ET & IoT Technology 2018 電子設計・EDA トラック講演資料

Artgraphics

UVMの概要と実践に適用する際のガイドライン

篠塚一也

アートグラフィックス

2018.11.16

www.artgraphics.co.jp

本講演の概要

- 検証作業における生産性向上と検証コードの再利用性を可能にするUVMが次第に普及し始めています。UVMはSystemVerilog上に構築された大規模、且つ、複雑な検証ライブラリーである為、UVMを理解する事は一朝一夕には行きません。
- UVMの正式な文書として、ユーザ・ガイド、及び、UVMクラス・リファレンス・マニュアルがありますが、何れもUVMに関する基礎知識が無いと理解するのは難しいと思えます。
- 本講演では、UVMの中枢を簡潔に解説し、それらの文書を理解する事が出来る様に道筋を描きます。
- 講演の前半はUVMの概要を解説し、後半はUVMを実践に適用する際のガイドラインを解説します。

Part One
UVM概要

UVMとは何かを説明し、UVMを使用する為に必要な手順を解説します。

UVMとは何か？

- UVM (Universal Verification Methodology)とは、検証分野で推奨されている技術、ルール、慣習、規律等をコードとして具体化し、検証技術の再利用性と生産性向上をさせる為のSystemVerilogのクラス・ライブラリーです。
- UVMはAccellera Systems Initiativeにより開発されました。
- UVMはSystemVerilogをベースにして記述されているので、SystemVerilogをサポートしている検証ツールの環境で使用する事が出来ます。
- UVMは、検証分野で推奨されている技術、ルール、慣習、規律等をベースにしている方法論である為、その開発の背景を最初に理解しなければなりません。

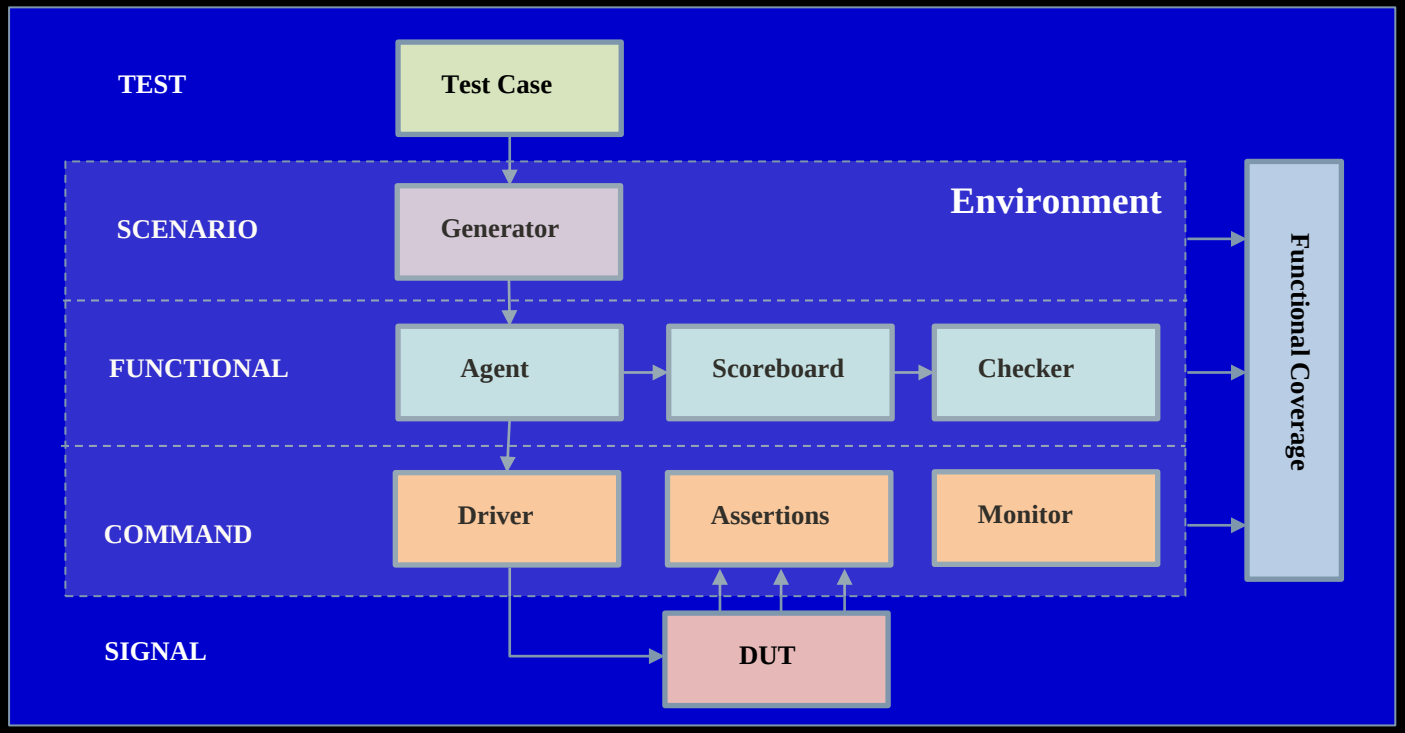
検証技術のトレンド（VMM）

- 近年の検証技術では階層的にテストベンチを記述する手法（layered testbench）が採用されています（[3]）。

Layer	検証目的及び機能
test layer	テストベンチのトップ・レベルのレイヤーです。テストを生成して下位の層に送ります。
scenario layer	シナリオに沿ってトランザクションのシーケンスを生成します。
functional layer	高位のレベルのコマンドを処理するレイヤーです。例えば、DMA read/write等のコマンドを受け取ると、個々のbus read/write等のコマンドに分解してcommand layerに送ります。
command layer	トランザクション・レベルのコマンドをシグナル・レベルの値に変換をしてsignal layerに送ります。DriverはDUTをドライブします。
signal layer（DUT）	DUTからのレスポンスは上位の層に送られます。アサーション等の結果も上位の層に戻されます。

