Artgraphics

初心者の為のUVM概説

篠塚一也

アートグラフィックス

Document Revision: 2.2,2020.09.12

www.artgraphics.co.jp

注意事項
（Caveat）

日本国内にはUVMに関する良書が皆無な為、本資料を作成しました。UVMの知識を個人的に習得する目的として本資料を活用して下さい。本資料を通して、業務（実践）で必要となるUVMに関する知識を習得して頂くのが本来の目的です。

転用目的（本来の目的と違った他の用途に使う事）で本資料を使用する事はご遠慮下さい。また、本資料から学んだ知識を転載する場合等は出典が本資料である事を明記して下さい。但し、他の著者の文書にも書かれている内容は、この限りではありません。本注意事項は現在及び過去に於ける弊社からの全てのフリー・ダウンロード資料に適用されます。

本注意事項に合意出来ない場合には、本資料を速やかに抹消して下さい。

本資料の目的と概要

- 本資料は初心者向けにUVMの概要を解説します。UVMに関する基礎的な知識を習得したい方に適した学習用素材です。
- 基礎的な知識といえども実践に役立つ技術を身に付ける事を目的にする為に、題材を広く選択して行きます。従って、本資料は一度限りの執筆では無く、数回に渡り内容を充実して行く予定です。
- 内容の更新は不定期に行ないますのでご了承下さい。
- 本資料は UVM1.2 をベースにして解説を進めます。
- 尚、本資料はSystemVerilogに関する基礎知識を仮定しています。

第一章
UVM 概要

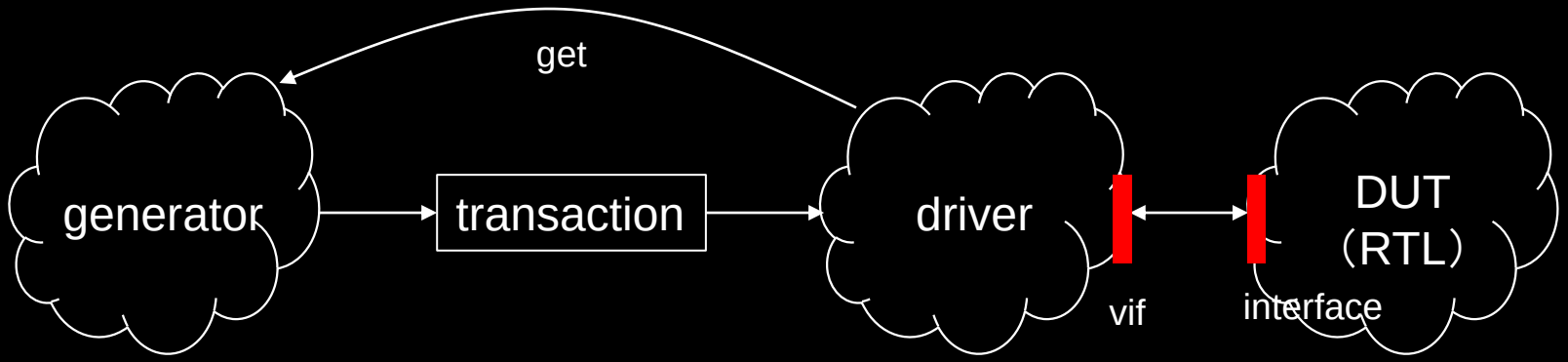
本章ではUVMとは何か、又、その目的等を解説します。そして、UVMがどの様に構成されているかを概説します。これらの知識は後章への準備となります。

UVMとは何か？

- Universal Verification Methodology (UVM)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たSystemVerilogのクラス・ライブラリーです。UVMは検証技術の再利用性を促進して生産性向上をさせる目的を持っています。
- 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。
- 但し、使用しているEDAツールがUVMの全ての機能を実装しているとは限りません。

トランザクション

- UVMはトランザクションを使用して実行します。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。
- driverがgeneratorにトランザクションを要求すると、generatorは制約を満たすトランザクションを作成してdriverに引き渡します。
- driverは取得したトランザクションをシグナル・レベル（RTL）に変換してDUTをドライブします。その際、interface、及び、virtual interface（vif）が使用されます。



vif: virtual interface

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。
- 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側からのレスポンス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。

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す。大きく分けて次の二種類のクラスがあります。

  - トランザクション、又は、シナリオを記述する為のクラス
  - トランザクションを処理して検証するUVMコンポーネント（メソドロジー・クラスと呼ばれます）

- ユーザは上記のUVMクラスを使用してトランザクション、及び、検証コンポーネントを定義して行きます。その際、SystemVerilogのクラス・インヘリタンス（extends）を使用します。

トランザクション関連のUVMクラス

UVMクラス	機能及び目的
uvm_sequence_item	トランザクションを記述する為のベース・クラスです。データ・オブジェクトであり、コンポーネントではありません。
uvm_sequence	トランザクションを生成する為に必要な手順を提供するベース・クラスです。手順の中には他のシーケンスへの引用を含む事が出来るので、階層的にテスト・シーケンスを構築する事が出来ます。シーケンスもデータ・オブジェクトです。

トランザクション（記述例）

```
 1   typedef enum {ZERO, SHORT, MEDIUM, LARGE, MAX} simple_item_delay_e;
 2
 3   class simple_item extends uvm_sequence_item;
 4   rand int unsigned addr;
 5   rand int unsigned data;
 6   rand int unsigned delay;
 7   rand simple_item_delay_e delay_kind;
 8   `uvm_object_utils_begin(simple_item)
 9           `uvm_field_int(addr,UVM_DEFAULT)
10           `uvm_field_int(data,UVM_DEFAULT)
11           `uvm_field_int(delay,UVM_DEFAULT)
12           `uvm_field_enum(simple_item_delay_e,delay_kind,
13                           UVM_DEFAULT|UVM_NOCOMPARE|UVM_NOPACK)
14   `uvm_object_utils_end
15
16   function new(string name="simple_item");
17           super.new(name);
18   endfunction
19   endclass
```

