PSL/SVA Assertions in SPICE

Donald O'Riordan
Cadence Design Systems, Inc.
2655 Seeley Ave
San Jose, CA
+1 408 428 5794
riordan@cadence.com

Prabal Bhattacharya
Cadence Design Systems, Inc.
2655 Seeley Ave
San Jose, CA
+1 408 894 2508
prabal@cadence.com

ABSTRACT

Assertion-based verification is a key aspect of any complete SoC or Silicon Realization flow. In this paper, we discuss how PSL (Property Specification Language)/SVA (System-Verilog Assertions) assertion semantics are extended for the first time to SPICE (Simulation Program with Integrated Circuit Emphasis)-level netlists and evaluated within a SPICE simulator, and present multiple examples and simulation results. Both inline pragma-based assertions (within SPICE subcircuits) and separate vunit/bind file based assertion methodologies (which reference objects within SPICE subcircuits) are covered. SPICE electrical quantities (analog node voltages, currents etc.) are referenced within the Boolean layers of the assertions, and analog behavioral modeling concepts borrowed from Verilog-AMS are also included both to enrich the expressiveness of the properties being asserted and to specify the clocking scheme for the assertion sampling. We also describe various control aspects i.e. how the SPICE simulator can be selectively programmed/controlled to:

(a) Read external vunit files,

(b) React to PSL/SVA assertions triggering (treating some as warnings, others as errors leading to early termination of the simulation),

(c) Flexibly specify for which properties to generate assertions waveforms to the simulation results database, etc.

With assertion based capabilities applied to SPICE simulators as described herein, we extend the benefits of assertion based verification to SPICE-based users (includes analog and mixed signal users) and introduce an important verification bridge between HDL (Hardware Description Language) and SPICE based design and verification communities.

1. INTRODUCTION

In analog (just as for mixed-signal or digital) block design, there is a need to specify certain design properties which are to be verified during simulation. While various ad-hoc methods have been employed to this end by analog designers over the years, it is increasingly being found that the lack of standardized approaches in this general area is leading to verification and interoperability problems when attempts are made to subsequently integrate the analog design IP (Intellectual Property) into a larger mixed signal context.

1.1 PREVIOUS WORK

Various ad-hoc methods have been employed in the past to specify various properties which are to be verified in analog/transistor level simulation. In a leading vendor solution, simulation waveform post-processing techniques require SKILL[1]-based calculator measurements, meaning they cannot be ported to digital/SOC-based verification environments (SKILL is not present in those environments), and even if they could, the performance impacts would be challenging. HSPICE[2] *.measure* statements, while popular, again suffer from portability problems. The lack of support for Ultrasim[3] device checks in Spectre[4], and the corresponding lack of support for Spectre checklimit analyses in Ultrasim, perhaps best serves to illustrate the issues with lack of standardization, even within single-vendor SPICE[5] and FastSPICE[6] simulators that otherwise consume the same input decks. Further, the existing approaches are very limited in terms of their ability to specify *predicated* functional behaviors *(if <pre_condition> |=> post_condition)*.

Additional problems are attributed to disparate user interfaces and environments both for setting up the properties to be verified, for interacting with the simulation results (e.g. even a simple assertion summary/dashboard), and the ability of properties or assertions to be ported across the multiple abstractions/models of a given cellview (e.g. Verilog[7] models, *wreal[8]*-based models, Verilog-AMS[9] models and SPICE subcircuit models) used in the accuracy/performance tradeoff (see Figure 1)

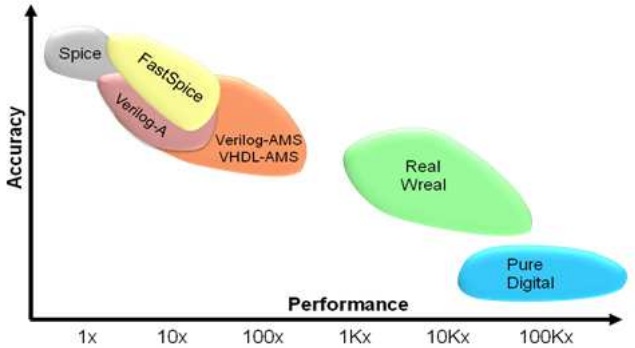

Figure 1 Model Abstraction Tradeoff

While many companies crave a standardized approach to such assertion management, some have found even *co-simulation-based* solutions employing the standard PSL/SVA languages to be either excessive in terms of setup cost (for a mixed signal/digital simulator such as NCSIM or VCS), or simply unpalatable to their heavily SPICE-based design community. Finally, due to these issues, many customers have actually abandoned efforts to continuously verify 'analog' blocks when integrating them in a mixed signal/SOC (System-On-Chip) environment, leading to 'plug and pray' based integration attempts (and the subsequent mixed signal tapeout nightmares).