テストベンチとレイヤーの関係（[3]）

- このlayered testbenchを良く考察するとUVM のメソドロジー・クラスの名称に対応している事が分かります。言い換えると、UVMは検証技術のトレンドに沿って開発されている事を確認する事が出来ます。

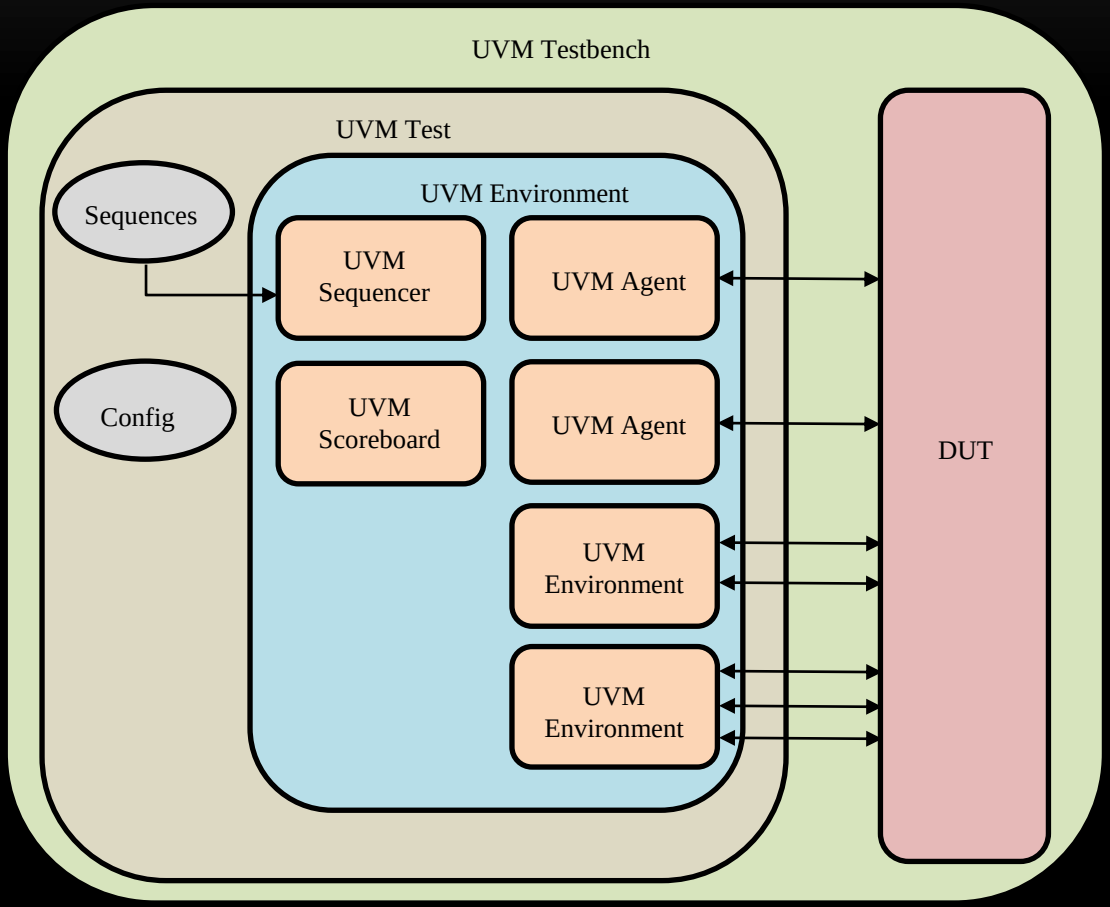


代表的なUVMクラス

- UVMには多くのクラスが定義されていますが、ユーザが直接使用するのはその内の一部のクラスで、メソドロジー・クラス（methodology class）と呼ばれます。

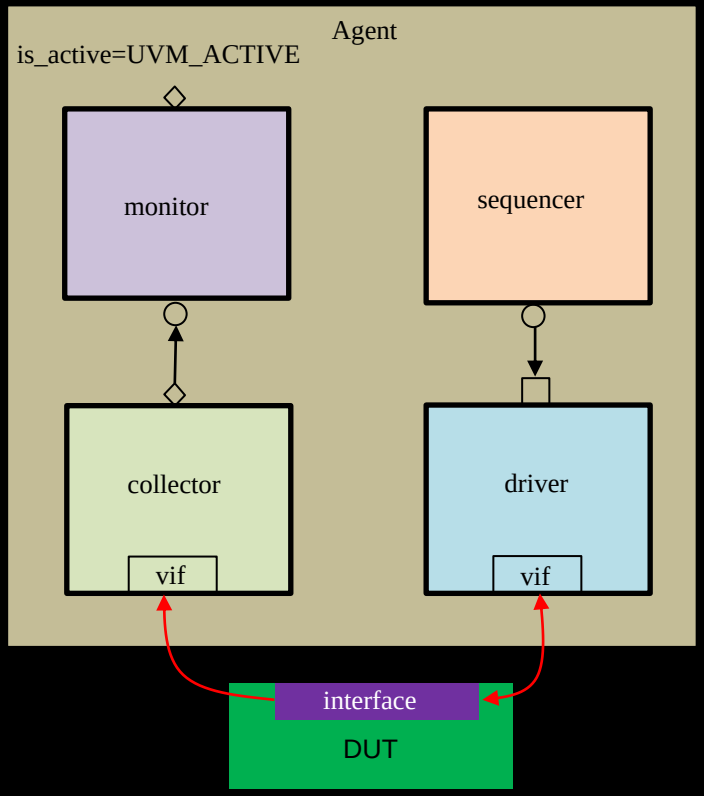
UVMクラス	種別
uvm_sequence_item	トランザクション関連
uvm_sequence	トランザクション関連
uvm_driver	メソドロジー・クラス
uvm_sequencer	メソドロジー・クラス
uvm_env	メソドロジー・クラス
uvm_agent	メソドロジー・クラス
uvm_test	メソドロジー・クラス
uvm_monitor	メソドロジー・クラス
uvm_scoreboard	メソドロジー・クラス

