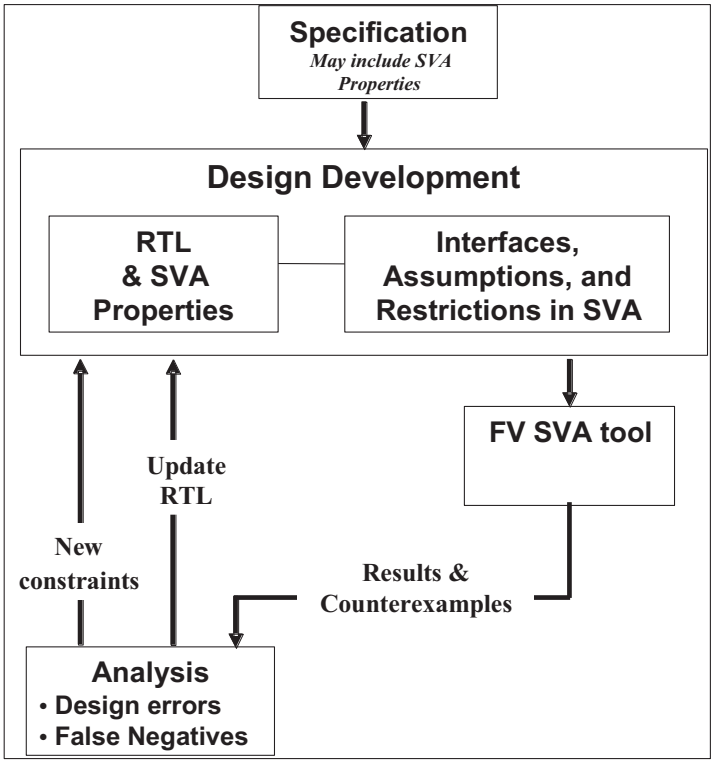

Figure 7.1-1 Typical Formal Verification Design Process with SystemVerilog Assertions

Model Checking Expectations and Rules

A first and foremost requirement for Model Checking is that designs need to be synthesizable and cycle-based. Model Checking tries to extract mathematical equations from the design and the properties and tries to prove/disprove the equations. A natural limitation of such an approach would be that designs that contain elements such as phase-locked loops (PLLs), behavioral memories, etc. cannot be converted into logic equations, thus they can't be proven using the formal engines. Typically, a model checker treats non-synthesizable blocks as black-boxes, making the outputs of those blocks free ranging variables, unless built-in models are available for those black boxes.

Another important aspect in Model Checking is the size of designs that it can handle. The size of a design is a function of state-levels, gates equivalents, number of assertions, classes of formal verification engine, and interestingly enough the tool algorithms. While a general statement on the size limits of today's tools is hardly possible, several observations can be made:

- Functional verification at the chip level is beyond today's tools.
- On the other hand, simple chip-level properties such as signal connectivity have frequently been reported to be successfully checked with formal techniques.
- Processor verification is making strong use of formal techniques, as reported e.g. by Infineon[88] and IBM[89], including complex pipelined processors and cache architectures.

[88] Jörg Bormann, Sven Beyer, Adriana Maggiore, Michael Siegel, Sebastian Skalberg Tim Blackmore, Fabio Bruno: Complete Formal Verification of TriCore2´and Other Processors, DVCon 2007

- Sometimes a "flop count" (i.e. number of state bits) is used as a rough measure of complexity. Raj Mitra of Texas Instruments estimates that, based on a number of projects, designs with more than 3000 flops are bad candidates for formal checking, while less than 1000 flops are usually no problem. [90]
- In such a rule of thumb, the flops in the cone of influence of each assertion should be counted rather than those in the DUV itself. This again is difficult to estimate for a given property.
- In terms of code size, compiling 100,000 lines of VHDL or Verilog into a formal model is no issue nowadays.
- When approaching their capacity limits, most tools offer some means of (manually) blackboxing modules, or otherwise abstracting unnecessary parts of the DUV.

Formal verification engine approach can be "bounded" (Bounded Model Checking, BMC) in the number of cycles the engine tries to use to check for assertion violations. They can also be "unbounded" (UBMC), where the number of cycles is not bounded (see section 7.6.1).

