Artgraphics

SCL

検証のための SystemVerilog クラスライブラリー

Document Identification Number: ARTG-TD-001-2023
Document Revision: 1.3, 2024.02.05
アートグラフィックス

SCL
検証のための SystemVerilog クラスライブラリー

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

SystemVerilog Class Library (SCL) for Verification

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

弊社製品、および、業務案内に関するご質問には下記の連絡先をご利用下さい。
連絡先：contact.us@artgraphics.co.jp

はじめに

本書は、SystemVerilog Class Library (SCL) の解説書です。SCL は小規模な検証パッケージなので、全体を理解する事は容易です。更に、SCL の軽量性はコンパイルの高速化と実行モジュールサイズの小型化に通じ、TAT の短縮効果があります。

UVM は SystemVerilog で記述された優れた検証パッケージですが、UVM の難易度が高い事も事実です。このため、多くの学習時間が初期コストとして発生し、UVM を十分に駆使する事ができる状況に至るまでの道のりが遠く見えるのが現状です。一方、膨大な行数からなる UVM の解読は殆ど不可能に近いだけでなく、基本的な機能に関しても多くのミステリーが存在します。UVM には深い理由があり現在のような複雑な実装になっていると考えられますが、高度な機能を必要としない検証処理では、むしろ簡潔明瞭な処理方式の方が望ましいと言えます。

SCL は、簡潔明瞭で誰でも短期間に理解できる検証ライブラリーの構造を備えています。例えば、UVM のクラス階層と異なり、SCL は理路整然としたクラス階層を実現しています。更に、UVM のトランザクション処理では、重要な処理部分が表面から隠されているため、殆どの人が理解し得ない難攻不落の要塞になっています。UVM を開発した者しか理解し得ないような処理方式は一般的なライブラリーとしては不適切です。SCL は、その様な設計思想を排除し、誰にでも分かり易い処理方式を採用しています。具体的には、SCL には以下のような特徴があります。

- UVM のクラス階層と異なり、理路整然としたクラス階層を実現。
- マクロを使用し記述を簡単に記述できるように入力の省力化。
- TLM ポートを標準装備したコレクタークラスを追加。
- モニタークラスには標準的な TLM ポートを定義。
- トランザクション処理には UVM とは異なる方式を採用し、トランザクションの流れを可視化。
- virtual インターフェースの使用手順を簡素化。
- 検証作業に重宝する汎用的な機能とユーティリティを追加。

例えば、virtual インターフェースの使用法が如何に簡単であるかを示します。先ず、インターフェースを定義した時に、以下のように SCL マクロを使用して virtual インターフェースを操作するための vif_config クラスを生成します。

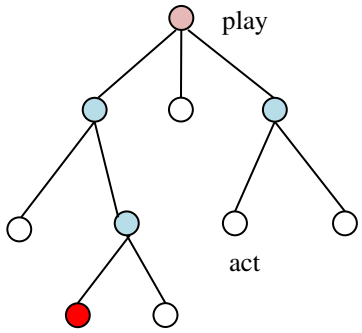
```
interface simple_if;
...
endinterface
`scl_vif_config_m(virtual simple_if,vif_config)
```

こうすると、トップモジュールではインターフェースのインスタンスを以下のように設定できます。

```
vif_config::set(SIF);
```

そして、検証コンポーネントでは、以下に示すような簡単な手順で virtual インターフェースを取得できます。こうする事により、検証コンポーネント内でインターフェース名をハードコーディングしないで済みます。

```
vif = vif_config::get();
```



もう一つ別の例を紹介します。UVM のトランザクション生成手順は非常に複雑であり理解し難いという欠点があります。その欠点を克服するために、SCL ではトランザクションの生成手順をツリーで表現します。ツリーのルートは play と呼ばれるオブジェクトで、他のノードは act と呼ばれるオブジェクトで表現されます。ツリーの内部ノードはトランザクションの生成手順を示し、リーフノードは一つのトランザクションを生成する役目を持ちます。テストがシナリオを決定するとジェネレータに play が割り当てられて右図のようなツリーが構築されてトランザクションを生成するための手順が確定します。ドライバーがジェネレータにトランザクション取得要求を発行するとリーフノードが呼ばれトランザクションが生成されてドライバーに戻されます。赤いリーフノードは、次にトランザクションを作る役目を持つノードを示しています。赤いリーフノードの処理が終わるとジェネレータは右隣のリーフノードを次の処理対象に設定します。このようにトランザクションとリーフノードが対応しているので、トランザクション生成の仕組みは理解し易くなっています。

