Artgraphics

SystemVerilog による検証環境の構築

Document Identification Number: ARTG-TD-005-2020
Document Revision: 1.0, 2020.07.05
アートグラフィックス
篠塚一也

SystemVerilog による検証環境の構築

〒124-0012 東京都葛飾区立石 8-14-1
www.artgraphics.co.jp

Experimental Verification Environment using SystemVerilog

8-14-1, Tateishi, Katsushika-ku, Tokyo, 124-0012 Japan
www.artgraphics.co.jp

はじめに

SystemVerilogの最新仕様は、2018年2月21日に規格IEEE Std 1800-2017（以降、LRMと略称）として公開され、2年以上の歳月が経過しました。最近では、VerilogからSystemVerilogへと移行する傾向も顕著に見られ始めていますが、SystemVerilog言語自体の難易度は依然として変わっていません。LRMの解説書を解読する事ですら容易な作業ではない事に加えて、SystemVerilogが提供する機能の実践的な適用知識を身に着ける事は更に困難です。しかも、UVMがIEEE Std 1800.2-2017規格となり、検証分野では今後ますます採用される事になると思われます。この様に高度化した知識環境での作業を効率よく進めるためには、SystemVerilog言語に対する基礎知識と技術を充実させる事が求められます。

本書は、SystemVerilogにより検証環境を構築するために必要な知識と技術を総括した実践的な資料です。具体的には、EVE (Experimental Verification Environment)を例にとり、検証環境の構築技術を解説しています。EVEは、UVMのアーキテクチャをモデルにしているので、EVEの実装技術を理解する事はUVMに関する理解を深める事に繋がります。しかも、EVEは小規模な検証パッケージなので、全体を解読する事は容易です。本書の最後の章では、参考用にEVEのソースコードを紹介してあります。

UVMはSystemVerilogで記述された優れた検証パッケージですが、UVMの難易度が高い事も事実です。このため、多くの学習時間が初期コストとして発生し、UVMを十分に駆使する事ができる状況に至るまでの道のりが遠く見えるのが現状です。一方、膨大な行数からなるUVMの解読は殆ど不可能に近いだけでなく、基本的な機能に関しても多くのミステリーが存在します。特に、UVMのコア機能に関しての詳細な解説があれば、多くの技術者は不必要な努力を経験せずに、より短期間に実践的な知識を身に着ける事ができる様になります。

前述した様に、本書で解説するEVEはUVMをモデルにしているので、EVEの実装法を理解する事は、UVMのミステリーを解決する事に繋がります。例えば、EVEで検証コンポーネント作るには、以下の様な機能を使用します。

```
collector = collector_t::registry::create("collector",this);
```

これは、UVMのファクトリ機能と全く同じです。従って、EVEの実装法を解読すれば、UVMのファクトリの仕組みを理解する事ができる事が分かります。シミュレーションフェーズも同様に、EVEでは以下の様に記述します。

```
function void agent_t::build_phase(eve_run_param_t param);
   super.build_phase(param);
   if( m_active ) begin
      driver = driver_t::registry::create("driver",this);
      generator = generator_t::registry::create("generator",this);
   end
   collector = collector_t::registry::create("collector",this);
   monitor = monitor_t::registry::create("monitor",this);
endfunction
```

上記の記述には、UVMとの類似性がある事は一目瞭然です。EVEの実行制御法を解読すれば、UVMに関する理解が深まる事は確かです。トランザクションの定義に関してもUVMとの類似性があります。例えば、EVEではトランザクションを以下の様に定義する事ができます。

```
class simple_item_t extends eve_item_object_t;
rand logic [3:0] a, b;
rand logic       cin;
logic [3:0]      sum;
logic            co;
```

```
`eve_object_begin_m(simple_item_t)
   `eve_field_integral_m(a,EVE_DEFAULT)
   `eve_field_integral_m(b,EVE_DEFAULT)
   `eve_field_integral_m(cin,EVE_DEFAULT)
   `eve_field_integral_m(co,EVE_DEFAULT)
   `eve_field_integral_m(sum,EVE_DEFAULT)
`eve_object_end_m

function new(string name="simple_item");
   super.new(name);
endfunction

endclass
```