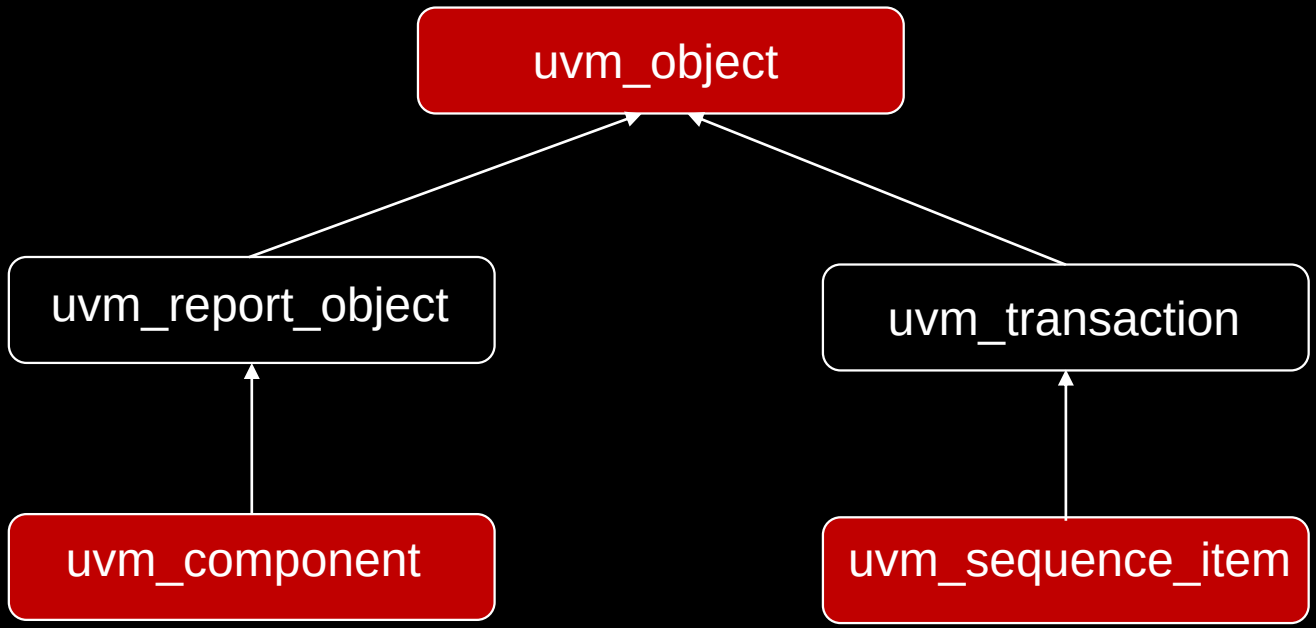
メソドロジー・クラス

UVMクラス	機能及び目的
uvm_driver	DUTをドライブするコンポーネントを定義する為のベース・クラスです。
uvm_sequencer	トランザクションを生成する手順を制御するコンポーネントのベース・クラスとして使用されます。
uvm_env	agent、montor等を含んだ検証コンポーネントを定義する為のベース・クラスです。
uvm_agent	sequencer、driver、monitor、collectorから構成される検証コンポーネントを定義する為のベース・クラスです。
uvm_test	テストベンチを構築する為のベース・クラスです。
uvm_monitor	monitorの基本機能を備えたベース・クラスです。
uvm_scoreboard	scoreboardの基本機能を提供するベース・クラスです。

メソドロジー・クラスは、uvm_componentのサブクラスです。

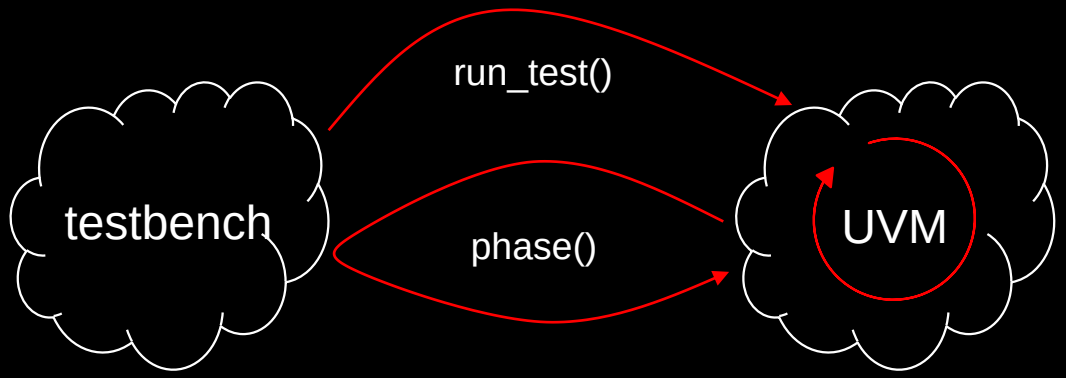
uvm_objectクラス

- UVMには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シミュレーション制御

- シミュレーションの実行制御は全てUVMが司ります。ユーザ側に実行制御権はありません。
- テストベンチからUVMのrun_test()メソッドを呼ぶと、UVMによるシミュレーションが開始します。それ以降の実行制御はUVMが行います。
- UVMコンポーネントが呼び出されるタイミングは予め決定されておりシミュレーション・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（simulation phases）

フェーズ・メソッド	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（解説）

- シミュレーション・フェーズはvirtualメソッドとして定義されています。従って、ユーザは必要なフェーズだけ記述すれば良い事になっています。
- run_phaseだけがtaskです。それ以外はすべてfunctionです。
- 前ページに於けるフェーズは、表に記述されている順に制御を受けます。例えば、build_phaseが最初に呼ばれた後、connect_phaseが次に呼ばれます。
- 従って、クラス内にこれらのメソッドを定義する際には、表の順序に従いメソッドを記述する事を勧めます。

シミュレーション・フェーズ（記述例）

```
 1   class simple_driver extends uvm_driver #(simple_item);
 2   simple_item     item;
 3   virtual dut_if  vif;
 4
 5   `uvm_component_utils(simple_driver)
 6
 7   function new(string name,uvm_component parent);
 8           super.new(name,parent);
 9   endfunction
10
11   function void build_phase(uvm_phase phase);
12   // ...
13   endfunction
14
15   task run_phase(uvm_phase phase);
16   // ...
17   endtask
18   // ...
19   endclass
```

シミュレーション・フェーズとsuper.method

- コンストラクタ、及び、シミュレーション・フェーズではベース・クラスのメソッドの呼び出しが必須の条件となる事があります。以下はその纏めです。

メソッド	必要性
new	必ず、super.newをコンストラクタの先頭で呼び出さなければなりません。これはUVMの規則となっています。
build_phase	super.build_phaseを呼び出さなければなりません。ベース・クラスがapply_config_settingを呼び出してコンフィギュレーション変更処理を行います。
connect_phase	一般的にはsuper.connect_phaseを呼び出した方が良いと考えられます。
end_of_elaboration_phase	super. end_of_elaboration_phaseの呼び出しは必要であると考えられます。UVMのソース・コードを観察するとベース・クラスのメソッドを呼び出している例が多く見当たります。

UVMを適用する手順

- UVMを検証作業に適用する為には、概略次の様な手順を踏みます。

- トランザクションを定義する。
- 検証コンポーネントを定義する。
- トップ・レベルのテストベンチを定義する。

- クラスを定義する際には、前述したクラスを利用して生産性を高めます。

トップ・レベルの検証コード例

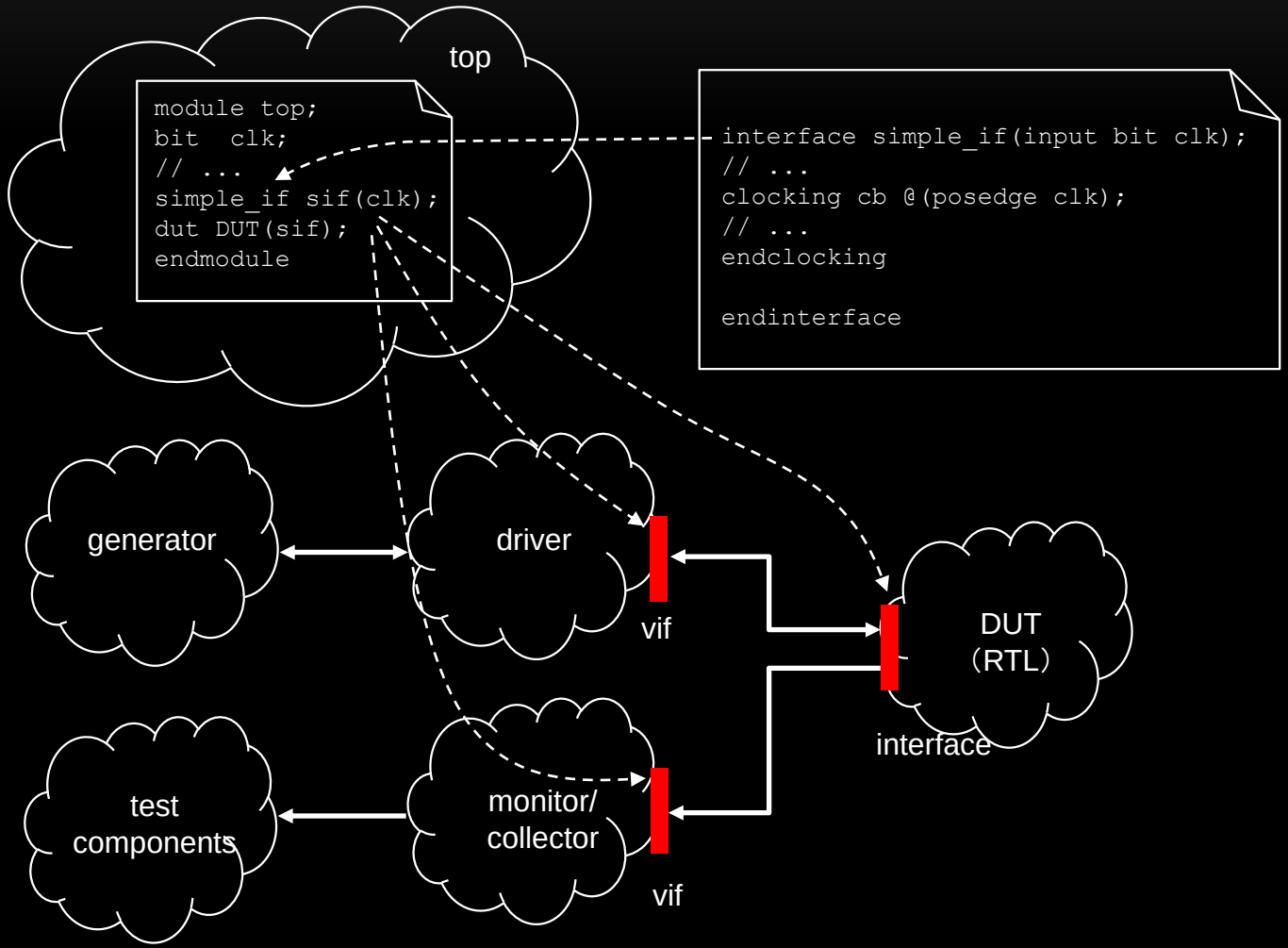
```
 1   `include "uvm_pkg.sv"
 2   `include "uvm_macros.svh"
 3   `include "simple_if.sv"
 4   `include "my_pkg.sv"
 5
 6   module top;
 7   import uvm_pkg::*;
 8   import my_pkg::*;
 9   bit     clk;
10
11   simple_if sif(clk);
12   dut DUT(sif);
13
14   initial begin
15           setup_config(); // set up configuration
16           run_test();     // start UVM
17   end
18   // ...
19   endmodule
```

