SystemVerilog AMS Checkers

Synopsys Inc.

Abstract

This document describes an implementation of AMS checkers in SystemVerilog (SV). The objective is to show how such checkers can be constructed within the existing standard. It is relatively at a low level of abstraction due to limitations imposed by the language. Nevertheless, it allows expressing most constructs defined and implemented in the Python assert_temporal and assert_window assertions. The SV checkers can execute online in an SV simulation, very much like the existing SV Assertions. If support for arithmetic on real numbers is provided the checkers can also execute on-line in emulation.

Some relatively simple extensions to the SV language are also proposed. They would simplify the development of such checkers. More profound extensions to SystemVerilog will be stated in a separate document. In that case the expressive power will match the Python version.

Keywords: SystemVerilog, Fourier transform, assertions, simulation, emulation, AMS.

1. Introduction

The document describes a simple architecture of an AMS checker entirely in SystemVerilog [1]. Due to limitations of the language, it is at a lower level of abstraction than the primary specification of the AMS language, currently implemented in Python [2]. Yet it does support assert_temporal and assert_window assertions described in the document. In addition, it allows using SystemVerilog Assertions (SVA) inside the checkers. Since the SV implementation is LRM compliant, it can be compiled and executed in simulation together with the design it is supposed to check. The checkers can be used in emulation, provided that support for real numbers is available.

The document is organized as follows:

Section 2 outlines the general structure of the checker: the encapsulation, main body blocks and the supporting SV package.

Section 2 illustrates the proposed architecture on 2 checkers, one that verifies a noisy sinusoidal signal, and the other a DAC device.

Section 4 discusses some minimal extensions to the SV language that would simplify writing AMS checkers.

2. Checker Architecture

The checkers support both frequency-based checks and temporal assertions. For the former, an SV package is provided to implement a sliding window over the signal valuations in time. It also includes a Discrete Fourier Transform (DFT) for generating a frequency spectrum for each such window. A more efficient implementation can always be substituted for the open source version of DFT (translated to SystemVerilog) used here. The windows and the spectra are

stored and passed around in dynamic arrays which allows creating a generic version of the package, independent of the window size.

The checker is encapsulated in a parameterized SV *module*. The parameters are used to define the dimensions of the windows, time / frequency resolution, and constants used in checks (e.g., expected gain, SNR, SFDR, Power limits, expected delays, etc.)

A better solution than *module* would be using the *checker* construct due to its flexibility in port definitions, however, dynamic arrays are not supported there. A parameterized class-based formulation could provide flexibility; however, concurrent assertions cannot be used within classes.

The organization of the checker body is as follows

```
module Name #(… parameters …) (… ref ports …);

   local parameters if any, derived from module parameters;

   dynamic array declarations for signal windows;
   dynamic array declarations for associated spectra;
   creation of the arrays based on window size;

// there are as many of the following three declarations as
// there are clock domains
   int startCount = 0; // counter of clock tics to fill the first window
   bit running = 1'b0; // indicates that the first windows has been filled
   clocking clk @ … endclocking // defines clock domain clock and signal
sampling

   function void action();
      // performs calls to FFT and any signals analysis needed for assertions
   endfunction

   Concurrent assertions, if any;

// an always procedure for each clock domain
   always @(clk) begin
      sample(); // call to sampling function from package
      if (!running) begin
         startCount++;
         running = startCount >= win_size-1;
      end else begin
         action();

         immediate assertions for quantities computed in action
      end
```

```
   end
endmodule
```

