Design Solution Forum 2025 講演資料

Artgraphics

UVMの基礎知識と実装技術

篠塚一也

アートグラフィックス

2025.10.07

www.artgraphics.co.jp

プロローグ

UVMはSystemVerilogで書かれているので、SystemVerilogに依存するのは明らかですが、SystemVerilogがUVMと共に進化しているのも確かです。先ずは、SystemVerilogの変遷とUVMとのかかわりあいから解説を始めます。

SystemVerilogとUVM

- UVMはSystemVerilogの到来によりはじめて可能となった検証環境構築のためのパッケージです。
- SystemVerilogの最新仕様は**IEEE Std 1800-2023**で2024年2月に公開されました。
- SystemVerilogにクラスの概念が導入された恩恵の下で汎用性の高いUVMクラス機能を実現できるようになりました。また、SystemVerilogが提供する機能に疑問を持てば、UVMのソースコードから理由を見つける事ができます。例えば、virtualインターフェースやvirtualメソッドの使用法が良い例です。あるいは、コンストラクタの定義法も良い例です。
- このように、UVMとSystemVerilogは切っても切れない関係にあります。この講演の第３部では、SystemVerilogの最新仕様がUVMのコード保守作業を確実にする機能を提供している事実を紹介します。
- また、第1部と第2部ではSystemVerilogの機能がUVMのアーキテクチャに的確に活用されている事実を体験できます。この活用法を通して、UVMだけでなくSystemVerilogの知識・技術を向上させる機会になります。
- なお、本講演を理解するにはSystemVerilogの基礎知識が必要です。

SystemVerilogの変遷とUVM

- SystemVerilog規格は、以下のような変遷を辿って来ています。

このリリースのSystemVerilogはIEEE Std 1364-2005 (Verilog HDL)の拡張言語であると宣言しているだけであり、SystemVerilogがVerilog HDLを基礎言語として包含しているとは明記していないようです。

このリリースはSystemVerilogとして機能が充実していますが、望ましくない機能も追加されています。例えば、現在では、非推奨となっているオペレータオーバーローディングの機能が含まれています。このリリースのLRMを読んだ方は、IEEE Std 1800-2017以降のリリースのLRMを読む必要があります。この時期にUVMがほぼ完成しています。

前仕様の解説上の改善と僅かですが新機能が含まれています。ただし、非常に難解な英文で書かれています。

SystemVerilogはIEEE Std 1364-2005 (Verilog HDL)とIEEE Std 1800-2005を統合した言語として定義されています。この頃にはUVMの開発は本格的になっています。

非推奨機能等を除外して非常に良く書かれたLRMです。このリリースに書かれた知識を持てば、世界で通用する技術者と言えます。UVMはIEEE Std 1800.2-2017規格となり、この時期に多くの企業がUVMを導入し始めました。

本講演の概要

第1部　UVM概要

1. UVMとは何か？
2. virtualインターフェース
3. TLM
4. トランザクション
5. ドライバー、シーケンサー、コレクター、モニター
6. エージェント
7. エンバイロンメント
8. テストとテストベンチ
9. シーケンス
10. UVMテストベンチの構造
11. ファクトリ
12. UVMシミュレーション制御
13. シミュレーションフェーズ
14. UVMの使用手順

第2部　UVMによる検証環境構築

1. UVMを適用する手順
2. トランザクションの定義法
3. ドライバーの定義法
4. シーケンサーの定義法
5. コレクターの定義法
6. モニターの定義法
7. エージェントの定義法
8. テストの定義法
9. raise_objection()とdrop_objection()
10. 検証環境開発において推奨される手法

第３部　IEEE Std 1800-2023とUVM

第1部
UVM概要

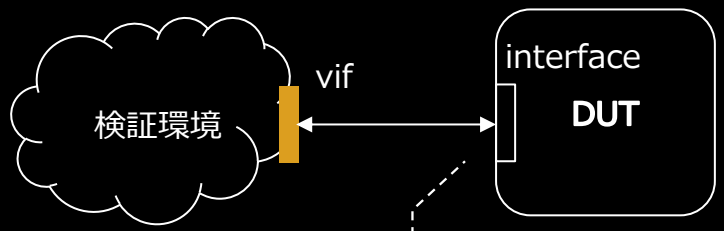
第1部ではUVMとは何か、また、その目的等を解説します。そして、UVMがどのように構成されているかを概説し、UVMにより検証環境を構築するための基礎知識を養います。

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ためのSystemVerilogのクラスライブラリーです。
- UVMを活用する事により再利用可能な検証環境を構築する事ができます。
- UVMは、**IEEE Std 1800.2-2020** 標準規格として知られています。
- 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できます。したがって、UVMはEDAツール間での互換性が保証されています。

virtualインターフェース

- DUTは検証環境から独立して存在するため、検証環境に接続されていません。
- DUTをドライブしたり、DUTからのレスポンスをサンプリングするためには、検証環境とDUTを結び付けなければなりません。
- DUTと検証環境を結び付けるために使用される手段が **virtualインターフェース** です。virtualインターフェースはインターフェースのインスタンスへのポインターです。
- インターフェースのインスタンスはDUTに接続されているので、virtualインターフェースにより検証環境はDUTと接続ができるようになります。
- virtualインターフェースはSystemVerilogの機能です。



vif：virtualインターフェース

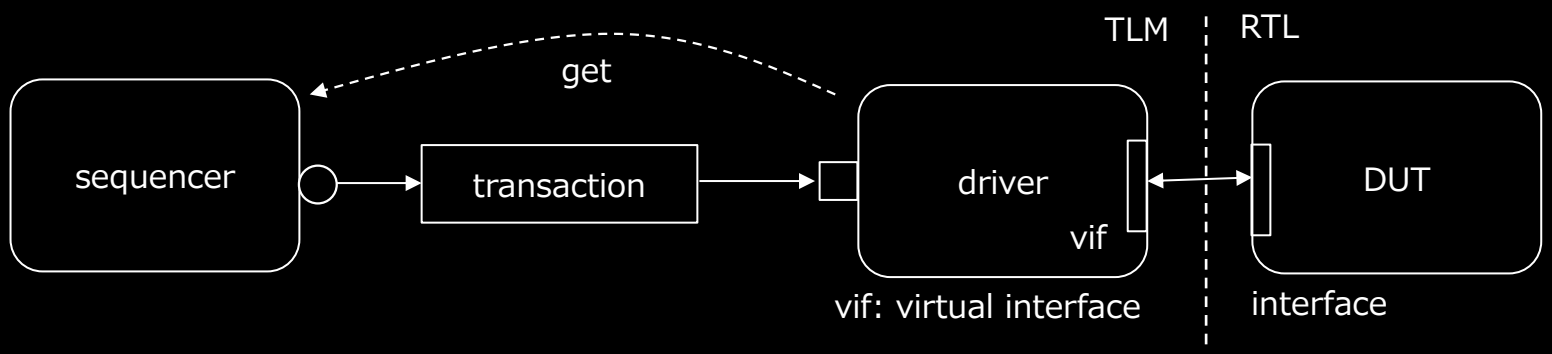
DUTがインターフェースを使用していなくてもvirtualインターフェースを使用できます。例えば、DUTがVerilog HDLで記述されていてもUVMを使用できます。