UVM と同じ様に、EVE にはフィールドマクロが実装されています。EVE の実装法を解読すれば、UVM に関する多くのミステリーを解決する事ができます。因みに、EVE で上記のトランザクションをプリントすると以下の様な表が生成されます。

```
-----------------------------------------------
Name              Type              Size    Value
-----------------------------------------------
simple_item       simple_item_t     -       @17
  a               integral          4       'h6
  b               integral          4       'h6
  cin             integral          1       'h0
  co              integral          1       'h0
  sum             integral          4       'hc
-----------------------------------------------
```

UVM と同じような形式の表が EVE でも生成されます。EVE の実装法を解読すれば、UVM のプリント機能の理解が深まるだけでなく、SystemVerilog に関する実践的な知識を身に着ける事ができます。

EVE の目的は、SystemVerilog に関する実践的知識を提供する事と UVM に関する使用面での知識を供給する事です。然し、EVE が UVM が備える検証機能を全て装備している訳ではありません。SystemVerilog 及び UVM に関する解説に必要な機能だけが EVE に実装されています。一方、EVE には、UVM に備えられていない便利な機能や改善が含まれている事も確かです。例えば、UVM ではフィールドマクロを以下の様に指定します。

```
`uvm_field_int(a,UVM_DEFAULT)
```

この`uvm_field_int マクロは、int 型だけでなく、bit、logic、byte、shortint、longint にも適用する事ができるので、適切な命名法とは言えません。EVE では、以下の様に指定します。

```
`eve_field_integral_m(a,EVE_DEFAULT)
```

この様な改善は、読者自身の設計において役立つと思います。

尚、本書の記述は UVM 1.2 をベースにしています。また、紙面の都合上、一部の記述は小さな書体で記述されています。

アートグラフィックス
篠塚一也

変更履歴

日付	Revision	変更点
2020.07.05	1.0	初版。

目次

1　EVE による検証環境構築

本章では EVE の機能を概説する目的で、EVE による検証環境の構築例を紹介します。本章で使用する EVE の機能は第 2 章以降での解説の主題となります。本章の構築例は、EVE が UVM を彷彿させるには十分な素材です。

1.1　検証環境

DUT を検証するために、図 1-1 の様な環境を準備します。DUT としては、簡単な組み合わせ回路を使用しますが、複雑なデザインの検証でも同様なアプローチが可能です。

