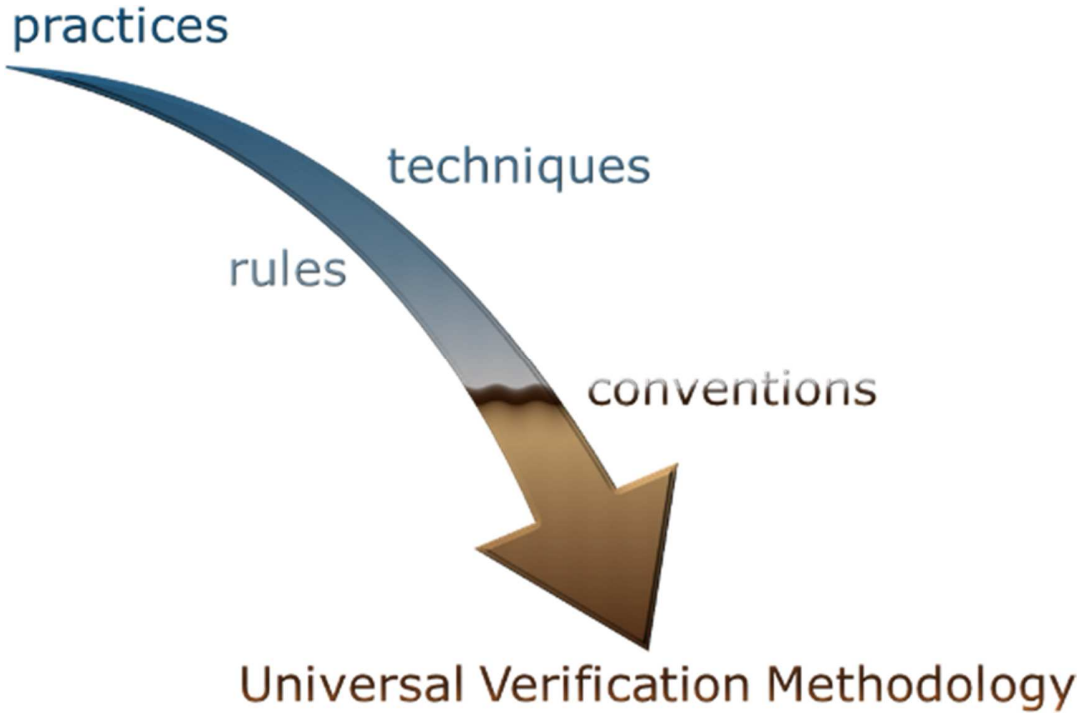
Artgraphics

実践 UVM 入門

Document Identification Number: ARTG-TD-002-2019
Document Revision: 1.0 2019.11.30
アートグラフィックス
篠塚一也



実践 UVM 入門

〒124-0012 東京都葛飾区立石 8-14-1
www.artgraphics.co.jp

A Practical Guide to Learning UVM

8-14-1, Tateishi, Katsushika-ku, Tokyo, 124-0012 Japan
www.artgraphics.co.jp

はじめに

UVM は再利用可能な検証コンポーネントを開発する手段として次第に普及しつつあります。然し、現在市販されている UVM に関する書物、及び入手可能な技術書（ユーザガイド、リファレンスマニュアル等）の殆どは英文で上級技術者向けに書かれているため、SystemVerilog の多くのユーザが UVM に関する基礎的な知識を得る事が難しいと感じているのが実情です。本書は、初心者を対象に SystemVerilog 以外の知識を仮定せずに、UVM とは何か、UVM は何故必要なのか、UVM はどの様に構成されているか、また、UVM をどの様に使用するかを具体例を通して解説しています。

予備知識なしで UVM を理解する事が難しいのは、近年の検証技術の動向、及び UVM が開発された背景等に関する解説が十分では無い事にも起因していると考えられます。歴史的な背景を理解する事により、UVM 誕生の所以、及び UVM が齎す効果を知る事ができます。その様な知識は、UVM が備える機能一つ一つを理解するのと同程度に重要な概念で、UVM の全容を知る手助けとなります。UVM の存在意義、及び UVM を構成する機能を合理的に理解する事ができる様に、背景を重視した解説手順を採用しているのが本書の特徴です。

本書の第 1 章では、近年の検証技術手法の概要を解説し、その検証手法と UVM の構成には多くの共通点が存在する事を指摘します。その共通点を知る事により、UVM の全容を理解し易くなります。更に、第 1 章では、UVM を使用した検証環境の概要を解説し、検証環境構築に必要となる主な検証コンポーネントを概説します。その知識を基にすれば、UVM の全体像を確実に理解する事ができ、第 2 章以降の UVM 機能の詳細解説へとスムーズに移行する事ができます。

本書の 2 章以降は、UVM を構成する個々の概念を具体的に解説しています。UVM では、トランザクションを用いて RTL よりも高位なレベルでシステム全体の流れを記述します。それが、第 2 章の TLM（Transaction Level Modeling）です。この章では、検証コンポーネント間でデータ授受をするための基本的な手順法を紹介します。シーケンサーとドライバー間に見られる 1 対 1 データ通信に加えて、モニターと他の検証コンポーネント間に見られる 1 対 N のデータ通信の詳細を解説します。TLM に関する知識は、UVM コンポーネントを開発する際には、必要不可欠な知識です。この章では、シーケンサーやドライバー等のメソドロジークラスを使用せずに、素朴なコンポーネントを使用して TLM 通信を詳しく解説します。これにより、TLM を適用するための厳密な技術を習得する事ができると共に、メソドロジークラスが持つ利便性を理解する事ができます。

さて、UVM は手法であり、ユーザは UVM が規定している原則、及びルールに従う事が必要です。例えば、SystemVerilog による記述としては正しくても、UVM の検証コードとして考えると不備な記述が多々存在します。しかも、それらの不備な記述の発見が遅れれば遅れるほど解決に時間がかかります。然し、UVM のマニュアルやユーザガイドには、そうした規定を簡潔に明記していないため、UVM への初心者は貴重な時間を浪費しかねない状況に陥るのが常です。第 3 章では UVM が推奨しているルール、記述法、及び手順を簡潔明瞭に解説し、第 4 章以降の検証環境構築の準備をします。

第 4 章は、検証環境を構築するための主要な要素であるトランザクション、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、エージェント、エンバイロンメント、テスト等を詳しく解説します。この章は、検証環境を構築するための重要な内容を含みます。特に、シーケンスは、UVM の中で最も重要で、且つ難解な概念の一つであるため、シーケンスの機能と使用法を具体例を通して詳しく解説します。更に、シミュレーションを進行させるために必要な raise_objection()と drop_objection()の呼び出しをシーケンスに自動的に行なわせる方法も解説します。シーケンサーとドライバーはシーケンスとの関わりにおいて機能、役割、及び使用法を詳しく解説します。総じて、第 4 章の内容を理解すれば、UVM を実践に適用する知識を身に付ける

事ができます。

第 5 章は、第 4 章までに習得した知識を応用した検証環境構築例を紹介します。各検証コンポーネントを省略せずに記述しているので、実践に適用する際のコードスニペットの役割を果たします。この章に記述されている例を通して、シーケンサー、ドライバー、コレクター、モニター等の UVM コンポーネント開発法を再確認する事ができます。また、シーケンスを利用して実行時に制約を与えてトランザクションを生成する技術を習得する事ができます。