検証コードを含むパッケージ例

- パッケージを`include文で構成します。
- 忘れずに、uvm_pkgをimportして下さい。

```
1   `ifndef MY_PKG_H
2   `define MY_PKG_H
3
4   package my_pkg;
5   import uvm_pkg::*;
6
7   `include "my_testbench.sv"
8   // Include other files as well.
9
10  endpackage
11
12  `endif
```

UVMを適用したテスト環境



第二章
トランザクション

この章ではUVMでトランザクションを定義する為の知識を解説します。

トランザクション

- トランザクションはDUTをドライブする為に使用されます。
- Driverはトランザクションをgeneratorから取得します。
- Generatorはトランザクションを作成し、乱数を発生した後、driverにトランザクションを引き渡します。
- トランザクションに含まれる項目は仕様から決定されます。

トランザクション（定義法）

- uvm_sequence_item、又は、そのサブクラスからトランザクションを定義する。
- トランザクションに必要なフィールドを定義する。
- UVMマクロを定義する。
- コンストラクタを定義する。

トランザクション（定義例）

- Generatorが乱数を発生させる為、プロパーティにはrand、又は、randc属性を付加する。

```
 1    typedef bit [15:0]       item_type_t;
 2
 3    class simple_item_base_t extends uvm_sequence_item;
 4    rand item_type_t         addr;
 5    rand item_type_t         data;
 6    rand item_type_t         delay;
 7
 8    `uvm_object_utils_begin(simple_item_base_t)
 9            `uvm_field_int(addr,UVM_DEFAULT)
10            `uvm_field_int(data,UVM_DEFAULT)
11            `uvm_field_int(delay,UVM_DEFAULT)
12    `uvm_object_utils_end
13
14    function new(string name="simple_item_base_t");
15            super.new(name);
16    endfunction
17    endclass
```

トランザクションとクラス・インヘリタンス

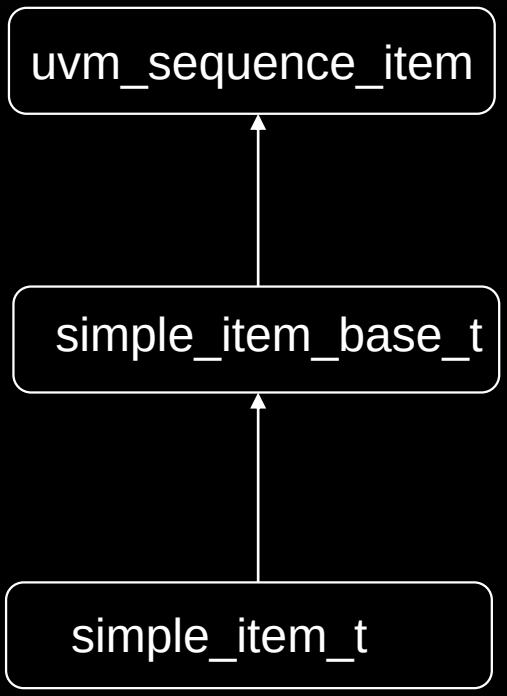
- トランザクションは様々なテスト・ケースに応じて柔軟な対応が求められます。その様な多様性に対して唯一つのトランザクション定義ではコードが複雑になり、維持する為のコストが増大します。
- 従って、様々なテスト・ケースに対応し得るトランザクション定義法を実装しなければなりません。
- クラス・インヘリタンスがその解答を提供します。
  - トランザクションのベース・クラスには、仕様から導いた項目だけを定義する。
  - テスト・ケースに依存する条件をクラス・インヘリタンスにより得られたサブクラスに追加して行く。

クラス・インヘリタンス（定義例）

- 下記の定義例は、先程のトランザクションに制約を課しています。
- この制約によると乱数発生時に、addrは4の倍数、dataは0~2048の間の数になる様に限定されます。

```
1   class simple_item_t extends simple_item_base_t;
2
3   constraint C_ADDR { addr[1:0] == 2'b00 ; }
4   constraint C_DATA { data inside { [0:2048] }; }
5
6   function new(string name="simple_item_t");
7           super.new(name);
8   endfunction
9   endclass
```

クラス・インヘリタンスによるトランザクションの詳細定義



第三章
TLM（Transaction Level Modeling）

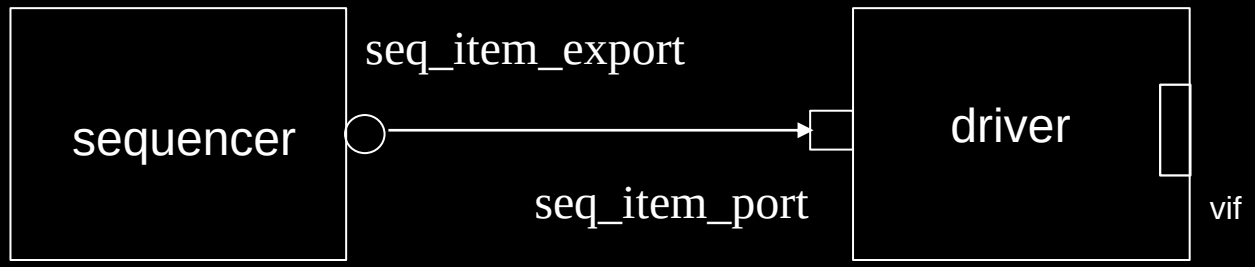
トランザクションはRTLより高位のレベルでのデータ処理に適用されます。UVMではTLMを使用します。SystemCのTLM 1.0に対応していますが、それよりも高速に動作します。

UVMコンポーネント間の通信

- 一対一の通信には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起動します。
- 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。
- コレクターとモニターは一対一の関係ですが、トランザクションの送信を即時に完了する為にTLM-portを使用しません。その代り、analysis portを使用します。
- モニターは、N個のsubscribersにトランザクションを伝達（broadcast）する為、analysis portを使用します。Subscribersはanalysis exportsを使用してwriteメソッドを実装します。