TLM (Transaction Level Modeling)

- UVMでは **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システムの動作を表現します。
- RTLではデバイス間はネットで接続されているため、接続されているネットの信号値が変化する事で通信が行われます。一方、TLMではネットが存在しないため**タスクやファンクションの呼び出し**を使用してトランザクションの授受を行います。
- トランザクションの授受には**TLMポート**が必要になります。TLMポートが検証コンポーネントに配置され、その検証コンポーネントのインスタンスを持つ親の検証コンポーネントがTLMポート同士を接続します。
- 検証環境がDUTと交信するためには、シグナルレベル（RTL）とTLM間の変換を行わなければなりません。検証環境でTLMをRTLに変換をする役目をするのが**ドライバー**です。DUTが出力する信号レベルの値をTLMに変換する役目を持つのが**コレクター**です。
- 文献[5]にはTLMに関する詳しい解説が含まれています。

TLM-portとTLM-export

- UVMコンポーネント間の通信には**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ための動作（putやget）を引き起こし、TLM-exportはトランザクションを処理するために必要な実処理（putやgetの実装内容）をvirtualメソッドとして定義します。
- TLM-portとTLM-exportは**一対一**の関係を確立しますが、モニターのように**一対N**の関係を持つ通信には適していません。その場合には、UVMでは**analysis-port**を使用します。一つのanalysis-portに対して複数個の**analysis-exports**を接続できます。
- UVMではTLM-portは□、TLM-exportとanalysis-exportは○、analysis-portは◇で図示するルールがあります。

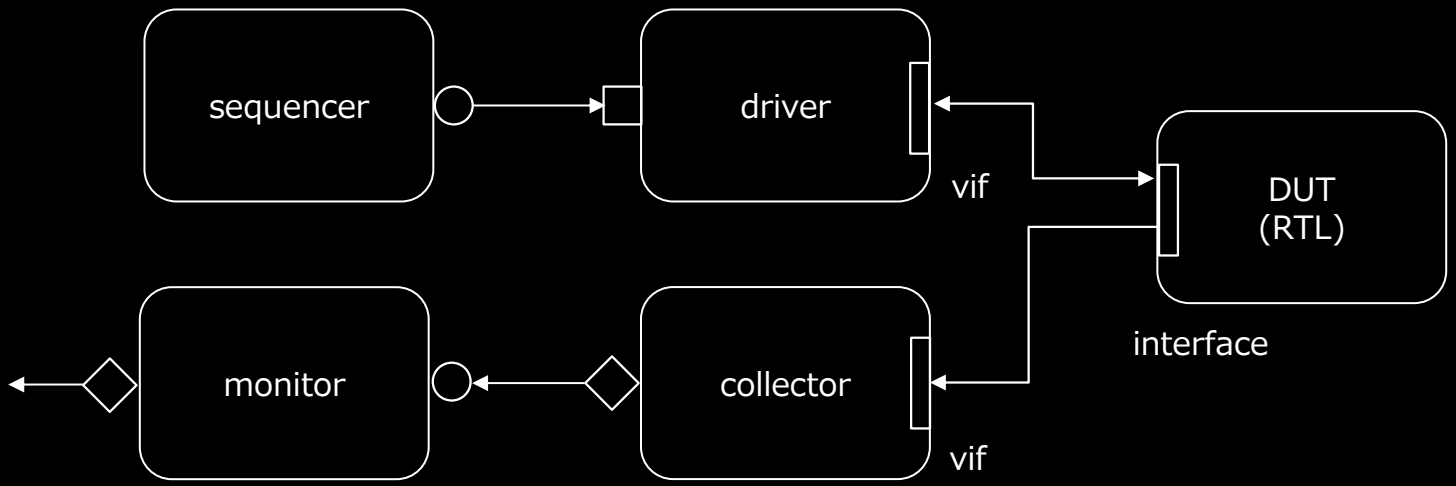


トランザクション

- 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ために必要な情報を意味します。
- **ドライバー**が**シーケンサー**にトランザクションを要求すると、シーケンサーは制約を満たすトランザクションを作成してドライバーに引き渡します。
- ドライバー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、virtual インターフェース（vif）が使用されます。

ドライバー、シーケンサー、コレクター、モニター

- **シーケンサー**は、ドライバーの要求に応じてトランザクションを作ります。**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をドライブする役目を持ちます。
- DUT側からのレスポンスをサンプリングする役目を持つUVMコンポーネントは、一般的には、**コレクター**と呼ばれます。
- **モニター**はコレクターからトランザクションを受け取り、簡単な検証作業を行い、詳細な解析を行うために、他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。他の検証コンポーネントとしては**スコアボード**等があります。



