### 7.1.1 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-boxed, making the outputs of those blocks free ranging variables.

In formal verification, clocks are not handled in the same manner as in simulation, which is typically event driven. Gated clocks are assumed to always be active and the actual gating logic is ignored. Assertions with multiple clocks that are integer ratio of each other may present a proof issue. For example an assertion described with one signal being sampled at 66 MHz clock and another signal at 100 MHz clock may not prove the assertion multiple times because of the way the clocks overlap (this process is tool dependent).

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, RTL gates, number of assertions, class of formal verification engine, and interestingly enough the tool algorithms. 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 7.4.1).

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 cycles. 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.\(^5\)

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)

\(^5\) NVIDIA Results with MG
http://biz.yahoo.com/prnews/041027/sfw068_1.html
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.

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 Role of SystemVerilog Assertions in FV

This section correlates SystemVerilog Assertions with the FV process, and addresses the similarities and differences between the application of SystemVerilog Assertions in Dynamic ABV and FV. It also presents some practical applications of SystemVerilog Assertions + FV as a combined methodology to tackle some of the niche areas of verification and design in general.

7.2.1 SystemVerilog Assertions in Formal Specifications

FV, as a technology, has been around for at least half-a-decade and has been used by some of the large semiconductor companies for verification with reasonable success.6 One of the main hurdles for FV to become accepted in main stream FPGA and ASIC Design Verification has been the lack of standardization in “specifying the behavior” of the system under verification. Almost every FV implementation used its own proprietary language to specify the design behavior. However, the standardization of property languages like PSL and SystemVerilog Assertions helped FV tool vendors to support those languages as their main interface. A FV tool will require at least two inputs: the design implementation (RTL), and a set of properties (e.g., SystemVerilog Assertions). It may also require information about design constraints. As shown in Figure 7.1-1, SystemVerilog Assertions properties can be part of a specification, and it is becoming common to have formal, machine-executable specifications defined for the protocols.7 A formal specification provides an unambiguous golden reference model of the properties and requirements of a design, independent of the implementation. Refer to Chapter 4 for examples of such specifications.

---

6 http://www.cl.cam.ac.uk/users/jrh/papers/fparith.pdf
A machine-checked theory of Floating point arithmetic, John Harrison, Intel Corporation

7 Verification of the Futurebus+ cache coherence protocol, by E.M.Clarke et. al. http://www-2.cs.cmu.edu/~emc/ Presents a formalization of an IEEE standard protocol and demonstrates that such formalization helps in finding errors in the Specification itself. Though this particular citing doesn't use SVA as their language, the concept still holds.