain is one such classic vulnerability. Once you specify that these key registers are a "secure area", the formal tool will exhaustively inspect your design source code for all possible path permutations of ways your key registers can be accessed, including the scan path. Common hardware cyber-security attacks can be in the form of:

- Integrity: operational attacks (e.g. hacking cars, drones, IoT devices)

- Confidentiality: theft of information (e.g. encrypted license keys)

The upside of using security verification apps is that they are automatic and exhaustive.

## CLOCK DOMAIN CROSSING (CDC) APP

The CDC app ensures clock signal integrity is maintained across the device. Large chips now have 100s of clock domains, especially when you look at all the permutations of derived clocks internal to the chip.

Large sections of a design can be asynchronous relative to each other and can have random phase and frequency relationships. When a signal transitions from one clock domain to another, it's a nightmare of race conditions, meta-stability, and glitches. Design engineers use double flip-flop (or sometimes even triple flip-flop) synchronizers at clock domain boundaries to eliminate signal instability between clock domains.

Without the app: It's humanly impossible to test the 1000's of permutations of clock phases, frequencies, glitches, and transients. The scary method is to build the chip with synchronizing circuitry on the clock domain edges and hope that it works with minimal testing.

With the app: The CDC app examines all chip signals where they cross clock domain boundaries. Depending on the app two tiers of info may be provided:

- It identifies all signals where they cross clock domain boundaries that might be subject to instability.

- It analyzes the synchronizers to ensure that for all clock phases,no signal instability is propagated in the chip.

This app is standard for all design flows of chips with multiple clocks.

## POWER DOMAIN MANAGEMENT APP

This app tests the sequential switching logic used to turn "on" and "off" all of the independent power domains, block-by-block on the chip. This is to reduce power consumption and encourage more "Dark Silicon" on the chip. Turning "on" a power domain while a chip is running typically requires a complex "reset" sequence that (hopefully) does not disrupt the normal operation of the rest of the device.

Without the app: Design teams must ensure that every single power domain "start" and "stop" sequence is correctly initiated. This is virtually impossible with Verilog/VHDL simulation due to unpredictable timing and the sheer number of scenarios that require testing.

With the app: This formal app does an exhaustive time-independent analysis of all power domain "start" and "stop" sequences. For example, a "reset" on a power-up sequence is rigorously examined to completely eliminate unknown states and X propagation issues. And this is all done without a single power test vector being applied. This app is only in limited use. The formal tool must look at the whole design for each power domain, not just pieces, which uses boatloads of memory. The designs today are so big that it takes weeks to run. Add the permutation problem of multiple power domains, and the runtime is months.

Further, since software-controlled power domains are becoming more common,eliminating the need for power switching logic, UPF usage is down.

## X-PROPAGATION CHECKING APP

This app checks how wires, registers, or flip-flops somehow get into an unknown state -- it's either a "1" or a "0" but you don't know which. This is commonly known as an "X" state.X states can be caused by timing errors, glitches, or insufficient "reset" wiring/sequences. X states that appear during a Verilog/VHDL simulation run indicate a possible bug in the design. It gets worse if the X values are propagated across your chip.

Without the app: RTL simulation suffers from "X-Optimism" where additional X states are inserted into situations where they shouldn't be. This can lead to false X propagation issues, which can hide actual bugs in the chip.

With the app: An X-propagation app will automatically, and more reliably, check the propagation of X-states to areas where they may be harmful in the design.For example, you have a flip-flop not controlled by "reset"; on power up, it goes to an X state. You expect the downstream logic to get rid of the X state, but for some reason it doesn't. This propagates errors downstream. This app detects both the X state creation and the propagation problems in your chip, instead of false X problems in your RTL simulation. In addition, you don't need RTL simulation vectors to find these errors.

## ABSTRACT ASSERTION AUTHORING APP

This app is for creating assertions at higher levels of abstraction. This lets you more easily visualize the intent of your assertions -- to better understand what you are trying to test.

Without the app: You would only write low-level assertions by hand, which is done using System Verilog 90% of the time.

With the app: This app loads in the System Verilog libraries, then you invoke these high level assertion types. These are easier to use because they look like timing diagrams, which results in an overall 2X to 5X speed up in assertion creation. Using these high level SVA libraries to create low level assertion is similar to UVM -- where you have sequences which drive sets of tests.

## SIM-TRACE PROPERTY SYNTHESIS APP

A trace is a sequence of events that make up a particular RTL source code operation leading to an assertion failure. This app is for creating assertions by using sim-trace property synthesis; also called "behavioral property synthesis" or "assertion synthesis". By using your existing Verilog/VHDL simulation tests, this approach reduces the time and expertise needed to create low level assertions.This is not a linting tool to test your RTL design code -- instead it's a method to create assertions from design intent.

