A Verification Synergy: Constraint-Based Verification

By Carl Pixley (Synopsys Inc.) and John Havlicek (Motorola, Inc.)

Introduction

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, computer costs 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 repeatedly" strategy for precious design information.

Constraint-Based Verification

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 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. We’re 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 [yuan99].

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 -- *with precisely the same constraints* -- to find the more subtle bugs that were harder to find in any sort of simulation. Of course, there was the possibility of actually *proving* an assertion with the model checker as well. 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 <u>is</u> 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. The block structure encourages the writing of a comprehensive specification about a block rather than just an ad hoc set of isolated assertions.

The goal of CBV was to give designers an intuitively easy way to write the kinds of properties that are of the most practical interest. Most practical properties are expected to hold all the time. Therefore CBV provides an implicit "always" on all top-level assertions. The most common way to combine assertions is by "and", so the "begin-end" operator in CBV is interpreted as a conjunctive fork. CBV encourages a forward-looking style of coding, in which nesting of many variations of temporal "if" operators can be used to define the precondition of an assertion. The "if" operators also allow limited expressions of disjunction. Negation and liveness operators in CBV are limited to Boolean expressions. The resulting language is richly expressive of safety properties and offers limited liveness. It covers the very important intersection of what is most commonly of interest to designers with what is efficient to implement, e.g. in thread-based simulation monitors or on-the-fly model checkers.

The thread-based semantics of CBV provides a unified conceptual framework for writing temporal assertions without having to rely on traditional temporal logic operators or regular expressions. The underlying multi-threading features give the user a convenient foundation: new threads begin every cycle due to the implicit top-level "always", and forking is as easy as opening a "begin-end". Since there is no join in CBV, the scoping for threads is particularly simple--each thread has its own local namespace, inherited initially from its parent, but free to evolve independently of its siblings. Put another way, the various concurrent CBV

threads grow as a *tree* [theoretically, this tree is nothing more than a computation tree of a forall-automaton], and as a result, CBV threads can carry local variables very intuitively.

CBV also has functions and tasks for modularity. The tasks can be temporally recursive, which makes the coding of ongoing assertions based on patterns of events routine.

These days, assertion languages based on regular expressions have become quite popular (Synopsys' OVA, Intel's ForSpec, IBM's Sugar, and the emerging PSL and SVA from Accellera). Regular expressions provide a convenient way to define temporal patterns, or "sequences", which can then be combined in various ways to define assertions. Admittedly, CBV does not offer the same convenience as regular expressions, but CBV does go a long way in letting the user define temporal patterns. The idea is to use the same intuitive thread-based semantics from assertion writing when defining the patterns. The user puts the pattern matching code into "matching tasks" and uses the leaf-level "return" statement in such a task to indicate that the pattern has been matched. The matching task can then be called in the condition of an "if" statement, and every thread that matches the pattern will return to execute the body of the "if". This makes the "if" work like a "sequence implication" in Sugar or the "triggers" operator in ForSpec. The neat thing in CBV is that the thread semantics that the user thinks about is the same for defining patterns and properties.

By the way, there are some challenges to adding local variables to regular expressions. Regular expressions provide "and" and "or" operators that cause somewhat different kinds of joining. At the end of an "and", there is typically a true join of the threads of the branches, so there must be defined a method for resolving inconsistent local variable assignments in the different threads. At the end of an "or", there may not be a true join, since it might be that only one branch matches. In case both branches match at the same time, one must decide whether to join the local variables as in the "and" case or let the threads continue separately. Whatever decisions are made to deal with these joins, it seems preferable to have static (compile time) rules about how local variables pass through joins. The unbounded repetition operator poses another challenge for supporting local variables in regular expressions.

A regular expression

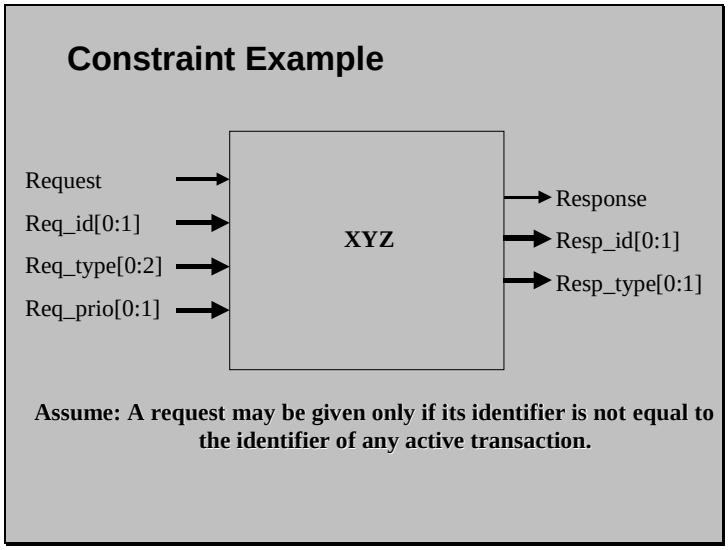
```
(!p*[0:inf]; !p(local x[31:0] = e[31:0]) ; !p*[0:inf]; p && (y == x));
```