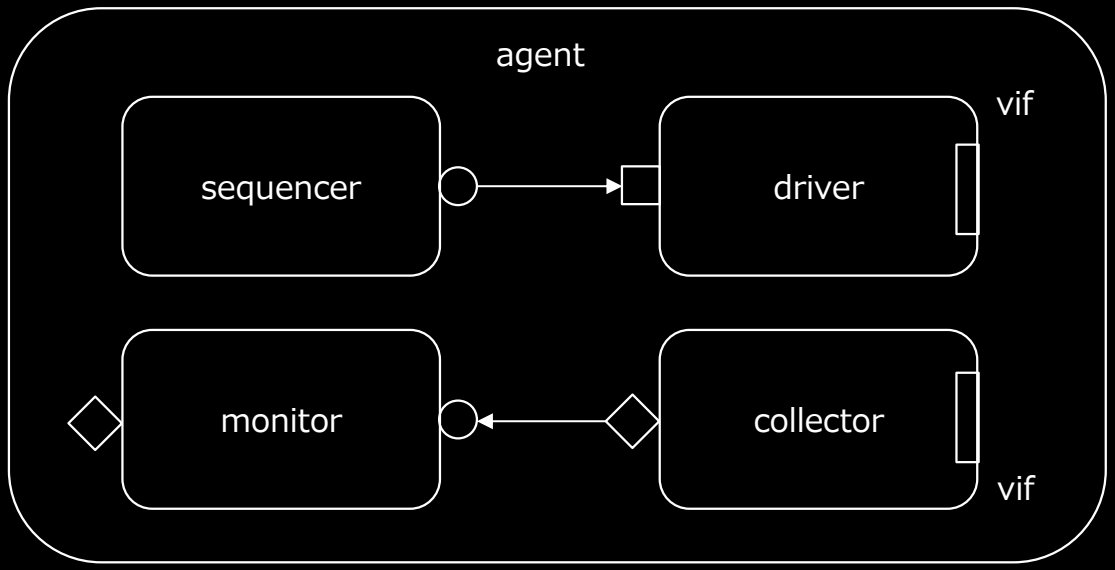
エージェント

- **エージェント**は、シーケンサー、ドライバー、モニター、コレクターから構成される階層的な検証コンポーネントですが、コンフィギュレーションの柔軟性を高めるために、**is_active**プロパーティを保有しています。

is_active	機能
UVM_ACTIVE	このモードでは、エージェント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、およびカバレッジをするためにモニター（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、およ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。

エージェントの構成

- ドライバー、シーケンサー、コレクター、モニターは一組として使用されるため、UVMではこれらのコンポーネントを含むためのコンテイナーを提供しています。それが、**uvm_agent**です。
- 下図は、is_activeがUVM_ACTIVEに設定されている場合のエージェントの構成を示しています。



,

エンバイロンメント

- エージェントは、最も簡単な階層的検証コンポーネントです。より複雑な検証タスクを遂行するためには、幾つかのエージェントを組み合わせて検証コンポネントを構築して行きます。このような階層的なコンポーネントを**エンバイロンメント**と呼びます。それに対応するUVMクラスは**uvm_env**です。
- 一般的には、エンバイロンメントには他のエンバイロンメント、モニター、コンフィギュレーションパラメータ等を含みます。

テストとテストベンチ

- UVMによる検証環境の最上位には**テスト**と呼ばれる検証コンポーネントが作られます。
- テストは、テストケースに対応して必要な検証コンポーネントのインスタンスを配置して構築されます。
- **テストベンチ**は、DUTとインターフェースのインスタンスを配置して、検証を実行する環境です。

シーケンス

- シーケンサーは、ドライバーの要求に応じてトランザクションを作りますが、実際には**シーケンス**がトランザクションを作ります。
- トランザクション、およびシナリオを作るためには、下表に示すクラスを使用します。

UVMクラス	機能および目的
uvm_sequence_item	トランザクションを記述するため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では、一般的には、シーケンスがトランザクションを生成します。
uvm_sequence	トランザクションを生成するために必要な手順を提供するクラスでシーケンスと呼ばれます。手順の中には他のシーケンスへの引用を含む事ができるので、階層的にシーケンスを構築する事ができます。シーケンスもデータオブジェクトです。 UVMでは、シーケンサーがシーケンスを実行してトランザクションを生成します。

