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: 1.1,2019.05.19

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。

変更履歴

日付	Revision	変更点
2019.05.12	1.0	初版。
2019.05.19	1.1	第二章を追加。

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

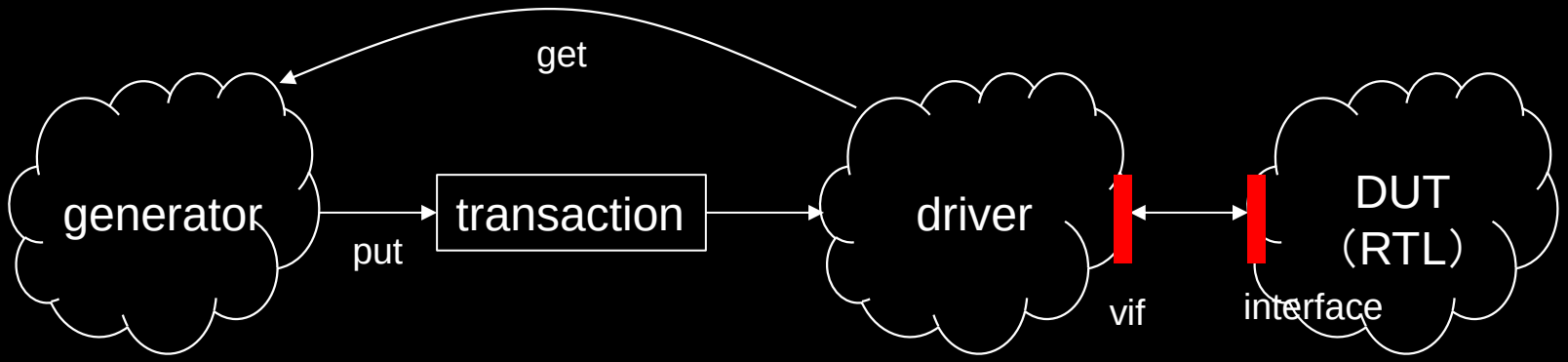
本章ではUVMとは何か、又、その目的等を解説します。そして、UVMがどの様に構成されているかを概説します。これらの知識は後章への準備となります。

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
```

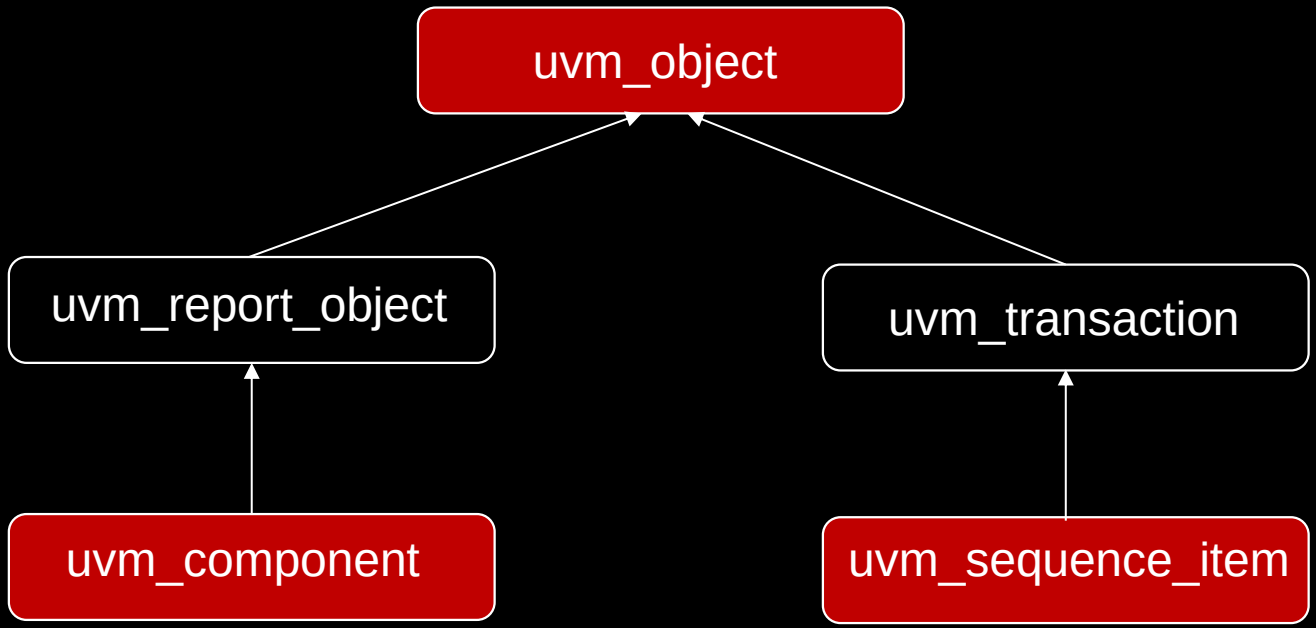
メソドロジー・クラス

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クラスです。
uvm_scoreboard	scoreboardの基本機能を提供するクラスです。

メソドロジー・クラスは、uvm_componentのサブクラスです。

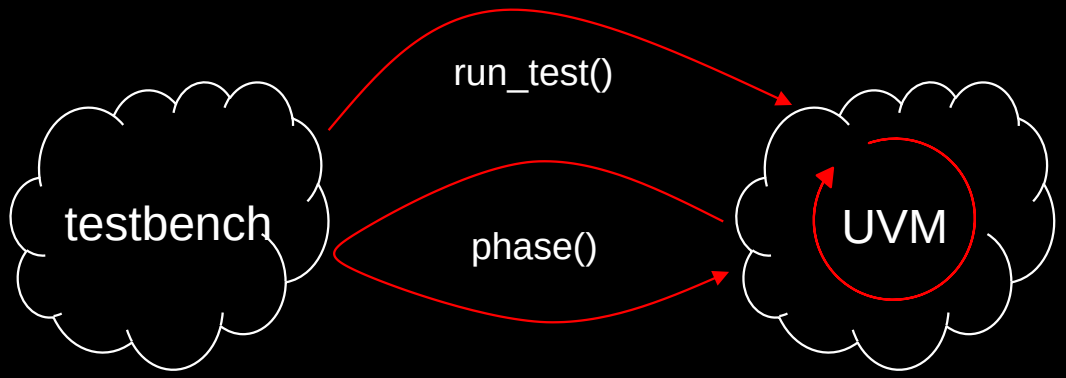
uvm_objectクラス

- 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
```