UVMテストベンチの構造（[2]）



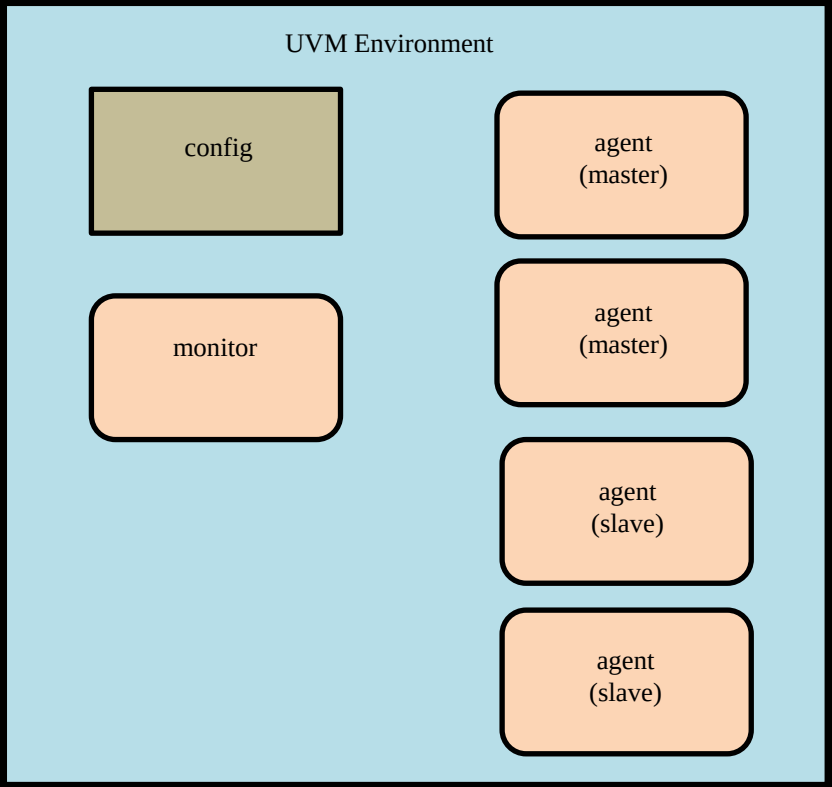
エージェント（uvm_agentのサブクラス）

- シーケンサー、ドライバー、及び、モニターはエージェントとして纏められます。



エンバイロンメント（uvm_envのサブクラス）

- エンバイロンメントはシーケンサー、スコアボード、エージェント、及び、下位のエンバイロンメント（uvm_envのサブクラス）から構成されます。



TLM (Transaction Level Modeling)

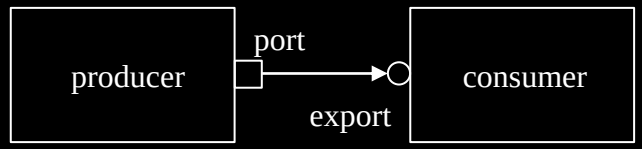
- UVMはTLMを採用し、シグナル・レベルよりも高位の記述法を用いて検証タスクを表現します。このアプローチはシステムの動作を考察する際の自然な方法です。
- トランザクションは二つのコンポーネント間の通信をモデルする為に必要な情報を意味します。トランザクションには、その情報を操作する為のメソッドが定義されます。
- UVMではトランザクションはオブジェクトであり、UVMコンポーネントがトランザクションを操作します。その内の特別なコンポーネントとしてドライバー（uvm_driverのサブクラス）が存在します。ドライバーはトランザクションをシグナル・レベルに変換してDUTを操作する役目を持ちます。
- DUT側のシグナルの変化を検知する役目を持つUVMコンポーネントも必要になります。そのコンポーネントは、一般的には、コレクター（collector）と呼ばれます。
- ドライバーとコレクターの存在により、UVMではトランザクション・レベルでシステムを記述する事が出来る様になります。尚、UVMコンポーネントとDUT間のデータ授受にはSystemVerilogのvirtual interfaceが使用されます。

コンポーネント間の通信

- 通信にはTLM-portとTLM-exportが使用されます。TLM-portはトランザクションを操作する為の方法を定義します。一方、TLM-exportではトランザクションを処理する為に必要な実処理を記述します。
- ここではブロッキング・メソッドを使用して説明をします。ノンブロッキングの方法も殆ど同様にして行えます。通信は、producer、及び、consumerの関係になります
- producer、及び、consumerは一対一の関係にあり、producerに対してconsumerが必ず存在しなければなりません。然し、producerを定義する時点ではconsumerに関する知識は一切必要ありません。
- 言い換えると、producerが使用するプロトコルに従うコンポーネントがあればproducerと通信を行なえる事になります。即ち、producerコンポーネントの再利用が可能となります。同様の事がconsumer側にも言えます。

ブロッキングput

- producerコンポーネントはTLM-portを使用してトランザクションを引き渡す操作を行います。一方、consumerコンポーネントはTLM-exportを使用してトランザクションを受け取り、トランザクションを処理する手順を記述します。この関係をUVMでは以下の様に記します。



- producer の定義例は次ページを参照して下さい。
- producerは相手方のコンポーネントに関する知識を持たずに記述している事に注意して下さい。producerのインスタンスを作る親コンポーネントがproducerの相手方のコンポーネントを指定します。
- 更に、producerはトランザクション・オブジェクトを作る必要があります。また、put()はブロッキングである為、producerはput()が終了する迄待ち状態に入る様になっても動作する様にしなければなりません。
- 尚、この例ではトランザクションを一つだけ生成していますが、一般的な実処理では、トランザクションを繰り返し生成する必要があります。