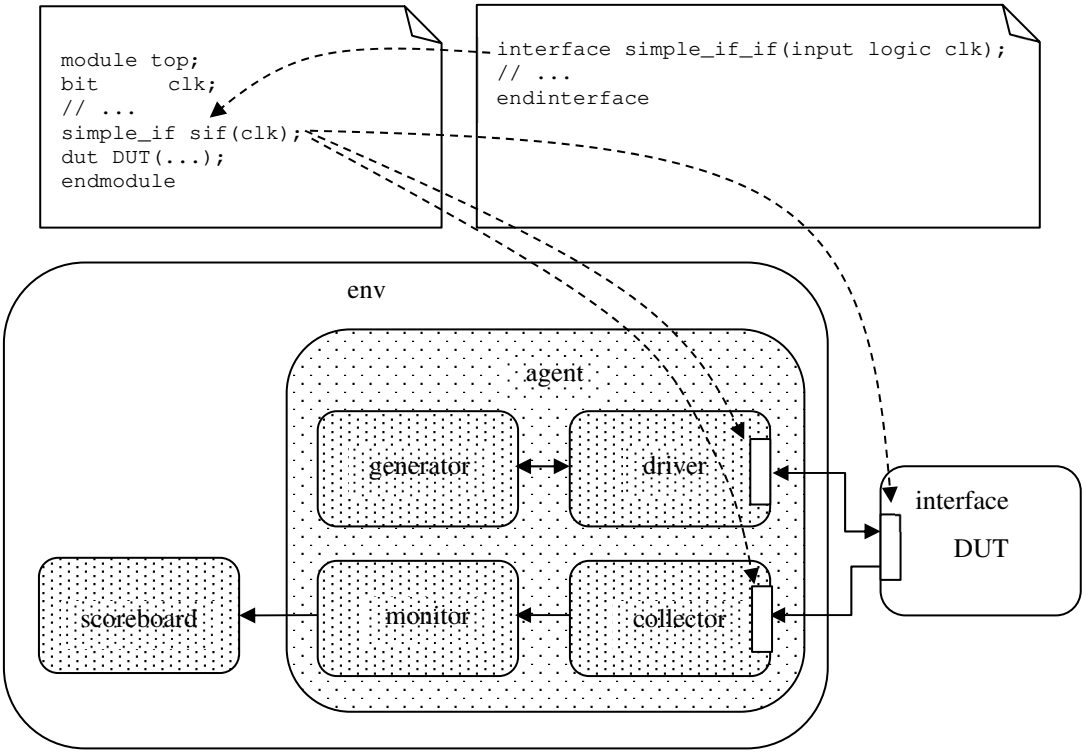

図 1-1　EVE による検証環境構築例

検証環境を構成する要素を表 1-1 にまとめます。

表 1-1　検証環境を構成する要素

クラス及び DUT	説明

dut	4 ビットの加算器で、以下の様な仕様です。但し、dut ではポートの定義にインターフェースを使用していません。
	`module dut(input logic [3:0] a,b,`
	`logic cin,output logic co,logic [3:0] sum);`

simple_if	DUT の信号を操作するために使用するインターフェースです。
simple_item_t	トランザクションのクラスです。
driver_t	ドライバーのクラスです。
generator_t	トランザクションを生成するためのジェネレータクラスです。
collector_t	DUT のレスポンスを収集して、トランザクションに変換するコレクターのクラスです。変換したトランザクションは、モニターに送信されます。
monitor_t	モニターのクラスです。モニターは簡単な解析と他の検証コンポーネントにトランザクションを送信する役目を持ちます。この例では、特別な解析処理をせずにトランザクションをスコアボード

	に送信します。
agent_t	ドライバー、ジェネレータ、コレクター、モニターから構成されるエージェントのクラスです。
scoreboard_t	モニターから送信されたトランザクションを解析する役目を持つスコアボードを定義するクラスです。
env_t	エージェント、スコアボードから構成される階層的な検証コンポーネントを定義するためのクラスです。
pkg	検証環境で使用するパッケージです。
top	トップモジュールです。

それぞれの検証要素を以下に解説します。

1.2 dut

DUT は、以下に示す様な簡単な組み合わせ回路です。

```
module dut(input logic [3:0] a,b,logic cin,
  output logic co,logic [3:0] sum);
assign {co,sum} = a+b+cin;
endmodule
```

1.3 simple_if

インターフェースの定義は以下の様に、DUT を操作するために必要な信号値のみを定義しています。

```
interface simple_if(input logic clk);
logic [3:0]   a, b, sum;
logic         co, cin;
endinterface
```

1.4 simple_item_t

トランザクションの定義は以下の様に eve_item_object_t クラスを使用して定義します。トランザクションはDUTをドライブする時に使用されるので、DUTへの入力に対応する変数をランダム変数として宣言する必要があります。トランザクションのフィールドには、UVM と同様にフィールドマクロを指定しなければなりません。

```
class simple_item_t extends eve_item_object_t;
rand logic [3:0] a, b;
rand logic       cin;
logic [3:0]      sum;
logic            co;

`eve_object_begin_m(simple_item_t)
   `eve_field_integral_m(a,EVE_DEFAULT)
   `eve_field_integral_m(b,EVE_DEFAULT)
   `eve_field_integral_m(cin,EVE_DEFAULT)
   `eve_field_integral_m(co,EVE_DEFAULT)
   `eve_field_integral_m(sum,EVE_DEFAULT)
`eve_object_end_m

function new(string name="simple_item");
   super.new(name);
endfunction

endclass
```