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

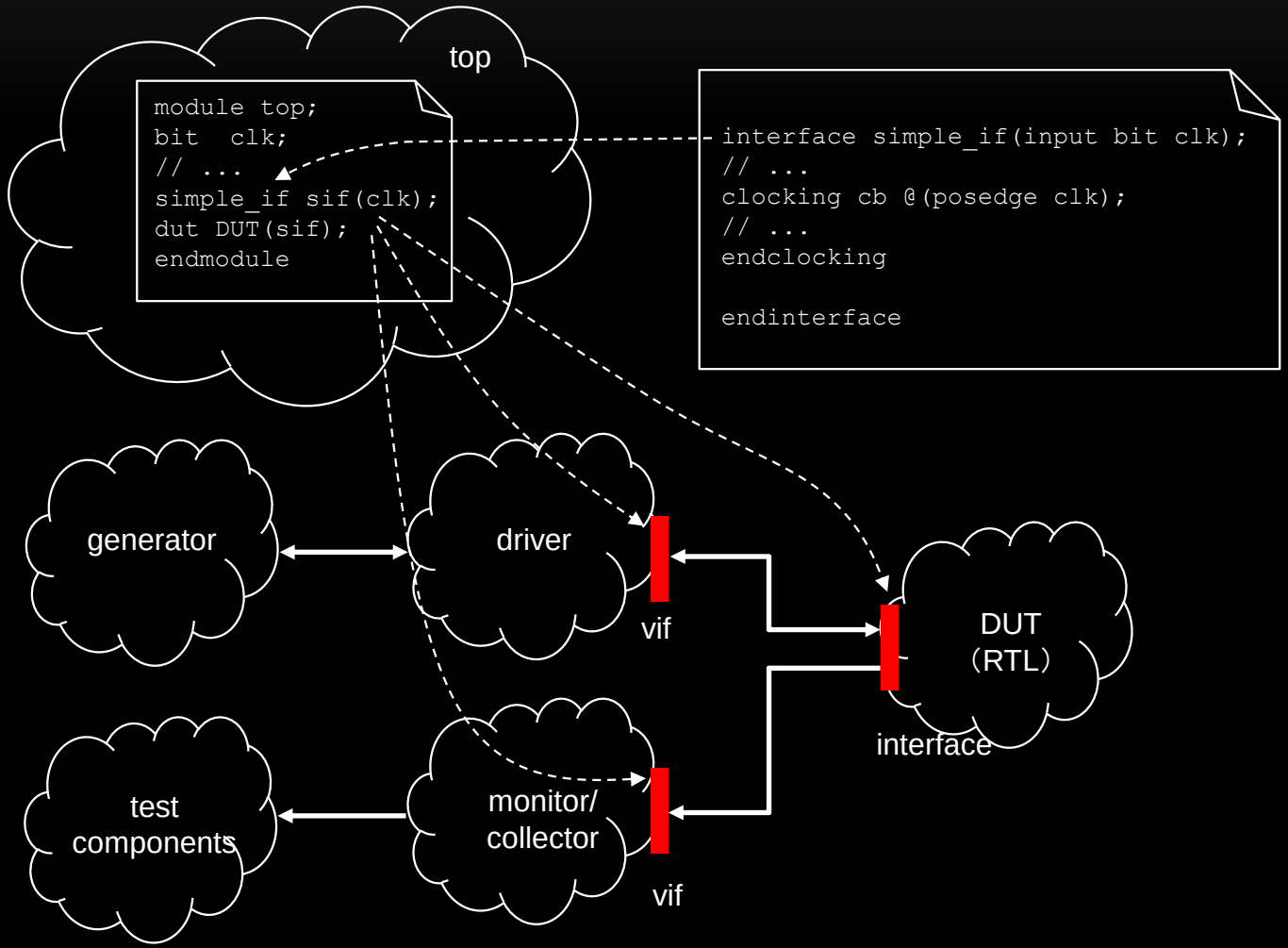
```
 1    `include "uvm_pkg.sv"
 2    `include "uvm_macros.svh"
 3    `include "simple_if.sv"
 4    `include "my_pkg.sv"
 5
 6    module top;
 7    import uvm_pkg::*;
 8    import my_pkg::*;
 9    bit     clk;
10
11    simple_if sif(clk);
12    dut DUT(sif);
13
14    initial begin
15            setup_config(); // 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

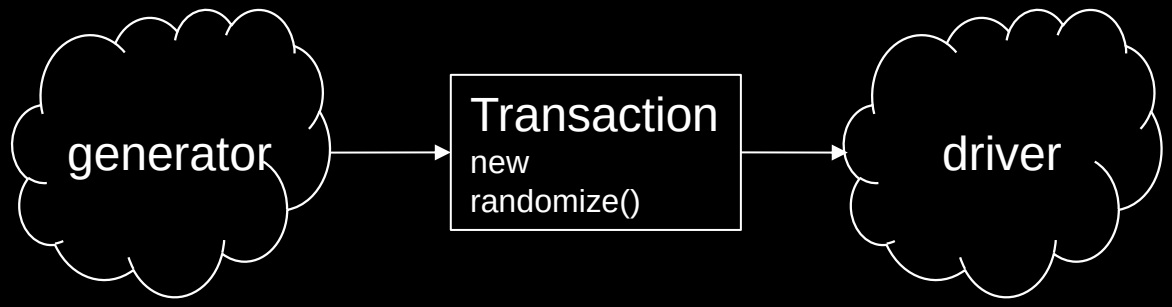


第二章
トランザクション