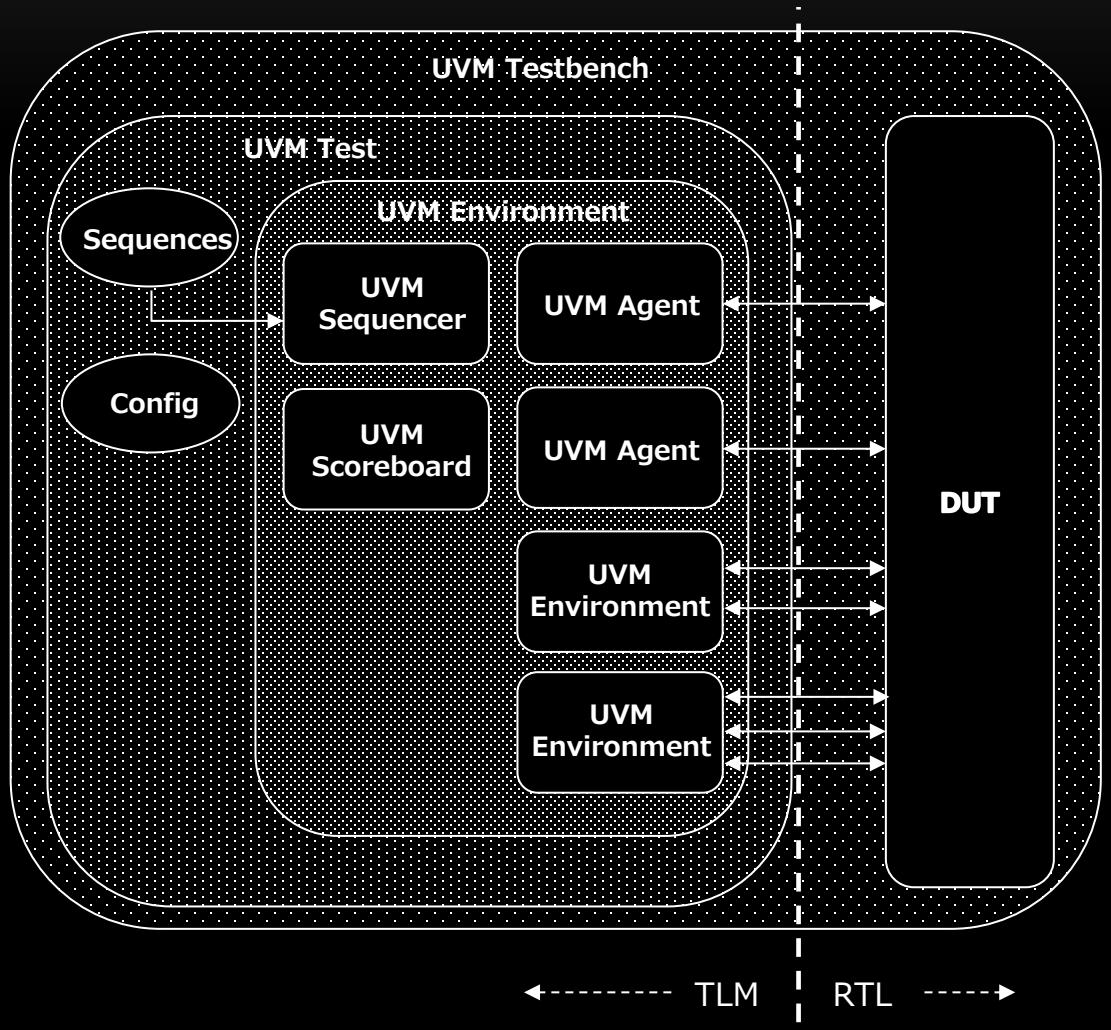
UVMテストベンチの構造（[3]）

UVM Testbenchはトップレベルのモジュールで SystemVerilogのmoduleで記述されます。

UVM Testのインスタンスはダイナミックに配置されます。

シーケンスはテストケースにしたがいトランザクションを生成して行く仕組みです。

シーケンサーは与えられたシーケンスを基にしてトランザクションを生成して行きます。



Environmentを再利用可能なように開発するのが特徴です。

Agentはドライバー、シーケンサー、コレクター、モニターから構成されます。

メソドロジークラス

- 検証コンポーネントを開発するためには、以下に示すクラスを使用します。

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ため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	エンバイロンメントを定義するためのベースクラスです。
uvm_agent	エージェントを定義するためのベースクラスです。
uvm_test	テストを構築する為のベースクラスです。
uvm_monitor	モニターの基本機能を備えたベースクラスです。
uvm_scoreboard	スコアボードの基本機能を提供するベースクラスです。

ファクトリ

- UVMでは、クラスのインスタンスを作る場合、newコンストラクタを使用しないルールを設けています。
- 以下のようにcreate()メソッドを使用してインスタンスを作ります。

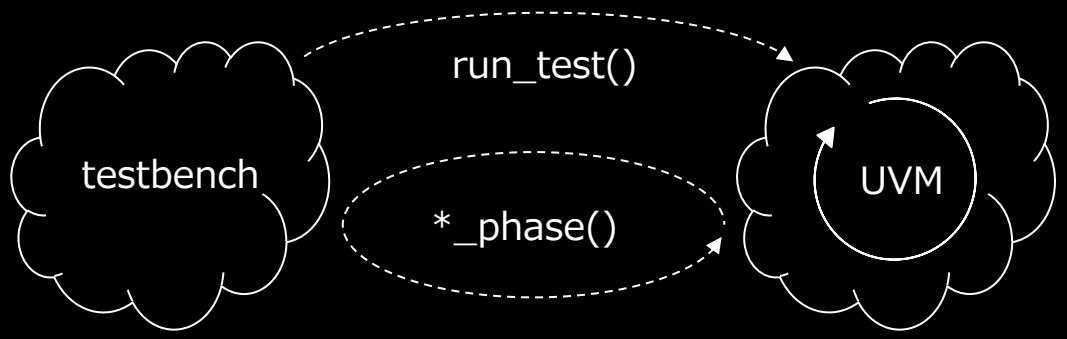
```
simple_collector    collector;

function void build_phase(uvm_phase phase);
          super.build_phase(phase);
          collector = simple_collector::type_id::create("collector",this);
          ...
endfunction
```

- こうすると、ソースコードを変更せずに、実行時にsimple_collectorと互換性のあるクラスのインスタンスで置き換えることができます。
- そのためには、検証コンポーネントを定義する際、コンポーネント名をUVMマクロで宣言する必要があります。

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が行います。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。
- ユーザの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

- 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。ユーザは必要なフェーズだけ記述すれば良いです。run_phaseだけがtaskです。それ以外はすべてfunctionです。表に記述されている順に制御を受けます。
- 通常使用するフェーズは、build_phase、connect_phase、run_phaseです。

代表的なフェーズ	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。サブコンポーネントをこのフェーズで作成します。トップダウンで呼び出されます。
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。ボトムアップで呼び出されます。
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ためのフェーズです。
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。

UVMの使用手順

- UVMの全ての機能は**uvm_pkg**パッケージに定義されています。したがって、このパッケージをインポートする事により、UVMを使用できるようになります。
- 実際にUVMを使用するスコープの先頭で uvm_pkgをインポートして下さい。

開発した検証環境をインクルードする。

```
`include "uvm_pkg.sv"
`include "uvm_macros.svh"
`include "simple_if.sv"
`include "simple_pkg.sv"

module top;
import uvm_pkg::*;
import simple_pkg::*;
...
simple_if SIF(.*);
design DUT(...);

initial begin
  uvm_config_db #(virtual simple_if)::set(null,"*env0*","vif",SIF);
  ...
  run_test();
end
...
endmodule
```

開発した検証環境を使用できるようにする。

インターフェースのインスタンスを検証環境に登録する。

① 最初の２行のインクルードをしておけばUVMを使用できます。uvm_pkgの中でuvm_macros.svhをインクルードしているので、2行目のインクルードは必要ないのですが、将来の変更に備えてインクルードしておくと安全です。

② 実際にUVMの機能を使用するスコープでUVMの機能をインポートします。

③ UVMを実行させます。UVMはユーザが開発した検証環境を起動し実行させていきます。

第2部
UVMによる検証環境構築

UVMを使用して検証環境を構築するために必要な手順を詳しく解説します。開発すべき検証機能は、ドライバー、シーケンサー、モニター、コレクター、エージェントに加えてその他の検証コンポーネントがあります。そして、それらの検証要素を開発するためには、一定のルールと推奨されているUVMの使用法に従わなければなりません。第2部では、検証環境開発手順と遵守すべきルールを解説します。

UVMを適用する手順

- UVMを検証作業に適用するためには、概略次のような作業があります。
- 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、前述したメソドロジークラスクラスを利用して生産性を高めます。
- 以下に、トランザクション、ドライバー、シーケンサー、コレクター、モニター、エージェント、テストの記述例を紹介します。