SCL では、一般的に普及されている用語を踏襲し、SCL 独自のアーキテクチャに依存する部分には SCL 固有の用語を使用します。例えば、トランザクションを処理するモデリングには一般的な TLM という用語を使用します。一方、トランザクション処理を制御する機能にはシナリオという用語を使用します。SCL は UVM とはトランザクション処理が異なるので、混乱を避けるために UVM と同じ用語を使用するのを避けています。例えば、SCL では UVM で使用されているシーケンスやシーケンサーという用語を使用していません。また、検証の対象である DUT は RTL で記述されていると仮定します。

本書は、幾つかの章から構成されていますが、第 1 章が最も重要な内容になっています。内容的には SCL の概要的な解説であるため、他の章を読むまでは理解できない部分もあると思います。既に触れたように、SCL の特徴は virtual インターフェースの使用手順とトランザクション生成処理にあるので、第 3 章、第 4 章、第 8 章は非常に重要な意味を持ちます。これらの章は丁寧にお読みください。第 9 章では、SCL を使用して検証環境を構築する例を紹介しているので、必要に応じて参照して下さい。

SCL には、トランザクションや検証コンポーネントの他に検証作業で必要になる汎用的な機能も含まれています。例えば、パターンマッチング機能、文字列操作機能、プリント書式機能、クロック生成機能、プロセス生成機能、プライオリティキュー、ファイル入出力等の機能も SCL に含まれています。特に、SCL は強力なプリント機能を備えているので、殆どのプリント処理は省力化されます。例えば、プリント用のレイアウトをヘッダと呼ばれるデータ構造で定義すると、カラムに表示する値を設定するだけで、カラムの自動調節が行われます。そして、2 進や 16 進の表現形式の変更は使用するマクロ名を変更するだけで済みます。また、ビット幅を変更しても自動的にカラム調節が行われます。第 10 章はこれらの便利な機能を詳細な使用例と共に解説をしています。

尚、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。ただし、書体が小さいからと言って内容の重要性が低いわけではありません。

アートグラフィックス

変更履歴

日付	Revision	変更点
2023.12.10	1.0	初版。
2024.01.08	1.1	• パターンマッチング機能（正則表現機能）を追加。 • コンフィギュレーション設定変更機能を追加。
2024.01.25	1.2	• オブジェクトとコンポーネントにコピー機能を追加したので、トランザクションのコピー等が簡単にできようになりました。 • プリント書式の自動カラム調節機能を追加しました。 • スコアボードに `predict()` メソッドを追加しました。 • スコアボードを使用して検証環境を構築する例を追加しました。
2024.02.05	1.3	• ノンブロッキング方式によるトランザクション送受信するためのポート機能を追加。 • トランザクションをプーリングする FIFO 機能を追加。

目次

1 概要 1

1.1 SCL の意義と目的 1
1.2 検証環境を構成する概念 1
1.3 インターフェースと VIRTUAL インターフェース 2
1.4 VIRTUAL インターフェースの操作 2
1.5 ダイナミックな検証環境 3
1.6 ドライバーとトランザクション 3
1.7 トランザクションの操作 4
1.7.1 トランザクションを取得する手順 4
1.7.2 トランザクションを送信する手順 5
1.7.3 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手順 5
1.8 トランザクションの生成 5
1.9 テストとシナリオ 6
1.10 検証環境の構成 6
1.11 ダイナミックなコンフィギュレーション設定変更 8
1.12 SCL マクロ 8
1.13 シミュレーション 10
1.13.1 実行制御 11
1.13.2 scl_run_test 12
1.13.3 scl_set_timeout 12
1.13.4 scl_set_max_simulation_time 13
1.13.5 scl_get_test_name 13
1.14 SCL の使用手順 13
1.15 検証環境の開発法 13
1.16 SCL の記法 14
1.17 本書の記法 14