There is a debate as to which type of engine (BMC, UBMC) can handle large designs. Proponents of BMC claim that bounded engines can run much larger assertions than unbounded. For example, bounded model checking has been successfully used on non-trivial designs (with reasonably complex assertions. 5000 flop, 80k gates) for 200 clock cycles. However, such a model may not complete when using an unbounded engine. The drawback of BMC is that the logic is only evaluated over a specific number of clock ticks. If a design is run using a bounded engine over 200 clock cycles and the assertion passes, there is no guarantee that the assertion will not fail at cycle 250. Proponents of UBMC claim that an unbounded engine is capable of incorporating more powerful reductions than a bounded engine. In fact, they have demonstrated that unbounded engines significantly outperform bounded model checking on a testbench. But in most of the real examples, BMC has been a better success. Another interesting observation has been that a hybrid formal verification yields better results than a pure FV, as the hybrid approach doesn't suffer from the state space explosion problem that limits the usage of pure FV.[91]

With formal verification, the size of the driving logic for each respective assertion may play a significant role in the performance of the tool, and reductions may be applied that do not scale in a monotonic fashion with the size of the driving logic. Some vendors may extract the cone of logic associated with the assertion, and that could define the size limit. Other formal tools may read the entire design into memory at once, and try and build a state space to be explored by the formal engine; those tools will indeed be limited by design size, not assertion size. The size limit is also more related to the number of flip-flops (i.e., the state-space) than the number of gates. In addition, there are other techniques that some formal verification vendors adopt to minimize the runtime. These techniques may include hierarchical, incremental, and distributed model checking. Some of the earlier generation model checkers used BDD (Binary Decision Diagrams) as a basis to build the state space, and that puts a tremendous size limitation on the number of state elements that could be handled. As the technology matured, Ordered, Reduced Ordered BDDs are being used, which helps in the handling of larger designs.

---

[89] Thuyen Le, Tilman Glökler, Jason Baumgartner: Formal verification of a pervasive interconnect bus system in a high-performance microprocessor. DATE 2007: 219-224

[90] Alok Jain, Raj S. Mitra, Jason Baumgartner, Pallab Desgupta,Formal Assertion based Verification in Industrial Setting, Tutorial, DAC 2007

[91] NVIDIA Results with MG
http://biz.yahoo.com/prnews/041027/sfw068_1.html

The types of assertion failures that a user can expect from formal verification generally do not run for many clock cycles. However, that issue is debatable as there are vendors that claim support of assertions verified over very long number of clock cycles. If a formal tool does not support a failure trace for thousands of clock cycles, then such failures are better debugged with simulation-based techniques. Another verification approach would be to split such long assertions into multiple small assertions.

Note: Depending on the design, formal tools often return the shortest possible failure trace. Thus, instead of returning a failure track of 1000 clock cycles, which can be very difficult to follow, the tools return the shortest trace. An assertion that requires a large number of clock cycles to fail can be troublesome for a bounded engine. However, formal verification tools (this is very design and vendor dependent) were successful in generating several failure traces over 1000 clock cycles.

7.2 Global Clocking Past and Future Sampled Value Functions

With recent maturity in FV EDA tools, Formal Verification has attained a significant role in modern day ASIC design flow. In fact, in many large semiconductor houses, formal verification is part of the standard design-verification flow. With wider adoption of this technology IEEE 1800-2009 leveraged the opportunity to add additional constructs to facilitate the use of formal verification. The following sub-sections details those constructs/features. It is important to appreciate that these constructs can also be used in simulation.

As previously mentioned, clocks are not handled in the same manner in formal verification as in simulation, which is typically event driven. Gated clocks are assumed to always be active and the actual gating logic is ignored. Formal Verification usually requires a primary/system clock. Formal verification of assertions controlled by the primary clock is more efficient than the verification of assertions controlled by any other clock. This is because with a globally defined clock, the tool assumes the clocking between states, and does need to check for the clocking conditions. Thus, the purpose of the global clocking construct is to provide a way to specify the primary clock that synchronizes transitions in formal models and thereby obtain efficiencies in formal verification computations. Using clocking events defined by regular clocking blocks, even the default for a particular scope, does not accomplish specification of the global primary clock for the entire formal model. SystemVerilog answered the needs of assertion-based formal verification of synchronous systems with the notion of the reference clock and with supporting functions. Prior to the definition of a global clocking scheme, formal verification tools required a separate input, such as a configuration file, to identify the system clocking event. Formal verification tools can use the specification of a global clocking to identify the system clock.

The introduction of the global cloking does not mean that all assertions should now use **$global_clock** as the clock for each attempt. Assertions should use the clock that is associated with the logic being tested, and **$global_clock** should be used when the fastest system clock is needed, such as when signal changes are being detected.