本書は、他の書物、及び技術情報と異なり、検証環境構築の具体例を簡潔に示してあります。簡単な DUT を使用して再利用可能な UVM 検証コンポーネントの開発例を示しているため、検証環境全体を把握し易いだけでなく、検証コンポーネントの開発自身に集中する事ができる利点を齎しています。

尚、本書の記述は UVM 1.2 をベースにしています。また、紙面の都合上、一部の記述は小さな書体で記述されています。

アートグラフィックス
篠塚一也

変更履歴

日付	Revision	変更点
2019.11.30	1.0	初版。

目次

次第に明らかになる様に、UVM は検証技術のトレンドである階層的テストベンチ技術を実現しています。実際問題として、上記のレイヤーの機能を遂行するための UVM クラスが存在します。ユーザは、それらのクラスを使用して検証作業の生産性を向上する事ができます。

1.3 UVM テストベンチの構成

図 1-3 は、UVM のテストベンチの構成を示しています。図において、包含関係はクラス、又はモジュールのインスタンスを持つという関係を意味します。例えば、テストベンチは DUT のインスタンスを保有します。包含関係は、一般的に、階層を意味します。図から明らかな様に、UVM のテストベンチの構成は階層的にテストベンチを記述す手法の構造を反映しています。階層を構成する要素の意味を表 1-2 に記します。

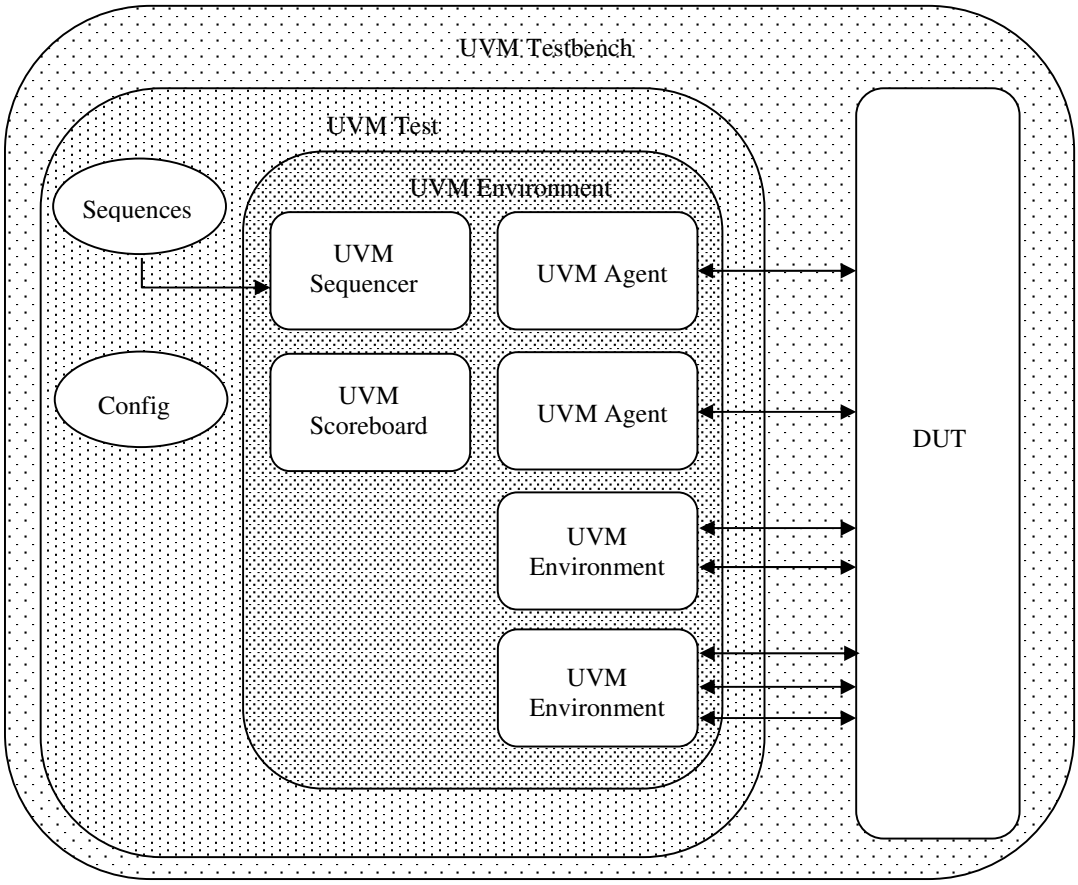


図 1-3　UVM のテストベンチ構成（[2]）

表 1-2　UVM のテストベンチを構成する要素（[2]）

階層を構成する要素	意味
UVM Testbench テストベンチ	これは、所謂、トップレベルのモジュールを指します。このテストベンチは DUT、及び検証機能等をインスタンスとして保有します。
UVM Test テスト	検証用のテストケースのトップレベルのクラスインスタンスを意味します。このインスタンスが該当するテストを実行します。コンフィギュレーション（Config）の設定を変更する事により、各種のテ

	ストケースの選別をする事ができます。
UVM Environment エンバイロンメント	階層的に構築した検証コンポーネントです。例えば、トップレベルのエンバイロンメントとしては、SoC、IP（PCIe エンバイロンメント、USB エンバイロンメント、Memory Controller エンバイロンメント）等があります。
UVM Scoreboard スコアボード	スコアボードの主目的は、DUT の動作を確認する事です。即ち、エージェントから取得した DUT のレスポンスが期待される結果と一致するかのチェックを遂行します。
UVM Agent エージェント	基本的な検証コンポーネントで、通常、シーケンサー、ドライバー、モニター、コレクター等のインスタンスで構成されます。エージェントは DUT と virtual インターフェースを介して接続されます。
UVM Sequencer シーケンサー	トランザクションを生成する制御を司るメインコンポーネントです。シーケンサーはシーケンスを作り、シーケンスを実行する事によりトランザクションを生成します。
UVM Sequences シーケンス	トランザクションを生成するための手順を含むオブジェクトです。オブジェクトであるため、コンポーネント階層には含まれません。シーケンスは複雑なテストシナリオを定義する事ができます。シーケンスはテストに依存するため、エンバイロンメントには含まれていません。
UVM Driver ドライバー	ドライバーはシーケンサーからトランザクションを取得し、シグナルレベルに変換した後、DUT をドライブします。シーケンサーとドライバーの通信には TLM が使用されます。
UVM Monitor モニター	モニターは DUT からのレスポンスを他の検証コンポーネントにトランザクションとして送信する役目を持ちます。モニター自身もカバレッジ計算、及びレスポンスに関するチェックも行います。DUT からのレスポンスはシグナルレベルであるため、レスポンスをトランザクションに変換する機能をコレクターとして分離するのが一般的です。

1.4　トランザクション

UVM はトランザクションを使用してシミュレーションを実行します。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。最も代表的なトランザクション処理はシーケンサーとドライバーに於いて行われます（図 1-4）。