The supporting package is as follows:

```
package AnalogAssertionsPkg;
    typedef int unsigned uint;
    const real  M_PI = 3.14159265358979323846264338327795;

    typedef struct { real re; real im; } Complex;
    typedef struct { real mag; real arg; } Polar;
    typedef struct { real mag; uint bin; } MagBin;

    function real abs(Complex a);
        return ($sqrt(a.re**2 + a.im**2));
    endfunction // abs
    function real arg(Complex a);
        return ($atan2(a.im, a.re));
    endfunction // arg
    function Complex add(Complex a, b);
        return ('{a.re+b.re, a.im + b.im});
    endfunction // add
    function Complex mult(Complex a, b);
        return ( '{ (a.re*b.re - a.im*b.im), (a.re*b.im + a.im*b.re) });
    endfunction // mult
    function Polar complex2Polar(Complex a);
        return ('{ abs(a), arg(a) });
    endfunction // complex2Polar
    function Complex polar2Complex(Polar a);
        Complex tmp;
        tmp.re = a.mag * $cos(a.arg);
        tmp.im = a.mag * $sin(a.arg);
        return tmp;
    endfunction // polar2Complex

    function bit cmpWTolerance(real x, value, tolerance);
        return ( (x < value + tolerance) &&
                (x > value - tolerance));
    endfunction // cmpWTolerance

    // Spectrum is expected to be of size win_size, but it may be the
    // we will have to preserve the preceding spectrum for incremental
    //  DFT?
    typedef Complex Spectrum[];

    // signal windows are expected to be of size win_size
    typedef real   RealWindow[];

    function void rSample(input real x, ref RealWindow w, input uint w_size);
        int        maxIdx;
        maxIdx = w_size-1;
        foreach(w[i])
            if (i<maxIdx)
                w[i] = w[i+1];
```

```
        w[maxIdx] = x;
    endfunction // RealWindow

    function void uiSample(input uint x, ref RealWindow w, input uint
w_size);
        int        maxIdx;
        maxIdx = w_size-1;
        foreach(w[i])
            if (i<maxIdx)
                w[i] = w[i+1];
        w[maxIdx] = $itor(x);
    endfunction // UintWindow

    function void iSample(input int x, ref RealWindow w, input uint w_size);
        int        maxIdx;
        maxIdx = w_size-1;
        foreach(w[i])
            if (i<maxIdx)
                w[i] = w[i+1];
        w[maxIdx] = $itor(x);
    endfunction // IntWindow

    function MagBin maxAbsBin(Spectrum Sp, input uint lowIdx, highIdx);
        real       mx;
        uint sz, bin;
        mx = 0;
        bin = 0;
        sz = Sp.size/2;
        for (uint i = lowIdx; i <= highIdx; i++)
          if (((i < sz) && (abs(Sp[i]) > mx))) begin
              mx = abs(Sp[i]);
                bin = i;
            end
        return '{mx, bin};
    endfunction // maxAbs

    function real getFrequency(int i, real fStep);
        return (i * fStep);
    endfunction // getFrequency

// convert data types from real window and spectrum to separate real
// and imaginary arrays needed by the windowing and DFT function.
// These functions should be modified to use the RealWindow and Spectrum
// data types.
function void realBuildSpectrum(ref Spectrum Sp, ref RealWindow rw);
        uint w_sz;
        real re [];
        real im [];
        w_sz = rw.size();
        re = new[w_sz];
        im = new[w_sz];
        applyHannWindow(rw, re, im, w_sz);
        Fft_transform(re, im, w_sz);
        foreach (re[i]) begin
            Sp[i].re = re[i];
            Sp[i].im = im[i];
```

```
        end
    endfunction // realBuildSpectrum


    function void deleteFrequencyBin
            (ref Spectrum sourceSp, ref Spectrum targetSp,
             real freq, real fStep);
   // replace bin at frequency freq by Complex 0, frequency step is fStep
        foreach (sourceSp[i]) begin
          if ( i*fStep < freq && (i+1)*fStep > freq ) begin // freq is in the
middle
              targetSp[i] = '{0, 0};
              targetSp[i+1] = '{0, 0};
          end
          else if ( i*fStep == freq ) begin
              targetSp[i] = '{0, 0};
          end else begin
              targetSp[i] = sourceSp[i];
          end
        end // foreach (sourceSp[i])
    endfunction // deleteFrequencyBin

    function void deleteSpectrumBin
            (ref Spectrum sourceSp, ref Spectrum targetSp, input uint index);