2. PSL/SVA ASSERTIONS IN SPICE

It is a desire of this present work to overcome the challenges associated with the previous work, in order to allow for *standard* property/assertion languages (PSL[10], SVA[11]) to be used directly within a Spectre/SPICE based simulator and simulation environment. A primary objective is to facilitate the transfer of assertions and properties across multiple representations of a design cell, such that the same basic assertions can 'travel with the design' as the IP is integrated (i.e. debugged) and verified in bigger contexts. The same assertions should therefore be able to travel with the design *in the same basic form* originating with the SPICE-based simulator, into the Verilog-AMS simulator, and even into the extremely fast RNM (Real-Number-Modeling)-based event driven simulator (Verilog with wreal extensions).

Both PSL and SVA standards were chosen, the former due to its largely 'language-agnostic' nature leading to possibilities to (relatively) easily create SPICE flavors thereof, while the latter (which is rapidly growing in popularity in the digital D&V (Design & Verification) communities) is also potentially closely aligned with Verilog-A/MS due to (some) common Verilog-based language roots.

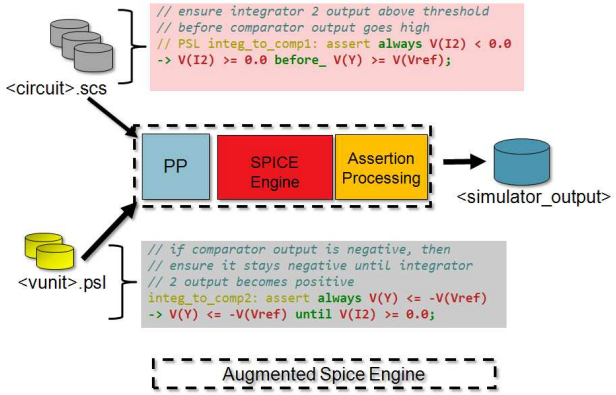

Figure 2 Prototype Overall Architecture

Our study of prior approaches (some[12] academic, some[13] from industrial/CAD settings) to PSL/SVA integration with analog type simulators or waveform processors led to the conclusion that it should be possible to achieve a high level of functional coverage in the analog assertion/property space without significant (or indeed any!) modifications to the assertion language syntax or semantics themselves, provided a few basic tenets were followed. These tenets include introduction of 'analog' terms to the Boolean layer of the assertion stack show in Figure 3 (for this we chose the expression sub-grammar of the Verilog-A[14] language), ensuring these expressions still return Boolean values (i.e. implication of analog thresholding/ relational operations) combined with a liberal usage of the assertion modeling layer (again, Verilog-A based) for any required state machine modeling or convenience functions.

Figure 2 illustrates the overall architecture deployed in the implementation of the prototype.

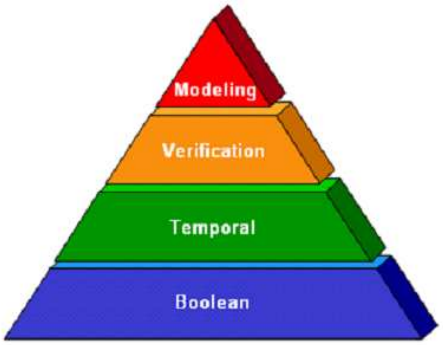

Figure 3 Primary Layers of an Assertion Stack

An assertion pre-processor was added to the simulator's parser to read PSL and SVA assertions either in the form of pragmas embedded within the SPICE subcircuit files or in the form of PSL/SVA statements embedded within PSL vunit files. A binding process was added to ensure that the assertion expressions are bound to the corresponding SPICE circuit nodes.

The SPICE engine (Spectre) was further augmented with an assertion evaluation engine which dynamically monitors the results of the analog expressions during transient simulation, coupling those results into appropriately constructed FSM (Finite State Machine) models in order to dynamically evaluate the temporal aspects of the assertion (e.g. sequences, implication operators enabling predicated assertions, etc). These FSM models were especially created to follow the temporal semantics of PSL and SVA.