SequencerとDriverの通信

- UVMではトランザクションを取得する操作を下記の様に表現します（[2]）。

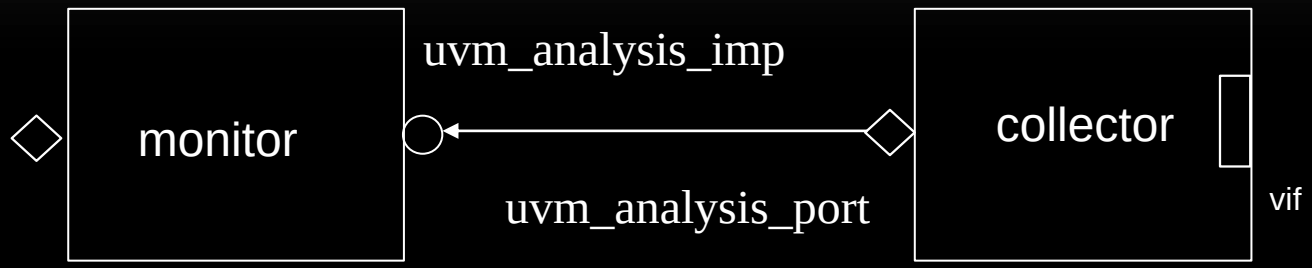


- ドライバーがseq_item_port.get_next_item(item)を呼ぶとシーケンサーがトランザクションを作成してドライバーに戻します。

```
class driver extends uvm_driver #(simple_item);
...
task run_phase(uvm_phase phase);
...
seq_item_port.get_next_item(item);
...
endtask
endclass
```

MonitorとCollectorの通信

- UVMでは下記の様に表現します（[2],[3]）。

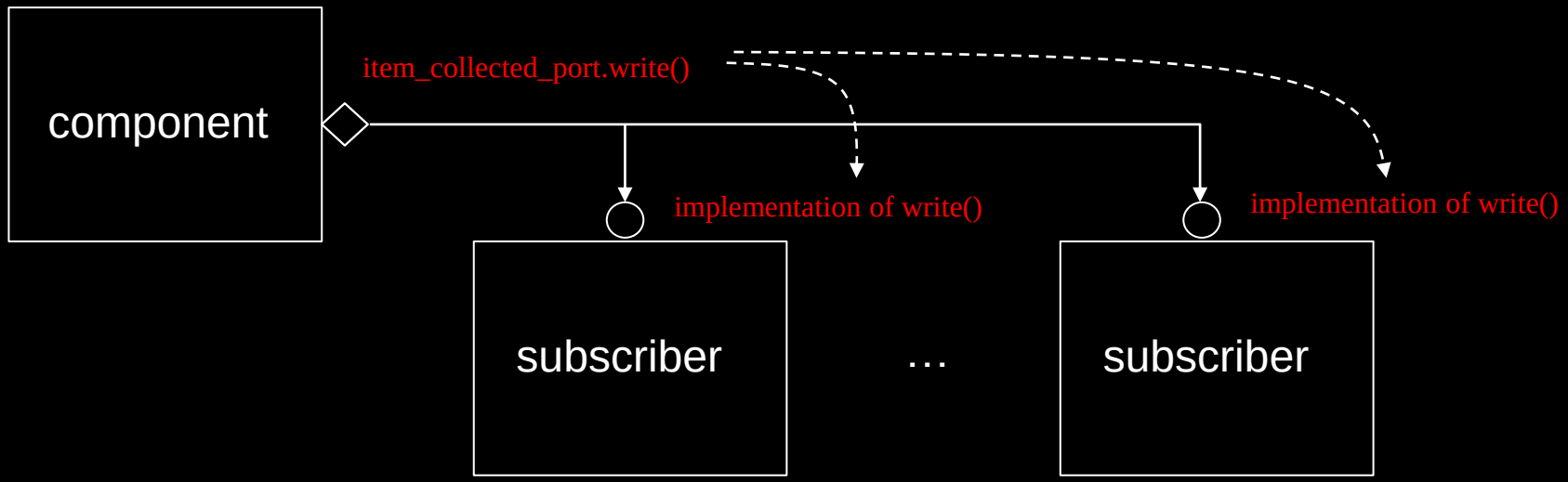


- コレクターはトランザクションを作成し、item_collected_port.write(item)を呼び出します。

```
class collector extends uvm_component;
uvm_analysis_port #(simple_item)        item_collected_port;
...
task collect_transactions();
...
@ vif.cb;
...
item_collected_port.write(item);
...
endtask
...
endclass
```

analysis_port (uvm_analysis_port)

- analysis_portは特殊なTLM-portで、それに接続されている analysis_exportのリストを保有しています。
- analysis_portのwrite()メソッドが呼ばれると、リストを走査して、 analysis_exportが接続されているコンポーネント（subscriber）のwrite()メソッドを呼び出します。



第四章
シーケンサーとドライバー

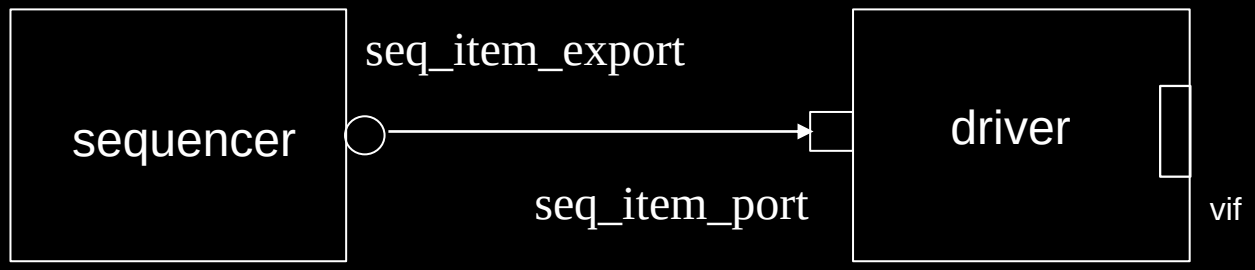
この章では、シーケンサーとドライバーの通信に関して解説します。

シーケンサーとドライバーの通信

- 既に紹介した様に、シーケンサーとドライバーはTLM-portsとTLM-exportsを使用して通信を行います。
- シーケンサーとドライバーが同じプロセス内で順に動作する場合には、両者を直接接続すれば目的を達成する事が出来ます。
- 一方、もし両者が独立したプロセス内で動作する場合には、シーケンサーが生成するトランザクションをプーリングしなければなりません。その際、FIFOリスト（tlm_fifo）が使用されます。
- FIFOリストが空の場合、ドライバーは待たなければなりません。FIFOリストがフルの場合、シーケンサーは待たなければなりません。
- シーケンサーとドライバーを結びつけるのは、親コンポーネント（通常はagent）が行います。