1.5 driver_t

ドライバーを eve_driver_t クラスから構築します。ドライバーの全容は以下の様になります。

必ず、virtual インターフェースの宣言が必要です。

```
class driver_t extends eve_driver_t#(simple_item_t);
virtual simple_if   vif;

`eve_component_m(driver_t)

function new(string name,eve_component_t parent);
   super.new(name,parent);
endfunction

extern function void connect_phase(eve_run_param_t param);
extern task run_phase(eve_run_param_t param);
extern task drive_dut(ITEM item);
endclass
```

connect_phase()は、以下の様に virtual インターフェースの設定を完了します。

```
function void driver_t::connect_phase(eve_run_param_t param);
   super.connect_phase(param);
   if( !eve_config_t#(virtual simple_if)::get(this,"","vif",vif) )
      eve_error($sformatf("Could not find assignment for '%s.vif'",
         get_full_name()));
endfunction
```

run_phase()は、以下の様になります。下記の、get()は、ドライバーに接続されているジェネレータの get()を呼び出します。ジェネレータからトランザクションを取得すると、タイミングを調節して、DUT をドライブする様にします。

```
task driver_t::run_phase(eve_run_param_t param);
   forever begin
      get(m_get_item);
      @(posedge vif.clk);
      drive_dut(m_get_item);
   end
endtask
```

この get()は、ジェネレータの get()を呼び出します。

drive_dut()は、virtual インターフェースを介して、DUT の入力信号を更新します。信号の更新に伴い、DUT が動作を開始します。

```
task driver_t::drive_dut(ITEM item);
   vif.a <= item.a;
   vif.b <= item.b;
   vif.cin <= item.cin;
endtask
```

1.6 generator_t

ジェネレータは以下の様に簡単な構成となります。通常、これ以外の記述は必要ありません。

```
class generator_t extends eve_generator_t#(simple_item_t);
ITEM   item;

`eve_component_m(generator_t)

function new(string name,eve_component_t parent);
   super.new(name,parent);
endfunction
```

```
extern task get(output ITEM item);
endclass
```

get()は、ドライバーから呼ばれて動作を開始します。ここでは、トランザクションに乱数を発生するだけの簡単な処理になっています。

```
task generator_t::get(output ITEM item);
    if( item == null )
        item = new;
    assert( item.randomize() );
endtask
```

1.7 collector_t

コレクターは以下の様になります。ドライバーと同様に、virtual インターフェースを宣言しなければなりません。

```
class collector_t extends eve_collector_t#(simple_item_t);
virtual simple_if   vif;
ITEM                item;

`eve_component_m(collector_t)

function new(string name,eve_component_t parent);
    super.new(name,parent);
endfunction

extern function void connect_phase(eve_run_param_t param);;
extern task run_phase(eve_run_param_t param);
extern function void collect_broadcast();
endclass
```

connect_phase()では、以下の様にして virtual インターフェースの設定を完了しなければなりません。

```
function void collector_t::connect_phase(eve_run_param_t param);
    super.connect_phase(param);
    if( !eve_config_t#(virtual simple_if)::get(this,"","vif",vif) )
        eve_error($sformatf("Could not find assignment for '%s.vif'",
            get_full_name()));
endfunction
```

run_phase()では、DUT からのレスポンスを収集して、トランザクションに変換をします。その後、トランザクションをモニターに送信します。そして、この処理を繰り返します。DUT は、入力信号が変化した時のみ動作するので、その特性を活かした検証法を記述しなければなりません。

```
task collector_t::run_phase(eve_run_param_t param);
    item = new;
    forever begin
        @(vif.a,vif.b,vif.cin);
        #0;
        collect_broadcast();
    end
endtask
```

DUT への入力が変化したら、出力が安定するのを待ちます。

collect_broadcast() がトランザクションへの変換とモニターへの送信を行います。broadcast(item)は、コレクターに接続されている全ての検証コンポーネントにトランザクショ

ンを送信しますが、この場合には、モニターだけが接続されています。

```
function void collector_t::collect_broadcast();
   item.a = vif.a;
   item.b = vif.b;
   item.cin = vif.cin;
   item.co = vif.co;
   item.sum = vif.sum;
   broadcast(item);
endfunction
```

1.8 monitor_t

モニターは、標準的な記述になります。但し、コレクターからトランザクションを受信するために write()メソッドを実装しなければなりません。一般的には、モニターは簡単な解析処理をしますが、ここでは何もせずに、トランザクションを他の検証コンポーネント、例えば、スコアボードに送信します。

```
class monitor_t extends eve_monitor_t#(simple_item_t);

`eve_component_m(monitor_t)

function new(string name,eve_component_t parent);
   super.new(name,parent);
endfunction

function void write();
   // pass the transaction to other analysis tools.
   broadcast(m_receive_port.m_item);
endfunction

endclass
```