2 SCL の構成 15

2.1 クラス階層 15
2.2 SCL_VOID_T 15
2.3 SCL_THING_T 16
2.3.1 new 17
2.3.2 get_name 17
2.3.3 get_id 17
2.3.4 get_type_name 17
2.3.5 is_object 17
2.3.6 is_component 18
2.3.7 scl_field_processor 18
2.3.8 print 18
2.3.9 copy 18
2.4 SCL_OBJECT_T 18
2.4.1 new 19
2.4.2 get_type_name 19
2.4.3 is_object 19
2.4.4 print 19
2.4.5 copy 19
2.5 SCL_COMPONENT_T 20
2.5.1 new 22
2.5.2 get_type_name 22
2.5.3 is_component 22
2.5.4 add_child 22
2.5.5 add_object 22
2.5.6 get_parent 23

2.5.7 get_objects 23
2.5.8 get_child 23
2.5.9 get_first_child 23
2.5.10 get_last_child 23
2.5.11 get_next_child 23
2.5.12 get_prev_child 23
2.5.13 get_name 23
2.5.14 get_full_name 24
2.5.15 print 24
2.5.16 copy 24
2.5.17 is_root 24
2.5.18 create_top 24
2.5.19 build_phase 24
2.5.20 connect_phase 24
2.5.21 setup_phase 24
2.5.22 run_phase 25
2.5.23 conclude_phase 25
2.5.24 final_phase 25
2.6 オブジェクトとコンポーネントの生成 25
2.6.1 `scl_create_object_m 25
2.6.2 `scl_create_component_m 25
2.7 オブジェクトとコンポーネントの定義確認 26
2.7.1 scl_get_object_types 26
2.7.2 scl_get_component_types 26
2.7.3 オブジェクトとコンポーネントの定義確認例 26
2.8 SCL_GET_INSTANCES_BY_PREORDER 27
2.9 コンフィギュレーション設定変更 28
2.9.1 クラスタイプの変更 29
2.9.2 クラスプロパーティの変更 29

3 TLM 32

3.1 ポート 32
3.1.1 scl_port_t 33
3.1.1.1 new 34
3.1.1.2 get_type_name 34
3.1.1.3 set_option 34
3.1.1.4 clear_option 35
3.1.1.5 is_resuse_transaction 35
3.1.2 scl_io_port_t 35
3.1.2.1 new 36
3.1.2.2 get_type_name 36
3.1.2.3 connect 36
3.1.2.4 get 36
3.1.2.5 put 36
3.1.3 scl_nbio_port_t 36
3.1.3.1 new 37
3.1.3.2 get_type_name 37
3.1.3.3 connect 37
3.1.3.4 try_get 37
3.1.3.5 try_put 38
3.1.4 scl_pass_port_t 38
3.1.4.1 new 38
3.1.4.2 get_type_name 39
3.1.4.3 write 39
3.2 GET 39
3.2.1 scl_get_port_t 39
3.2.1.1 new 40
3.2.1.2 get_type_name 40
3.2.1.3 get 40

3.2.2 scl_get_server_t 40
3.2.2.1 new 41
3.2.2.2 get_type_name 41
3.2.2.3 get 41
3.2.3 get の使用例 41
3.3 PUT 43
3.3.1 scl_put_port_t 43
3.3.1.1 new 44
3.3.1.2 get_type_name 44
3.3.1.3 put 44
3.3.2 scl_put_server_t 44
3.3.2.1 new 45
3.3.2.2 get_type_name 45
3.3.2.3 put 45
3.3.3 put の使用例 45
3.4 SEND 46
3.4.1 scl_send_port_t 47
3.4.1.1 new 48
3.4.1.2 get_type_name 48
3.4.1.3 add_receiver 48
3.4.1.4 send 48
3.4.2 scl_receive_port_t 48
3.4.2.1 new 49
3.4.2.2 get_type_name 49
3.4.2.3 write 49
3.4.3 send の使用例 50
3.5 TRY_GET 51
3.5.1 scl_nbget_port_t 51
3.5.1.1 new 52
3.5.1.2 get_type_name 52
3.5.1.3 try_get 52
3.5.2 scl_nbget_server_t 52
3.5.2.1 new 53
3.5.2.2 get_type_name 53
3.5.2.3 try_get 53
3.5.3 try_get の使用例 53
3.6 TRY_PUT 54
3.6.1 scl_nbput_port_t 54
3.6.1.1 new 55
3.6.1.2 get_type_name 55
3.6.1.3 try_put 55
3.6.2 scl_nbput_server_t 55
3.6.2.1 new 56
3.6.2.2 get_type_name 56
3.6.2.3 try_put 56
3.6.3 try_put の使用例 56
3.7 FIFO 57
3.7.1 scl_fifo_t 57
3.7.1.1 new 59
3.7.1.2 put 59
3.7.1.3 try_put 59
3.7.1.4 get 59
3.7.1.5 try_get 60
3.7.1.6 fifo_size 60
3.7.1.7 num_items 60
3.7.1.8 is_empty 60
3.7.1.9 is_full 60
3.7.1.10 can_put 60
3.7.1.11 can_get 60
3.7.1.12 empty 60
3.7.2 FIFO の使用例 60