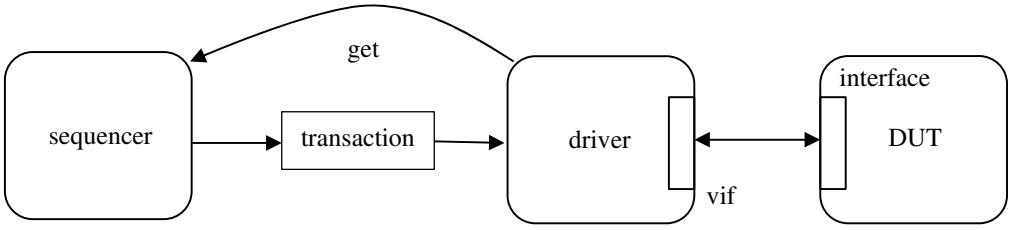


図 1-4　典型的なトランザクション処理の例

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に変換して DUT をドライブします。その際、ドライバーは virtual インターフェ

ース（vif）を使用します。

尚、UVM では、uvm_sequence_item クラス、又は、そのサブクラスからトランザクションを定義しなければなりません。SystemVerilog クラスにはビルトイン randomize()メソッドが定義されているので、トランザクションのフィールドに乱数を発生する事は容易です。

一方、DUT からのレスポンスはシグナルレベルなので、トランザクションに変換する必要があります。この変換をするコンポーネントは、一般的に、コレクターと呼ばれています。コレクターは、DUT からのレスポンスをサンプリングし、トランザクションに変換します。変換されたトランザクションは、更に詳細な検証をするためにモニターを経由して他の検証コンポーネントに送られます。

1.5 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。大きく分けて以下の様な二種類のクラスがあります。

- トランザクション、及びシナリオを記述するためのクラス
- 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

1.5.1 トランザクションとシナリオに関連する UVM クラス

トランザクション、及びシナリオを作成するためには表 1-3 のクラスを使用します。これらは、データオブジェクトを定義するために使用するクラスです、

表 1-3 トランザクションとシナリオに関連するクラス

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。 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。

例 1-1 トランザクションの定義例

トランザクションを定義するためには、以下の様に uvm_sequence_item クラスを使用します。

```
class simple_item extends uvm_sequence_item;
rand int unsigned addr;
rand int unsigned data;
rand int unsigned delay;

`uvm_object_utils_begin(simple_item)
    `uvm_field_int(addr,UVM_DEFAULT)
    `uvm_field_int(data,UVM_DEFAULT)
    `uvm_field_int(delay,UVM_DEFAULT)
`uvm_object_utils_end
```

UVM の効果を最大限に引き出すためには、メソドロジークラスを使用して検証コンポーネントを開発します。以下に、典型的なモニターの定義例を紹介します。

例 1-2　モニターの定義例

モニターを定義するためには、以下に示す様にメソドロジークラス uvm_monior を利用します。尚、モニターの定義法は、第 4 章で詳しく解説します。

```
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
```

■

1.6　Virtual インターフェース

UVM は SystemVerilog クラスの集合体です。UVM が DUT と通信するためには、virtual インターフェースを使用します。virtual インターフェースとは DUT に接続されているインターフェースのインスタンスへのポインターです。UVM の検証コンポーネントは virtual インターフェースを使用して、DUT の信号を操作したり、DUT からのレスポンスをサンプリングします。UVM と DUT との接点はインターフェースに宣言されている信号だけです。このため、UVM で開発した検証機能は、自ずから汎用的になります。

ドライバー、及びコレクターは DUT とのデータ授受に virtual インターフェース（vif）を使用します（図 1-6）。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、インターフェース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 と関連する検証コンポーネントの virtual インターフェースに引き渡さなければなりません。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ため、実行時（通常は、build_phase()、又は、connect_phase()）に virtual インターフェースの情報を vif に設定する様にします。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利用されます。

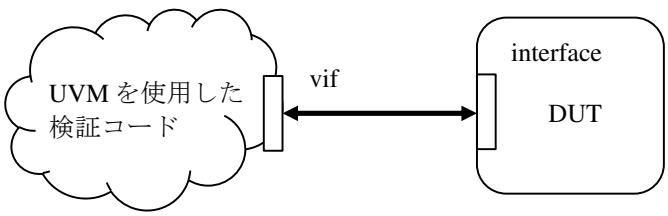


図 1-6　Virtual インターフェースに依る UVM と DUT の信号授受

virtual インターフェースを使用する UVM コンポーネントは、主としてドライバーとコレクター

です。テストベンチにはインターフェースのインスタンスが実在するので、テストベンチではそのインスタンスと UVM 側の virtual インターフェースのプロパーティ名称 vif との対応表を準備しておきます。この準備を UVM の run_test()を呼ぶ前に完了しなければなりません。UVM 側は、UVM コンポーネントの build_phase()、又は、connect_phase()実行中に、その対応表を使用して vif にインターフェースのインスタンスを割り当てます。

参考 1-2

DUT 側ではポートとしてインターフェースを使用していなくても構いません。例えば、DUT が Verilog スタイルのモジュールでも virtual インターフェースは正しく動作します。インターフェースには DUT のポートが接続されているため、ドライバーがインターフェースの信号を操作する事により、DUT をドライブする事ができます。
■

参考 1-3

SystemVerilog のルールにより、パッケージ、及びクラス内にインターフェースを定義する事はできません。従って、インターフェースはグローバルスコープに定義されなければなりません（図 1-7）。また、当然の帰着として、インターフェース内では UVM に関する機能を使用する事はできません。

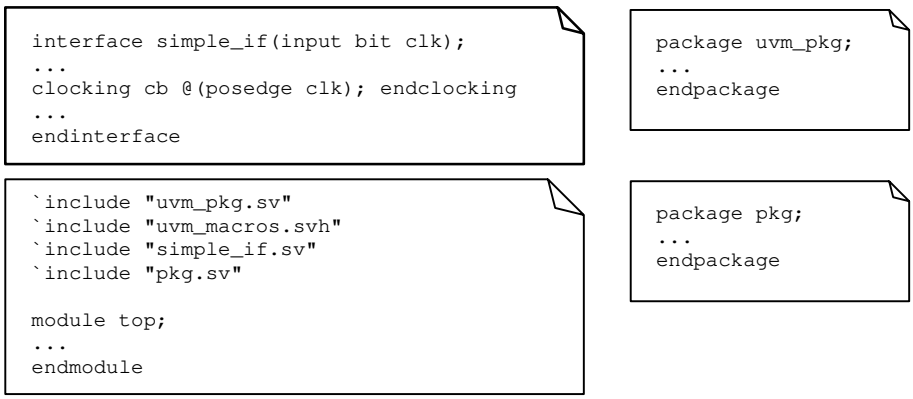


図 1-7　インターフェースを定義するスコープ

■

1.7　UVM の使用手順

本節では、UVM を使用する際に必要な手順をまとめます。UVM をサポートする機能は EDA ベンダーにより異なるため、本節では UVM を SystemVerilog で記述した一般的なライブラリーとして解説を進めます。