ブロッキングputの例

- 例えば、producerの記述は以下の様になります。

```
 1  // producer
 2  class producer extends uvm_component;
 3  uvm_blocking_put_port #(simple_item)    put_port;
 4  `uvm_component_utils(producer)
 5
 6  function new(string name,uvm_component parent);
 7          super.new(name,parent);
 8          put_port = new("put_port",this);
 9  endfunction
10
11  virtual task run_phase(uvm_phase phase);
12  simple_item     tr;
13  // prepare transaction.
14  make_transaction(tr);
15  // send transaction to consumer.
16  put_port.put(tr);
17  endtask
18
19  task make_transaction(output simple_item item);
20  // ...
21  endtask
22  endclass
```

Virtual Interface（SystemVerilogの機能）

- DUTとのデータ授受はvirtual interfaceを介して行われます。virtual interfaceはinterfaceのインスタンスへのポインターです。
- interfaceのインスタンスはテストベンチ（トップ・モジュール）で作られます。そのインスタンスがDUTと関連するコンポーネントのvirtual interfaceに引き渡さなれればなりません。その割り当てをハード・コーディングすると検証技術の再利用性に反する為、実行時にvirtual interfaceの情報を設定します。その際、前述したコンポーネント・インスタンスの階層構造が使用されます。

uvm_object

- UVMはSystemVerilogクラスとUVMマクロから構成されます。UVMには多くのクラスが定義されていますが、特別なクラスとしてuvm_objectが存在します。このクラスはアブストラクト・クラスで他の全てのUVMクラスのベース・クラスになっています。
- uvm_objectクラスでは他の全てのクラスに共通する属性、及び、手順を宣言しています。具体的な手順の内容はサブクラスで行います。手順としては、例えば、print及びcopyがあります。
- uvm_objectクラスから二つの重要なサブクラスが定義されています。それらは、uvm_sequence_itemとuvm_componentです。前者は、トランザクションを定義する為に使用します。一方、後者はテストベンチを構築する為に使用します。全てのメソドロジー・クラスはuvm_componentのサブクラスです。

UVMコンポーネント（uvm_componentのサブクラス）

- UVMコンポーネントはシミュレーションの対象となるので、デザインにおけるmoduleの様な役目を果たします。即ち、UVMにおいて、コンポーネントは自然にコンポーネント・インスタンスの階層構造を構成します。
- 階層のトップには、uvm_topと呼ばれるインスタンスが存在します。UVMはその階層構造を使用してダイナミックなシミュレーションを制御します。例えば、階層構造を使用して、テストベンチの構成をシミュレーション開始直前に変える事が出来ます。この機能により、一度のコンパイルで複数のテスト・ケースを実行する事が出来ます。
- 階層構造は、シミュレーション開始直前に決定し、一度シミュレーションが開始（コンポーネントのタスクrun_phase() の実行を開始した時点）するとコンポーネント・インスタンスの階層構造は変化しません。

UVMを使用する為に必要な手順

```
1  `include "uvm_pkg.sv"
2  `include "uvm_macros.svh"
3
4  module top;
5  import uvm_pkg::*;
6
7  class my_comp extends uvm_component;
8          int     data[];
9          int     data_size = 2;
10         string  header_name = "data[0..1]";
11
12         // ...
13 endclass
14
15 my_comp         test;
16
17         initial begin
18                 // fix the configuration.
19                 uvm_config_db#(string)::set(null,"test","header_name","data[0..7]");
20                 uvm_config_db#(int)::set(null,"test","data_size",8);
21                 // create components.
22                 test = my_comp::type_id::create("test",null);
23                 run_test();
24         end
25
26 endmodule
```

① UVMクラス・ライブラリーを`includeする。

② uvm_pkgをimportする。

③ テストすべき内容を定義する。

④ run_test()を呼び出しテストを実行する。

重要な概念：run_test()

- ユーザが準備したトップ・モジュールからrun_test()を呼び出すと、UVMが実行権を得て、シミュレーション終了時まで実行制御権を持ち続けます。
- メソドロジー・クラスを使用してユーザが記述したUVMコンポーネントはUVMにより呼び出し制御を受けます。ユーザ側に実行制御権はありません。
- ユーザのUVMコンポーネントに制御が渡るタイミングは予め決定されていて、シミュレーション・フェーズ（simulation phases）と呼ばれています。

シミュレーション・フェーズ（virtual function又はtask）

フェーズ	機能
build_phase	コンポーネントの階層を構築するフェーズです。従って、階層のトップから順に呼ばれて行きます。通常、チャイルド・コンポーネントをこのフェーズで作成します。
connect_phase	コンポーネント間の接続を完成するフェーズです。例えば、TLMポートの接続を定義します。
end_of_elaboration_phase	全ての接続が終了するとこのフェーズに制御が移ります。通常は、コンフィギュレーションをプリントする等の処理を記述します。
start_of_simulation_phase	シミュレーションが開始する直前にこのフェーズが呼ばれます。初期化処理等を行う事が出来ます。
run_phase	シミュレーションを行う為のフェーズです。
extract_phase	シミュレーションが終了すると、このフェーズに制御が移ります。シミュレーション結果を抽出する為の処理を記述する事が出来ます。
check_phase	抽出したシミュレーション結果をチェックする為の処理を記述します。
report_phase	シミュレーション結果のレポートを出力する処理を記述します。