トランザクションの定義法

- トランザクションは検証環境の要素ではありませんが、UVMの一部なので一定のルールに従い定義されなければなりません。手順は以下のようになります。

① uvm_sequence_item、またはそのサブクラスからトランザクションを定義します。

② トランザクションに必要なフィールド（つまり、変数名）を定義する。

③ フィールドを定義する場合、幾つかのフィールドにはrand、またはrandc属性を付加する必要性が出てきます。

④ 必要に応じて制御フィールドマクロを定義します。

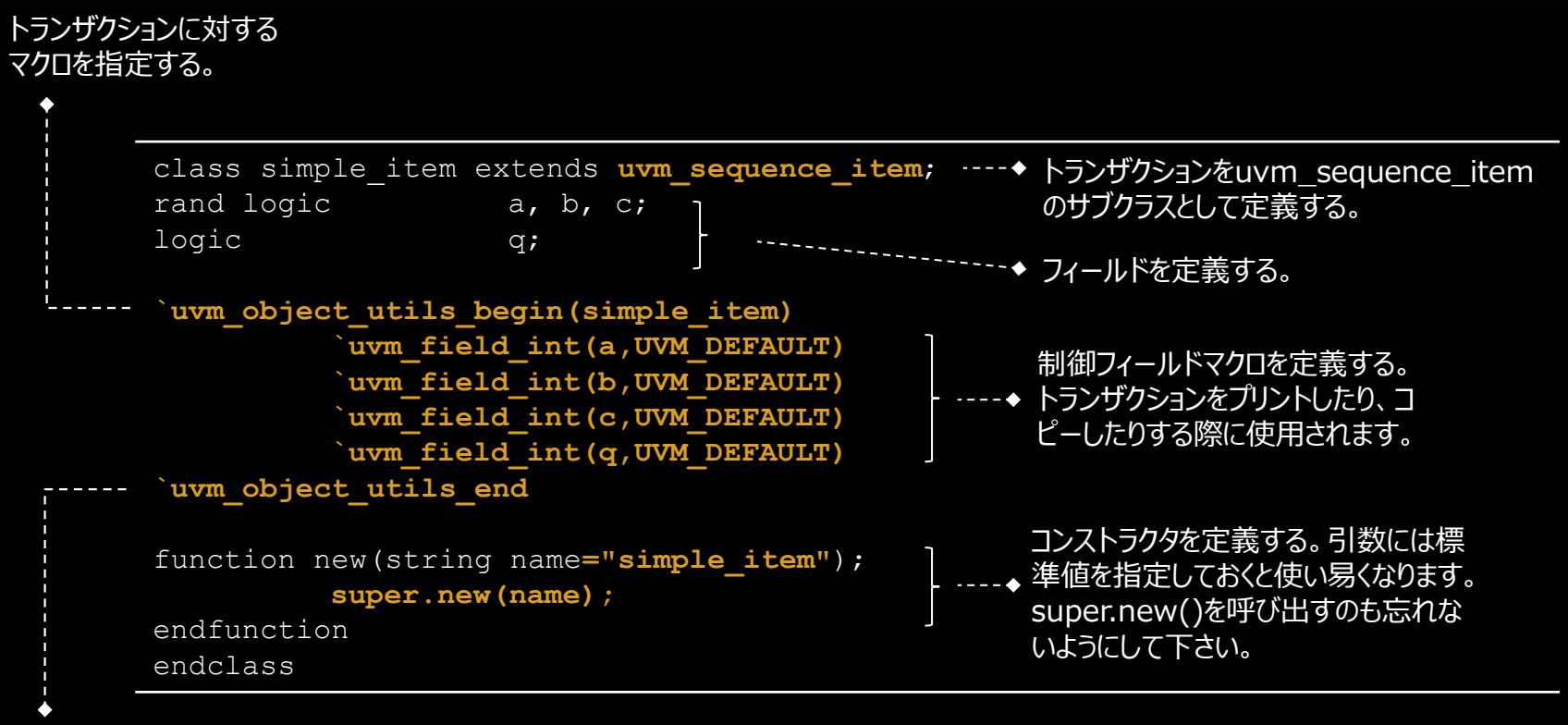
⑤ UVMマクロを定義します。フィールドの存在に関わらずUVMマクロは必須です。

⑥ コンストラクタを定義します。省略可能ですが、定義する方が望ましいと言われています。

⑦ コンストラクタ内でsuper.new(name)を呼び出します。

トランザクションの定義例

- トランザクションの定義例を以下に紹介します。



ドライバーの定義法

- ドライバーはDUTをドライブする機能を持ちますが、以下の手順で定義します。

① uvm_driver、または、そのサブクラスからドライバーを定義する。uvm_driverがseq_item_portという名称でTLMポートを定義しているので、ドライバーはget_next_item ()メソッドを呼び出すだけでトランザクションを取得できます。

② virtual インターフェースを宣言する。

③ コンストラクタを定義する。

④ UVMマクロを定義する。

⑤ connect_phase()でvirtualインターフェースを設定する。

⑥ run_phase()にトランザクションの処理手順を定義する。

ドライバーの定義例（その1）

- ドライバーの定義例を以下に紹介します。

```
class simple_driver extends uvm_driver #(simple_item);
virtual simple_if   vif;

`uvm_component_utils(simple_driver)

function new(string name,uvm_component parent);
        super.new(name,parent);
endfunction
extern function void connect_phase(uvm_phase phase);
extern task run_phase(uvm_phase phase);
extern task get_and_drive();
extern task drive_dut(input simple_item item);
endclass

function void simple_driver::connect_phase(uvm_phase phase);
        super.connect_phase(phase);
        if( !uvm_config_db#(virtual simple_if)::get(this,
                    get_full_name(),"vif",vif) )
          `uvm_error("NO-VIF",{"VIF error for ",get_full_name(),".vif"})
endfunction
```

このマクロを定義しないとファクトリ機能を使用できません。

コンストラクタの引数に標準値を指定しない方が安全です。

connect_phaseとrun_phaseを宣言する。

DUTをドライブる処理手順を宣言する。

virtualインターフェースを設定する。

ドライバーの定義例（その２）

- run_phase()は以下のようになります。

```
task simple_driver::run_phase(uvm_phase phase);
          forever begin
                    get_and_drive();
          end
endtask

task simple_driver::get_and_drive();
          // Get the next data item from sequencer.
          seq_item_port.get_next_item(req);
          // Execute the item.
          drive_dut(req);
          // Tell sequencer that item is done.
          seq_item_port.item_done();
endtask

task simple_driver::drive_dut(simple_item item);
...
endtask
```

シーケンサーからトランザクションを取得する。
reqはトランザクションを示す変数でuvm_driverに定義されています。

トランザクションの処理が終了したら、シーケンサーに通知する。

シーケンサーの定義法

- uvm_sequencer、またはそのサブクラスからシーケンサーを定義します。
- シーケンサーは、通常、以下に示すような簡単な定義で済みます。

```
class simple_sequencer extends uvm_sequencer #(simple_item);

`uvm_component_utils(simple_sequencer)

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

endclass
```