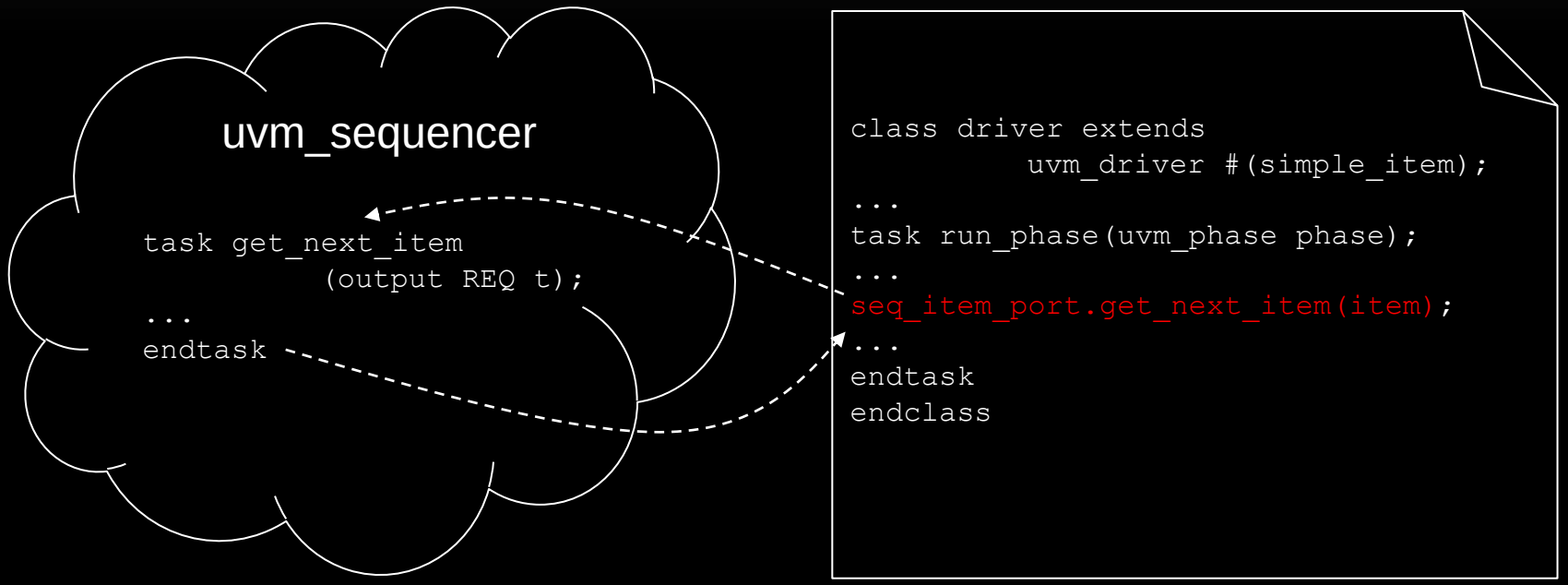
直接接続

- 既に紹介した図を再度引用します。



- ドライバーがseq_item_port.get_next_item(item)を呼ぶとシーケンサーがトランザクションを作成してドライバーに戻します。

トランザクションを取得する仕組み



ドライバーのrun_phaseの構造

- 概して、ドライバーのrun_phaseは以下の様な構造となります。トランザクション取得を無限回繰り返します。

```
24    task simple_driver::run_phase(uvm_phase phase);
25            forever begin
26                    get_and_drive();
27            end
28    endtask
29
30    task simple_driver::get_and_drive();
31                    // Get the next data item from sequencer.
32                    seq_item_port.get_next_item(req);
33                    // Execute the item.
34                    drive_item(req);
35                    // tell sequencer that item is done.
36                    seq_item_port.item_done();
37    endtask
38
39    task simple_driver::drive_item(input simple_item item);
40            vif.a <= item.a;
41            vif.b <= item.b;
42            vif.op_code <= item.op_code;
43    endtask
```

ドライバーのrun_phaseに関して

- 前頁で示した様にrun_phaseはforeverで無限回ループしますが、正しい記述です。実は、無限ループしません。
- raise_objectionでraiseしたobjectionsの数がdrop_objectionにより0になるとシミュレーションは終了します。即ち、run_phaseは終了させられます。
- 通常は、シーケンスがobjectionsの数を0にします。詳細は文献[6]に詳しく書かれています。
- この機能はUVMの優れた特徴の一つです。

シーケンサーの例

- 概して、以下の様な記述で充分です。

```
1   class simple_sequencer extends uvm_sequencer #(simple_item);
2
3   `uvm_component_utils(simple_sequencer)
4
5   function new(string name,uvm_component parent);
6           super.new(name,parent);
7   endfunction
8
9   endclass
```

ドライバーの例
（クラス全体の定義）

```
 1   class simple_driver extends uvm_driver #(simple_item);
 2   virtual simple_if       vif;
 3
 4   `uvm_component_utils(simple_driver)
 5
 6   function new(string name,uvm_component parent);
 7           super.new(name,parent);
 8   endfunction
 9
10   extern function void build_phase(uvm_phase phase);
11   extern task run_phase(uvm_phase phase);
12   extern task get_and_drive();
13   extern task drive_item(input simple_item item);
14   endclass
```

ドライバーの例（外部宣言の定義）

```
16    function void simple_driver::build_phase(uvm_phase phase);
17            super.build_phase(phase);
18            if( !uvm_config_db#(virtual simple_if)::get(this,"","vif",vif) )
19                    `uvm_error("NO-VIF",{"VIF error for ",
20                                         get_full_name(),".vif"})
21
22    endfunction
23
24    task simple_driver::run_phase(uvm_phase phase);
25            forever begin
26                    get_and_drive();
27            end
28    endtask
29
30    task simple_driver::get_and_drive();
31            // Get the next data item from sequencer.
32            seq_item_port.get_next_item(req);
33            // wait n clocks.
34            repeat( req.delay ) @(posedge vif.clk);
35            // Execute the item.
36            drive_item(req);
37            // tell sequencer that item is done.
38            seq_item_port.item_done();
39    endtask
40
41    task simple_driver::drive_item(input simple_item item);
42            vif.a <= item.a;
43            vif.b <= item.b;
44            vif.op_code <= item.op_code;
45    endtask
```

tlm_fifo

- シーケンサーとドライバーが独立したプロセスで動作する場合には、下図に示す様な FIFO を使用します（[2]）。

- 図から容易に分かる様に、producer、及び、consumerはそれぞれ、put()、及び、get()を呼び出すだけで済みます。

シーケンス

- シーケンスを定義しないと検証を自動的に実行する事は出来ません。
- uvm_sequence、又は、そのサブクラスからシーケンスを定義します。
- シーケンスのタスクbody()を必ず実装しなければなりません。このタスクがトランザクションを生成してドライバーに引き渡します。
- 通常は、以下の様な簡単な記述で済みます。制約を課する為には、`uvm_do_withマクロを使用して下さい。
- `uvm_do_withマクロはSystemVerilogのin-line constraints機能を利用します。
- トランザクションを生成してドライバーに引き渡します処理がnum_data回繰り返します。一連の詳しい処理は、文献[7]に記述されています。

```
18    task simple_sequence::body();
19            repeat( num_data )
20                    `uvm_do(req);
21    endtask
```


シーケンスの例

```
 1   class simple_sequence extends uvm_sequence #(simple_item);
 2   rand int         num_data;
 3
 4   `uvm_object_utils_begin(simple_sequence)
 5           `uvm_field_int(num_data,UVM_DEFAULT)
 6   `uvm_component_utils_end
 7
 8   constraint C_NUM_DATA { num_data inside { [8:16] }; }
 9
10   function new(string name="simple_sequence");
11           super.new(name);
12   endfunction
13
14   extern virtual task body();
15   endclass
16
17   task simple_sequence::body();
18           repeat( num_data )
19                   `uvm_do(req);
20   endtask
```

第五章
モニターとコレクター

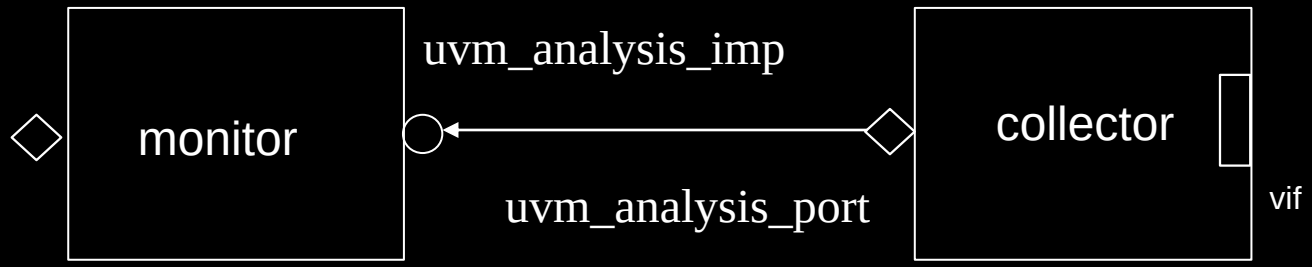
モニターからコレクター機能を分離する方法は処理を明確化します。本章では、この点を強調します。

コレクター

- モニターはDUTのレスポンスの変化を検知して他のコンポーネントに知らせる（broadcast）役割を果たします。
- モニターはその他の機能も持つ為、DUTからのレスポンス収集機能をコレクターとして分離すると構造的に分かり易くなります。
- コレクターはDUTからのレスポンスをvirtualインターフェースを介して取得し、トランザクションに変換してモニターに送信します。