シミュレーション・フェーズの記述例

```
 1  //
 2  //      my_testbench
 3  //
 4  class my_testbench extends uvm_test;
 5  `uvm_component_utils(my_testbench)
 6  //      new
 7  function new(string name="my_testbench",uvm_component parent=null);
 8          super.new(name,parent);
 9  endfunction
10  //      build_phase
11  function void build_phase(uvm_phase phase);
12          super.build_phase(phase);
13          `uvm_info("MY_TESTBENCH","build_phase running.",UVM_LOW)
14  endfunction
15  //      run_phase
16  task run_phase(uvm_phase phase);
17          `uvm_info("MY_TESTBENCH","run_phase running.",UVM_LOW)
18  endtask
19  endclass
20
```

データ・オブジェクトとフィールド・マクロ

- データ・オブジェクトを定義する場合には、プロパーティ名に対して`uvm_object_utilsマクロを使用しなければなりません。

```
21  class simple_item extends uvm_sequence_item;
22  rand int unsigned addr;
23  rand int unsigned data;
24  rand int unsigned delay;
25  rand simple_item_delay_e delay_kind;
26  `uvm_object_utils_begin(simple_item)
27          `uvm_field_int(addr,UVM_DEFAULT)
28          `uvm_field_int(data,UVM_DEFAULT)
29          `uvm_field_int(delay,UVM_DEFAULT)
30          `uvm_field_enum(simple_item_delay_e,delay_kind,
31                  UVM_DEFAULT|UVM_NOCOMPARE|UVM_NOPACK)
32  `uvm_object_utils_end
33
34  //
35  //      new
36  //
37  function new(string name="simple_item");
38          super.new(name);
39  endfunction
40  endclass
```

データ・オブジェクトとフィールド・マクロ（フィールド・マクロを持たない場合）

- フィールド・マクロを持たない場合にも、UVMマクロは必要です。

```
 1  class mem_response_seq extends uvm_sequence #(apb_transfer);
 2
 3    function new(string name="mem_response_seq");
 4      super.new(name);
 5    endfunction
 6
 7    rand logic [7:0] mem_data;
 8
 9    `uvm_object_utils(mem_response_seq)
10    `uvm_declare_p_sequencer(apb_slave_sequencer)
11
12    //simple assoc array to hold values
13    logic [7:0] slave_mem[int];
14
15    apb_transfer req;
16    apb_transfer util_transfer;
17
18    virtual task body();
19          // ...
20    endtask : body
21  // ...
22  endclass
```

コンポーネントとフィールド・マクロ

- コンポーネントに関しては、`uvm_component_utilsマクロを使用しなければなりません。

```
 8  class my_comp extends uvm_component;
 9  int      data[];
10  int      data_size = 2;
11  string   header_name = "data[0..1]";
12
13  `uvm_component_utils_begin(my_comp)
14          `uvm_field_string(header_name,UVM_DEFAULT)
15          `uvm_field_int(data_size,UVM_DEFAULT)
16          `uvm_field_array_int(data,UVM_DEFAULT)
17  `uvm_component_utils_end
18
19  function new(string name,uvm_component parent);
20          super.new(name,parent);
21  endfunction
22
23  // ...
24
25  endclass
```

コンポーネントとフィールド・マクロ（フィールド・マクロを持たない場合）

- フィールド・マクロを持たない場合にも、UVMマクロは必要です。

```
 1  class slave_comp extends uvm_component;
 2  `uvm_component_utils(slave_comp)
 3
 4  function new (string name,uvm_component parent);
 5  super.new(name,parent);
 6  endfunction
 7
 8  task run_phase(uvm_phase pahse);
 9  `uvm_info("SLAVE","run_phase: Executing.",UVM_LOW)
10  endtask
11
12  endclass
```

raise_objection()とdrop_objection()（UVMで最も難解な概念の一つ）

- UVMコンポーネントを正しく記述してもシミュレーションは実行しません。厳密に言えば、シミュレーションは時刻0で終了してしまいます。
- シミュレーションをする為には、UVMに実行すべき内容がある事を知らせなければなりません。そして、実行が終了したらその旨通知しなければなりません。
- raise_objection() はUVMに実行すべき内容がある事を知らせます。一方、drop_objection() は実行すべき内容が完了した事をUVMに知らせます。全ての実行すべき内容が完了した時点でUVMはシミュレーションphaseを終了して、次のphase（通常は、extract_phase）に進みます。
- テストベンチを構成する少なくとも一つのコンポーネントが、run_phase()の初めにraise_objection()を呼び、処理終了後にdrop_objection()を呼ばなければなりません。

raise_objection()とdrop_objection()の使用例

- 実行開始前にraise_objection()を呼び、実行を終了した時点で、drop_objection()を呼び出します。

```
 1  class env extends uvm_env;
 2
 3    local pin_vif pif;
 4    driver d;
 5
 6    function new(string name, uvm_component parent = null);
 7      super.new(name, parent);
 8      d = new("driver", this);
 9    endfunction
10
11    task run_phase(uvm_phase phase);
12      phase.raise_objection(this);
13      do_something();
14      phase.drop_objection(this);
15    endtask
16
17  // ...
18
19  endclass
```

Part Two
実践に適用する際のガイドライン

UVMを検証作業に適用する際、遵守すべき一般的なルールがあります。それらの代表的な例を紹介します。ルールに従う事で、予期しないトラブルを未然に防ぐ事が出来ます。

パッケージ

- 開発した検証コンポーネントは、パッケージに包含する事が望ましいと言われています。下記の様に、パッケージを定義したファイルを作成します。

```
1  package my_pkg;
2          import uvm_pkg::*;
3          `include "uvm_macros.svh"
4          `include "my_packet.sv"
5          // Include other files as needed.
6  endpackage
```

- テストベンチでは、準備したパッケージを `includeし、importします。

```
1  `include "uvm_pkg.sv"
2  `include "my_pkg.sv"
3
4  module simple_example;
5  import uvm_pkg::*;
6  import my_pkg::*;
7  `include "uvm_macros.svh"
8  my_packet       packet;
9
10         initial begin
11                 packet = my_packet::type_id::create("packet");
12                 // ...
13         end
14 endmodule
```