1.9 agent_t

エージェントの定義は、以下に示す様に極めて標準的な記述で済みます。

```
class agent_t extends eve_agent_t;
driver_t      driver;
generator_t   generator;
collector_t   collector;
monitor_t     monitor;

`eve_component_begin_m(agent_t)
   `eve_field_integral_m(m_active,EVE_DEFAULT)
   `eve_field_integral_m(m_coverage_enabled,EVE_DEFAULT)
`eve_component_end_m

function new(string name,eve_component_t parent);
   super.new(name,parent);
endfunction

extern function void build_phase(eve_run_param_t param);
extern function void connect_phase(eve_run_param_t param);
endclass
```

build_phase()でドライバー、ジェネレータ、コレクター、モニターのインスタンスを作ります。ベースクラスの eve_agent_t には、コンフィギュレーションパラメータ m_active が定義されているので、そのパラメータに従いサブコンポーネントを定義します。

```
function void agent_t::build_phase(eve_run_param_t param);
   super.build_phase(param);
   if( m_active ) begin
      driver = driver_t::registry::create("driver",this);
      generator = generator_t::registry::create("generator",this);
   end
   collector = collector_t::registry::create("collector",this);
   monitor = monitor_t::registry::create("monitor",this);
endfunction
```

connect_phase()では、build_phase()で作成したサブコンポーネントの接続を完了します。

```
function void agent_t::connect_phase(eve_run_param_t param);
   if( m_active )
      driver.m_port.connect(generator.m_port);
   collector.m_send_port.connect(monitor.m_receive_port);
endfunction
```

1.10 scoreboard_t

スコアボードは、DUT のレスポンスを検証する役目を持ちますが、ここでは、単にトランザクションをプリントするだけの処理となっています。

```
class scoreboard_t extends eve_scoreboard_t#(simple_item_t);
bit       header_done;

`eve_component_begin_m(scoreboard_t)
   `eve_field_integral_m(header_done,EVE_DEFAULT)
`eve_component_end_m

function new(string name,eve_component_t parent);
   super.new(name,parent);
endfunction

extern function void write();
extern function void print_header();
endclass
```

スコアボードはモニターに接続されているので、下記の write()はモニターから直接呼び出されます。write()は、トランザクションをプリントして処理を終了します。

```
function void scoreboard_t::write();
ITEM   item;
   if( !header_done ) begin
      print_header();
      header_done = 1;
   end
   item = m_receive_port.m_item;
   eve_system($sformatf("@%3t: %2d %2d  %1d    %2d",
      $time,item.a,item.b,item.cin,{item.co,item.sum}));
endfunction
```

print_header()は、プリント用のヘッダーをプリントします。

```
function void scoreboard_t::print_header();
   eve_system($sformatf("%5s  a  b  ci {co,sum}"," "));
endfunction
```

1.11 env_t

この検証環境例では、エンバイロンメントはスコアボードと唯一つのエージェントから構成されます。

```
class env_t extends eve_env_t;
agent_t       agent;
scoreboard_t  scoreboard;

`eve_component_m(env_t)

function new(string name,eve_component_t parent);
   super.new(name,parent);
endfunction

extern function void build_phase(eve_run_param_t param);
extern function void connect_phase(eve_run_param_t param);
endclass
```

build_phase()では、エージェントとスコアボードのインスタンスを作り配置します。

```
function void env_t::build_phase(eve_run_param_t param);
   super.build_phase(param);
   agent = agent_t::registry::create("agent",this);
   scoreboard = scoreboard_t::registry::create("scoreboard",this);
endfunction
```

connect_phase()では、モニターとスコアボードの接続を完了します。

```
function void env_t::connect_phase(eve_run_param_t param);
   super.connect_phase(param);
   agent.monitor.m_send_port.connect(scoreboard.m_receive_port);
endfunction
```