samples x at a non-deterministic point prior to the first assertion of p. Thus, this regular expression matches provided that at the first assertion of p (after the start of the match), y has a value that appeared in e at some prior point (the prior point also being after the start of the match). Checking for matches of such a regular expression in simulation can be very expensive.

We do not wish to get involved in a language controversy but we would like to make the point that CBV (a) is easy to learn, (b) is used on real designs by both verification engineers and designers, and (c) fit into our total verification flow. Which brings us back to the main point of this talk. It is synergy among languages, tools and methodology that has maximum impact on design.

A Simple Constraint Example:

Assume the following bus interface unit. Attached are two simple constraints on the interface to the unit: the environment is not allowed to generate a request for a request id that is already active.




```
module xyz;

  /* Definitions Block */

  function activate(id[0:1])[0:0]   = request  & (req_id == id) ;
  function deactivate(id[0:1])[0:0] = response & (resp_id == id) ;
  function active_next(id[0:1])[0:0] =
  (
    deactivate(id) ? 1'b0    :
    activate(id)   ? 1'b1    :
                     active[id]
  ) ;

  var active[0:1] =
  {
    active_next(0),
    active_next(1),
    active_next(2),
    active_next(3),
  } ;
```

Constraint: A request may be given only if its identifier is not equal to the identifier of any active transaction

```
    constraint(request ? ~active[req_id] : 1'b1) ;

endmodule
```

SimGen Constraint Generation

As noted above, simulation generation from constraints a la SimGen [2] works very simply. At compile time the user supplies a set of constraints to the SimGen compiler. The constraints are compiled either into a

Verilog module [kukula00] suitable for emulation or into a C program [pixley99] that runs during simulation. During simulation a user can give a set of biases to the runtime program to control the likelihood that an input bit will get set to 1. The user can also set an initial state using Verilog directives or can give the design a synchronizing sequence. At any point the user can call a SimGen task, which turns the simulation over to the SimGen executable. After that point SimGen generates simulations compliant with the constraints and biases every clock cycle.

There is a logical (and very real) possibility that the simulation will encounter a deadend state, i.e., a state for which there is no solution for the inputs. At that point the simulation will stop and the offending trace will be generated. The cause of a deadend state can either be an error in constraints (for which we had a simple debugger) or an error in the design that allowed it to get into an unanticipated "bad" state. In either case, the deadend must be analyzed.

Summary

The thesis of this paper is that synergies of verification technologies are most effective. As an example, constraint-based verification was examined. Constraints can be used (1) to generate simulations, (2) to monitor simulations, (3) as formal environment definitions for model checking, (4) properties for model checking. Therefore, constraints can be used in an assume/guarantee methodology with both simulation and formal verification. A few key advantages of this approach are that constraints can be developed incrementally starting with inexpensive early animation of designs. They can be introduced into an existing verification methodology easily. Constraints will mature while the design matures. Most importantly, constraints can easily be "flipped" to become monitors as well as generators. Hence, they support reuse.

Bibliography

[pixley99] C. Pixley, K. Shultz, J. Yuan, "Integrated Formal and Informal Design Verification of Commercial Integrated Circuits", Proc. of the international Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), pp. 1061-1067, June 28, 1999.

[yuan99] J. Yuan, K. Shultz, C. Pixley, H. Miller, A. Aziz, "Modeling Design Constraints and Biasing in Simulation Using BDDs", ICCAD'99, pp. 584-589.

[kaufmann98] M. Kaufmann, A. Martin, C. Pixley; "Design Constraints in Symbolic Model Checking", CAV'98, pp. 477-487.

[park00] J. Park, C. Pixley, M. Burns, H. Cho, "An Efficient Logic Equivalence Checker for Industrial Circuits", JETTA vol. 16:1/2, pp. 91-106, 2000.

[jolly02] S. Jolly, A. Parashkevov, T. McDougall, "Automated Equivalence Checking of Switch Level Circuits", DAC'02, pp. 199-205.

[kukula00] J. Kukula, T. Shiple, "Building Circuits from Relations" CAV 2000.
We thank to Jim Kukula and Robert Damiano who commented on this paper.