`includeする。

importする事が必要。

コンストラクタ（データ・オブジェクト）

- UVMでは、コンストラクタに対して一般的なルールがあります。データ・オブジェクトでは、名称に対して標準値を設定します。例えば、次の様に名称の標準値を定義します。更に、super.new()を呼び出す事が必要です。

```
21  class simple_item extends uvm_sequence_item;
22  rand int unsigned addr;
23  rand int unsigned data;
24  rand int unsigned delay;
25  rand simple_item_delay_e delay_kind;
26  `uvm_object_utils_begin(simple_item)
27          `uvm_field_int(addr,UVM_DEFAULT)
28          `uvm_field_int(data,UVM_DEFAULT)
29          `uvm_field_int(delay,UVM_DEFAULT)
30          `uvm_field_enum(simple_item_delay_e,delay_kind,
31                  UVM_DEFAULT|UVM_NOCOMPARE|UVM_NOPACK)
32  `uvm_object_utils_end
33
34  //                          標準値を定義して置く。
35  //      new                 ↓
36  //
37  function new(string name="simple_item");
38          super.new(name);    ← super.new()を呼ぶ。
39  endfunction
40  // ...
41  endclass
```

コンストラクタ（コンポーネント）

- コンポーネントは階層を構築する為、親コンポーネントの指定が必要になります。また、インスタンス名称には標準値を設定しません。同じ階層レベル内には、ユニークなインスタンス名称が必要な為、この様な配慮が必要になります。例えば、次の様に定義します。更に、super.new()を呼び出す事が必要です。

```
 8  class my_comp extends uvm_component;
 9  int      data[];
10  int      data_size = 2;
11  string   header_name = "data[0..1]";
12
13  `uvm_component_utils_begin(my_comp)
14          `uvm_field_string(header_name,UVM_DEFAULT)
15          `uvm_field_int(data_size,UVM_DEFAULT)
16          `uvm_field_array_int(data,UVM_DEFAULT)
17  `uvm_component_utils_end
18
19  function new(string name,uvm_component parent);
20          super.new(name,parent);
21  endfunction
22  //...
23  endclass
```

標準値を定義しない。

親コンポーネントを指定。

ファクトリ
（絶対に遵守すべきルール）

- オブジェクト、及び、コンポーネントのインスタンスを作る場合、newオペレータを使用する代わりに、factoryメソッドを使用する事が推奨されています。

```
37  my_comp          test;
38
39  initial begin
40          // fix the configuration.
41          uvm_config_db#(string)::set(null,"test","header_name","data[0..7]");
42          uvm_config_db#(int)::set(null,"test","data_size",8);
43          // create components.
44          test = my_comp::type_id::create("test",null);
45          run_test();
46  end
```

newオペレータを使用せずにファクトリ・メソッドを使用してインスタンスを作成する。

virtual interface名称

- Virtual interfaceは頻繁に使用する為、出来るだけ短い名称を使用する事が望ましい。

```
30  class apb_master_driver extends uvm_driver #(apb_transfer);
31  virtual apb_if vif;
32  apb_config cfg;
33  // Provide implementations of virtual methods such as get_type_name and create
34  `uvm_component_utils_begin(apb_master_driver)
35          `uvm_field_object(cfg, UVM_DEFAULT|UVM_REFERENCE)
36  `uvm_component_utils_end
37  // Drive all signals to reset state
38  task apb_master_driver::reset();
39          // If the reset is not active, then wait for it to become active before
40          // resetting the interface.
41          wait(!vif.preset);
42          // APB_MASTER_DRIVER tag required for Debug Labs
43          `uvm_info("APB_MASTER_DRIVER", $sformatf("Reset observed"), UVM_MEDIUM)
44          vif.paddr     <= 'h0;
45          vif.pwdata    <= 'h0;
46          vif.prwd      <= 'b0;
47          vif.psel      <= 'b0;
48          vif.penable   <= 'b0;
49  endtask : reset
50  // ...
51  endclass : apb_master_driver
```

短い名称を使用すると便利。

Interfaceは多くの信号を含む為、短い名称を使用するとタイピングの負担が軽減されると同時にミスが少なくなる。

サブコンポーネントの生成（絶対に遵守すべきルール）

- 通常、build_phase()でサブコンポーネントの生成をします。コンストラクタが実行する時点ではコンフィギュレーションが決定されていない為、コンストラクタ内でサブコンポーネントを作成する事は正しくありません。

```
49  class simple_master_agent extends uvm_agent;
50  protected int master_id;
51  simple_driver           driver;
52  simple_sequencer        sequencer;
53  simple_collector        collector;
54  simple_monitor          monitor;
55  `uvm_component_utils_begin(simple_master_agent)
56          `uvm_field_int(master_id, UVM_DEFAULT)
57          `uvm_field_enum(uvm_active_passive_enum,is_active,UVM_DEFAULT)
58  `uvm_component_utils_end
59  // ...
60  function void build_hase(uvm_phase phase);
61          collector = simple_collector::type_id::create("simple_collector",this);
62          monitor = simple_monitor::type_id::create("simple_monitor",this);
63          if( is_active == UVM_ACTIVE ) begin
64                  sequencer = simple_sequencer::type_id::create("simple_sequencer",this);
65                  driver = simple_driver::type_id::create("simple_driver",this);
66          end
67  endfunction
68  // ...
69  endclass
```