コレクターの定義法

- 近年の検証法ではDUTからのレスポンスをサンプリングする機能をモニターではなくコレクターと定義しています。ここでも、その定義に従う事にします。
- しかし、UVMにはコレクターに対応するメソドロジークラスが定義されていません。以下の手順でコレクターを定義します。

① コレクターをuvm_componentのサブクラスとして定義する。

② virtual インターフェースを宣言する。

③ 他の検証コンポーネントにトランザクションを送るためのTLMポートを定義する。

④ コンストラクタを定義する。

⑤ UVMマクロを定義する。

⑥ connect_phase()でvirtualインターフェースを設定する。

⑦ run_phase()でDUTからのレスポンスを監視する。

コレクターの定義例（その１）

- コレクターの定義例を以下に紹介します。

```
class simple_collector extends uvm_component;
virtual simple_if   vif;
simple_item         item;
uvm_analysis_port #(simple_item)        analysis_port;   ------◆ TLMポートを定義する。

`uvm_component_utils(simple_collector)

function new(string name,uvm_component parent);
        super.new(name,parent);
        analysis_port = new("analysis_port",this);     ---------◆ TLMポートのインスタンスを作る。
endfunction
extern function void connect_phase(uvm_phase phase);
extern task run_phase(uvm_phase phase);
extern task collect_response();
extern task collect_reset();
extern function void send_item();
endclass

function void simple_collector::connect_phase(uvm_phase phase);
...                                                    ---------◆ ドライバーと同じようにしてvirtualインターフェースの設定を行う。
endfunction
```

コレクターの定義例（その2）

- run_phase()では、DUTからのレスポンスをチェックし、レスポンスを得たらトランザクションに変換してモニターに送ります。

```
task simple_collector::run_phase(uvm_phase phase);
          item = simple_item::type_id::create("dut_response");
          fork
                    collect_response();
                    collect_reset();
          join
endtask
```

DUTがシーケンシャル回路であれば、クロッキングイベントと非同期信号のイベントを監視します。

```
task simple_collector::collect_response();
          forever begin
                    @vif.cb;
                    send_item();
          end
endtask

task simple_collector::collect_reset();
          forever begin
                    @vif.cbr;
                    send_item();
          end
endtask
```

```
function void simple_collector::send_item();
          item.a = vif.a;
          item.b = vif.b;
          item.c = vif.c;
          item.q = vif.q;
          ...
          analysis_port.write(item);
endfunction
```

トランザクションをモニターに送る。

モニターの定義法

- モニターはコレクターからトランザクションを受け取り、チェック、およびカバレッジの収集を行い、トランザクションを他のコンポーネントに送信します。
- モニターをuvm_monitor、またはそのサブクラスを使用して以下の手順で定義します。

① モニターをuvm_monitorのサブクラスとして定義する。

② トランザクションを受け取るためのTLMポートとトランザクションを他の検証コンポーネントに送信するためのTLMポートを定義する。

③ コンストラクタを定義する。

④ UVMマクロを定義する。

⑤ コレクターからトランザクションを受け取るためにwrite()メソッドを定義する。

モニターの定義例

- モニターを以下のように定義します。

```
class simple_monitor extends uvm_monitor;
uvm_analysis_imp #(simple_item,simple_monitor)    analysis_export;
uvm_analysis_port #(simple_item)                  item_collected_port;

`uvm_component_utils(simple_monitor)

function new(string name,uvm_component parent);
        super.new(name,parent);
        analysis_export = new("analysis_export",this);
        item_collected_port = new("item_colleted_port",this);
endfunction

extern function void write(simple_item data);
endclass

function void simple_monitor::write(simple_item data);
        item_collected_port.write(data);
        ...
endfunction
```

TLMポートを定義する。

TLMポートのインスタンスを作る。

トランザクションを他の検証コンポーネントに送る。

エージェントの定義法

- エージェントは、ドライバー、シーケンサー、モニター、コレクターから構成されますが、コンフィギュレーションパラメータの設定にしたがって構成を変える必要があります。エージェントをuvm_agent、またはそのサブクラスを使用して以下の手順で定義します。

① ドライバー、シーケンサー、モニター、コレクターのためのハンドルを宣言する。

② UVMマクロを定義する。

③ コンストラクタを定義する。

④ build_phaseでドライバー、シーケンサー、モニター、コレクターのインスタンスを作る。

⑤ connect_phaseでドライバー、シーケンサー、モニター、コレクターを接続する。

エージェントの定義例（その1）

- エージェントを以下のように定義します。

```
class simple_agent extends uvm_agent;
simple_driver       driver;
simple_sequencer    sequencer;
simple_collector    collector;
simple_monitor      monitor;

`uvm_component_utils_begin(simple_agent)
          `uvm_field_enum(uvm_active_passive_enum,is_active,UVM_DEFAULT)
`uvm_component_utils_end

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

extern function void build_phase(uvm_phase phase);
extern function void connect_phase(uvm_phase phase);
endclass
```

uvm_agentにはis_activeと呼ばれるコンフィギュレーションパラメータが定義されているので、プリントするためにUVMマクロを指定しておきます。

build_phaseで階層を作り、connect_phaseでサブコンポネントを接続します。

エージェントの定義例（その2）

- build_phase()とconnect_phase()は以下のようになります。

```
function void simple_agent::build_phase(uvm_phase phase);
        super.build_phase(phase);
        collector = simple_collector::type_id::create("collector",this);
        monitor = simple_monitor::type_id::create("monitor",this);
        if( is_active == UVM_ACTIVE ) begin
          sequencer = simple_sequencer::type_id::create("sequencer",this);
          driver = simple_driver::type_id::create("driver",this);
        end
endfunction

function void simple_agent::connect_phase(uvm_phase phase);
        super.connect_phase(phase);
        collector.analysis_port.connect(monitor.analysis_export);
        if( is_active == UVM_ACTIVE ) begin
          driver.seq_item_port.connect(sequencer.seq_item_export);
        end
endfunction
```