        // replace bin at index by Complex 0
        foreach (sourceSp[i])
          if (i == index)
              targetSp[i] = '{0, 0};
          else
              targetSp[i] = sourceSp[i];
    endfunction // deleteSpectrumBin

    function real sumPwr(ref Spectrum sp, input uint lowIdx, highIdx);
        uint i, win_size;
        real pwr, tmp;
        pwr = 0;
        win_size = sp.size();
        for (i = lowIdx; i <= highIdx; i++) begin
            tmp = (2 * abs(sp[i])/win_size)**2;
            pwr += tmp;
        end
        return pwr;
    endfunction // sumPwr

    function real sumAbs(ref Spectrum sp, input uint lowIdx, highIdx);
        uint i, win_size;
        real mg, tmp;
        mg = 0;
        win_size = sp.size();
        for (i = lowIdx; i <= highIdx; i++) begin
            tmp = (2 * abs(sp[i])/win_size);
            mg += tmp;
        end
        return mg;
    endfunction // sumAbs
```

```
    /*
     * Free FFT and convolution (C)
     * Recoded in SystemVerilog (E.C.)
     *
     * Copyright (c) 2017 Project Nayuki. (MIT License)
     * https://www.nayuki.io/page/free-small-fft-in-multiple-languages
     *
     * For commercial use obtains license or use another open source DFT
     */

    function bit Fft_transform(ref real re[], ref real im[], input uint n);
        if (n == 0)
          return 1'b0;
        if ((n & (n - 1)) == 0)  // Is power of 2
          return Fft_transformRadix2(re, im, n);
        else  // More complicated algorithm for arbitrary sizes
          return Fft_transformBluestein(re, im, n);
    endfunction // Fft_transform

    function bit Fft_inverseTransform(ref real re[], ref real im[], input
uint n);
        return Fft_transform(im, re, n);
    endfunction

    function bit Fft_transformRadix2(ref real re[], ref real im[], input uint
n);
        // Length variables
        uint j;
        uint l;
        real   tpre;
        real   tpim;
        real   temp;
        uint halfsize;
        uint tablestep;
        real   cos_table[];
        real   sin_table[];
        uint levels;  // Compute levels = floor(log2(n))
        uint size;
        size = (n / 2);
        cos_table = new[size];
        sin_table = new[size];
        levels = 0;
        for (uint temp = n; temp > 1; temp >>= 1) begin
          levels++;
        end

        if (1 << levels != n) begin
            $display(" 1 << levels %0d, n = %0d", 1 << levels, n);
          return 1'b0;  // n is not a power of 2
        end
      // Trignometric tables
        for (uint i = 0; i < n / 2; i++) begin
          cos_table[i] = $cos(2 * M_PI * i / n);
          sin_table[i] = $sin(2 * M_PI * i / n);
        end
```

```
        // Bit-reversed addressing permutation
        for (uint i = 0; i < n; i++) begin
            j = reverse_bits(i, levels);
          if (j > i) begin
              temp = re[i];
              re[i] = re[j];
              re[j] = temp;
              temp = im[i];
              im[i] = im[j];
              im[j] = temp;
          end
        end

        // Cooley-Tukey decimation-in-time radix-2 FFT
        for (uint size = 2; size <= n; size *= 2) begin
          halfsize = size / 2;
          tablestep = n / size;
          for (uint i = 0; i < n; i += size) begin
              for (uint j = i, k = 0; j < i + halfsize;
                                  j++, k += tablestep) begin
                l = j + halfsize;
                tpre =  re[l] * cos_table[k] + im[l] * sin_table[k];
                tpim = -re[l] * sin_table[k] + im[l] * cos_table[k];

                re[l] = re[j] - tpre;
                im[l] = im[j] - tpim;
                re[j] += tpre;
                im[j] += tpim;
                end
            end // for (uint i = 0; i < n; i += size)
          if (size == n)  // Prevent overflow in 'size *= 2'
              break;
        end // for (uint size = 2; size <= n; size *= 2)
        return 1'b1;
    endfunction // Fft_transformRadix2

    function bit Fft_transformBluestein(ref real re[], ref real im[], input
uint n);
        real cos_table[];
        real sin_table[];
        real areal[];
        real aimag[];
        real breal[];
        real bimag[];
        real creal[];
        real cimag[];
        real angle;
