LibQA - Library Quality Assurance for VHDL Synthesis and Simulation

Ronald B. Stewart, SGS THOMSON Microelectronics

Jon Phillips, Philips Semiconductors

Abstract

LibQA is a quality assurance tool for VHDL synthesis and simulation models which also performs timing characterization. The synthesis model is translated into an FSM, then graph exploration generates stimuli for VHDL and electrical simulation. Function, propagation delay, timing constraint violation, and hazard response are all tested.

1. Introduction

Correct standard cell models are a good foundation for successful synthesis and simulation. LibQA is a quality-assurance tool currently in industrial service to compare Synopsys synthesis models against VHDL simulation models and against electrical models extracted from layout. Starting from a synthesis model, LibQA generates an equivalent FSM. Using graph algorithms, LibQA generates stimuli for simulations. The results of the simulations give these benefits:

- Synopsys Technology Library cell model testing
- VHDL cell model testing
- Cell timing characterization

In most cases, LibQA is automatic. However, cells which cannot be defined in the syntax of the synthesis model require manually supplied stimuli.

The synthesis model is analyzed formally, and the simulation stimuli generated from it are formally correct. Assuming there are no unexpected states in the models, the test coverage is total, showing that the electrical and VHDL models implement the specification of the synthesis model. Others have proven equivalence of certain VHDL models using formal methods, but this is not practical for a general production tool. Abstraction of electrical models has been done so far without proof, and is limited in scope. Progress on those two fronts should create new opportunities in model validation.

LibQA also supports inspection of timing models:

- Cell timing range and monotonicity checks
- Cell timing graph viewer
- Cell timing calculator

2. FSM Construction

LibQA translates the Synopsys Technology Library synthesis model into a Mealy machine[1] $\mathbf{M} = (\mathbf{Q}, \Sigma, \Delta, \delta, \lambda, q_0)$ where $\mathbf{Q} = \{q_0, q_1, \ldots q_n\}$ is the set of states, $\Sigma = \{a_0, a_1, \ldots a_n\}$ is the input alphabet, $\Delta = \{a_0, a_1, \ldots a_n\}$ is the output alphabet, δ is the state transition function mapping $\mathbf{Q}\times\Sigma$ to $\mathbf{Q}$, λ is the output function mapping $\mathbf{Q}$ to Δ, and q_0 is the initial state.

For this application, some preliminary definitions relate the abstract machine to the physical cell model. Define V $=\{0, 1, Z\}$ the set of signal values, $v \in V$, and $I = \{i_0, i_1, \ldots i_n\} \subset V^n$ the set of inputs. The single input transition function $I_{n+1} = f(I_n, i, v)$ produces a new input combination from an old one, so that $i_{j+1} = i_{j,\ j\neq n,}$ and $i_n = v$.

Σ is defined as a set of all one-signal input signal transitions (i,v), **Q** is a set of input signal and state variable current values, and Δ is a set of output signals defined in {0, 1, Z}. q_0 is defined for each test set so that values on primary inputs are assigned to the associated q_i, and values taken from the state variables of Synopsys sequential primitives ff and latch are assigned ‘U’. These conventions are well suited to the waveform generation task needed to run the simulators.

3. FSM for Combinational Cells

To benefit from the simple graph search algorithms that generate the test vectors, even the combinational cells are translated into FSMs. To illustrate the process, here is the functional part of the synthesis specification for a tristate buffer with resistive input pull-up:

```
cell(TRI_BUF) {
 pin(Z) { direction : output;
  function : “A”;
  three_state : “E’” }
 pin(A) { direction : input;
  driver_type : pull_up; }
 pin(E) { direction : input; }
}
```

The steps are:

1. Define the set of states. There is one state variable for each input. Variables are Boolean unless the

driver_type attribute is present, in which case such variables are on {0, 1, Z}.

Given input signals A and E, **Q** = A × E is the set of states, where A ∈ {0, 1} and Z ∈ {0, 1, Z}. For readability, label them qAE = {q00, q01, q0Z, ...}

2. Define the input alphabet as the set of all signal transitions on individual signals. For a Boolean input signal, the transitions are {01, 10}. For signals on {0, 1, Z} the transitions are {01, 0Z, 10, 1Z, Z1, Z0}.

Thus, label the input alphabet Δ = {e01, e10, a01, a0Z, a10, a1Z, aZ1, aZ0}.

3. Define the state transition function. $I_{n+1} = f(I_n, i, v)$. The table contents are generated by evaluating the definition over its range.