1.7.1　UVM の引用

UVM を使用するファイルの先頭で、次の様にファイルをインクルードして下さい。UVM を使用する時には、これらのファイルは不可欠です。

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

1.7.2 uvm_pkg パッケージ

UVM の全てのクラスは uvm_pkg パッケージに定義されています。従って、このパッケージをインポートする事により、UVM を使用する事ができる様になります。

1.7.3 uvm_pkg のインポート

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。但し、uvm_pkg をグローバルにインポートすると名称に関する矛盾が発生する可能性があります。その問題を未然に防ぐためには、以下の様に UVM を使用するスコープの先頭で uvm_pkg をインポートする様にして下さい。

```
module top;
import uvm_pkg::*;
...
endmodule
```

1.7.4 UVM の全体的な使用例

以上の使用ルールをまとめると以下の様になります。尚、UVM のシミュレーションを実行するためには、run_test()を呼び出す事が必要です。

```
`include "uvm_pkg.sv"
`include "uvm_macros.svh"
`include "my_interface.sv"
`include "my_pkg.sv"

module top;
import uvm_pkg::*;   // import UVM before using
import my_pkg::*;

initial begin
    ...
    run_test();      // start UVM
end
...
endmodule
```

例 1-3　UVM の簡単な使用例

以下にメッセージをプリントするだけの簡単な処理を示します。この例ではシミュレーションを実行しないので、run_test()を呼んでいません。

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

module top;
import uvm_pkg::*;   // import UVM before using

initial begin
    `uvm_info("INFO","My first UVM example.",UVM_LOW)
end

endmodule
```

この例をコンパイルして実行すると以下の様にメッセージがプリントされます。

2 TLM

本章では、UVM 検証コンポーネント間の通信の基礎を成す TLM を詳しく解説します。本章では、意図的に、メソドロジークラスを使用せずに検証コンポーネントを記述します。第 4 章では、メソドロジークラスの使用がそれらの検証コンポーネントの記述を簡略化する効果がある事を学びます。

メソドロジークラスを使用すると準備作業が簡略化されます。例えば、uvm_driver クラスは TLM-port として seq_item_port を定義しているので、uvm_driver クラスを利用すればトランザクションを取得する際に特別な準備は必要ありません。また、シーケンサーのベースクラス uvm_sequencer クラスも同様に、TLM-export として seq_item_export を定義しています。従って、シーケンサーとドライバー間でのトランザクションを送受信する処理は、特別な準備をせずに開始する事ができます。即ち、シーケンサーとドライバーを、それぞれ uvm_sequencer と uvm_driver のサブクラスとして定義すれば、TLM を使用するための準備処理は軽減されます。

然し、本章の目的は、UVM で TLM 通信をするために必要な基本手順を解説する事が目的です。そのためには、TLM-port と TLM-export を具体的に宣言して、put 及び get という基本的な操作をモデルにして解説を進めます。

2.1 概要

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。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。以降の節で示す例により明らかになりますが、同じ TLM-port のプロトコルに従う限り、UVM コンポーネント間の通信は相手方のコンポーネントを選びません。例えば、図 2-1 で示す様に put_producer と get_consumer を変更せずに、FIFO キューに接続する事ができます。