        uint temp;
        uint m;
        // Find a power-of-2 convolution length m such that m >= n * 2 + 1
        m = 1;
        while (m / 2 <= n) begin
          m *= 2;
        end
        cos_table = new[n];
        sin_table = new[n];
```

```
        areal = new[m];
        aimag = new[m];
        breal = new[m];
        bimag = new[m];
        creal = new[m];
        cimag = new[m];

        // Trigonometric tables
        for (uint i = 0; i < n; i++) begin
          temp = i * i;
          temp %= n * 2;
            angle = M_PI * temp / n;
        // Less accurate version if long long is unavailable:
        // double angle = M_PI * i * i / n;
          cos_table[i] = $cos(angle);
          sin_table[i] = $sin(angle);
        end
        // Temporary vectors and preprocessing
        for (uint i = 0; i < n; i++) begin
          areal[i] =  re[i] * cos_table[i] + im[i] * sin_table[i];
          aimag[i] = -re[i] * sin_table[i] + im[i] * cos_table[i];
        end
        breal[0] = cos_table[0];
        bimag[0] = sin_table[0];
        for (uint i = 1; i < n; i++) begin
          breal[i] = cos_table[i];
            breal[m - i] = cos_table[i];
          bimag[i] = sin_table[i];
            bimag[m - i] = sin_table[i];
        end
        // Convolution
        if (!Fft_convolveComplex(areal, aimag, breal, bimag, creal, cimag,
m))
          return 1'b0;

        // Postprocessing
        for (uint i = 0; i < n; i++) begin
          re[i] =  creal[i] * cos_table[i] + cimag[i] * sin_table[i];
          im[i] = -creal[i] * sin_table[i] + cimag[i] * cos_table[i];
        end

        return 1'b1;
    endfunction // Fft_transformBluestein


    function bit Fft_convolveReal(ref real x[], ref real y[], ref real out[],
input uint n);
        bit status;
        real ximag[];
        real yimag[];
        real zimag[];
        ximag = new[n];
        yimag = new[n];
        zimag = new[n];
        status = Fft_convolveComplex(x, ximag, y, yimag, out, zimag, n);
        return status;
```

```
    endfunction // Fft_convolveReal

    // currently works only for n being power of 2
    function bit Fft_convolveComplex(ref real xreal[], ref real ximag[],
                                     ref real yreal[], ref real yimag[],
                                     ref real outreal[], ref real outimag[],
input uint n);

        real                                    temp;
        real                                    xr[];
        real                                    xi[];
        real                                    yr[];
        real                                    yi[];
        xr = new[n];
        xi = new[n];
        yr = new[n];
        yi = new[n];

        xr = xreal;
        xi = ximag;
        yr = yreal;
        yi = yimag;

        //    $display("Fft on xr, xi");
        if (!Fft_transform(xr, xi, n))
          return 1'b0;
        //    $display("fft on yr, yi");
        if (!Fft_transform(yr, yi, n))
          return 1'b0;

        for (uint i = 0; i < n; i++) begin
          temp = xr[i] * yr[i] - xi[i] * yi[i];
          xi[i] = xi[i] * yr[i] + xr[i] * yi[i];
          xr[i] = temp;
        end
        if (!Fft_inverseTransform(xr, xi, n))
          return 1'b0;

        for (uint i = 0; i < n; i++) begin  // Scaling (because this FFT
implementation omits it)
          outreal[i] = xr[i] / n;
          outimag[i] = xi[i] / n;
        end

        return 1'b1;
    endfunction


    function uint reverse_bits(uint x, uint n);
        uint result;
        result = 0;
          result = (result << 1) | (x & 1);
        end
        return result;
    endfunction
```

```
    // Funstion to adapt BuildSpectrum to use Fft_transform
    // also to add hann window

    // apply hann to real window and return complex equivalent in
    // re and im arrays  suitable for calling Fft transform
    function void applyHannWindow(ref RealWindow rw, ref real re[],
                                  ref real im[], input uint n);
        real radians;
        radians = 2*M_PI/(n-1);
        foreach (rw[i]) begin
            re[i] = 0.5 * (1 - $cos(radians*i)) * rw[i];
            im[i] = 0;
        end
    endfunction // applyHannWindow

endpackage : AnalogAssertionsPkg
```