モニターとコレクターの通信

- 以前紹介した図を再度引用します。



コレクター

- クロッキング・イベントに同期（@vif.cb）してDUTからのレスポンスをチェックし、トランザクションに変換します。
- 変換後、write()メソッドでモニターにトランザクションを送信します。

```
class collector extends uvm_component;
virtual dut_if      vif;
uvm_analysis_port #(simple_item)        item_collected_port;
...
task run_phase(uvm_phase phase);
simple_item         item;
          item = simple_item::type_id::create("simple_item");
          forever begin
                    @vif.cb;
                    // ...
                    item_collected_port.write(item);
          end
endtask
...
endclass
```

コレクターの例

```
1    class simple_collector extends uvm_component;
2    virtual simple_if   vif;
3    uvm_analysis_port #(simple_item)        analysis_port;
4
5    `uvm_component_utils(simple_collector)
6
7    function new(string name,uvm_component parent);
8        super.new(name,parent);
9        analysis_port = new("analysis_port",this);
10   endfunction
11
12   extern function void connect_phase(uvm_phase phase);
13   extern task run_phase(uvm_phase phase);
14   extern task get_response();
15   endclass
16
17   function void simple_collector::connect_phase(uvm_phase phase);
18       super.connect_phase(phase);
19       if( !uvm_config_db #(virtual simple_if)::get(this,get_full_name(),"vif",vif) )
20           `uvm_error("NO-VIF",{"VIF error for ",get_full_name(),".vif"})
21   endfunction
22
23   task simple_collector::run_phase(uvm_phase phase);
24       get_response();
25   endtask
26
27   task simple_collector::get_response();
28   simple_item data;
29       data = simple_item::type_id::create("dut_response");
30       forever begin
31           @(vif.q);   // wait until DUT response changes.
32           data.a = vif.a;
33           data.b = vif.b;
34           data.op_code = vif.op_code;
35           data.q = vif.q;
36           analysis_port.write(data);  // send transaction to monitor
37       end
38   endtask
```

モニター

- uvm_monitor又はそのサブクラスからモニターを定義します。
- コレクターからトランザクションを受信する為に、write()メソッドを定義します。
- 他の検証コンポーネントにトランザクションを伝達する。

```
class simple_monitor extends uvm_monitor;
bit       checks_enable = 1;
bit       coverage_enable = 1;
uvm_analysis_port #(simple_item)        item_collected_port;
uvm_analysis_imp #(simple_item)         coll_mon_export;
simple_item         collected_item;

`uvm_component_utils(simple_monitor)

// ...
function void write(simple_item item);
          collected_item = item;
          if( checks_enable ) perform_checks();
          if( coverage_enable )         perform_coverage();
          item_collected_port.write(collected_item);
endfunction
endclass
```

Monitorの例

```
 1   class monitor extends uvm_monitor;
 2   uvm_analysis_imp #(simple_item,monitor) analysis_export;
 3   uvm_analysis_port #(simple_item)        item_collected_port;
 4
 5   `uvm_component_utils(monitor)
 6
 7   function new(string name,uvm_component parent);
 8           super.new(name,parent);
 9           analysis_export = new("analysis_port",this);
10           item_collected_port = new("item_colleted_port",this);
11   endfunction
12
13   extern function void write(simple_item data);
14   endclass
15
16   function void monitor::write(simple_item data);
17           $display("@%4t: a=%0d b=%0d op_code=%3s q=%3d",
18                   $time,data.a,data.b,data.op_code.name,data.q);
19
20           item_collected_port.write(data);
21   endfunction
```

第六章
検証コンポーネント

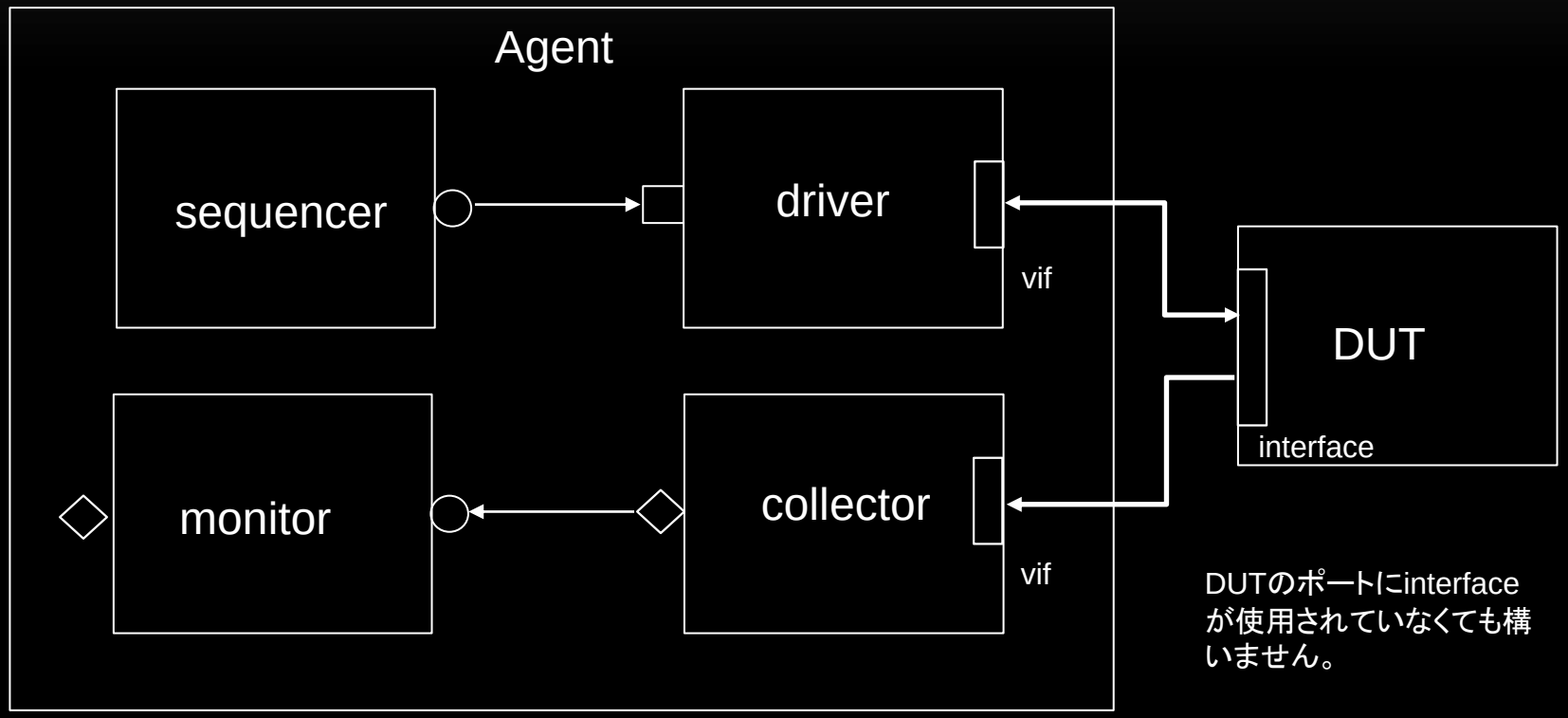
本章ではagent、environment等のコンポーネントを紹介します。

Agent

- 通常、ドライバー、シーケンサー、モニター（コレクターを含む）は一組として一緒に使用されます。この為、UVMではこれらのコンポーネントを含む為のコンテイナーを提供しています。そのコンポーネントがuvm_agentです。
- uvm_agentはトポロジーの柔軟性を高める為に、コンフィギュレーション・パラメータを装備しています。
- 代表的なパラメータとしてis_activeプロパーティがあります。

is_active	機能
UVM_ACTIVE	このモードでは、agentはデバイスをエミュレーションしてDUTをドライブします。また、チェック、及び、カバレッジをする為にモニター（コレクターを含む）も使用します。
UVM_PASSIVE	このモードでは、シーケンサー、及び、ドライバーを装備しません。モニター（コレクターを含む）のみ装備します。