サブコンポーネント

サブコンポーネントとフィールド・マクロ

- サブ・コンポーネントにはフィールド・マクロを指定する必要はありません。寧ろ、指定してはいけません。
- uvm_agentのサブクラスの場合、is_activeに対してフィールド・マクロを定義しなければならない。

```
49  class simple_master_agent extends uvm_agent;
50  protected int master_id;
51  simple_driver           driver;
52  simple_sequencer        sequencer;
53  simple_collector        collector;
54  simple_monitor          monitor;
55
56  `uvm_component_utils_begin(simple_master_agent)
57          `uvm_field_int(master_id, UVM_DEFAULT)
58          `uvm_field_enum(uvm_active_passive_enum,is_active,UVM_DEFAULT)
59  `uvm_component_utils_end
60
61  function new(string name,uvm_phase phase);
62          super.new(name,phase);
63  endfunction
```

サブコンポーネント

is_activeのフィールド・マクロが必要。

制約とノブ（constraints and knobs）

- トランザクションに定義されているデータ項目は、通常、randomize()メソッドの対象となります。
- 様々な値を漏れなくテストする為には、重複した値を発生しない様にしなければなりません。然し、全ての値を試す事が出来ない場合もあります。
- 例えば、32ビットのdelayに値を設定する場合、全ての値を試す事は不可能です。この様な場合には、制約を与えて意味のある値域を定義します。
- ノブは制約を効果的に働かせる手段です。
- ノブはDUTに引き渡されないデータである為、フィールド・マクロのオプションにUVM_NOPACK、及び、UVM_NOCOMPAREを指定しなければなりません。

制約とノブの使用例

```
21  class simple_item extends uvm_sequence_item;
22  rand int unsigned addr;
23  rand int unsigned data;
24  rand int unsigned delay;
25  rand simple_item_delay_e delay_kind;
26  `uvm_object_utils_begin(simple_item)
27          `uvm_field_int(addr,UVM_DEFAULT)
28          `uvm_field_int(data,UVM_DEFAULT)
29          `uvm_field_int(delay,UVM_DEFAULT)
30          `uvm_field_enum(simple_item_delay_e,delay_kind,
31                  UVM_DEFAULT|UVM_NOCOMPARE|UVM_NOPACK)
32  `uvm_object_utils_end
33
34  constraint addr_c { addr < 16'h2000; }
35  constraint data_c { data < 16'h1000; }
36  constraint delay_c {
37          (delay_kind == ZERO) -> delay == 0;
38          (delay_kind == SHORT) -> delay inside { [1:10] };
39          (delay_kind == MEDIUM) -> delay inside { [11:99] };
40          (delay_kind == LARGE) -> delay inside { [100:999] };
41          (delay_kind == MAX) -> delay == 1000;
42  }
43  // ...
```

ノブ

ノブには不必要な処理を省略する指示をする。

トランザクションのデータ項目属性

- 定義項目はpublicであり、且つ、rand又はrandc属性が付加されるべき。

```
 1  class simple_item extends uvm_sequence_item;
 2  rand int unsigned addr;
 3  rand int unsigned data;
 4  rand int unsigned delay;
 5  rand simple_item_delay_e delay_kind;
 6  `uvm_object_utils_begin(simple_item)
 7          `uvm_field_int(addr,UVM_DEFAULT)
 8          `uvm_field_int(data,UVM_DEFAULT)
 9          `uvm_field_int(delay,UVM_DEFAULT)
10          `uvm_field_enum(simple_item_delay_e,delay_kind,
11                  UVM_DEFAULT|UVM_NOCOMPARE|UVM_NOPACK)
12  `uvm_object_utils_end
13
14  constraint addr_c { addr < 16'h2000; }
15  constraint data_c { data < 16'h1000; }
16  constraint delay_c {
17          (delay_kind == ZERO) -> delay == 0;
18          (delay_kind == SHORT) -> delay inside { [1:10] };
19          (delay_kind == MEDIUM) -> delay inside { [11:99] };
20          (delay_kind == LARGE) -> delay inside { [100:999] };
21          (delay_kind == MAX) -> delay == 1000;
22  }
23  // ...
24  endclass
```

protected又はlocal属性を付加しない。

コンフィギュレーションの設定変更

- コンフィギュレーションを実行時に変更する事を可能にする為には、uvm_config_db#(type)クラスを使用しなければなりません。このクラスを使用する事により、ソース・コードを修正せずに、コンポーネントの構成を変える事が出来ます。
- 例えば、ソース・コード中ではコンポーネント内のdata_sizeが2と定義されていても、テストベンチでdata_size を8に変更する事が出来ます。但し、この変更情報をbuild_phase() が実行する前に指定しなければなりません。build_phase()では、テストベンチで指定した情報が使用されるので、階層構造が指定した状態に変更されます。
- 具体的には、build_phase() でsuper.build_phase(phase)を呼び出すとapply_config_settings()が呼ばれてdata_sizeに変更値（この場合には8）が設定されます。この様に、super.build_phase(phase)は必須な呼び出しとなります。テストベンチでは、uvm_config_db#(int)::set()メソッドを使用して変更値を設定します。そして、この変更が有効になる為には、`uvm_fieldマクロが正しく設定されていなければなりません。

コンフィギュレーションの設定変更（使用例:その1）

```
 7  class my_comp extends uvm_component;
 8  int      data[];
 9  int      data_size = 2;
10  string   header_name = "data[0..1]";
11
12  `uvm_component_utils_begin(my_comp)
13          `uvm_field_string(header_name,UVM_DEFAULT)
14          `uvm_field_int(data_size,UVM_DEFAULT)
15          `uvm_field_array_int(data,UVM_DEFAULT)
16  `uvm_component_utils_end
17
18  function new(string name,uvm_component parent);
19          super.new(name,parent);
20  endfunction
21
22  function void build_phase(uvm_phase phase);
23          super.build_phase(phase);
24          data = new[data_size];
25          foreach(data[i])
26                  data[i] = i;
27  endfunction
28
29  function void end_of_elaboration_phase(uvm_phase phase);
30          super.end_of_elaboration_phase(phase);
31          this.print();
32  endfunction
33  endclass
```

