Artgraphics

デザインの検証と **UVM**

Document Revision: 1.0 2019.07.28

デザインの検証と UVM

© 2019 アートグラフィックス
〒124-0012 東京都葛飾区立石 8-14-1
www.artgraphics.co.jp

An Application of UVM to Verification

© 2019 Artgraphics. All rights reserved.
8-14-1, Tateishi, Katsushika-ku, Tokyo, 124-0012 Japan
www.artgraphics.co.jp

目次

1 概要 1

1.1 再利用性 1
1.2 UVM の適用 2

2 UVM 支援機能 3

2.1 UVM クラス・ウィザード 3
2.2 デザイン・スタイル・チェック 3
2.3 コード・スニペット 4
2.4 UVM クラスの情報検索 4

3 参考文献 6

1 概要

1.1 再利用性

UVM と言えば、大規模なデザイン・プロジェクトに適用されるべき検証メソドロジーと理解しているのが一般的であると考えられています。然し、どの様な検証にしても再利用可能性の概念が存在し得る検証作業であれば、UVM を適用する事が出来ます。寧ろ、UVM を適用すべきです。

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。階層のレイヤー（layer）には以下の様な種類があります（[3]）。

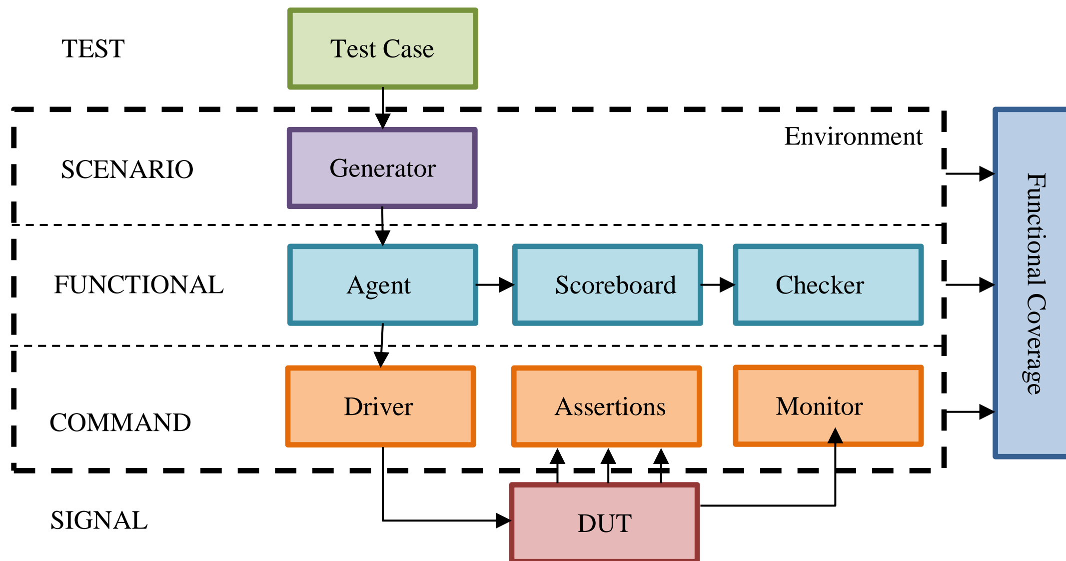


図 1-1　テストベンチとレイヤーの関係（[3]）

TestCase には多くの種類が存在しますが、各テストに使用する検証コンポーネントを共有するのが手法の主たる狙いです。即ち、上図の Environment に含まれる検証要素を再利用可能な様に開発します。テスト・ケース（シナリオ）に対応して、それぞれの検証コンポーネントが適切な検証動作を遂行します。UVM はこの精神を受け継ぎ具現化した SystemVerilog クラス・ライブラリーです。例えば、Generator はシーケンサー（uvm_sequencer のサブクラス）です。

同様な作業を何度も繰り返す事を避けて生産性を向上するのが手法のゴールであるとすれば、小規模な検証作業でも UVM の効果を期待する事が出来ます。従来のモジュール（modules）をベースにした検証手法では柔軟性は全くありません。例えば、モジュール・インスタンスは static である為、実行時に構成を変更する事は出来ません。例えば、master と slave のコンポーネント数を変更する事は出来ません。この為、従来の手法ではテスト・ケース毎に異なる検証環境が存在する事になります。自ずから、検証し得るテスト・ケースが限定されてしまいます。

モジュールの代わりに SystemVerilog クラスを使用する事により検証環境を dynamic に変更する事が出来ます。dynamic 性が、即ち、検証環境の共有（再利用）です。クラスのインスタンスをファクトリでダイナミックに作り出す事が出来る為、テスト・ケースに即した構成を実現する事が出来ます。

Copyright 2019 © Artgraphics. All rights reserved.