We finally extended the simulator's 'back end' to produce simulation waveforms (see Figure 8) that represent the assertion evaluation status as a function of simulation time.

While some attention was paid in the prototype implementation to minimize simulation overhead in both the evaluation of and processing of assertion waveforms,

we expect that further attention can lead to additional optimizations in a commercial offering.

3. ASSERTION CONTROL STATEMENTS

In order to provide a large degree of control and flexibility to users, the simulator's parser was also extended to parse a variety of new *assertion control statements* by which the user can instruct the simulator:

- which vunit files to process/include
- which actions to take upon an assertion triggering (e.g. issue a warning and continue simulating, or error out, terminating simulation)
- for which assertions to save status waveforms

The syntax and semantics for these new assertion control statements is intentionally quite similar to existing simulator control statements for general options, saving regular node waveforms, and for message/action control, supporting scoping, wildcarding via regular expressions[15], and exclusion lists. (While the exact syntax and semantics may vary somewhat in a production release of the software, we do expect that they will at least be similar in spirit/intent to our prototype implementation.) Based on these control statements, the simulator monitors the assertion waveforms and issues messages to the simulation logfile, governed appropriately by the user specified options.

The assertion *control* statements are maintained separately from the assertion properties themselves i.e. they are placed in the SPICE netlist along with the regular SPICE control statements, and not in the assertion pragmas or vunit files. This intentional separation promotes reuse of the core assertions themselves as much as possible across different types of simulators, which often provide different assertion control capability (such as Tcl statements in a mixed-signal Verilog-AMS simulator).

```
a1 aoptions vunit_files=["basic.psl"
"extended.psl"]
asave ids=[".*"] excludes=[".*pos*"]
subckts=["ADC"]
a2 aaction ids=[".*pos.*"] message="OH, DEAR!"
level="warning"
a3 aaction ids=[".*"] excludes=[".*pos.*"]
message="THE SKY IS FALLING!" level="fatal"
```

Figure 4 Assertion Control Statements

Sample assertion control statements are listed in Figure 4. Here, three new types of statements have been added to the SPICE simulator's parser:

- `aoptions`. These act as global assertion options, analogous to regular simulator options.
- `asave`. These act as assertion 'save' statements, indicating for which assertions waveforms are to be produced and saved to the waveform database. (The obvious analogy is the simulator's typical ability to save simulation nodes/currents)
- `aaction`. These act as action statements, instructing the simulator on what action to take when assertions trigger.

The leading 'a' on these statements are used to denote (a)ssertion control.

Further details on these three types of assertion control statements and their provided flexibility follow below.

3.1 `aoptions` Statement Details

The `aoptions` statement is a global assertion options statement which instructs the simulator about the list of separate vunit files (which contain assertion properties) that are to be bound to the simulation

```
a1 aoptions vunit_files=["basic.psl"
"extended.psl"]
```

The `aoptions` statement accepts a vector argument `vunit_files`. This argument specifies a space-separated list of file pathnames, each file of which is expected be a PSL vunit file, containing standard PSL statements, including clocking statements, modeling layer statements, and of course assertion statements. In the above example, two files "basic.psl" and "extended.psl" are specified.

3.2 `asave` Statement Details

The `asave` statements instruct the simulator on specifically which assertions are to be saved as waveforms.

```
asave ids=[".*"] excludes=[".*pos*"]
subckts=["ADC"]
```

In order to provide maximum flexibility, a regular expression (*regexp*) scheme is used in order to specify a list of assertion inclusion id's (each assertion specified in a vunit file or embedded as a pragma within a SPICE subcircuit/netlist is associated with a unique id). The `ids` parameter allows a list of such regular expressions (space separated, double quote delimited) to be provided. An exclusion list (again, a list of regular expressions) can also be provided via the `excludes` parameter. Finally, each `asave` statement can be associated with a list of subcircuits to which it applies; this list is specified in the `subckts` parameter, again a list of regular expressions.

For each of the SPICE subcircuits that match any of the *regexps* specified in the `subckts` parameter, its list of associated assertions are traversed, and any found which match the *regexp* list specified in the `ids` parameter are tentatively marked for waveform saving. Any which are additionally found to match the `excludes` list of *regexps* are removed from that tentative list. Waveforms are then saved for those which remain in the list (see highlighted waveform example in Figure 8). This scheme provides a

huge amount of flexibility to the user in order to reduce waveform database size.