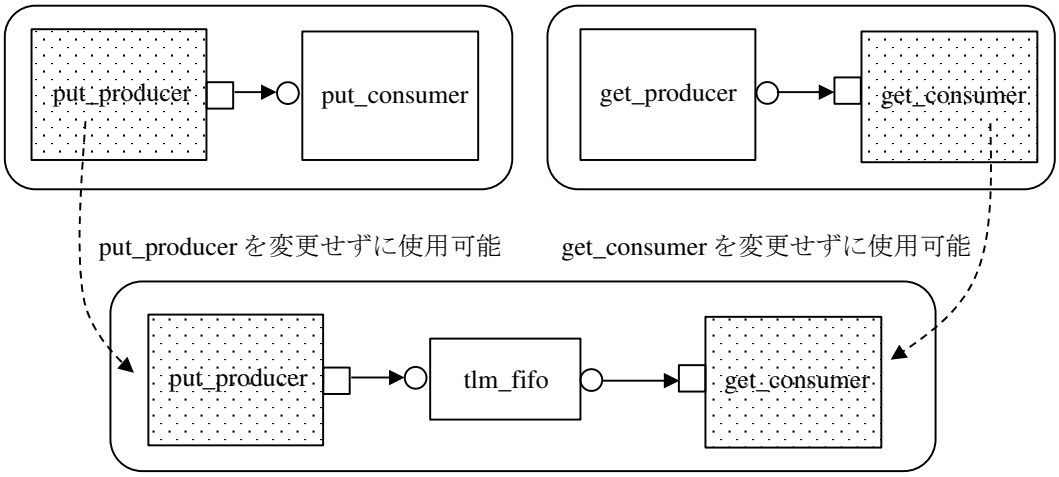


図 2-1 TLM 接続の効果

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処理します。そ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 をドライブする役目を持ちます。DUT 側からのレスポンスを収集する役目を持つ 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できる様になります

（図 2-2）。

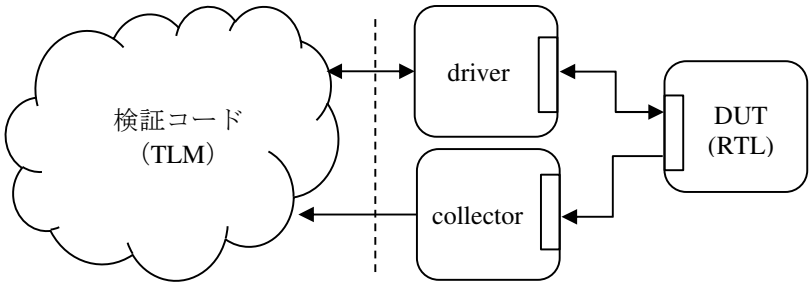


図 2-2　TLM と RTL の分離

2.2　UVM コンポーネント間の通信

2.2.1　概要

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を記述します。TLM-port と TLM-export の概念は、トランザクションを取得（get）する場合と作成（put）する場合では異なります。

TLM-port と TLM-export は一対一の関係を確立しますが、モニターの様に一対 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 analisys-port に対して複数個の analysis-exports を接続する事ができます。UVM では analysis-port を菱形◇で表現します。これらのポートの接続図を図 2-3 に示します。また、UVM で使用する TLM ポートを表 2-1 の様にまとめる事ができます。

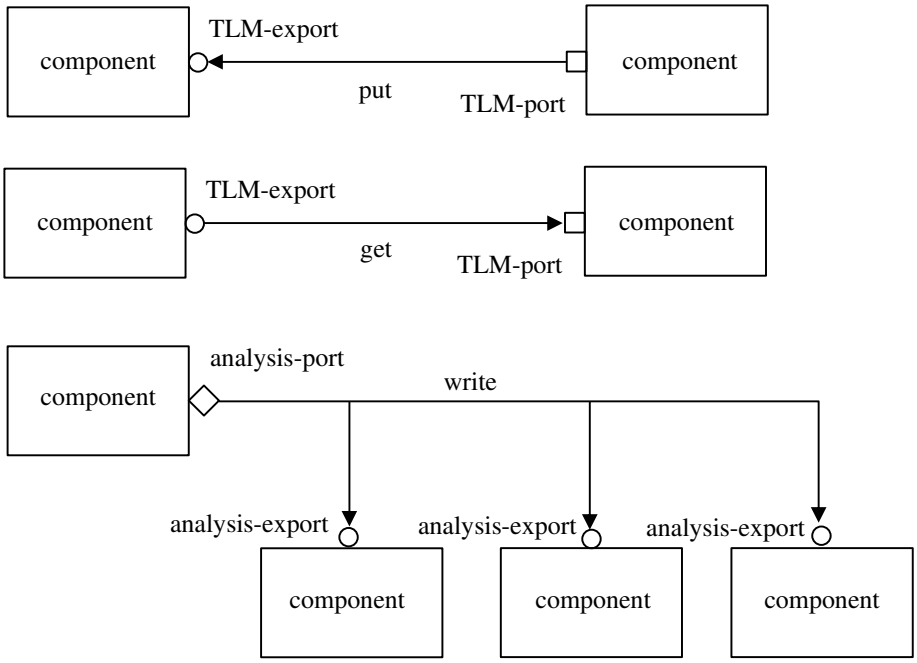


図 2-3　TLM-port、TLM-export、analysis-port、及び analysis-export（[2]）

表 2-1　TLM 接続に使用される記号一覧

port タイプ	記号	意味
TLM-port	□	トランザクションに対する動作を引き起こすポートを表現します。例えば、get()や put()メソッドの呼び出しが該当します。TLM-port でこれらのメソッドを呼び出すと、接続されている TLM-export に対して定義されているメソッドが起動されます。
TLM-export analysis-export	○	トランザクションに対する処理法を定義するポートを表現します。処理法を定義するタスク及びファンクションが該当します。export 側ではこれらのメソッドを実装しなければなりません。TLM-export に対してはタスクを定義し、analysis-export に対してはファンクションを定義します。
analysis-port	◇	一対 N の N 個のサブスクライバ―に対する動作を引き起こすポートを表現します。例えば、write()メソッドの呼び出しが該当します。N 個のサブスクライバ―では write()メソッドの内容を実装しなければなりません。

2.2.2　put 操作

UVM ではトランザクションを送信するための通信を図 2-4 の様に表現します。この操作では、put_producer が put 動作を起こす主人公になり、put_consumer は呼ばれる側で put 動作の内容を定義します。

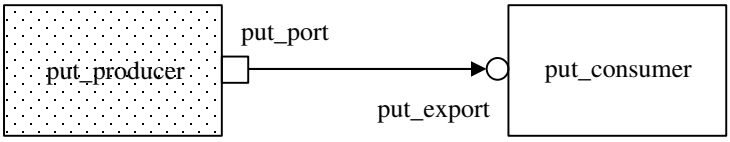


図 2-4　UVM の put 操作の表現法（[2]）

図 2-5 に示す様に、put_port の put()メソッドを呼び出すと、put_consumer に定義されている put()メソッドが起動します。従って、put_consumer はトランザクションを処理するための手順を put()メソッドに実装しなければなりません。

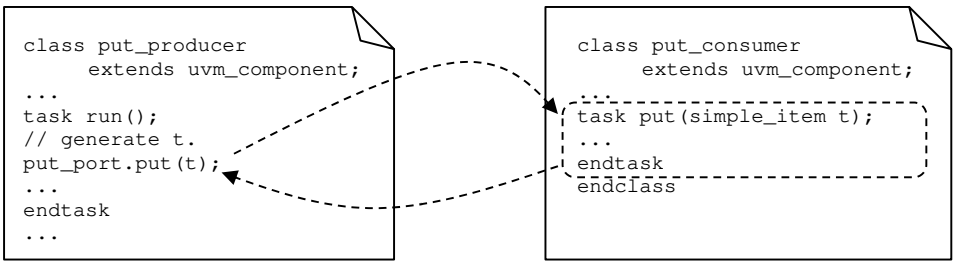


図 2-5　put_port と put_export の機能関連

put_port と put_export の接続は put_producer と put_consumer のインスタンスを持つ親コンポーネントが行います。従って、put_producer、及び put_consumer はお互いに他を意識していません。同じ種類の TLM-port と TLM-export のプロトコルを使用する限り、両者は常に接続可能です。