1.2 UVM の適用

UVM を適用するとなると、先ず、以下の様な懸念が出て来ます。但し、UVM の知識習得を除外してあります。デザイナーにしても検証技術者にしても UVM に関する知識を持つ事は必須です。

① 検証環境を整える事が多岐に渡り、準備に手間がかかりそう。
② UVM では、タイプする負荷が多い。
③ 同じ様な命令を繰り返し入力しなければならない。
④ マクロは複雑で分かり難い。
⑤ フィールド・マクロは分かり難いと同時に、些細な間違いが多くのコンパイル・エラーを誘発する。
⑥ コンパイルに時間がかかる。

モジュール・ベースの検証環境に比べて、UVM の環境は複雑である為、コンパイル時間が増加するのは止むを得ない事実です。然し、その他の懸念は EDA ツールを使用する事により、軽減されます。

以下では、SystemVerilog Checker の UVM 支援機能を紹介します。

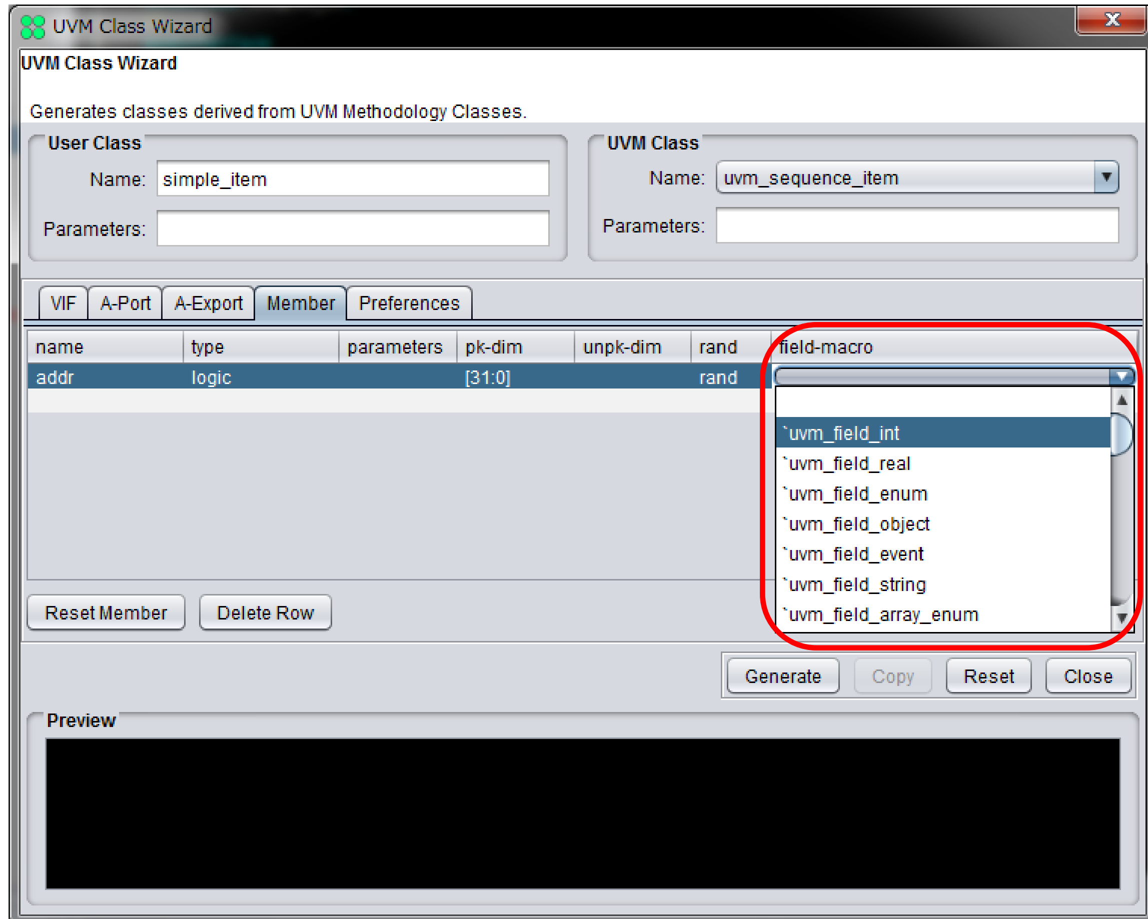
Copyright 2019 © Artgraphics. All rights reserved.