Agentの構造

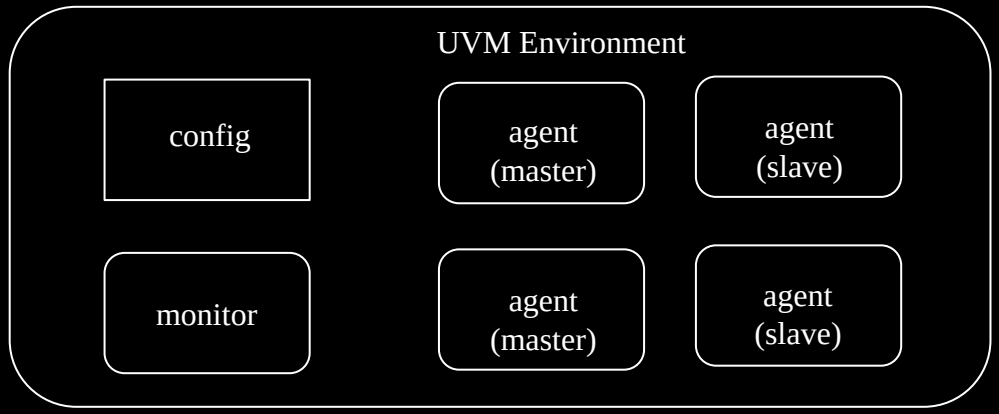


Agentの例

```
1    class simple_agent extends uvm_agent;
2    simple_driver       driver;
3    simple_sequencer    sequencer;
4    simple_collector    collector;
5    simple_monitor      monitor;
6
7    `uvm_component_utils_begin(simple_agent)
8        `uvm_field_enum(uvm_active_passive_enum,is_active,UVM_DEFAULT)
9    `uvm_component_utils_end
10
11   function new(string name,uvm_component parent);
12       super.new(name,parent);
13   endfunction
14
15   extern function void build_phase(uvm_phase phase);
16   extern function void connect_phase(uvm_phase phase);
17   endclass
18
19   function void simple_agent::build_phase(uvm_phase phase);
20       super.build_phase(phase);
21       collector = simple_collector::type_id::create("collector",this);
22       monitor = simple_monitor::type_id::create("monitor",this);
23       if( is_active == UVM_ACTIVE ) begin
24           sequencer = simple_sequencer::type_id::create("sequencer",this);
25           driver = simple_driver::type_id::create("driver",this);
26       end
27   endfunction
28
29   function void simple_agent::connect_phase(uvm_phase phase);
30       super.connect_phase(phase);
31       collector.analysis_port.connect(monitor.analysis_export);
32       if( is_active == UVM_ACTIVE ) begin
33           driver.seq_item_port.connect(sequencer.seq_item_export);
34       end
35   endfunction
```

Environment

- Environmentクラスは検証コンポーネントのトップ・レベル・コンテイナーです。
- Environmentは、幾つかのagents、及び、Environment専用のモニターを含みます。その他、Environmentトポロジーの柔軟性を高める為にコンフィギュレーション・パラメータも保有します。
- Environmentクラスは、メソドロジー・クラスuvm_env、又は、そのサブクラスを使用して定義します。

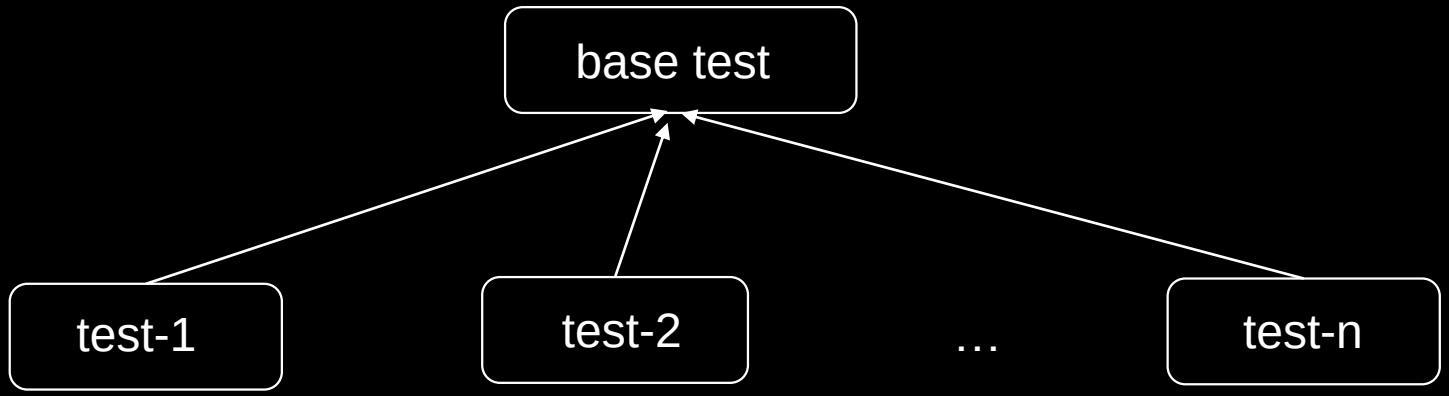


Environmentの例([2])

```
 1   class ahb_env extends uvm_env;
 2   int                     num_masters;
 3   ahb_master_agent        masters[];
 4
 5   `uvm_component_utils_begin(ahb_env)
 6           `uvm_field_int(num_masters, UVM_ALL_ON)
 7   `uvm_component_utils_end
 8
 9   function new(string name, uvm_component parent);
10           super.new(name, parent);
11   endfunction : new
12
13   virtual function void build_phase(phase);
14   string inst_name;
15           super.build_phase(phase);
16           if(num_masters ==0))
17                   `uvm_fatal("NONUM",{"'num_masters' must be set";
18           masters = new[num_masters];
19           for(int i = 0; i < num_masters; i++) begin
20                   $sformat(inst_name, "masters[%0d]", i);
21                   masters[i] = ahb_master_agent::type_id::create(inst_name,this);
22           end
23           // Build slaves and other components.
24   endfunction
25
26   endclass
```

テスト

- 検証を実行する為にはシナリオを使用します。シナリオに対応するテスト環境を構築するのがテストの役目です。
- 通常、多くのテスト・ケースが存在する為、それらに共通する処理が発生します。共通処理をベース・クラスとして定義する方法が一般的です。
- ベース・テストはuvm_testを使用して定義します。



ベース・テスト

- テストを実行する為に必要なコンポーネント階層を定義するのが主たる役割です。
- 通常、トップ・レベルのエンバイロンメントのインスタンスを作成します。

ベース・テストの例（[2]）

```
 1   class ubus_example_base_test extends uvm_test;
 2   `uvm_component_utils(ubus_example_base_test)
 3   ubus_example_env ubus_example_env0;
 4
 5   // The test's constructor
 6   function new (string name = "ubus_example_base_test",
 7           uvm_component parent = null);
 8   super.new(name, parent);
 9   endfunction
10
11   // Update this component's properties and create the ubus_example_tb component.
12   virtual function build_phase(); // Create the top-level environment.
13           super.build_phase(phase);
14           ubus_example_env0 =
15                   ubus_example_tb::type_id::create("ubus_example_env0", this);
16   endfunction
17
18   endclass
```

テスト・ケース（test-i）

- テスト・ケースに対応するシナリオを設定します。
- コンポーネント階層に従って、使用するシーケンスをdefault_sequenceとしてシーケンサーのrun_phaseに割り当てます。
- default_sequenceが設定されていると、test-iコンポーネントを実行する事が出来ます。