3. Checker Examples

The checkers included here are for illustration only. The calculation used for SNR, SFDR, etc. may not be as needed in a real application, but they can be easily changed.

The clock that runs on real time in the examples is expected to be implemented outside the checker using

```
always begin
    #1;
    ->clk;
end
```

or something similar but it has to preserve the time scale expected in the checker. This time unit management is critical for obtaining correct results from the frequency and temporal analysis.

A noisy sinusoid signal checker

```
module SINchecker
    #(parameter real sampFreq = 48000.0,
      // check limits
      parameter real fundFreq = 2500.00, // specified conversion factor
      parameter real freqTolerance = 600.00,
      parameter real absFundFreq = 1.0,
      parameter real absTolerance = 0.05,
      parameter real powerFundFreq = 0.433,
      parameter real powerTolerance = 0.01,
      parameter real minSNR = 57.25,
      parameter real snrTolerance = 2.0,
      parameter real minSFDR = 49.3,
      parameter real sfdrTolerance = 5.0,
      parameter uint win_size = 1024)
    (
```

```
 ref real  vIn,
 ref event clk // sampling clock
 );
localparam real       freqStep = sampFreq /win_size ;
int                   startCount = 0;
bit                   running = 1'b0;
real                  vInNoise, spurAbs, snr, sfdr;
real                  carrierAbs, fundamentalAbs;
real                  sideMin1Abs, sidePlus1Abs, fundamentalPwr;
uint                  carrierIndex, spurIndex;
real                  carrierFreq;
MagBin absBin1, absBin2;

AnalogAssertionsPkg::RealWindow vIn_w = new[win_size];
AnalogAssertionsPkg::Spectrum VIN = new[win_size];

clocking aClk @clk;
    default input #1step;
    input  vIn;
endclocking // aClk

function void action();
    // do DFT on signal
    //       foreach(vIn_w[i])
    //            $display("i = %0d, vIn_w[i] %10.3g", i, vIn_w[i]);
    realBuildSpectrum(VIN, vIn_w);
    $display("ACTION time = %0t", $realtime);
    foreach (VIN[i]) begin
        $display("i %0d, VIN[i].re %10.3g, VIN[i].im %10.3g",
                 i, VIN[i].re, VIN[i].im);
        $display("i %0d, power abs**2 / win_size %10.3g",
                 i, (2*abs(VIN[i])/win_size)**2);
    end
    // remove input frequencies from Vout
    vInNoise = 0; // cmpute RMS
    absBin1 =  maxAbsBin(VIN, 0, win_size-1);
    carrierIndex = absBin1.bin;
    carrierAbs = 2 * absBin1.mag/win_size;
    carrierFreq = getFrequency(carrierIndex, freqStep);
    fundamentalAbs = sumAbs(VIN, carrierIndex-1, carrierIndex+1);
    fundamentalPwr = sumPwr(VIN, carrierIndex-1, carrierIndex+1);
    vInNoise = sumPwr(VIN, 0, carrierIndex-2);
    vInNoise += sumPwr(VIN, carrierIndex+2, win_size/2-1);
    snr = $log10(fundamentalPwr / vInNoise); // check for 0 divisor
    snr = 20.0 * snr;
    absBin1 = maxAbsBin(VIN, 0, carrierIndex-2);
    absBin2 = maxAbsBin(VIN, carrierIndex+2, win_size/2-1);
    spurAbs = 2 * absBin1.mag/win_size;
    spurIndex = absBin1.bin;
    if (spurAbs < 2 * absBin2.mag/win_size) begin
        spurAbs = 2 * absBin2.mag/win_size;
        spurIndex = absBin2.bin;
    end
    sfdr = $log10(fundamentalAbs**2 / spurAbs**2); // check for 0 divisor