Table 1: δ for TRI_BUF

	e01	*e10*	*a01*	*a0Z*	*a10*	*a1Z*	*aZ1*	*aZ0*
q00	*q10*	-	*q01*	*q0Z*	-	-	-	-
q01	*q11*	-	*q11*	-	*q00*	*q0Z*	-	-
q0Z	*q1Z*	-	*q1Z*	-	-	-	*q01*	*q00*
q10	-	*q00*	-	*q1Z*	-	-	-	-
q11	-	*q01*	-	-	*q10*	*q1Z*	-	-
q1Z	-	*q0Z*	-	-	-	-	*q11*	*q10*

4. Define the output alphabet. This is the set of values for the output pins. Each occurrence of `pin(){direction:output}` is an output. The values are normally Boolean, but they become {0,1,Z} if the record `three_state` is present.

There is one three_state output, Z. The output alphabet is Δ = {0, 1, Z}.

5. Define the output function mapping **Q** to Δ. For each output, evaluate the function and three_state expressions. If three_state is present and evaluates as true, the output value is Z; otherwise, the output is the value of the function expression. The independent variables of the two expressions correspond to the state variables, but the range of the expressions is Boolean, while that of the state variables is {0,1,Z}. Because Z is caused only by the driver_type attribute, a Boolean value is obtained for every occurrence of Z: pull_up maps Z to 1, and pull_down maps Z to 0. The evaluation then proceeds by converting the infix to a postfix tree and evaluating from leaf to summit. λ is the output function mapping **Q** to Δ.

Table 2: λ for TRI_BUF

State qAE	*q00*	*q01*	*q0Z*	*q10*	*q11*	*a1Z*
Output Z	*1*	*0*	*0*	*Z*	*Z*	*Z*

Not surprisingly, the table turns out to be simply the truth table for the combinational cell. All the other machinery has been created to give these conveniences:

- The data structures are identical to those to be used for sequential cells, supporting the graph algorithms used to generate test stimuli.
- Only single input transitions exist, supporting the definition and application of timing events.

4. Sequential Cell FSM Construction

Synopsys models sequential cells use just two primitives: `ff` for flip-flops, and `latch` for latches. There is also an obsolete `state` primitive we do not use.

Here is a Synopsys description of a sample flip-flop:

```
cell(FD3S) {
 ff( “IQ”, “IQN”) {
  next_state : “(D TE’)+(TI TE)”;
  clocked_on : “CP”;
  clear : “CD’”;
  preset : “SD’”;
  clear_preset_var1 : L;
  clear_preset_var2 : H;
 }
 pin(Q) { direction : output;
  function “IQ”;}
 pin(D) { direction : input };
 pin(TE) { direction : input };
 pin(TI) { direction : input };
 pin(CP) { direction : input };
 pin(CD) { direction : input };
 pin(SD) { direction : input };
}
```

The steps LibQA uses to construct its FSM model are:

1. Call construct_ff, which returns an FSM where boolean expressions next_state, clocked_on, and all the other ff inputs are represented by a single term.

2. Expand the FSM for each ff input expression.

3. Construct the output function Δ.

`FSM * construct_ff()` is a function which constructs a complete FSM structure which models an elementary ff. These come in several variations depending on attributes which concern the preset and clear functions:

- whether preset and clear exist.
- the state when clear and set are applied together.

For example, a multiplexed D ff is expressed as:

```
ff( “IQ”, “IQN”) {
 next_state : “(D TE’)+(TI TE)”;
 clocked_on : “CP”;
}
```

The FSM will be constructed by first calling `construct_ff` to return an elementary FSM for the ff. Then, new state variables are created for each input, and each arc to the (now replaced) next_state and clocked_on variables is replaced by a set of arcs accounting for all the cases due to the expressions. In the present case, clocked_on is simple, so re-labeling is all that is needed.

The ff primitive is expanded to:

Table 3: δ for simplest ff

(IQ, next, clock)	next state Rise	next state Fall	clock Rise	clock Fall
(0,0,0)	(0,1,0)	-	(0,0,1)	-
(0.0,1)	(0,1,1)	-	-	(0,0,0)
(0,1,0)	-	(0,0,0)	(1,1,1)	-
(0,1,1)	-	(0,0,1)	-	(0,1,0)
(1,0,0)	(1,1,0)	-	(0,0,1)	-
(1,0,1)	(1,1,1)	-	-	(1,0,0)
(1,1,0)	-	(1,1,0)	(1,1,1)	-
(1,1,1)	-		-	(1,1,0)
(U,0,0)	(U,1,0)	-	(0,0,1)	-
(U,0,1)	(U,1,1)	-	-	(U,0,0)
(U,1,0)	-	(U,0,0)	(1,1,1)	-
(U,1,1)	-	(U,0,1)	-	(U,1,0)