3.8 メソドロジークラスと TLM ポート 64
4 VIRTUAL インターフェースの設定と取得 65
4.1 VIRTUAL インターフェースの使用準備 65
4.2 トップモジュールにおける VIRTUAL インターフェースの設定準備 66
4.3 VIRTUAL インターフェースの取得 66
5 トランザクション 68
5.1 SCL_TRANSACTION_T 68
5.1.1 new 68
5.1.2 get_type_name 68
5.2 トランザクションの定義 69
6 フィールドマクロ 71
6.1 単数形変数 72
6.1.1 `scl_field_integral_m 72
6.1.2 ` scl_field_byte_m 72
6.1.3 `scl_field_shortint_m 73
6.1.4 `scl_field_int_m 73
6.1.5 `scl_field_longint_m 73
6.1.6 `scl_field_integer_m 73
6.1.7 `scl_field_time_m 73
6.1.8 `scl_field_string_m 73
6.1.9 `scl_field_real_m 73
6.1.10 `scl_field_enum_m 73
6.1.11 `scl_field_object_m 73
6.1.12 単数形変数のフィールドマクロ使用例 74
6.2 アレイ変数 74
6.3 固定サイズのアレイ 75
6.3.1 `scl_field_fa_integral_m 75
6.3.2 `scl_field_fa_byte_m 75
6.3.3 `scl_field_fa_shortint_m 75
6.3.4 `scl_field_fa_int_m 75
6.3.5 `scl_field_fa_longint_m 75
6.3.6 `scl_field_fa_integer_m 75
6.3.7 `scl_field_fa_time_m 75
6.3.8 `scl_field_fa_string_m 75
6.3.9 `scl_field_fa_real_m 75
6.3.10 `scl_field_fa_enum_m 76
6.3.11 `scl_field_fa_object_m 76
6.3.12 固定サイズのアレイのフィールドマクロ使用例 76
6.4 ダイナミックアレイ 77
6.4.1 `scl_field_da_integral_m 77
6.4.2 `scl_field_da_byte_m 77
6.4.3 `scl_field_da_shortint_m 77
6.4.4 `scl_field_da_int_m 77
6.4.5 `scl_field_da_longint_m 77
6.4.6 `scl_field_da_integer_m 77
6.4.7 `scl_field_da_time_m 77
6.4.8 `scl_field_da_string_m 78
6.4.9 `scl_field_da_real_m 78
6.4.10 `scl_field_da_enum_m 78
6.4.11 `scl_field_da_object_m 78
6.5 キュー 78
6.5.1 `scl_field_queue_integral_m 78
6.5.2 `scl_field_queue_byte_m 78