3.3 `aaction` Statement Details

The `aaction` statements instruct the simulator on what explicit actions to take when certain assertions are triggered or 'fire'.

```
a3 aaction ids=[".*"] excludes=[".*pos.*"]
message="THE SKY IS FALLING!" level="fatal"
```

This statement again takes an inclusion list, a optional exclusion list, and an optional scope modifier (subckts list), allowing with a message, and a severity level.

For any matching regular expression (see `asave` statement details section above for a definition of matching) which fires during simulation, the given message is printed to the simulator's output/log file, and treated with the given severity level. This allows some assertions to be treated as informational, others to be treated as warnings, and yet others to be treated as fatal. Fatal errors will terminate the (transient) analysis being performed by the simulator.

4. ASSERTION PRAGMAS

Assertions can be created in separate *vunit* files as noted previously, or can alternately be embedded directly within SPICE subcircuits via assertion *pragmas*. These pragmas act just like assertion pragmas in leading Verilog/VHDL simulators, appearing in a *// comment-like syntax*, such as the `atest0`, `atest1` and `atest2` assertion pragmas which appear in the listing of Figure 5.

```
simulator lang=spectre
global gnd

// SVA atest0: assert property ( @( "cross(V(A)-
1.0)" )  ("V(B) > 0.8" ##[3:5] "V(C) < 0.0" ) )
;

// PSL atest1: assert always {"V(B) > 0.8";
[*3:5]; "V(C) < 0.0" } @( "cross(V(A)-1.0)" );

// PSL atest2: assert always {"V(B,gnd) < 0.8";
[*2:6]; "V(C) > 0.0" } @( "cross(V(A)-1.0)" );

parameters pvdd = 1.1 \
           pR = 1.0 \
           pSt = 0.1

R1 C gnd resistor r=pR
C1 C gnd capacitor c=0.01
Cin B C capacitor c=0.02
Rin A B resistor r=2

EA A gnd vsource type=pulse \
                 val0=0    val1=pvdd    period=5
rise=0.5 fall=0.5 width=2 delay=pSt

tran1 tran stop=100
```

Figure 5 SPICE Netlist containing Assertion Pragmas

Note that in Figure 5, two syntaxes are evident in the pragma statement, the first is an (abbreviated) SVA syntax and the second is a PSL syntax. The first two of these assert that once node B has been determined to have a voltage > 0.8 volts, then shortly thereafter node C must have a voltage which is < 0.0 volts. Sampling (assertion check and update) occurs whenever node A voltage cross above or below a threshold of 1.0 volts, according to the semantics of the Verilog-A cross statement used as an explicit assertion clocking expression. 'Shortly thereafter' refers to a range of 3 to 5 such sampling points or clock cycles. The remaining portion of the listing of Figure 5 instantiates some SPICE level devices (resistors, capacitors) and stimulus (vsource), along with instructions on how long to perform a transient analysis simulation.

5. EXPERIMENTAL EVIDENCE

The assertion-capable SPICE prototype was employed to perform ABV (Assertion-Based Verification[16]) of a second order Sigma-Delta (17) based ADC (Analog-to-Digital Converter) circuit. Figure 6 shows a typical implementation choice for the modulator portion of such a circuit, employing switched-capacitor based integrators.

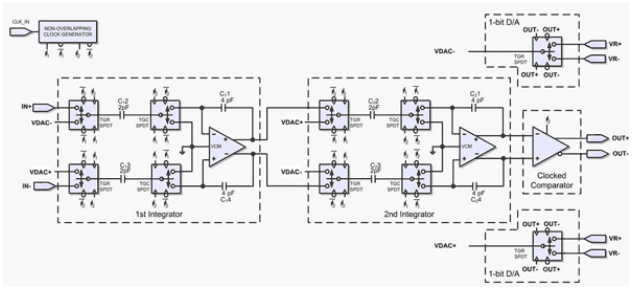

Figure 6 Second order switched capacitor implementation of Sigma-Delta modulator

In order to decrease simulation time, Verilog-A models were employed for both the analog modulator and the digital decimating filter components of the ADC, but SPICE subcircuits could have equally been substituted. Figure 7 shows the block diagram of the Verilog-A model used to represent the modulator. All nodes (X, E1, I1, E2, I2, Y) within the modulator are of type (Verilog-A) `electrical`, and the Spectre simulator was used to perform the simulations running on the Linux operating system.