    sfdr = 20.0 *  sfdr;
endfunction // action
```

```
// time domain check for fundamental frequency
// disabled - too much noise to detect crossing
//    property FundFreq_p;
//        realtime tCross;
//        @(posedge (vIn > 0.0))
//            (1, tCross = $realtime) |=>
//             @(posedge (vIn > 0.0) )
//             cmpWTolerance(($realtime-tCross)* 1/freqStep,
                              fundFreq, freqTolerance);
//    endproperty
//    FundFreq_a: assert property(FundFreq_p);

    always @(aClk) begin : run
        rSample(aClk.vIn, vIn_w, win_size);
        if (!running) begin
          startCount++;
          running = startCount >= win_size-1;
        end else begin
          action();
          // frequency domain check on fundamental frequency
          FundFreqDFT_a: assert(cmpWTolerance(carrierFreq,
                                fundFreq, freqTolerance))
                $display("Fundamental Frequency PASSED, actual %10.8g,
required %10.8g, tolerance %10.8g, bin %0d",
                     carrierFreq, fundFreq, freqTolerance, carrierIndex);
                   else
                 $display("Fundamental Frequency FAILED, actual %10.8g,
required %10.8g, tolerance %10.8g, bin %0d",
                     carrierFreq, fundFreq, freqTolerance,carrierIndex);

          // check amplitude at fundamental frequency
          FundFreqAmplitude_a: assert
                (cmpWTolerance(fundamentalAbs, absFundFreq, absTolerance))
                $display("Fund Freq Abs PASSED, abs %10.3g, ref %10.3g,
toler %10.3g",
                      fundamentalAbs, absFundFreq, absTolerance);
                else
                    $display("Fund Freq Abs FAILED, abs %10.3g, ref %10.3g,
toler %10.3g",
                      fundamentalAbs, absFundFreq, absTolerance);

          // check max power at fundamental frequency
          FundFreqPower_a: assert
                (cmpWTolerance(fundamentalPwr, powerFundFreq,
                               powerTolerance))
                   $display("Fund Freq Pwr PASSED, pwr %10.3g, ref %10.3g,
toler %10.3g",
                      fundamentalPwr, powerFundFreq, powerTolerance);
                else
                    $display("Fund Freq Pwr FAILED, pwr %10.3g, ref %10.3g,
toler %10.3g",
                      fundamentalPwr, powerFundFreq, powerTolerance);

          // check SNR
          SNR_a: assert (cmpWTolerance(snr, minSNR, snrTolerance))
```

```
                $display("SNR PASSED, snr %10.3g, ref %10.3g, toler %10.3g",
                         snr, minSNR, snrTolerance);
                else
                    $display("SNR FAILED, snr %10.3g, ref %10.3g,
toler %10.3g",
                             snr, minSNR, snrTolerance);

          // check SFDR
          SFDR_a: assert(cmpWTolerance(sfdr, minSFDR, sfdrTolerance))
                $display("SFDR PASSED, sfdr %10.3g, ref %10.3g, toler %10.3g,
index %0d",
                         sfdr, minSFDR, sfdrTolerance, spurIndex);
                else
                    $display("SFDR FAILED, sfdr %10.3g, ref %10.3g,
toler %10.3g index %0d",
                             sfdr, minSFDR, sfdrTolerance, spurIndex);
        end
    end

endmodule // SINchecker
```