```
task tlm_fifo_parent::run_phase(uvm_phase phase);
    `uvm_info("TLM_FIFO_PARENT","Running ...",UVM_LOW)
endtask
```

この例で使用するパッケージ pkg は以下の様になります。

```
`ifndef PKG_H
`define PKG_H

package pkg;
import uvm_pkg::*;

`include "simple_data.sv"
`include "put_producer.sv"
`include "get_consumer.sv"
`include "tlm_fifo_parent.sv"

endpackage

`endif
```

最後に、トップモジュールを準備します。

```
`include "uvm_pkg.sv"
`include "uvm_macros.svh"
`include "pkg.sv"

module test;
import  uvm_pkg::*;
import  pkg::*;
tlm_fifo_parent  parent;

initial begin
    parent = tlm_fifo_parent::type_id::create("parent",null);
    run_test();
end
endmodule
```

コンパイルして実行すると、以下様な結果を得ます。トランザクションが FIFO を経由して consumer に引き渡されている事を確認する事ができます。

```
UVM_INFO tlm_fifo_parent.sv(29) @ 0: parent [TLM_FIFO_PARENT] Running ...
UVM_INFO put_producer.sv(13) @ 0: parent.producer [PUT_PRODUCER] Passing data.
UVM_INFO get_consumer.sv(14) @ 0: parent.consumer [GET_CONSUMER] Got data...
----------------------------------------
Name         Type           Size  Value
----------------------------------------
simple_data  simple_data_t  -     @385
  addr       integral       16    'hf25c
  data       integral       32    'hcd
----------------------------------------
```

■

2.2.5 analysis-ports と analysis-exports

モニター（コレクター）は受信専用で、DUT からのレスポンスの変化をトランザクションに変換し、他のコンポーネントに知らせます。知らせる相手（サブスクライバー）は不特定多数であるため、一対一の関係ではありません。従って、TLM-port、及び TLM-export の概念を適用する

事ができません。

UVM では、一対 N の関係を確立するために、analysis-ports、及び analysis-exports の機能を持っています。analysis-port は特殊な TLM-port で、それに接続している analysis-exports 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（つまり、サブスクライバー）の write()メソッドを呼び出してトランザクションを送信します（図 2-14）。

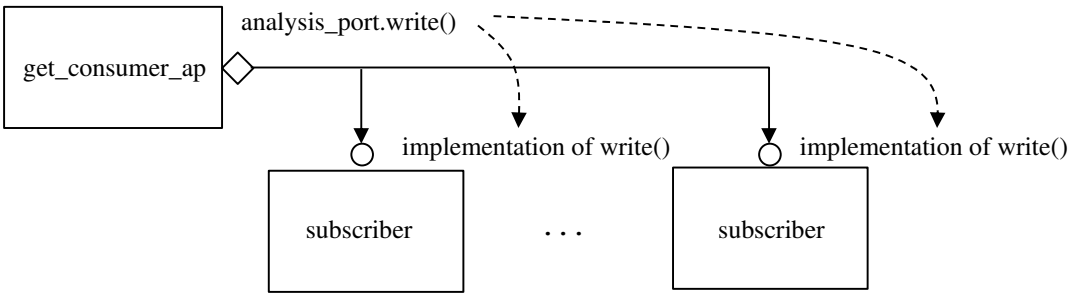


図 2-14　analysis-port によるトランザクションの送信（[2]）

put()や get()メソッドと異なり、write()メソッドはファンクションとして実装されなければなりません。換言すれば、原則として、write()メソッドでは時間を消費する事はできません。詰まり、トランザクションは全てのサブスクライバ―に同時に送信されます。

例 2-4　analysis-port と analysis-export の使用例

この例では、トランザクションを複数の検証コンポーネントに送信するために必要な analysis-port と analysis-export の機能を例示します。図 2-15 において、get_consumer_ap は get_producer からトランザクションを受信します。受信したトランザクションは、全てのサブスクライバ―に一斉に送信されます。以前開発した get_producer をそのまま使用する事ができる事に注意して下さい。構成要素は表 2-5 の様になります。

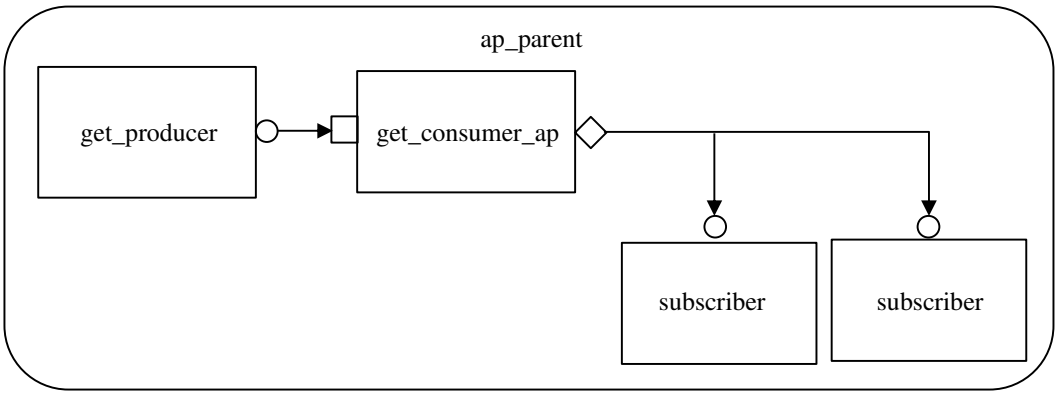


図 2-15　ap_parent コンポーネントの構成

表 2-5　ap_parent の構成要素

スコープ	機能
simple_data_t クラス	以前の例で開発したトランザクションを使用します。
get_producer クラス	以前の例で開発したクラスをそのまま使用する事ができます。一般的には、get_producer の代わりに uvm_tlm_fifo が接続されますが、この例では簡単のために get_producer を使用しています。
get_consumer_ap クラス	トランザクションを他のコンポーネントに送信します。N 個のコンポーネントに送信するためには、analysis_port を定義しなければなりません。analysis_port.write(data)はトランザクションを全てのサブスクライバ―に送信します。 また、このコンポーネントでは、get_producer からトランザクションを受け取るために get_port を定義しなければなりません。
subscriber クラス	トランザクションを受信するクラスです。analysis_export を定義して、subscriber クラス内に write()メソッドを定義しなければなりません。 ここでは、同じ subscriber のインスタンスを使用していますが、N 個のサブスクライバーが全て異なるタイプでも構いません。
ap_parent クラス	get_producer、get_consumer_ap、subscriber 等を接続するためのクラスです。build_phase()でそれらのインスタンスを作り、connect_phase()に於いてそれらの TLM 接続を行います。
pkg パッケージ	検証コンポーネントの使用を簡略化するために全てのクラスをパッケージに登録します。

先ず、サブスクライバ―にトランザクションを送信するコンポーネントを定義します。このクラスは以下の様な役割を持ちます。

- get_producer からトランザクションを受信するために、get_port を定義する。
- サブスクライバ―にトランザクションを送信するために、analysis_port を定義する。
- get_producer から受信したトランザクションを analysis_port.write(data)により、全てのサブスクライバ―に送信する。

get_consumer_ap クラスの定義は以下の様になります。コンストラクタで、analysis_port と get_port を作成しなければなりません。そして、run_phase()においてトランザクションを取得して、analysis_port.write(data)により他の検証コンポーネントにトランザクションを一斉に送信します。write()メソッドは function なので、全てのサブスクライバ―は同時にトランザクションを受信します。

```
class get_consumer_ap extends uvm_component;
uvm_analysis_port #(simple_data_t)        analysis_port;
uvm_blocking_get_port #(simple_data_t)    get_port;

`uvm_component_utils(get_consumer_ap)

function new(string name,uvm_component parent);
    super.new(name,parent);
    analysis_port = new("analysis_port",this);
    get_port = new("get_port",this);
endfunction

task run_phase(uvm_phase phase);
```

実行結果は以下の様になります。comp2 が simple_comp のインスタンスになっている事を確認する事ができます。

```
------------------------------
Name   Type       Size  Value
------------------------------
comp1  comp_base  -     @272
------------------------------
--------------------------------
Name   Type         Size  Value
--------------------------------
comp2  simple_comp  -     @281
  a    integral     8     'hd9
  b    integral     8     'haa
  sum  integral     8     'h0
  co   integral     1     'h0
--------------------------------
```

■

3.6 UVM シミュレーション

UVM を使用した検証環境では、UVM の定めたルールでシミュレーションが進行します。本節では、UVM によるシミュレーションを正しく進行させるための手順を解説します。具体的には、以下の技術を解説します。

- 実行制御を受けるシミュレーションフェーズの機能
- サブコンポーネントを作るための適切なタイミング
- raise_objection()と drop_objection()の役割と使用法

3.6.1 シミュレーションフェーズ

シミュレーションの実行制御は全て UVM が司り、ユーザ側に実行制御権はありません。但し、シミュレーションを開始するためにはテストベンチから UVM の run_test()メソッドを呼ばなければなりません。run_test()が実行を開始すると、ユーザが準備した UVM コンポーネントが順に呼び出されてシミュレーションが進行します（図 3-4）。

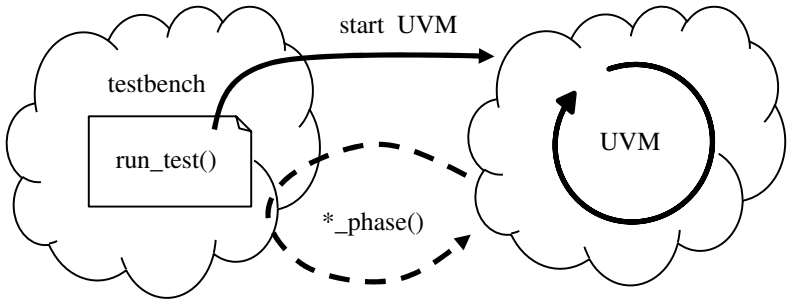


図 3-4　UVM を使用した環境での実行の流れ

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いてシミュレーションフェーズと呼ばれています。これらのフェーズは virtual タスク、又は、ファンクションとして定義されています。表 3-8 のシミュレーションフェーズは記述されている順に UVM から実行制御を受けます。例えば、run_test()が呼ばれて実行が開始すると、選択されているトップレベルのコンポーネントの build_phase()が呼ばれて階層構造を順に構築します。全てのコンポーネントに対して build_phase()が完了すると、構築された階層構造を基にして、connect_phase()が順に呼ばれて実行