テストの定義法

- テストを記述するためには、uvm_testまたはそのサブクラスを拡張して定義します。
- テストケースにより、テストの環境は多少異なる事がありますが、テスト間では多くの機能を共有します。したがって、共有される部分をベースクラスとして定義しておくと再利用が可能となります。
- テストの目的は、テスト環境に具体的なテスト項目を割り当てる事です。つまり、シーケンスをシーケンサーに割り当てる事です。
- シーケンサーはテストに関する情報を持たないので、default_sequenceという名称でシーケンスを認識します。したがって、テストではテストケースをdefault_sequenceに割り当てるのが仕事になります。

テストのベースクラス定義例

- 以下のようにベースクラスを定義します。テスト間で共有する機能を定義しておきます。以下の場合では、テストの階層構造を定義しています。

```
class simple_test_base extends uvm_test;
simple_env          env0;

`uvm_component_utils(simple_test_base)

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

extern function void build_phase(uvm_phase phase);
endclass

function void simple_test_base::build_phase(uvm_phase phase);
        super.build_phase(phase);
        env0 = simple_env::type_id::create("env0",this);
endfunction
```

テスト環境の階層構造を定義しています。

テストの定義例

- テストケースをシーケンサーに定義するためには、以下のような記述になります。

```
class simple_test1 extends simple_test_base;

`uvm_component_utils(simple_test1)

function new(string name="simple_test1",uvm_component parent=null);
        super.new(name, parent);
endfunction

extern function void build_phase(uvm_phase phase);
endclass

function void simple_test1::build_phase(uvm_phase phase);
        uvm_config_db#(uvm_object_wrapper)::set(this,
          "env0.agent0.sequencer.run_phase",
          "default_sequence",test_seq1::type_id::get());
        super.build_phase(phase);
endfunction
```

テストのベースクラスを拡張する。

コンストラクタの引数には標準値を設定しておく。

テストケースをdefault_sequenceに割り当てています。

- シミュレーションを実行する際、ここで定義したテストクラス名をコマンドラインで指定すると、このクラスが実行されます。

重要
raise_objection()とdrop_objection()

- 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ためには、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。
- 検証環境階層上のどこかのインスタンスで二つのメソッドを呼び出していれば、そのインスタンスの階層以下の全てのインスタンスにも適用されるので、これらのメソッドを個別に呼び出す必要はありません。
- しかし、検証環境階層上のどこかで呼び出す方法は検証環境がテストケースに依存する可能性が高くなるため、再利用可能性の原則に反するので望ましいとは言えません。
- 一般的には、シーケンスの実行でシミュレーションが開始し、シーケンスの終了でシミュレーションが終了するので、シーケンス内でraise_objection()とdrop_objection()の呼び出し制御をすると便利です。詳細は文献[4,5]にあります。

検証環境開発における推奨される手法

- トランザクションのコンストラクタでは全ての引数に標準値を設定しておくと便利です。
- 検証コンポーネントのコンストラクタでは、名称と親コンポーネントを示す引数には標準値を指定しない方が安全です。
- ただし、テストは階層のルートなので、引数に標準値を設定しておく方がテストケースを実行時に作り易くなります。
- virtualインターフェースの設定は、build_phaseでもconnect_phaseでも可能ですが、どちらのフェーズで設定するかを統一しておいた方が間違いが少なくなります。
- UVMにはコレクターのクラスが定義されていないので、コレクターのベースクラスを開発しておくと便利です。
- コンポーネントのインスタンスを作る時にはファクトリを利用する事をすすめます。つまり、newコンストラクタを使用せずにインスタンスを作ると良いです。オブジェクトに関しても同様です。
- SystemVerilogマクロを活用するとコードを記述する際に省力化の効果があります。例えば、シーケンサーは殆ど同じ記述になるので、マクロを準備しておくとシーケンサーの記述は一行で済みます。
- メソッドの内容をクラス外に記述すると検証機能の仕様を見易くなります。ただし、短い記述のコンストラクタはクラス内に書いた方が見易くなります。
- 一つのファイルに一つのクラスを定義し、クラス名とファイル名を対応させておくと管理し易くなります。

第3部
IEEE Std 1800-2023とUVM

SystemVerilog言語仕様の改訂版が2024年2月28日にIEEE Std 1800-2023として公開されました。改訂版には、コード開発の質を高める機能と保守を容易にするための機能が追加されました。以下では、UVMによる検証環境開発過程に役立つ知識を解説します。

IEEE Std 1800-2023の適用

- SystemVerilogの改訂版に追加されている機能でUVMによる検証環境構築で役に立つ機能は以下のようになります。
- クラスに**:final**を付けるとクラスを拡張できなくなります。
- クラスタイプを取り出すための機能が追加されました。
- ベースクラスのコンストラクタを呼ぶ際には引数として**default**を使用できます。
- virtualメソッドに**:extends**または**:initial**を付けて間違いを回避できます。
- 制約に**:extends**または**:initial**を付けて間違いを回避できます。
- ベースクラスで定義されたカバーグループをサブクラスで拡張できます。
- **weak_reference#(T)**機能が追加されてメモリーを有効に活用できるようになりました。
- これらの新機能をUVMに適用して保守作業の質を高める事ができます。

クラスと:final

- サブクラスが定義されるのを防ぐためには、**:final**を付けると良いです。
- 最も適切な例は、std::processクラスの定義になると思えます。std::processクラスは、以下のように定義されているので、ユーザはprocessクラスを拡張する事はできません。

```
class :final process;
          typedef enum {FINISHED, RUNNING, WAITING, SUSPENDED, KILLED} state;
          static function process self();
          function state status();
          function void kill();
          task await();
          function void suspend();
          function void resume();
          function void srandom(int seed);
          function string get_randstate();
          function void set_randstate(string state);
endclass
```

type(this)

- SystemVerilog改訂版では、使用しているスコープのクラスタイプを参照できるようにtypeの機能が拡張されました。
- UVMによる開発では、多くのパラメータを伴うクラスの参照が頻繁に発生します。指定を間違うとコンパイルエラーまたは誤動作の原因にもなります。この種の間違いを回避するために、**type(this)**機能が追加されました。
- 例えば、以下のように使用します。type(this)は、registry#(T)を表現しています。クラスタイプをtype(this)で参照しているので、クラスのパラメータに変更が生じてもget()メソッド内の記述に影響がありません。たとえ、クラス名が変更されても影響がありません。

```
class registry #(type T=int);
// ...
static function type(this) get();
static type(this)   m_inst;
          if (m_inst == null)
                    m_inst = new();
          return m_inst;
endfunction
...
endclass
```