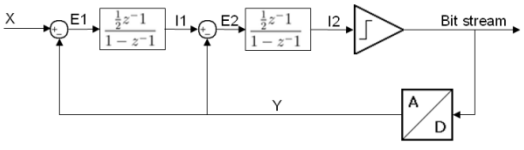

Figure 7 2nd order modulator architecture with integrators modeled in Z-domain

The code listing of Figure 9 illustrates some of the basic modulator properties for integrator and comparator functionalities within the feedback loop, captured in the PSL language. Verilog-A is used (expressions within double quoted strings, another prototyping shortcut) for the Boolean layer terms. The properties themselves are combined with some (convenience) variables in Verilog-A modeling statements, along with a Verilog-A timer-based clocking expression used to strobe the expression evaluation, and all encapsulated within a `vunit` that is bound to the ADC subcircuit.

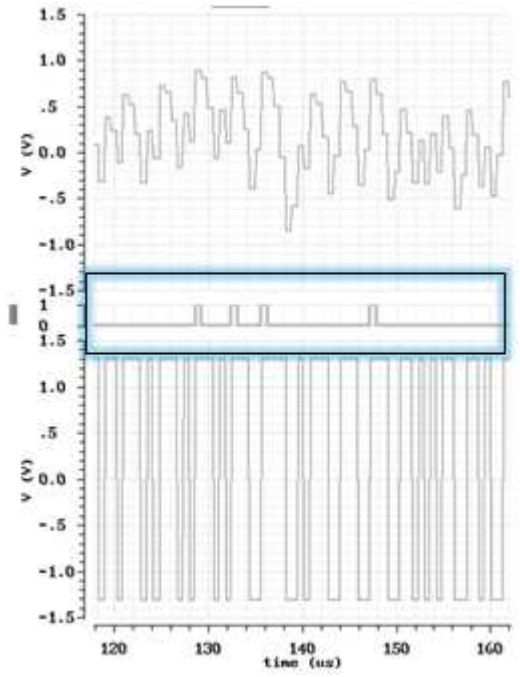

Figure 8 Simulation waveforms of analog integrator voltage (top), related digitized assertion status waveform (middle) and modulator feedback stream (bottom)

```
vunit my_psl_vunit(ADC) {
// DEFAULT CLOCK FOR ASSERTIONS
default clock = ("timer(254.5*80e-9, 8*80e-9)");

// modeling layer.
// Create some expression placeholders
// (used in pos_integ1 assertion)
integer i1_pos, i1_inputs_pos;
i1_pos = V(I1) > 0.0;
i1_inputs_pos = (V(X) > 0.0) && (V(I1) > 0.0) && (V(Y)
<= -V(Vref));

// INTEGRATORS and DIFF JUNCTIONS,
// basic behavior

// ensure preservation of arithmetic sign, positive
pos_integ1: assert always { "i1_inputs_pos" } |=>
"i1_pos";
// ensure preservation of arithmetic sign, negative
neg_integ1: assert always { "(V(X) < 0.0) && (V(I1) <
0.0) && (V(Y) >= V(Vref))" } |=> "V(I1) < 0.0";

// COMPARATOR BASIC FUNCTIONALITY

// if the input to the comparator
// (integrator 2 output) is positive,
// ensure the comparator detects that
// immediately, and vice versa
comparator_pos: assert always "V(I2) > 0.001" -> "V(Y)
>= V(Vref)";
comparator_neg: assert always "V(I2) < -0.001" ->
"V(Y) <= -V(Vref)";

// ensure integrator 2 output above threshold
// before comparator output goes high
integ_to_comp1: assert always "V(I2) < 0.0" -> "V(I2)
>= 0.0" before_ "V(Y) >= V(Vref)";

// if comparator output is negative, then
// ensure it stays negative until integrator
// 2 output becomes positive
integ_to_comp2: assert always "V(Y) <= -V(Vref)" ->
"V(Y) <= -V(Vref)" until "V(I2) >= 0.0";

}
```

Figure 9 PSL Code listing Basic Modulator Properties

By way of example, the `neg_integ1` assertion checks a fundamental property that whenever the input to the I1 integrator is negative, and its current output is negative, and the feedback voltage Y is positive, then the next expected output from the integrator must again be negative. (Such a property could be violated in a post-extracted simulation via substrate noise coupling for example). The `pos_integ1` assertion checks the mirror property, and the remaining assertions are explained via the in-lined comments in Figure 9.