6.5.3 `scl_field_queue_shortint_m 78
6.5.4 `scl_field_queue_int_m 78
6.5.5 `scl_field_queue_longint_m 79
6.5.6 `scl_field_queue_integer_m 79
6.5.7 `scl_field_queue_time_m 79
6.5.8 `scl_field_queue_string_m 79
6.5.9 `scl_field_queue_real_m 79
6.5.10 `scl_field_queue_enum_m 79
6.5.11 `scl_field_queue_object_m 79
6.6 ASSOCIATIVE アレイ 79
6.6.1 インデックスが整数系の associative アレイ 79
6.6.1.1 `scl_field_aa_numeric_integral_m 79
6.6.1.2 `scl_field_aa_numeric_byte_m 80
6.6.1.3 `scl_field_aa_numeric_shortint_m 80
6.6.1.4 `scl_field_aa_numeric_int_m 80
6.6.1.5 `scl_field_aa_numeric_longint_m 80
6.6.1.6 `scl_field_aa_numeric_integer_m 80
6.6.1.7 `scl_field_aa_numeric_time_m 80
6.6.1.8 `scl_field_aa_numeric_string_m 80
6.6.1.9 `scl_field_aa_numeric_real_m 80
6.6.1.10 `scl_field_aa_numeric_enum_m 80
6.6.1.11 `scl_field_aa_numeric_object_m 81
6.6.2 インデックスが実装型の associative アレイ 81
6.6.2.1 `scl_field_aa_real_integral_m 81
6.6.2.2 `scl_field_aa_real_byte_m 81
6.6.2.3 `scl_field_aa_real_shortint_m 81
6.6.2.4 `scl_field_aa_real_int_m 81
6.6.2.5 `scl_field_aa_real_longint_m 81
6.6.2.6 `scl_field_aa_real_integer_m 81
6.6.2.7 `scl_field_aa_real_time_m 81
6.6.2.8 `scl_field_aa_real_string_m 81
6.6.2.9 `scl_field_aa_real_real_m 82
6.6.2.10 `scl_field_aa_real_enum_m 82
6.6.2.11 `scl_field_aa_real_object_m 82
6.6.3 インデックスが string 型の associative アレイ 82
6.6.3.1 `scl_field_aa_string_integral_m 82
6.6.3.2 `scl_field_aa_string_byte_m 82
6.6.3.3 `scl_field_aa_string_shortint_ 82
6.6.3.4 `scl_field_aa_string_int_m 82
6.6.3.5 `scl_field_aa_string_longint_m 82
6.6.3.6 `scl_field_aa_string_integer_m 83
6.6.3.7 `scl_field_aa_string_time_m 83
6.6.3.8 `scl_field_aa_string_string_m 83
6.6.3.9 `scl_field_aa_string_real_m 83
6.6.3.10 `scl_field_aa_string_enum_m 83
6.6.3.11 `scl_field_aa_string_object_m 83
6.6.4 インデックスが enum 型の associative アレイ 83
6.6.4.1 `scl_field_aa_enum_integral_m 83
6.6.4.2 `scl_field_aa_enum_byte_m 83
6.6.4.3 `scl_field_aa_enum_shortint_m 83
6.6.4.4 `scl_field_aa_enum_int_m 84
6.6.4.5 `scl_field_aa_enum_longint_m 84
6.6.4.6 `scl_field_aa_enum_integer_m 84
6.6.4.7 `scl_field_aa_enum_time_m 84
6.6.4.8 `scl_field_aa_enum_string_m 84
6.6.4.9 `scl_field_aa_enum_real_m 84
6.6.4.10 `scl_field_aa_enum_enum_m 84
6.6.4.11 `scl_field_aa_enum_object_m 84
6.6.5 インデックスがオブジェクトの associative アレイ 84
6.6.5.1 `scl_field_aa_object_integral_m 85
6.6.5.2 `scl_field_aa_object_byte_m 85
6.6.5.3 `scl_field_aa_object_shortint_m 85

6.6.5.4 `scl_field_aa_object_int_m 85
6.6.5.5 `scl_field_aa_object_longint_m 85
6.6.5.6 `scl_field_aa_object_integer_m 85
6.6.5.7 `scl_field_aa_object_time_m 85
6.6.5.8 `scl_field_aa_object_string_m 85
6.6.5.9 `scl_field_aa_object_real_m 85
6.6.5.10 `scl_field_aa_object_enum_m 86
6.6.5.11 `scl_field_aa_object_object_m 86
6.6.6 associative アレイのフィールドマクロ使用例 86