2　UVM 支援機能

EDA ツールを使用しなくても自身で済ませる事は可能ですが、ツールよりも多くの時間を費やす事になります。

2.1　UVM クラス・ウィザード

UVM のコーディングに慣れるまでは、この種のツールは重宝します。特に、フィールド・マクロの指定に関しては大きな支援となります。



2.2　デザイン・スタイル・チェック

この種のツールを使用するとコンパイル・エラー、及び、実行時のエラーを未然に防ぐ事が出来ます。例えば、以下の様にドライバーを定義した後、デザイン・スタイル・チェックを適用すると、virtual インターフェースが未定義である事が判明します。

```
class driver_no_vif extends uvm_driver #(packet);

`uvm_component_utils(driver_no_vif)

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

endclass
```

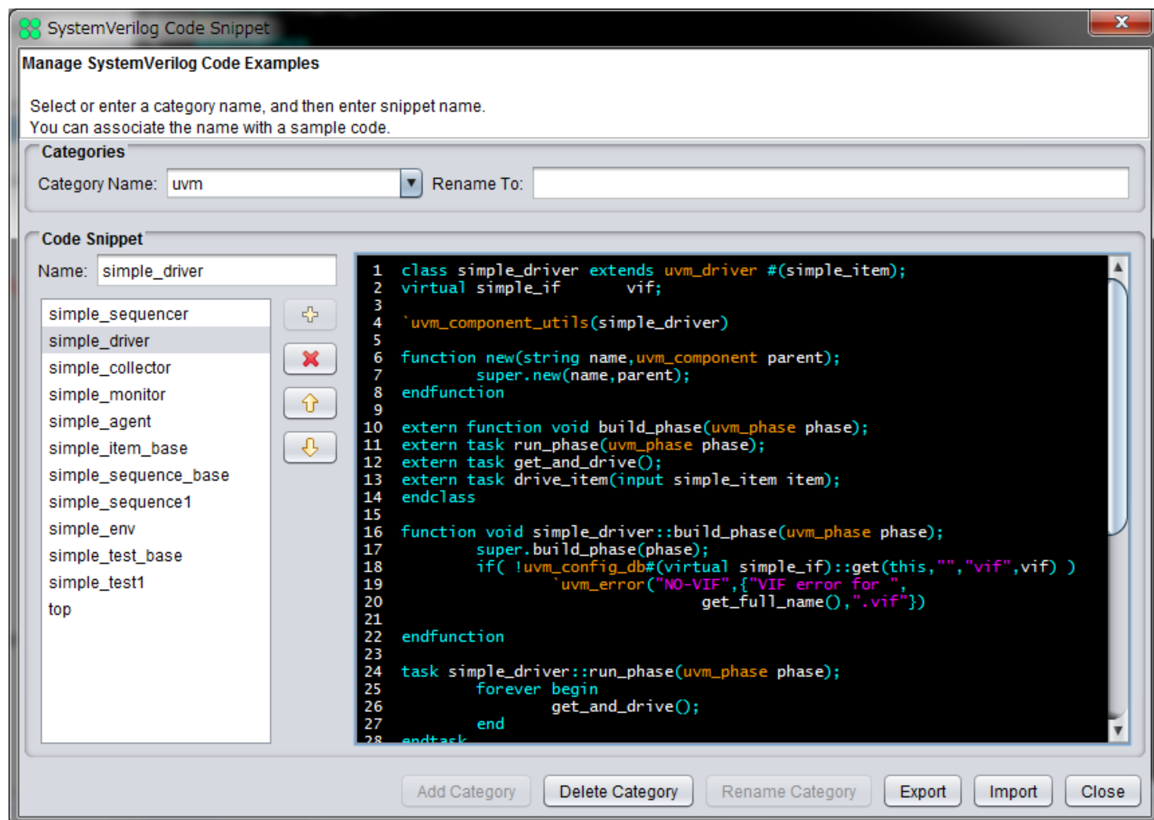

Copyright 2019 © Artgraphics. All rights reserved.