5. Test Sequence Generation

The different types of test sequence LibQA can make are:

- Functional exercise
- Timing exercise
- Timing constraint violation
- Hazard exercise

The C procedures which implement them are described below. Each returns a linked list of `path` data structures:

```
typedef Path {
  struct Path *p_next;
  Arc arc;
  Boolean isConstraintTest;
}
```

5.1 FindPath Procedure

This is a depth-first path enumeration. The `GoodArc` and `Done` functions are assigned to suit the kind of path needed: state-to-state, all arcs, or all timing arcs. The complexity of standard-cell FSM models is low, so performance is acceptable. The search algorithm is elementary:

```
TestSequence *FindPath(node, fsm, GoodArc, Done)
{
  for(arc = node->arcs; arc; arc = arc->next) {
    if(GoodArc(arc)){
      ++arc->explored;
      path = FindPath(fsm, arc->toState, budget);
      --arc->explored;
      if(path)
        return AppendArc(path, arc);
    }
    if(budget > 0) {
        ++arc->explored;
        path = FindPath(
          fsm, arc->toState, budget-1);
        --arc->explored;
        if(path)
          return AppendArc(path, arc);
    }
  }
  if(Done(fsm))
    path = NewPath();
  else
    path = NULL;
  return path;
}
```

5.2 FindTest Procedure

A test sequence is selected so that every arc of the FSM is traversed in optimal or near-optimal order. Optimal order is not useful when finding it costs more than the improvement in simulation time it buys, so sub-optimal paths are accepted when complexity exceeds a limit.

```
TestSequence *FindTestSequence(fsm, Done)
{
  if(fsm->arcCount > 512)
    return FindBudgetCover(fsm, MaxInt);
  else {
   budget = 0;
   do {
     test = FindBudgetCover(fsm, budget);
     if(test) return test;
     ++budget;
   } while (1);
}
TestSequence *FindBudgetCover(fsm, budget, Done)
{
 rootNode = ConnectAllStatesToRoot(fsm);
 ZeroAllStatesExplored(fsm);
 return FindPath(rootNode, fsm);
}
```

5.3 Functional Test Sequence

The functional test sequence traverses every arc of the FSM at least once.

```
test = FindTestSequence(fsm, IsUnexploredArc,
 AllArcsExplored);

Boolean IsUnexploredArc(arc) {
 return (arc->explored == 0);
}

Boolean AllArcsExplored(fsm);
```

5.4 Timing Test Sequence

Only the arcs that result in an output signal change are required in the timing test sequence. The resulting sequence is much shorter than the functional test.

```
test = FindTestSequence(fsm,
 IsUnexploredTimingArc,AllTimingArcsExplored);

Boolean IsUnexploredArc(arc)
{
 if ((arc->explored != 0)
   return FALSE
 beginOutput = fsm->output[arc->beginState];
 endOutput = fsm->output[arc->endState];
 return(beginOutput != endOutput);
}

Boolean AllTimingArcsExplored(fsm);
```

5.5 Violation Test Sequence

Knowledge of the state of a cell can be lost if a timing constraint is violated. A set of classically recognized constraint violations is produced for the latch and ff primitives, suitably expanded according to their associated expressions. No attempt was made to inspect the FSM trying to recognize these, but this a good opportunity for future improvement.

The catalog of constraints is:

- Minimum Pulse Width. All terms of `clocked_on`, clear, and `preset` expressions are exercised for minimum pulse width.
- Setup and Hold. All terms of `next_state` are exercised for setup and hold.
- Recovery. All terms of `preset` and `clear` are exercised for recovery time in combination with all terms of `clocked_on`.

5.6 Hazard Test Sequence

VHDL models can be configured to respond to hazards in a variety of ways: with transport delays, inertial delays, glitch X generation, spike X generation, and message generation. All arcs leading to an output change can be subjected to hazards. Testbenches are generated for all these behaviors because all possible choices need to be tested.

6. Electrical Testbench Generation

For functional testing, an Eldo[4] electrical simulation file is generated from the functional stimuli and the environmental information in the synthesis model. The simulation is carried out, and the output values are captured for every input change. A defect is reported if any output differs from the predicted value. If there are hidden states, or electrical problems, this test may not notice them. The timing characterization phase, where a range of operating conditions is applied, can help to spot more subtle problems.