7 検証コンポーネント 88

7.1 ドライバー 88
7.1.1 ドライバーのベースクラス 88
7.1.1.1 new 89
7.1.1.2 get_type_name 89
7.1.2 ドライバーの定義法 89
7.2 ジェネレータ 90
7.2.1 ジェネレータのベースクラス 90
7.2.1.1 new 91
7.2.1.2 get_type_name 91
7.2.1.3 get 91
7.2.2 ジェネレータの定義法 92
7.3 コレクター 92
7.3.1 コレクターのベースクラス 92
7.3.1.1 new 93
7.3.1.2 get_type_name 93
7.3.2 コレクターの定義法 93
7.4 モニター 94
7.4.1 モニターのベースクラス 94
7.4.1.1 new 94
7.4.1.2 get_type_name 95
7.4.1.3 build_phase 95
7.4.1.4 write 95
7.4.2 モニターの定義法 95
7.5 エージェント 96
7.5.1 エージェントのベースクラス 96
7.5.1.1 new 97
7.5.1.2 get_type_name 97
7.5.1.3 build_phase 97
7.5.2 エージェントの定義法 97
7.6 エンバイロンメント 98
7.6.1 エンバイロンメントのベースクラス 98
7.6.1.1 new 99
7.6.1.2 get_type_name 99
7.6.2 エンバイロンメントの定義法 99
7.7 スコアボード 99
7.7.1 スコアボードのベースクラス 99
7.7.1.1 new 100
7.7.1.2 get_type_name 100
7.7.1.3 write 100
7.7.1.4 predict 100
7.7.2 スコアボードの定義法 101
7.8 テスト 101
7.8.1 テストのベースクラス 101
7.8.1.1 new 102
7.8.1.2 get_type_name 102
7.8.2 テストの定義法 102
7.9 一般的な検証コンポーネントの定義法 103

7.10 検証コンポーネント用マクロ一覧 104
7.10.1 `scl_component_m 104
7.10.2 `scl_component_new_m 104
7.10.3 `scl_extern_build_phase_m 105
7.10.4 `scl_build_phase_m 105
7.10.5 `scl_end_build_phase_m 105
7.10.6 `scl_extern_connect_phase_m 105
7.10.7 `scl_connect_phase_m 105
7.10.8 `scl_end_connect_phase_m 106
7.10.9 `scl_extern_setup_phase_m 106
7.10.10 `scl_setup_phase_m 106
7.10.11 `scl_end_setup_phase_m 106
7.10.12 `scl_extern_run_phase_m 106
7.10.13 `scl_run_phase_m 106
7.10.14 `scl_end_run_phase_m 107
7.10.15 `scl_extern_conclude_phase_m 107
7.10.16 `scl_conclude_phase_m 107
7.10.17 `scl_end_conclude_phase_m 107
7.10.18 `scl_extern_final_phase_m 107
7.10.19 `scl_final_phase_m 107
7.10.20 `scl_end_final_phase_m 108
7.10.21 `scl_generator_m 108
7.10.22 `scl_agent_m 108

8 検証手順とシナリオ 110

8.1 シナリオ 110
8.1.1 シナリオのベースクラス 110
8.1.1.1 new 111
8.1.1.2 get_type_name 111
8.1.1.3 allocate 111
8.1.1.4 get 111
8.1.2 シナリオの使用法 111
8.2 プレイ 112
8.2.1 プレイのベースクラス 112
8.2.1.1 new 113
8.2.1.2 get_type_name 113
8.2.1.3 reuse_transaction 114
8.2.1.4 new_item 114
8.2.1.5 pre_play 114
8.2.1.6 post_play 114
8.2.1.7 get_item 114
8.2.2 プレイの定義法 114
8.3 アクト 116
8.3.1 アクトのベースクラス 116
8.3.1.1 new 117
8.3.1.2 get_type_name 117
8.3.1.3 set_play 117
8.3.1.4 get_item 117
8.3.1.5 get_group 117
8.3.1.6 make_item 117
8.3.2 アクトの定義法 117
8.3.3 アクトを生成するマクロ 118
8.3.3.1 `scl_create_act_m 119
8.3.3.2 `scl_act_get_item_m 119
8.3.3.3 `scl_act_get_group_m 119
8.4 ドライバー、ジェネレータ、プレイによるハンドシェーク 119
8.4.1 テストとシナリオ 119
8.4.2 シミュレーション実行開始の動作 119