1.12 pkg

検証環境で使用するパッケージの定義は、以下の様になります。

```
`ifndef   PKG_H
`define   PKG_H

package pkg;
import eve_pkg::*;

`include "simple_item.sv"
`include "driver.sv"
`include "generator.sv"
`include "collector.sv"
`include "monitor.sv"
`include "agent.sv"
`include "scoreboard.sv"
`include "env.sv"

endpackage : pkg

`endif
```

1.13 top

トップモジュールでは以下の処理を行います。

- インターフェースと DUT のインスタンスを作成し、接続する。
- インターフェースのインスタンスと virtual インターフェースの対応表を準備する。
- クロックを生成する。
- run_test()を呼び出して、EVE を開始する。

トップモジュールは以下の様になります。

```
module top;
import eve_pkg::*;
import pkg::*;
bit    clk;
env_t  env;

simple_if SIF(clk);
full_adder DUT(.a(SIF.a),.b(SIF.b),.cin(SIF.cin),
                 .co(SIF.co),.sum(SIF.sum));

   initial begin
      set_timeout(10,100);
      eve_config_t#(virtual simple_if)::set(null,
         "env0*","vif",SIF);
      env = env_t::registry::create("env0",null);
      run_test();
   end

   initial forever #10 clk = ~clk;

endmodule
```

以下の呼び出しは、タイムアウトを設定しています。この指定によると、$time が 100 を超えると EVE によるシミュレーションが終了します。タイムアウトを確認する時間間隔は、#10 となっています。

```
set_timeout(10,100);
```

1.14 実行結果

実行すると以下の様な結果を得ます。前述した様に、スコアボードが結果をプリントしています。また、$time==100 を超えると、シミュレーションは終了している事が分かります。

```
        a  b  ci {co,sum}
@ 10:   6  7  1    14
@ 30:   1  3  1     5
@ 50:   6  7  1    14
@ 70:  15 12  0    27
@ 90:   6  6  0    12
```

1.15 プリント機能

オブジェクトやコンポーネントのプリント機能も UVM に類似した形式で実装されています。以下は、トランザクションをプリントした例です。

```
-----------------------------------------------
Name              Type              Size    Value
-----------------------------------------------
simple_item       simple_item_t     -       @17
  a               integral          4       'h6
  b               integral          4       'h7
  cin             integral          1       'h1
  co              integral          1       'h0
  sum             integral          4       'he
```

```
----------------------------------------------
```

次に、コンポーネントのプリントとして、env_t クラスの例を以下に紹介します。

```
----------------------------------------------------------------------------------------
Name                        Type                                          Size    Value
----------------------------------------------------------------------------------------
env0                        env_t                                         -       @2
  agent                     agent_t                                       -       @5
    m_active                integral                                      1       'h1
    m_coverage_enabled      integral                                      1       'h1
    collector               collector_t                                   -       @12
      send_port             eve_send_port_t#(collector_t,simple_item_t)   -       @13
    driver                  driver_t                                      -       @8
      io_port               eve_io_port#(driver_t)                        -       @9
    generator               generator_t                                   -       @10
      io_port               eve_io_port#(driver_t)                        -       @11
    monitor                 monitor_t                                     -       @14
      receive_port          eve_receive_port_t#(simple_item_t)            -       @15
      send_port             eve_send_port_t#(monitor_t,simple_item_t)     -       @16
  scoreboard                scoreboard_t                                  -       @6
    header_done             integral                                      1       'h0
----------------------------------------------------------------------------------------
```

1.16 まとめ

以上の例から、EVE には UVM との類似性が多く存在し、EVE を学習する事は以下の様な理解に繋がる事が分かります。

- UVM 機能の実装技術を理解する事ができる。
- SystemVerilog クラスの使い方に関する実践的知識を身につける事ができる。
- SystemVerilog マクロの使い方を完全に習得する事ができる。

本書の最後の章ではソースコードを紹介するので、各クラスの解説は概要に留めます。実装技術に関しては、ソースコードを参考にして下さい。

EVE は、小規模な SystemVerilog パッケージなのでコンパイルも瞬時に終了します。SystemVerilog、及び UVM の学習素材として、EVE は最適です。EVE を活用すれば、UVM の学習時間を短縮する事ができます。