A Verification Synergy: Constraint-Based Verification
By Carl Pixley (Synopsys Inc.)

Functional verification (as opposed to verification for timing, power, manufacturability and so forth) is a bottleneck in design. We know why this is so. IC’s have become so complex that it is very difficult to specify and verify their behaviors. In the last ten years, the semiconductor industry has moved from directed simulation and directed random simulation, based solely on golden models, to more creative verification solutions, including comprehensive testbench tools, temporal assertion- and constraint-based verification, emulation, widespread use of automated Boolean equivalence checking, formal property checking and various hybrid schemes for verification.

It has also become obvious that verification technologies can have synergistic relationships. For example, we know that by writing constraint assertions in a certain way, they can be used as random simulation drivers for unit level verification and then can “flip” to become assertion monitors when units are combined with other units. These constraint assertions can also be used in a formal verification environment. There is a huge cost savings when information is used for more than one purpose. Historically, a similar synergy was discovered when it was realized that a subset of simulation language could be used for synthesis. This synergy was the basis for a revolution in IC design. But synthesis of models for verification (rather than design) is also attractive for getting emulation models running early in the design cycle. Typically, behavioral models of a design exist before detailed RTL models are available. Being able to synthesize and emulate these models is another attractive synergy. Several companies embed test generation and assertion monitoring in an emulator so that verification can proceed at emulation speed.

Language can facilitate synergies of technology. The important point is that language is not just about features for a single use (e.g., simulation, formal verification or synthesis) but also about how it facilitates synergy among several technologies. For example, an ideal assertion specification language would facilitate use with formal, simulation, and emulation engines (and even synthesis engines!) and be convenient for designers to use. However, language without supporting algorithms is not very useful. Verification algorithms also benefit from technology synergies. For example, Binary Decision Diagrams (BDDs), Boolean satisfaction algorithms (SAT), ATPG algorithms, uninterpreted functions, word-level algorithms and so forth and so on have been combined in various ways to create hybrid systems superior to any individual algorithm.

These synergies are having profound consequences for how verification is done. One issue that always haunts the design community is the cost effectiveness of a design strategy. Currently, most companies do the maximum verification that they can afford -- counting time to market, human effort, machine cost and so forth. Very few companies claim that any IC claim has been “totally verified” -- whatever that means. With very large expenditures some companies such as Intel have been able to totally formally verify certain key parts of their IC’s, such as floating-point units and memories. One way to reduce the cost of verification is to share information among different tools and methods. This implies a “capture once, use often” strategy for precious design information.

One example of verification synergy is constraint-based verification. To some degree, all commercial test bench tools allow constraints to be used in the generation of stimulus vectors during simulation. One way, pioneered at Motorola, is to capture constraints, which define the “environment” of a unit being verified. The idea was very simple, given that the design is in a certain state there is a set of vectors that are appropriate inputs for that state. At Motorola, we captured this as a Boolean formula, i.e., a constraint – that depends, of course, on the state of the design or of some useful auxiliary finite state machine. This was natural, since constraints were already built into our model checker, Verdict [kaufmann98], as a native capability. For example, we used constraints to define the valid set of initial states for model checking purposes. Actually, we used precisely the same constraint syntax for our Boolean equivalence checker, MET [park00], our switch-level extraction tool [jolly02] and other tools in our verification suite- another nice synergy. Since we used both formal model checking and in informal, simulation-based verification, it was natural for us to use constraints for both. On the one hand this was obviously useful because we could use constraints to represent the environment of a unit for formal verification, such as SAT-based bounded model checking. And as an added benefit, such constraints (i.e., assertions formulas) could easily become assertion monitors when the DUV was connected up to its real chip environment. This became the basis of an
assume/guarantee methodology whereby assertions about a unit could be proven under constraint assumptions and then “flip” and become a property to check in a larger context.

However, how would we use these constraint assertions in simulation? There was no obvious method for taking Boolean formulas and generating solutions on the fly that would not excessively slow down simulation. We would have to solve a SAT problem (np-hard) each clock cycle of simulation! So we invented a tool called SimGen [pixley99, yuan99] that could be used for non-backtracking, on-the-fly stimulus generation. The idea was actually pretty easy: first we compiled the constraints into Binary Decision Diagrams (BDDs) involving state variables and input variables of the design. As everyone who has used BDDs knows, there is always the possibility of BDD blowup so we had to think of some clever ways to ameliorate that problem. However, if you can compile the BDDs, the rest is easy. At each clock cycle, the state of the design is sampled and then a “walk” of the BDD is performed in linear time to get a satisfying assignment of inputs that satisfies the constraints. One then drives the inputs to the design with the input values just calculated, toggles the clock and does the same thing next clock cycle. I am leaving out lots of details but that is the general idea. It is possible to bias the inputs (even depending upon the state of the design!) so that one got a good mix of inputs.

It was quite a breakthrough when we discovered a “biasing” scheme. This solved two problems: how to bias inputs to the design and (surprisingly) how to allow for efficient BDD ordering. We proved that the biasing scheme was independent of BDD order, which allowed us to interleave state and input variables in any order. Of course, the whole point of this research was to find a way to generate inputs to the design that satisfied the constraints and was not inordinately time consuming. SimGen’s overhead during simulation was usually far less than 20% based upon the complexity of the constraint set. It turned out that most of the time was spent in the PLI. These days, we all know a lot more about how to make the compile phase and the runtime much more efficient. The interesting thing is that when we tried the new scheme on real verification problems, it turned out to be amazingly effective in generating interesting corner cases. So what evolved is a methodology in which SimGen was used on module, block and unit levels of the design to quickly locate bugs. This was called “maturing the logic early”. Then when the bug rate fell off, we used our model checker to find the more subtle bugs that were harder to find in any sort of simulation. The advantage of unit verification is that one is not finding silly bugs during full chip or SoC verification. Also, constraints-as-checkers (i.e., assertions) were left all over the design as booby traps to catch bugs at their source. My dream is that various reusable parts, like bus interfaces, can be fully verified at the factory and that verification IP, in the form of assertions and constraints can be delivered with the design IP.

Another very important synergy concerns the language that is used to define assertions and constraints. Our colleagues at Motorola Israel devised a language call CBV (Cycle-Based Verilog) that is extensively used directly by designers to express properties about a design. As the name implies, CBV looks as much like Verilog as possible. For example, the expression language of CBV is the Verilog expression language. So learning time is very short. We have known designers to be writing useful CBV with a half day of training. The language is block structured and its semantics is essentially thread based. The block structure encourages the writing of a comprehensive specification about a block – rather than just an ad hoc set of isolated assertions. The thread-based semantics allows convenient scoping rules -- which, among other things, allows local variable assignments and is very intuitive. CBV also has functions and tasks. It is straightforward to monitor CBV expressions in simulation and to model check properties written in CBV. I do not wish to get involved in a language controversy but I would like to make the point that CBV (a) was easy to learn, (b) was used on real designs by both verification engineers and designers, and (c) fit into our total verification flow. Which brings me back to the main point of my talk. It is synergy among languages, tools and methodology that has maximum impact on design.


Thanks to Jim Kukula and Robert Damiano who commented on this abstract.