コンストラクタとdefault

- サブクラスのコンストラクトを定義する際、ベースクラスのコンストラクタの引数のリストをdefaultで参照できるようになりました。この機能により、サブクラスのコストラクタは、ベースクラスのコンストラクタの仕様変更による影響を受け難くなります。
- 以下のように**default**を使用すれば、ベースクラスのコンストラクタの引数が変化してもサブクラスには影響が及びません。

```
class base_t;
string    name;
local int m_id;
function new(string name,output int id);
    this.name = name;
    id = m_id++;
endfunction
endclass
```

```
class sub_t extends base_t;
byte      m_value;
function new(default,byte v);
    super.new(default);
    m_value = v;
endfunction
endclass
```

ベースクラスの引数リストに変更が発生しても直接的な影響が及び難い

virtualメソッドと:extends / :initial

- ベースクラスでvirtualとして定義されているメソッドをサブクラスで拡張するには、**:extends** を付けておくと間違いが起こりません。
- :extendsを指定すると、そのメソッドがベースクラスでvirtualメソッドとして定義されていなければコンパイルエラーが出ます。
- ベースクラスで定義されているvirtualメソッドを不注意に書き換える間違いを予防するためには、サブクラスではメソッドに **:initial** を付けておくと安全です。
- サブクラスで定義するメソッドに:initialを付けておくと、そのメソッドがベースクラスでvirtualに宣言されていればコンパイルエラーになります。つまり、:initialはvirtualメソッドをnon-virtualメソッドに書き換える間違いを防ぐ手段です。

defaut / :initial / :extendsの使用例

- ドライバーの定義例を紹介しますが、以下の点に注意が必要です。
- コンストラクタの引数にdefaultを指定する。
- UVMでvirtualメソッドとして定義されていれば:extendsを指定する。
- ユーザのクラスで独自に定義するメソッドには:initialを指定する。

```
class simple_driver_t extends uvm_driver #(simple_item_t);
virtual simple_ifvif;
`uvm_component_utils(simple_driver_t)
function new(default);
    super.new(default);
endfunction
extern function :extends void build_phase(uvm_phase phase);
extern task :extends run_phase(uvm_phase phase);
extern task :initial get_and_drive();
extern task :initial drive_dut(input simple_item_t item);
endclass
```

制約と:extends / :initial / :final

- ベースクラスに定義されている制約をサブクラスで拡張するには、**:extends** を付けておくと間違いが起こりません。
- サブクラスで定義する制約に:extendsを付けると、同じ制約名称を持つ制約がベースクラスに定義されていなければコンパイルエラーになります。
- ベースクラスで定義されている制約を不注意に書き換える間違いを予防するためには、サブクラスでは制約に **:initial** を付けておくと安全です。
- サブクラスで定義する制約に:initialを付けると、同じ制約名称がベースクラスで定義されていればコンパイルエラーになります。
- 制約を定義する際に**:final**を付けておくと、サブクラスではその制約を再定義できなくなります。

:extends / :initial / :final の使用例

- 以下に使用例を紹介しますが、コンパイルエラーが出る理由を説明すると理解が深まります。

```
class base_t;
constraint C1 {}
constraint :final FC {}
endclass
```

```
class sub1_t extends base_t;
constraint :initial C1 {} // error
constraint :extends C2 {} // error
constraint FC {}          // error
endclass
```

weak_reference#(T)

- SystemVerilogにはJavaと同様にガーベッジコレクションの機能が備わっています。ダイナミックに作られたオブジェクトは使用されなくなるとガーベッジコレクションにより再利用される対象になります。一つのオブジェクトは複数個所から参照されるため、それらの全ての参照が終了しなければ、再利用の対象にはなりません。
- オブジェクトのハンドルが存在し続ける場合には参照したままの状態になるため、ハンドルにnullを設定して参照の終了宣言をしなければなりません。
- 弱い参照は、ガーベッジコレクションの働きを妨げずにオブジェクトを参照できる仕組みです。つまり、オブジェクトが弱い参照しかされていない場合には、オブジェクトは強い参照がされていないと見なされてガーベッジコレクションの対象になります。
- 弱い参照は高度な概念ですが、シミュレーション時のメモリー使用効率を向上させる役割を果たします。
- 詳細は、次ページにある資料をダウンロードして下さい。

IEEE Std 1800-2023の概要

- 以上紹介した機能以外にも重要な機能がIEEE Std 1800-2023に追加されていますが、時間とスライド数が許さないため説明を割愛します。
- 追加された機能の概要は、共立出版社の『SystemVerilog超入門』または『SystemVerilog入門』の書籍ウェブサイトより無償でダウンロードできます。
- サイトに行き「関連情報タブ」をクリックし、「補足資料」をクリックすると資料を得られます。
- その他、文献[8]にも要約があります。

エピローグ

以上、UVMの基礎概念、実装法、実装時における守るべきルール、およびコードの保守作業におけるSystemVerilog機能の活用法を述べました。UVMはSystemVerilogから独立してはいるもののSystemVerilogの仕様に依存をしているのは明らかです。今後もSystemVerilogの仕様は進歩していくので、UVMの機能も進化していく可能性は十分にあります。UVMだけでなくSystemVerilogの最新情報に注意してUVMを活用していく必要があります。

参考文献

文献[1]は最新版のSystemVerilog仕様書です。是非一読下さい。文献[2]はUVMのリファレンスマニュアルです。予備知識なしでは難しいと思いますが、本講演での基礎知識があれば十分に読めます。文献[3]はUVMの概要を比較的詳しくまとめた資料です。文献[4,5]はUVMに関する入門書です。SystemVerilogに関する知識を復習するためには、文献[6,7]を参考にして下さい。最新のSystemVerilog仕様は、文献[8]にも要約されています。

[1] IEEE Std 1800-2023: IEEE Standard for SystemVerilog – Unified Hardware Design, Specification and Verification Language.
[2] IEEE Std 1800.2-2020: IEEE Standard for Universal Verification Methodology Language Reference Manual.
[3] Universal Verification Methodology (UVM) 1.2 User's Guide, Accellera, October 8, 2015.
[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、実践UVM入門、森北出版 2021.
[6] 篠塚一也、SystemVerilog入門, 共立出版 2020.
[7] 篠塚一也、SystemVerilog超入門, 共立出版 2023.
[8] Kazuya Shinozuka, A Subjective Review on IEEE Std 1800-2023, DVCon Japan 2024.