DAC checker

```
import AnalogAssertionsPkg::*;

// sampling of analog signal is on finer time step than digital clk
// if period of clk is Td and time step is Ta, then Td == Kt * Ta holds for
some Kt, and
// the digital window size WSd and the analog window size WSa should
// satisfy VSa == Kw * Kt * WSd, for some Kw.
module DACchecker
    #(parameter real clkFrequency = 1000000.0, // digital clock frequency
      parameter real gain = 10.0, // specified conversion factor
      parameter real maxNoise = 0.5, // RMS noise on frequencies not in
dataIn
      parameter int  aWin_size = 2048, // analog window size
      parameter int  dWin_size = 256, // digital window size, Kw=1, Kt=8
      parameter real vOut_tolerance = 0.5 // absolute value
      )
    (
     ref uint   dataIn, // input digital unsigned int
     ref real   vOut, // analog output
     ref event  digitalClk, // digital side clock
     ref event  analogClk // analog sampling clock should be 8x faster
     );

    localparam real dFreqStep = clkFrequency/dWin_size;
    localparam real aFreqStep = dFreqStep;

    int        startCount = 0;
    bit        running = 1'b0;

    real       voutNoise;
```

```
    uint       aBin;

    AnalogAssertionsPkg::RealWindow dataIn_w = new[dWin_size];
    AnalogAssertionsPkg::RealWindow vOut_w = new[aWin_size];

    AnalogAssertionsPkg::Spectrum Din = new[dWin_size+1];
    AnalogAssertionsPkg::Spectrum Vout = new[aWin_size+1];

    clocking aClk @analogClk;
        default input #1step;
        input  vOut;
    endclocking // aClk

    clocking dClk @digitalClk;
        default input #1step;
        input  dataIn;
    endclocking // dClk

    function action;
      // do DFT on signals
        realBuildSpectrum(Din, dataIn_w);
        realBuildSpectrum(Vout, vOut_w);
      // remove input frequencies from Vout
        foreach (Din[i]) begin
          aBin = i * aWin_size / dWin_size; // need to associate i with time
step size
          nullSpectrumBin(Vout, aBin); //replace by 0 corresponding bin
      end
      voutNoise = 0; // cmpute RMS
      for(uint i = 0; i < aWin_size/2 -1; i++)
         voutNoise += (2 * abs(Vout[i])/aWin_size) **2;
      voutNoise = $sqrt(voutNoise/aWin_size);
      // Should the RMS be computed over the entire simulation or only
      // on a per window basis as done here?
    endfunction : action

    // check that DAC converts correctly digital to analog
    // assumed that dataIn sampled at register output
    // use of clocking vars does not introduce double sampling, used for
clarity
    DAC_conversion_a: assert property
    (@dClk cmpWTolerance(aClk.vOut, gain * dClk.dataIn, vOut_tolerance));

    always @(aClk) begin
       rSample(aClk.vOut, vOut_w, aWin_size);
    end

    always @(dClk) begin
       uiSample(dClk.dataIn, dataIn_w, dWin_size);
       if (!running) begin
        startCount++;
        running = startCount > dWin_size;
       end else begin
        action();
        // check that voutNoise is within limits
        DAC_noise_a: assert (voutNoise < maxNoise);
```

```
      end
    end

endmodule : DACchecker
```

4. SV Extensions

Would be nice extensions:

1. Built-in support for complex number data type, declaration and operations.
2. Allow $time or $realtime as a clock in assertions and always procedures, it would tick on every change of time.
3. Provide parameterized functions, it could avoid using dynamic arrays.
4. Allow arithmetic operations on dynamic arrays of same size.
5. Provide a built-in efficient FFT function.
6. Functions to return time step in appropriate time units.

If *checker* encapsulation is desired:

1. Allow dynamic array and real number operations in the construct.

If *class* encapsulation is desired:

1. Allow concurrent assertions in classes.
2. Reintroduce operator overloading.

References

[1] IEEE, "IEEE Standard for SystemVerilog - Unified Hardware Design, Specification, and Verification Language," IEEE, New York, 2017.

[2] Synonpsys, Inc., “amstaff Documentation”, March 2019.