7. VHDL Testbench Generation

The properties of the VHDL models which need to be tested include functional operation, backannotated timing, X and report generation in response to glitches and timing constraint violations. The VHDL testbenches themselves consist of stimuli and procedures. We need to know if the models survive the stimuli without fatal errors, if the functional operation is correct, and if the timing values from the SDF are respected.

7.1 VHDL Procedures

7.1.1 qa_input

Apply STIMULUS_VALUE to the cell.

```
procedure qa_input(
 signal STIMULUS: out STD_ULOGIC_VECTOR;
 STIMULUS_VALUE: in STD_ULOGIC_VECTOR);
```

7.1.2 qa_function

This procedure verifies the function only - not the timing.

```
procedure qa_function(
 signal ACTUAL: in STD_ULOGIC_VECTOR;
 signal EXPECT: out STD_ULOGIC_VECTOR;
EXPECT_VALUE: STD_ULOGIC_VECTOR;
OLD_STIMULUS, CURRENT_STIMULUS: string;
```

```
 DELAY: time);
```

7.1.3 qa_timing1

This procedure verifies function and timing for a single-output cell.

```
 procedure qa_timing1(
  signal ACTUAL: in STD_ULOGIC_VECTOR;
  signal EXPECT: out STD_ULOGIC_VECTOR;
  OLD_EXPECT_VALUE, EXPECT_VALUE:
    STD_ULOGIC_VECTOR;
  OLD_STIMULUS, CURRENT_STIMULUS: string;
  DELAY: time);
```

An excerpt shows how functional and timing errors are detected:

```
wait for DELAY;
expect <= EXPECT_VALUE;
if OLD_EXPECT_VALUE /= EXPECT VALUE then
  wait for 1 ps;
  if(DELAY = 0 nS) then
    wait for 10nS;
    if(ACTUAL /= EXPECT_VALUE) then
      NEVER := true;
    end if;
  elsif (ACTUAL /= EXPECT_VALUE) then
      LATE := true;
      wait for 10 nS;
    if ( ACTUAL /= EXPECT_VALUE) then
        NEVER := true;
      end if;
  elsif (ACTUAL’last_event /= 1 ps) then
      EARLY := true;
  end if;

  if( LATE or EARLY or NEVER) then
    -- write an error description to L
    assert false report L.all severity Warning;
  end if;
end if;
```

7.1.4 qa_timing2

Verify function and timing for a two-output cell. The code is a development of the method of qa_timing1. All functional and timing errors on both signals are reported.

```
 procedure qa_timing2(
  signal ACTUAL: in STD_ULOGIC_VECTOR;
  signal EXPECT: out STD_ULOGIC_VECTOR;
 OLD_EXPECT_VALUE, EXPECT_VALUE:
STD_ULOGIC_VECTOR;
 OLD_STIMULUS, CURRENT_STIMULUS: string;
 DELAY1, DELAY2: time);
```

7.2 VHDL Stimuli

VHDL header and closing statements are generated from the FSM structure. The test sequence generation procedures return a `path` data structure for each kind of test: function, timing, violation, and glitch. Each of these is translated into VHDL stimuli. LibQA generates the constants that provide delay values for the stimuli and the constants for the SDF, the test benches can detect timing errors in the models. This is an excerpt from the timing test for an inverter; other tests follow similar lines.

```
architecture TEST of TB_IV is
  signal STIMULUS: std_ulogic_vector(1 to 1);
  signal EXPECT, ACTUAL:
   std_ulogic_vector(1 to 1);
  signal CLOCK: std_logic := ‘0’;
  begin

test : process
  begin
    qa_input(STIMULUS,”0”);
    wait for 100 ns;
    qa_input(STIMULUS,”1”);
    qa_timing1(ACTUAL, EXPECT,
      ”1”, ”0”, ”0”, ”1”, IV_tpdA_Z_F);
    wait for 100 ns;
    qa_input(STIMULUS,”0”);
    qa_timing1(ACTUAL, EXPECT,
      ”0”, ”1”, ”1”, ”0”, IV_tpdA_Z_R);
    wait for 100 ns;
    assert false report ”end of test”
     severity failure;
  end process test;
  inst: IV
    port map(A=>STIMULUS(1),Z=>ACTUAL(1));
end TEST;
```

8. Timing Characterization

LibQA starts with an existing synthesis model, generates test patterns, runs electrical simulations, and produces a characterization database. Another program, U2STF [3], generates the new synthesis model set from the characterization database. A separate synthesis model is necessary for each operating condition (process, temperature, voltage) to avoid derating errors.