```
function void my_agent::end_of_elaboration_phase(uvm_phase phase);
uvm_component    children[$];

    get_children(children);
    foreach(children[i])
        `uvm_info("MY_AGENT",children[i].get_full_name(),UVM_LOW)
endfunction
```

■

3.6.5 シミュレーションの進行

シミュレーションの実行に関して、UVM には二つの重要なメソッドがあります。それらは、raise_objection()と drop_objection()です（表 3-14）。簡単に言えば、これらのメソッドを呼び出さないと、UVM はシミュレーションを実行しないので、これらのメソッドを何れかのコンポーネントで呼び出さなければなりません。

表 3-14　シミュレーションの進行を制御するためのメソッド

メソッド	機能
raise_objection	UVM に実行すべき内容がある事を知らせます。
drop_objection	UVM に実行すべき内容が完了した事を知らせます。

全ての実行すべき内容が完了した時点で UVM は run_phase()を終了して、次のフェーズ（通常は、extract_phase()）に進みます。

raise_objection() で提起された全てのオブジェクションを drop しても、シミュレーションは直ぐには終了しません。UVM では drain time 等の概念をサポートしています。詳細に関しては UVM のマニュアル（[2]）を参照して下さい。

3.6.5.1 raise_objection()と drop_objection()

raise_objection() メソッドの使用法は以下の様になっています。

```
virtual function void raise_objection (
    uvm_object   obj = null,
    string       description = "",
    int          count = 1
)
```

コンポーネントのインスタンス（obj）のオブジェクション数を count だけインクリメントします。コンポーネント階層で上位にあるインスタンスのオブジェクション数も一斉にインクリメントされます。従って、何処かのインスタンスが raise_objection() を呼び出せば、UVM シミュレーションは進行します。obj==null の場合には、uvm_top が仮定されます。

drop_objection() メソッドの使用法は以下の様になっています。

```
virtual function void drop_objection (
    uvm_object   obj = null,
    string       description = "",
    int          count = 1
)
```

このメソッドは raise_objection() と正反対の動作をします。コンポーネントのインスタンス（obj）のオブジェクション数を count だけ減少します。コンポーネント階層で上位にあるインスタンスのオブジェクション数も一斉に減少されます。obj==null の場合には、uvm_top が仮定されます。

インスタンスのオブジェクションカウントが 0 の場合、drop_objection() を適用するとエラーになるので注意して下さい。

3.6.5.2 raise_objection()と drop_objection()の使用例

以下の様に raise_objection()、及び drop_objection() を使用します。

```
class testbench_comp extends uvm_component;
...
task run_phase(uvm_phase phase);
    phase.raise_objection(this); // tell UVM that there is a job
    do_something();
    phase.drop_objection(this);  // ok to proceed to next phase
endtask
endclass
```

重要な機能であるため、簡単な例を使用して二つのメソッドの使用効果を確認します。

例 3-17　raise_objection()と drop_objection()の使用例

先ず、これらのメソッドを使用した検証コンポーネントを定義します。この例に於ける実行は、一定時間待った後に、メッセージをプリントする簡単な処理内容となっています。

```
class comp_objection extends uvm_component;
`uvm_component_utils(comp_objection)

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

task run_phase(uvm_phase phase);
    phase.raise_objection(this); // tell UVM that there is a job
    do_something();
    phase.drop_objection(this);  // ok to proceed to next phase
endtask

task do_something();
    repeat( 3 ) begin
        #10 `uvm_info("COMP_OBJECTION",$sformatf("@%0t triggered",
                $time),UVM_LOW)
    end
endtask

endclass
```

トップモジュールを以下の様に定義します。

```
`include "uvm_pkg.sv"
`include "uvm_macros.svh"
`include "pkg.sv"

module top;
import uvm_pkg::*;
import pkg::*;
comp_objection   comp;
```

```
 operation_e  op_code;

 clocking cb @(posedge clk); endclocking

 endinterface
```
■

4.3　ドライバー

4.3.1　概要

ドライバーは以下の様な機能を持ちます。

- シーケンサーからトランザクションを取得する。
- 取得したトランザクションをシグナルレベルに変換する。
- ドライバーは virtual インターフェースを使用して DUT に信号を送る。インターフェースには DUT のポートが接続されているので、virtual インターフェースでその信号値を変更すると、DUT にその変化が反映されて DUT が反応する。

UVM では、シーケンサーとドライバーの接続は、図 4-1 の様に表現されます。

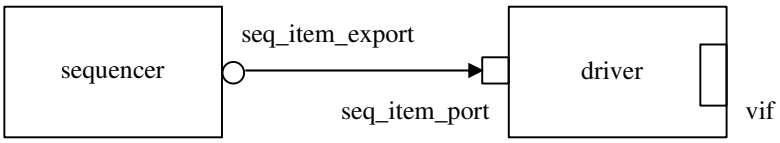


図 4-1　シーケンサーとドライバーの通信（[2]）

ドライバーを定義する手順をまとめると以下の様になります。

① uvm_driver、又は、そのサブクラスからドライバーを定義する。直接 uvm_driver から定義する場合には、トランザクションの型（REQ）、及びシーケンサーに戻すトランザクションの型（RSP）を指定する。
② virtual インターフェースを宣言する。
③ UVM マクロを定義する。
④ build_phase()又は connect_phase()で virtual インターフェースを設定する。
⑤ run_phase()にトランザクションの処理手順を定義する。

ベースクラスの uvm_driver は以下の様に定義されています。RSP を省略すると RSP==REQ と仮定されます。

```
class uvm_driver #(type REQ=uvm_sequence_item,
                   type RSP=REQ) extends uvm_component;
```

uvm_driver には、表 4-2 の様な重要なプロパーティが定義されているので、ドライバーの記述で自由に使用する事ができます。特に、seq_item_port はトランザクションを取得する際に使用する事ができるので非常に便利です。また、トランザクションをシーケンサーから受け取るために変数 req を指定する事ができます。この様に、重要なプロパーティが定義されている事実から、メソドロジークラスを使用する利点があります。もし、uvm_driver を使用しないと、全ての準備をユーザ自身で記述しなければなりません。

表 4-2　uvm_driver が備えている重要なプロパーティ

uvm_driver の重要なプロパーティ	意味
req	REQ 型のプロパーティでトランザクションを意味します。
rsp	RSP 型のプロパーティでシーケンサーに戻すレスポンスを指定する際に使用します。
seq_item_port	シーケンサーと通信するための TLM-port です。 この TLM-port は uvm_driver のコンストラクタで作成されます。

実際問題として、seq_item_port は uvm_driver クラス内で以下の様に定義されています。