例えば、以下の様な警告が発行されます。

```
#-W dsc_uvm_driver.sv (L18): virtual interface not declared in driver: driver_no_vif
#-W dsc_uvm_driver.sv (L18): driver must have connect_phase() defined: driver_no_vif
#-W dsc_uvm_driver.sv (L18): driver must have run_phase() defined: driver_no_vif
```

2.3 コード・スニペット

検証規模の大小に依らず、検証コードのテンプレートを準備して置く事は生成性向上に必要な手段です。通常、Environment を構成する検証コンポーネントのテンプレートを保存しておくと備忘録にもなります。



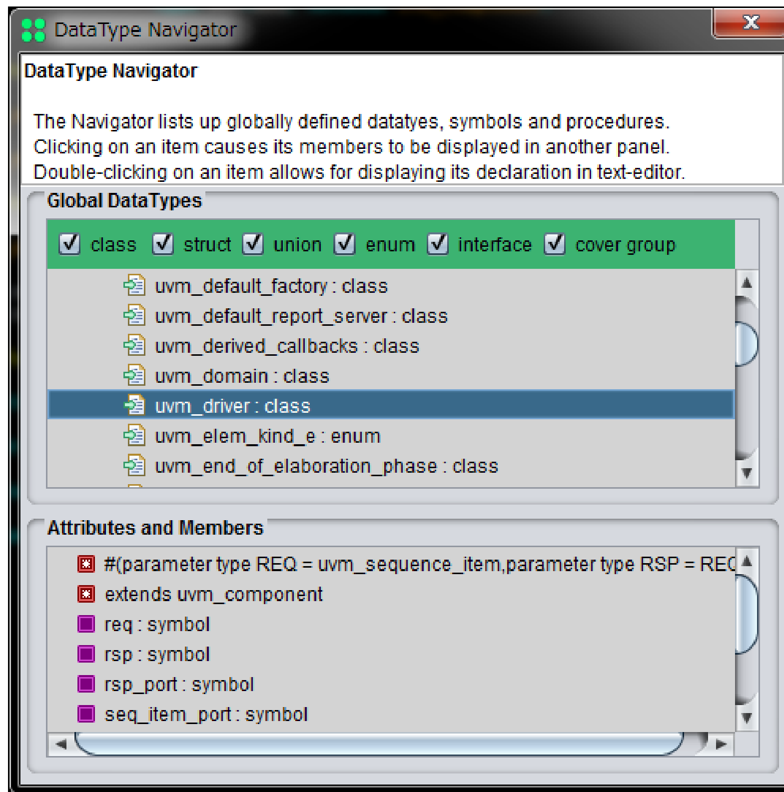
殆どの場合、スニペットに多少の変更を加えるだけで済みます。逆に、その様にテンプレートを作成しておきます。特に、トップ・モジュール、及び、シーケンサーは、殆どそのままで使用する事が出来ます。

2.4 UVM クラスの情報検索

時として、UVM のメソドロジー・クラス、シーケンス、トランザクション、TLM 等に関する詳しい情報が必要になります。UVM ユーザ・ガイド、及び、Reference マニュアルが存在しますが、使用範囲が限定されています。その様な場合、UVM ソース・コードを効率良く検索する為の機能が必要になります。ナビゲータはその要望に応えます。

ナビゲータ内のクラス名をクリックするだけで、対応するファイルをエディタに表示する事が出来ます。また、クラス内に定義されている、プロパティ、及び、メソッド等もナビゲータに表示されます。

Copyright 2019 © Artgraphics. All rights reserved.



Copyright 2019 © Artgraphics. All rights reserved.

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

Copyright 2019 © Artgraphics. All rights reserved.