Propagation delays are measured with the timing test patterns previously described. The Eldo electrical simulator has primitive functions to do the measurements. One simulation is done for each point in the timing table. Both propagation delay and rise time are measured for all timing events in each run, while load capacitance and input rise time are parameters which change for each run. One Eldo description used for an inverter is:

```
* LibQA Characterization Simulation
.GLOBAL VDD GND
VVDD VDD 0 2.700
VGND GND 0 0
.TEMP 125.00000
.option eps=1e-8
.include global/models.worst
.include IV/subckt
X0 Z A IV
C0 Z 0 0.018140pF
VA A 0 PWL(
+ 0US 0.00
+ 250PS 0.00 1000000PS 0.00
+ 1000250PS 2.70 2000000PS 2.70
+ 2000250PS 0.00 3000000PS 0.00
+ )
.extract tpdUD(A, Z,
+ vthin=1.080000,vthout=1.620000, after=1US)
.extract tfall(Z,
+ vh=2.295000, vl=1.350000, after=1US)
.extract tpdDU(A, Z,
+ vthin=1.620000, vthout=1.080000, after=2US)
.extract trise(Z,
+ vh=1.350000, vl=0.405000, after=2US)
.print tran V(Z)
.tran 1us 3us
.END
```

Often, a set of related events have negligible differences in propagation delay. LibQA groups these into equivalence classes. Where necessary, the classes are differentiated using sdf_cond attributes. This ability to decide which timing events to use gives an important contribution to accuracy.

Timing violations are measured with the violation test patterns previously described. A violation test tells whether the observable outputs failed to arrive at the expected value. A binary search on pass/fail gives a numerical result, but at more expense than the simpler delay measurements. Because each simulation run tests all the constraints, a separate binary search record is maintained for each one.

9. Timing Report Generation

LibQA tests the characterization database rather than the completed synthesis model because individual derating information is merged into global derating k_factors, so good scrutiny is not possible there. LibQA checks for properties that experience has linked to characterization problems. Statistics for each class of event are calculated, and individual events are compared to the statistics. These statistics are also used during synthesis model regeneration to calculate the k_factors. Some specific tests are:

- Monotonicity. Timing functions are tested for monotonicity with respect to all of their independent variables: input rise time, load, temperature, voltage, and process.
- Range. Check the range and domain of all timing functions. No timing should ever be negative, non-numeric, or infinite.
- Extreme values. For each class of event, the ten extreme cases are reported in ranked order. This can be interesting reading.

10. Timing Graph Viewer and Calculator

The timing graph viewer reveals timing properties difficult to recognize in tabular data. Timing characterization can produce subtle problems, but looking at every timing graph for a library will often reveal defects. The graphs can be output in printable format.

LibQA's calculator gives a numerical evaluation for any timing in the library.

11. Results

LibQA was used to develop a 151-cell library of 0.5micron CMOS standard cells. Two of the cells (buskeepers) required hand-generated stimuli because they had the ‘dont_use’ property and no functional description. Using a Sparc 10, functional verification using Eldo took 72 minutes, and some flip-flops having set and clear inputs were found incorrect when set and clear are applied together. Characterization for the library took 49 hours. As a result of the timing report, some of the max_cap values were corrected, and the choice of transition time and net_capacitance sample points were optimized. The VHDL simulation took 130 minutes, and an array out of bounds error due to a hand edited cell, and a number of undetected violations were revealed. The development time for LibQA was less than the time previously spent preparing tests, the coverage was much higher, and new classes of testing were applied. Finally, the time-to market for our latest library was significantly faster than the previous generation, and there have been no bug reports on the library.

12. Conclusions

LibQA is an industrial tool used to support the production and test of VHDL synthesis and simulation models. The application of formal methods increases precision and lowers costs. LibQA requires manual stimuli for cells which cannot be expressed in the syntax of the synthesis models, but it is fully automatic for 147 of 151 cells in a 0.5 micron CMOS standard cell library.

13. References

[1] “Introduction to Automata Theory, Languages, and Computation”, Hopcroft and Ullman, Addison-Wesley Publishing Co., 1979

[2] “Library Compiler Reference Manual” Synopsys, Inc., March 1994

[3] “U2STF User Guide”, SGS-Thomson Microelectronics, November 1994

[4] “ELDO Electrical Circuit Simulator”, Anacad, November 1994