Without the app: You would only write low level SVA assertions by hand.

With the app: These apps use simulation traces and RTL design information to automatically generate assertions. You get FSDB or VCD trace files from your RTL simulation. You then load in these trace files and your RTL design code, and the app generates your assertions. These property synthesis tools convert a single Verilog/VHDL simulation run into multiple assertions. It expands the same test to more scenarios without you having to learn the assertion syntax.

Caveat: This method depends on your design and simulation results being correct. If either are incorrect, then you risk your generated assertions also being incorrect.

## PROTOCOL COMPLIANCE APPS

These apps confirm whether your chip design source code complies with common bus and other protocol standards -- such as AMBA, APB, and AXI.

Without the app: Using behavioral/RTL/gate-level simulation to test for potential protocol violations takes a very long runtime and it can easily miss problems due to incomplete, hand-created tests.

With the app: A formal protocol compliance app uses a group of pre-defined assertions to exhaustively verify your interface for compliance. By using libraries, you can test for multiple protocols throughout your design. The app will often also have a library of verification IP to test specific protocols. Protocol checking uses formal proofs to answer: "did my source code generate only standard-compliant correct AXI bus transactions with correct timing?".In contrast, simulation verification IP uses multiple Verilog/VHDL runs to try to test if your entire PCIe block complies with the PCI standard. Using these apps, teams can expect a 2-4X reduction in project verification time; and more importantly the tests are exhaustive.

## SEQUENTIAL VERILOG/VHDL RTL EQUIVALENCY CHECKING APP

This app answers "are these two RTL blocks functionally equivalent?". The app goes beyond just structural equivalency. It also exhaustively searches for the functional equivalency over time of these two blocks in your design. It's used to quickly test ECOs/RTL power optimizations/scan-test without having to run extensive Verilog/VHDL simulation.

Without the app: Typically designers use combinatorial equivalency checking to verify ASIC synthesis. However, this approach is limited to two code representations with an unchanged register configuration -- and reports errors on equivalency differences that may come from changed registers or other clocked elements -- even if the functionality is the same.

With the app: This app automatically compares two Verilog/VHDL code segments for full functional equivalence, handling both combinatorial and temporal equivalency. The only limitation is state-space equivalency is not infinite -- it only goes to a certain cycle depth, e.g. 30 clock cycles. This cycle depth is a differentiator between vendor's apps. In addition to late ECOs/RTL code optimization, the app is also used when synthesis tools optimize register configurations, as is typical for FPGAs and High Level Synthesis (HLS).

## ARITHMETIC PRECISION ANALYSIS APP

This app reports whether the number representations used throughout your chip datapath have the correct bit width for the required accuracy. This is often necessary for abstract algorithm code. Ignoring the complexity of fixed vs. floating point number systems, and whether the numbers are signed, etc., a simple DSP example would be: two 8-bit unsigned integer numbers multiplied together to create a 16-bit number. You may not need 16-bit accuracy, you may only need the top 8 or 12 bits. Anything more than 16 bits would result in wasted transistors.The reality is the arithmetic in the logic is extremely complex, as are the representations of the numbers through the datapath. This app tests all the number precision representations throughout the algorithmic block, given the arithmetic performed, any normalization that occurs, etc, and it ensures a correct and optimal bit-width.

Without the app: Engineers use pencil and paper to go through the design by hand to make sure the number precision is correct for every number representation in the datapath.

With the app: The app uses formal verification to test all the number precision representations throughout an algorithmic block. It eliminates the possibility of human error, ensuring a correct and optimal bit-width.

- If your bit-width is too large, circuit elements, and therefore power consumption and chip real estate, will be wasted.

- If the bit-width is too small, overflow can occur leading to false results.

Untimed algorithmic code, usually written in SystemC/C++ for High LevelSynthesis, typically makes use of signed fixed-point number systems.

## AND THE 3RD PARTY APP MARKETPLACE...

his isn't an app per se, but more about a new business model which makes formal apps available to the common engineer. For example, you have a guy who's a serious expert in security flaws in AMBA busses. Instead of having Mr. AMBA explain these flaws to your verification engineers, Mr. AMBA can write a special 3rd party app that tests for his exact expertise. To do this, your formal tool provider needs:

- an API in their formal tool that lets a 3rd party app developer pass control and property info in and out of it.

- an encrypted licensing scheme so not everyone can use this API.

- OEM agreements and reduced pricing so users who want just the 3rd party app itself don't have the buy the full blown formal tool, too, just to use it.

- a way to lock the 3rd party app with the formal tool, so the user doesn't also get a general purpose full copy of the formal tool for the discounr price of one app.

The idea behind all of this is to open formal apps into a similar 3rd party market that both Apple and Goggle phone apps now have.