テスト・ケースの例（[2]）

```
1   class test_read_modify_write extends ubus_example_base_test;
2   `uvm_component_utils(test_read_modify_write)
3
4   // The test's constructor
5   function new (string name = "test_read_modify_write",
6                             uvm_component parent = null);
7           super.new(name, parent);
8   endfunction
9
10  // Register configurations to control which
11  // sequence is executed by the sequencers.
12  virtual function void build_phase(uvm_phase phase);
13          // Substitute the default sequence.
14          uvm_config_db#(uvm_object_wrapper)::set(this,
15                  "ubus0.masters[0].sequencer.main_phase",
16                  "default_sequence", read_modify_write_seq::type_id::get());
17          uvm_config_db#(uvm_object_wrapper)::set(this,
18                  "ubus0.slaves[0].sequencer.main_phase",
19                  "default_sequence",slave_memory_seq::type_id::get());
20          super.build_phase(phase);
21  endfunction
22
23  endclass
```

default_sequeneの設定

トップ・レベル・モジュール

- トップ・レベルのソース・コードは以下の様になります。但し、ファイルpkg.svに検証コンポーネントが含まれているとします。

```
 1   `include "uvm_pkg.sv"
 2   `include "uvm_macros.svh"
 3   `include "simple_if.sv"
 4   `include "pkg.sv"
 5
 6   module top;
 7   import  uvm_pkg::*;
 8   import  pkg::*;
 9   bit             clk;
10
11   simple_if sif(clk);
12   dut DUT(.a(sif.a),.b(sif.b),.op_code(sif.op_code),.q(sif.q));
13
14           initial begin
15                   uvm_config_db #(virtual simple_if)::set(null,"*test0*","vif",sif);
16                   run_test();
17           end
18
19           initial forever #10 clk = ~clk;
20
21   endmodule
```


第七章
結論

まとめ

- 本資料はUVMの意義を明確にし、UVMを構成する基本要素を概説しました。
- UVMは検証手法のトレンドである階層的テストベンチを記述する手法（layered testbench）を実現した優れた検証ライブラリーです。
- UVMに関するより高度な知識を習得する為には、UVMユーザ・ガイド（[2]）をお読み下さい。本資料の知識を持てば、ユーザ・ガイドを読破するのは難しい事では無いと思います。
- TLM, ドライバー、シーケンサー、シーケンス、モニター、コレクターの概念と機能を正しく理解する事が必要です。その知識があれば、文献[2]を読み、実践にUVMを適用する事ができます。もし、知識が不十分であると感じる場合には、文献[5,6]を薦めます。

演習問題－1

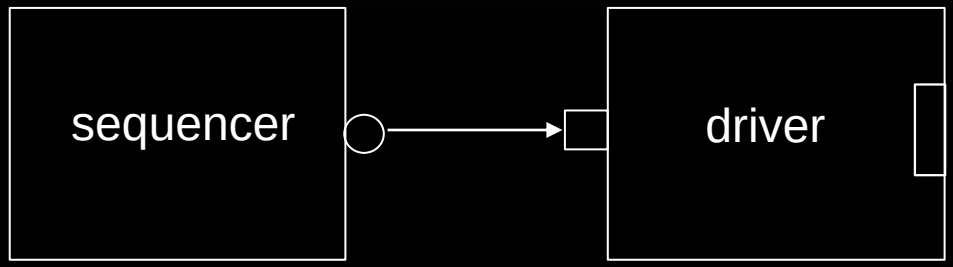
- 次ページに示す、binary_counterを検証する為の環境をUVMで記述して下さい。但し、シーケンスを使用してテストを実行する様にして下さい。
  - 最初に、カウンターをリセットする様にシーケンスを設計して下さい。
  - シーケンスを二つ定義して、コマンド・ラインからテスト・ケースを選択する事が出来る様にして下さい。
  - インターフェース、トランザクション、ドライバー、シーケンサー、シーケンス、コレクター、モニター、エージェント、エンバイロンメント、ベース・テスト、テスト・ケース、トップ・モジュール等の定義が必要です。
  - DUTはシーケンシャル回路なので、DUTをドライブするタイミングとDUTのレスポンスを収集するタイミングに注意して下さい。SystemVerilogのactive region、inactive region、NBA regionの概念を正しく理解していないと、この問題を解く事は出来ません。
- 文献[6]を読むと、この問題を簡単に解ける様になります。

binary_counter

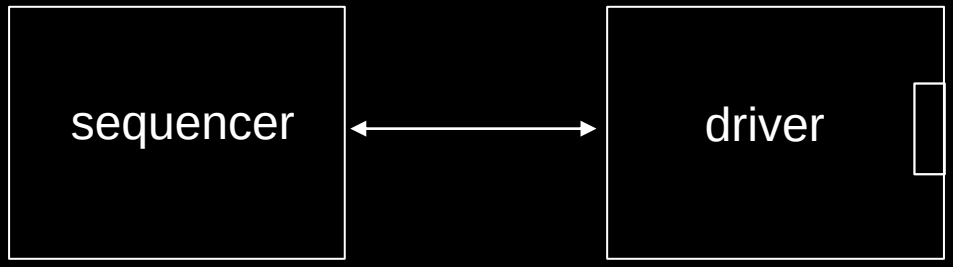
```
 1   module binary_counter (input clk,up_down,preset_clear,load_data,
 2                          input [7:0] data_in,output [7:0] q,qbar);
 3   logic [7:0]            counter;
 4
 5   always @(posedge clk)
 6           if( preset_clear )
 7                   counter <= 0;
 8           else if( ~load_data )
 9                   counter <= data_in;
10           else if( up_down )
11                   counter <= counter + 1;
12           else
13                   counter <= counter - 1;
14
15   assign q = counter;
16   assign qbar = ~counter;
17
18   endmodule
```

演習問題－2

- UVMではドライバーとシーケンサーの関係を下図の様に表現します。



- 然し、実際には次の図に示す様にドライバーとシーケンサーは交信します。
それぞれの矢印に対応する動作を説明して下さい。



演習問題－３

- TLMとRTLの差異を簡潔に説明して下さい。

演習問題－4

- スコアボードは、検証環境のどの階層に配置するのが適切であるかを検討して下さい。例えば、エージェントにスコアボードを配置する事はしません。では、どの検証コンポーネント内にスコアボードのインスタンスを作れば良いかを考えて下さい。
- スコアボードのインスタンスを持つ検証コンポーネントを決定すると、スコアボードとモニターを接続しなければなりません。どのように接続するかを考えて下さい。

演習問題－５

- カバレッジ計算を行う場合、どの検証コンポーネントで行うのが最も適切であるかを考えて下さい。明確な理由が必要です。

演習問題－6

- 簡単なトランザクションを定義して下さい。ただし、幾つかのフィールドはランダム変数と宣言して下さい。
- シーケンスを定義し、上記のトランザクション生成時に制約を与える実装例を示して下さい。

参考文献

文献[1]は、誰もが一度は目を通す言語仕様書です。SystemVerilog は、数年に一度改訂されます。その前に一読する事を薦めます。

文献[2]はUVMのユーザ・ガイドです。UVMに関する使用法を概説しています。比較的読み易い構成を取っているので一読を薦めます。

文献[3]はUVMを実践に適用する際に必要な知識を記述している優れた書物です。

文献[4]は、SystemVerilog へ の初心者向け の入門書です。検証には深入りしていないので、設計技術者にも薦めます。UVMに関する概説が含まれているので、初心者には参考になります。

文献[5]は、SystemVerilogの基礎知識の復習からファンクショナル・カバレッジ、アサーション、UVM適用までの知識を詳しく解説した参考書です。この文献一冊でSystemVerilogに関する必要な全ての知識を習得する事が出来ます。SystemVerilogの基礎知識を持つ検証技術者にお薦めします。

文献[6]は、検証作業にUVMを適用する際に必要な実践知識を詳しく解説した参考書の決定版です。この資料を読めば、UVMを使用する為の知識を完全にマスターする事が出来ます。文献[2]、及び、[3]と異なり、簡単なDUTを用いて検証環境を構築しているので、検証環境構築技術を理解し易い特長があります。例示した検証環境構築法をそのまま実践に適用する事が出来る利点も備えています。SystemVerilogを使用している検証技術者にお薦めします。

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

[4] 篠塚一也, SystemVerilog入門, 共立出版 2020.

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

[6] 篠塚一也, 実践UVM入門, 森北出版 2021.