The ADC sub-circuit and testbench listing for Spectre (in `.scs` format) follows in Figure 10.

```
// ADC TESTBENCH
simulator lang=spectre

// include modulator and filter models
```

```
ahdl_include "sd_behav.va"
ahdl_include "filter_decimator_behav.va"

parameters Tsig=3.2768000e-04 Tclock=8*80e-9
// Tsig = 512*Tclock => OSR=256
// Nyquiest rate



subckt ADC (X dout)
// instantiate the analog modulator
mod1 (X E1 I1 E2 I2 Vref Y) sd period=Tclock Vref=1.3
outStart=0 gn1=0.5 gn2=0.5

// instantiate the digital
// filter/decimator, with decimation
// rate 1/16th of OSR
df1 (Y dint1 dint2 dint3 ddiff0 ddiff1 ddiff2 ddiff3
dout)      filter_decimator      tperiod=Tclock      +
osr=Tsig/Tclock/16
ends // ADC

// instantiate the ADC
i1 (X dout) ADC

// give it some STIMULUS.
v1   (X   0)   vsource   type=sine   ampl=0.65   sinedc=0
freq=1/Tsig

// save all subckt nodes
save * i1.* depth=all

// the transient analysis
timedom         tran         stop=0.01*2**7*Tsig-Tclock
maxstep=0.05*Tclock
```

Figure 10 Spectre source listing of ADC circuit and Stimulus

6. CONCLUSIONS

With assertion based capabilities applied to SPICE simulators as described herein, we extend the benefits of assertion based verification to SPICE-based users (includes analog and mixed signal users) and introduce an important verification bridge between the long-isolated HDL and SPICE based design and verification communities.

7. REFERENCES

1. Cadence SKILL. [Online] http://en.wikipedia.org/wiki/Cadence_SKILL.
2. HSPICE. [Online] http://www.synopsys.com/Tools/Verification/AMSVerification/CircuitSimulation/HSPICE/Pages/default.aspx.
3. Virtuoso UltraSim Full-Chip Simulator. [Online] http://www.cadence.com/products/cic/UltraSim_fullchip/pages/default.aspx.
4. Cadence Virtuoso Spectre Circuit Simulator. *www.cadence.com.* [Online] http://www.cadence.com/products/rf/spectre_circuit/pages/default.aspx.
5. SPICE. [Online] http://en.wikipedia.org/wiki/SPICE.
6. Second generation circuit simulation - Fast-SPICE. [Online] http://www.allabouteda.com/second-generation-circuit-simulation-fast-spice/.
7. Verilog. [Online] http://en.wikipedia.org/wiki/Verilog.
8. Real Valued Modeling for Mixed Signal Simulation. [Online] http://www.cadence.com/rl/Resources/application_notes/real_number_appNote.pdf.
9. Verilog-AMS Language Reference Model. [Online] http://www.eda.org/verilog-ams/htmlpages/public-docs/lrm/2.3.1/VAMS-LRM-2-3-1.pdf.
10. 1850-2010 IEEE Standard for Property Specification Language (PSL). *IEEE Xplore.* [Online] April 6 , 2010 . http://ieeexplore.ieee.org/servlet/opac?punumber=5445949.
11. 1800-2009 IEEE Standard for System Verilog-Unified Hardware Design, Specification, and Verification Language. *IEEE Xplore.* [Online] 2009. http://ieeexplore.ieee.org/servlet/opac?punumber=5354133.
12. Towards Assertion based Verification of Analog and Mixed Signal Designs using PSL. [Online] http://hvg.ece.concordia.ca/Publications/Conferences/FDL%2707_02.pdf.
13. Assertion Based Analog Mixed Signal Verification. [Online] http://www.vhdl.org/verilog-ams/htmlpages/public-docs/AMS_Assertions/AnalogAssertions_AccelleraProposal_2008_08_12.pdf.
14. Verilog-A Language Reference Manual. [Online] August 1, 1996. http://www.vhdl.org/verilog-ams/htmlpages/public-docs/lrm/VerilogA/verilog-a-lrm-1-0.pdf.
15. Regular expression. [Online] http://en.wikipedia.org/wiki/Regular_expression.
16. Everything Assertion Based -- Assertion-Based Verification (ABV) Comes of Age for Complete Block-Level Verification. [Online] http://www.cadence.com/Community/blogs/fv/archive/2010/12/02/everything-assertion-based-assertion-based-verification-abv-comes-of-age-for-complete-block-level-verification.aspx.
17. Delta-sigma modulation. [Online] http://en.wikipedia.org/wiki/Delta-sigma_modulation.