```
uvm_seq_item_pull_port #(REQ, RSP) seq_item_port;
```

ドライバーはトランザクションを取得するために、get_next_item()メソッドを呼び出します。すると、uvm_sequencer に定義されている get_next_item()メソッドが呼ばれてトランザクションが準備されます（図 4-2）。

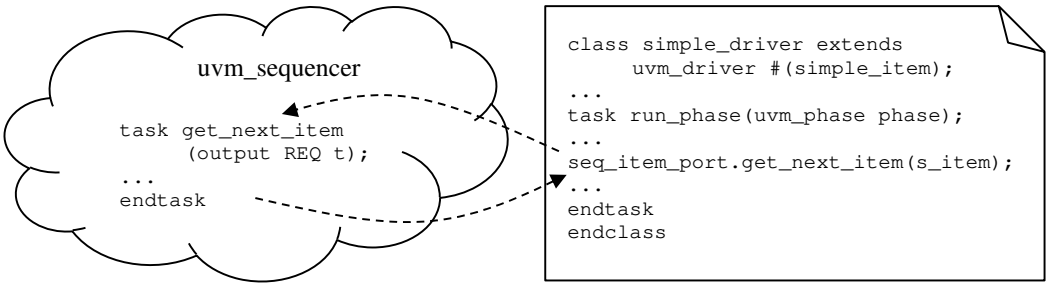


図 4-2　ドライバーがトランザクションを取得する流れ

ドライバーには二つのアプローチがあります。全ての機能を含んだドライバーを必要に応じて記述する標準的な方法と、ドライバーの一般的な機能をベースクラスと定義して再利用可能にする方法があります。両者の方法を順に解説します。

4.3.2　標準的なドライバー記述法

標準的なアプローチでドライバーを記述する事から始めます。即ち、ドライバーのクラスに全ての機能を実装する様にします。

例 4-7　標準的なアプローチによるドライバー記述例

ドライバーの全容は以下の様になります。

```
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
```

```
    // Execute the item.
    drive_item(req);
    // tell sequencer that item is done.
    seq_item_port.item_done();
endtask
```

drive_item()タスクでは virtual インターフェースを使用して DUT の入力信号に変更を与えます。この変更に反応して DUT が実行します。

```
task simple_driver::drive_item(input simple_item item);
    vif.a <= item.a;
    vif.b <= item.b;
    vif.op_code <= item.op_code;
endtask
```

■

4.4 シーケンス

4.4.1 概要

シーケンスは単純にトランザクションを作るだけでなく、他のシーケンスと組み合わせて複雑なシナリオを生成する事もできます。例えば、シーケンス 1 とシーケンス 2 から複雑なシーケンスを構築する事ができます。この機能を利用する事によりシナリオの再利用性は拡大します。個々の目的別シーケンスをライブラリーとして準備し、ライブラリーに登録されている小さなシーケンスを組み合わせて意義のある大きなシナリオを構築する事ができます。図 4-4 は 3 つのシーケンスを組み合わせ一つのシナリオを構築している様子を示しています。シーケンス main sequence は階層のルートシーケンスになります。

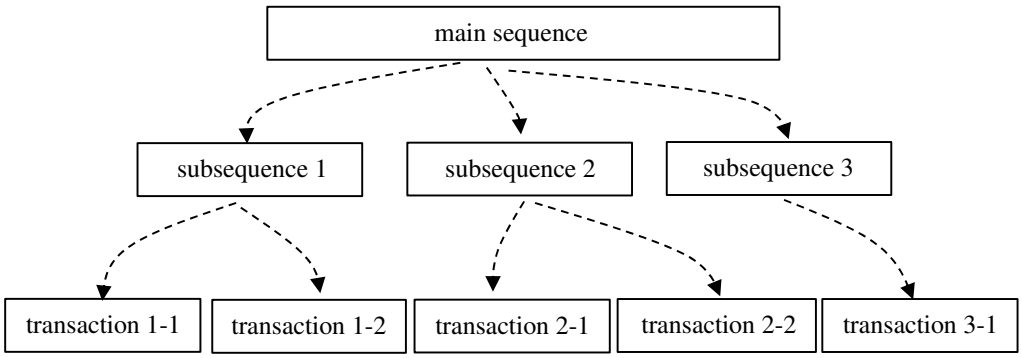


図 4-4　階層的シーケンスに依るシナリオ構築

4.4.2 シーケンスの定義手順

テストシナリオの作成はシーケンスの作成に尽きます。シーケンスが実行する事により有意義なトランザクションが作成されます。シーケンスを定義する手順は以下の様になります。

① uvm_sequence、又は、そのサブクラスからシーケンスを定義する。その際、トランザクションのタイプ（REQ）、及びドライバーがシーケンサーに戻すレスポンスのタイプ（RSP）を指定する。
② シーケンスに対して、`uvm_object_utils マクロを定義する。シーケンスは、コンポーネントではないので、`uvm_component_utils マクロを使用する事はできません。
③ 必要に応じて、`uvm_declare_p_sequencer マクロを使用してシーケンサーへのポインターを

宣言する。
④ シーケンスに body()タスクを定義する。

ベースクラスの uvm_sequence は以下の様に定義されています。

```
virtual class uvm_sequence #(type REQ = uvm_sequence_item,
                type RSP = REQ) extends uvm_sequence_base;
```

uvm_sequence クラスはプロパーティとして req（REQ 型）、及び rsp（RSP 型）を以下の様に確保するので、ユーザが定義したシーケンス内ではそれらの変数を自由に使用する事ができます。

```
virtual class uvm_sequence #(type REQ = uvm_sequence_item,
    type RSP = REQ) extends uvm_sequence_base;
...
REQ     req;
RSP     rsp;
...
endclass
```

実際問題として body()タスクを除くと、シーケンスは殆ど標準的な記述になります。

4.4.3 body()タスク

一般的に、body()タスクの内容は以下の様になります。

```
task body();
    repeat( num_seq )
        `uvm_do(req)
endtask
```

即ち、シーケンスはトランザクションを作りシーケンサーに渡します。これを num_seq 回繰り返して終了します。`uvm_do マクロに加えて`uvm_do_with マクロを使用する事ができます。num_seq の値は、シーケンスが作られた後に randomize()メソッドで決定されます。

4.4.4 `uvm_do マクロと`uvm_do_with マクロ

`uvm_do マクロは以下の機能を展開します。

- トランザクションの要求があるまで待ちます。
- トランザクションの要求があると、トランザクションを生成しトランザクションのフィールドに乱数を発生させます。
- トランザクションをシーケンサーに送信し、シーケンサーはドライバーにトランザクションを送信します。
- ドライバーが item_done()でトランザクションの処理を終了するのを待ちます。

別のマクロ`uvm_do_with はトランザクションの乱数発生に対する制約を追加する場合に使用します。例えば、以下の様に制約を追加する事ができます。

```
`uvm_do_with(req,{(req.a > 2} && (req.b > 2);})
```

ここで、制約を{…}で指定しましたが、制約内はトランザクションのスコープを意味するので、以下の様に簡略化した記述が可能です。但し、a、及び b と同じ名称がシーケンス内に定義されている時は、この簡略記述法を避けなければなりません。