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

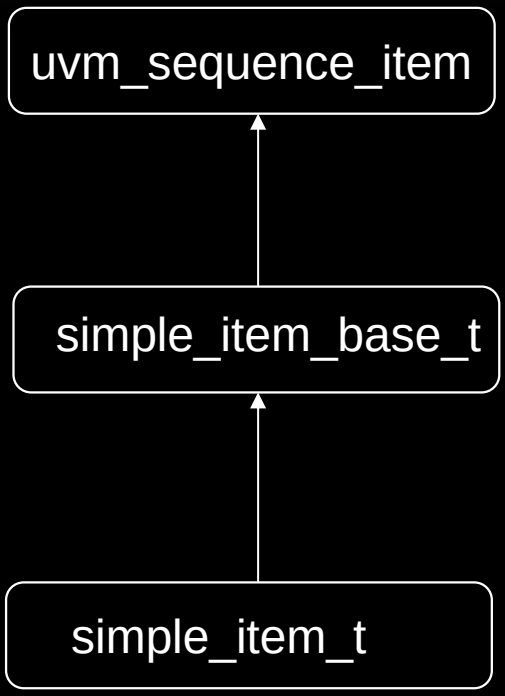
- トランザクションは様々なテスト・ケースに応じて柔軟な対応が求められます。その様な多様性に対して唯一つのトランザクション定義ではコードが複雑になり、維持する為のコストが増大します。
- 従って、様々なテスト・ケースに対応し得るトランザクション定義法を実装しなければなりません。
- クラス・インヘリタンスがその解答を提供します。
  - トランザクションのベース・クラスには、仕様から導いた項目だけを定義する。
  - テスト・ケースに依存する条件をクラス・インヘリタンスにより得られたサブクラスに追加して行く。

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

To be described.

参考文献

[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] UVM入門, アートグラフィックス, May 12, 2019.