これらの設定を変更します。

コンフィギュレーションの設定変更（使用例:その2）

- テストベンチでは、次の様にdata_size、及び、header_nameに新しい値を設定します。

```
35  my_comp          test;
36
37          initial begin
38                  // fix the configuration.
39                  uvm_config_db#(string)::set(null,"test","header_name","data[0..7]");
40                  uvm_config_db#(int)::set(null,"test","data_size",8);
41                  // create components.
42                  test = my_comp::type_id::create("test",null);
43                  run_test();
44          end
```

uvm_config_db#(type) に関する注意（重要）

- typeに関して厳密です。即ち、設定するプロパーティのタイプと完全に一致しなければなりません。例えば、もしdata_sizeがint型ではなくbyte型であれば、uvm_config_db#(byte)::set()を使用しなければなりません。
- 型の不一致は設定変更の失敗を導く為、UVMはより寛大なクラスを提供しています。例えば、uvm_config_int::set()を使用すると、整数型の全てのプロパーティに対して設定変更をする事が出来ます。詳細に関しては、UVMのマニュアルを参照して下さい。

UVMプリンター

- クラスのメンバーをプリントする為には、print()メソッドを使用します。このメソッドが正しく動作する為には、`uvm_fieldマクロを設定する必要があります。UVMは次の三種類のプリンターを定義しています。

プリンター	機能
uvm_default_table_printer	表形式にプリントします。最も複雑なプリント処理となります。
uvm_default_tree_printer	各行に名称と値の対をプリントします。
uvm_default_line_printer	名称と値の対を一行にプリントします。

UVMプリンター（使用例）

```
 1  my_obj           obj;
 2
 3  initial begin
 4          obj = my_obj::type_id::create();
 5
 6          obj.request = 8'hf0;
 7          obj.grant = 8'h01;
 8          obj.data = { 1, 2, 3, 4, 5, 6, 7, 8 };
 9
10          obj.print();
11          obj.print(uvm_default_tree_printer);
12          obj.print(uvm_default_line_printer);
13  end
```

UVMプリンター（使用例）

- 以下の例では、三種類のプリンターを使用しています。

```
 1  my_obj           obj;
 2
 3  initial begin
 4          obj = my_obj::type_id::create();
 5
 6          obj.request = 8'hf0;
 7          obj.grant = 8'h01;
 8          obj.data = { 1, 2, 3, 4, 5, 6, 7, 8 };
 9
10          obj.print();
11          obj.print(uvm_default_tree_printer);
12          obj.print(uvm_default_line_printer);
13  end
```

- 次の様にして標準プリンターを定義する事が出来ます。

```
initial begin
        uvm_default_printer = uvm_default_tree_printer;
        // ...
end
```

uvm_default_table_printer

```
------------------------------------
Name        Type           Size  Value
------------------------------------
my_obj      my_obj         -     @272
  request   integral       8     'hf0
  grant     integral       8     'h1
  data      sa(integral)   8     -
    [0]     integral       32    'h1
    [1]     integral       32    'h2
    [2]     integral       32    'h3
    [3]     integral       32    'h4
    [4]     integral       32    'h5
    [5]     integral       32    'h6
    [6]     integral       32    'h7
    [7]     integral       32    'h8
------------------------------------
```

uvm_default_tree_printer

```
my_obj: (my_obj@272) {
  request: 'hf0
  grant: 'h1
  data: {
    [0]: 'h1
    [1]: 'h2
    [2]: 'h3
    [3]: 'h4
    [4]: 'h5
    [5]: 'h6
    [6]: 'h7
    [7]: 'h8
  }
}
```

uvm_default_line_printer

```
my_obj: (my_obj@272) { request: 'hf0  grant: 'h1  data: { [0]: 'h1  ...  [7]: 'h8  } }
```

テストの選択

- UVMにはrun_test()というタスクがあります。run_test()は準備したテストを実行する為の機能です。run_test()にテスト名を指定する事が出来ますが、実行時にテスト名を指定する方法が一般的です。この方法を使用すると、一回のコンパイルで複数のテストを実行する事が出来ます。
- シミュレーションを実行する時、コマンド・ラインでテスト名を指定する方法は以下の様になります。ここで、svsimはツールにより異なります。ベンダー提供のマニュアルを参照して下さい。

```
svsim +UVM_TESTNAME=test_rmw
```

programブロック（重要な制限）

- SystemVerilogにはprogramというコンストラクトがありますが、UVMでは使用すべきではありません。
- program内の論理はreactive regionで実行するのに対して、UVMはactive regionで実行します。従って、タイミング制御が正しく行われません。

まとめ

以上、UVMの概要、及び、UVMを適用する際に考慮すべき点をガイドラインとして解説しました。これらの知識があれば、UVMユーザ・ガイドを読む事が出来ます。

UVMの使用法をマスターして、検証作業の効率を向上する事が出来る事を願っています。

参考文献

本資料公開後、多くの部分が読み易い書物として編纂されています。興味のある方は文献[5-7]を参照ください。

[1] IEEE Std 1800-2017: IEEE Standard for SystemVerilog – Unified Hardware Design, Specification and Verification Language.

[2] Universal Verification Methodology (UVM) 1.2 User's Guide, Accellera, October 8, 2015.

[3] Chris Spear: SystemVerilog for Verification, 2nd Edition, Springer 2008.

[4] Kathleen A. Meade and Sharon Rosenberg: A Practical Guide to Adopting the Universal Verification Methodology (UVM), 2nd Edition Cadence Design Systems, Inc. 2013.

[5] 篠塚一也、SystemVerilogによる検証の基礎、森北出版 2020.

[6] 篠塚一也、SystemVerilog入門、共立出版 2020.

[7] 篠塚一也、実践UVM入門、森北出版 2021.