8.4.3 ジェネレータとアクトのハンドシェーク 120
9 検証環境構築例 122
9.1 検証環境（モニターによる検証） 122
9.1.1 up_down_counter 123
9.1.2 pkg_definitions 124
9.1.3 simple_if 124
9.1.4 simple_item_t 124
9.1.5 act_down_t 125
9.1.6 act_load_t 125
9.1.7 act_loaddown_t 125
9.1.8 act_reset_t 126
9.1.9 act_up_t 126
9.1.10 act_upup_t 127
9.1.11 act_upuploaddown_t 127
9.1.12 play_t 127
9.1.13 play_reset_down_t 128
9.1.14 play_reset_up_t 128
9.1.15 driver_t 129
9.1.16 generator_t 130
9.1.17 collector_t 130
9.1.18 monitor_t 131
9.1.19 agent_t 132
9.1.20 env_t 133
9.1.21 test_base_t 133
9.1.22 test1_t 134
9.1.23 test2_t 134
9.1.24 pkg 134
9.1.25 top 135
9.1.26 テストの実行 136
9.1.26.1 +scl_testname=test1_t 136
9.1.26.2 +scl_testname=test2_t 136
9.2 検証環境（スコアボードによる検証） 136
9.2.1 パッケージ 137
9.2.2 モニター 138
9.2.3 エンバイロンメント 138
9.2.4 スコアボード 139
9.2.5 トップモジュール 141
9.2.6 テストの実行 141
9.2.6.1 +scl_testname=test1_t 141
9.2.6.2 +scl_testname=test2_t 142
9.3 検証結果の一括出力法 142
10 検証支援機能 145
10.1 データタイプと定数 145
10.1.1 enum タイプ 145
10.1.1.1 scl_agent_function_e 145
10.1.1.2 scl_ascii_mode_e 146
10.1.1.3 scl_print_option_e 146
10.1.1.4 scl_print_message_option_e 146
10.1.1.5 scl_field_option_e 147
10.1.1.6 scl_finish_code_e 147
10.1.1.7 scl_item_status_e 147
10.1.1.8 scl_level_e 148
10.1.1.9 scl_port_option_e 148
10.1.1.10 scl_print_field_e 148
10.1.1.11 scl_return_e 149
10.1.1.12 scl_sprint_string_e 149

10.1.1.13 scl_re_state_e 149
10.1.1.14 scl_re_code_e 150
10.1.1.15 scl_header_option_e 150
10.1.2 クラスとストラクチャ 151
10.1.2.1 scl_heap_item_t 152
10.1.2.2 scl_heap_t（プライオリティキュー用のデータ構造） 152
10.1.2.3 scl_max_heap_t（プライオリティキューの実装） 154
10.1.2.4 scl_min_heap_t（プライオリティキューの実装） 154
10.1.2.5 scl_instance_s 154
10.1.2.6 scl_parallel_array_t 155
10.1.2.7 scl_process_t 156
10.1.2.8 scl_run_param_t 157
10.1.2.9 scl_vif_config_t 158
10.1.2.10 scl_print_header_s 159
10.1.2.11 scl_print_data_s 160
10.2 文字列処理 161
10.2.1 scl_max_len 162
10.2.2 scl_is_alnum 162
10.2.3 scl_is_alpha 162
10.2.4 scl_is_digit 162
10.2.5 scl_is_lower 163
10.2.6 scl_is_upper 163
10.2.7 scl_is_space 163
10.2.8 scl_tolower 163
10.2.9 scl_toupper 163
10.2.10 scl_reverse_string 163
10.2.11 scl_starts_with 163
10.2.12 scl_ends_with 164
10.2.13 scl_first_index 164
10.2.14 scl_first_index_from 164
10.2.15 scl_last_index 164
10.2.16 scl_last_index_from 164
10.2.17 scl_find_first_substr 164
10.2.18 scl_find_last_substr 164
10.2.19 scl_trim 165
10.2.20 scl_replace_head_ws 165
10.2.21 scl_replace_tail_ws 165
10.2.22 scl_replace_first 165
10.2.23 scl_replace_last 165
10.2.24 scl_replace_all 165
10.2.25 scl_replace_substr 165
10.2.26 scl_separate_string 166
10.2.27 scl_strncmp 166
10.2.28 scl_strnicmp 166
10.2.29 scl_strnset 166
10.2.30 scl_ascii_to_hex 167
10.2.31 scl_ascii_to_bin 167
10.2.32 scl_ascii_to_oct 167
10.3 ランダム文字列の生成 167
10.4 正則表現 167
10.4.1 scl_reg_expr_t 168
10.4.1.1 new 169
10.4.1.2 build_nfa 169
10.4.1.3 can_match 169
10.4.2 クラスを使用しない正則表現 169
10.4.2.1 scl_pattern_matching 170
10.4.2.2 scl_pattern_matching_multi 170
10.4.2.3 scl_pattern_match_index 170
10.5 ファイル入出力 170

10.5.1 scl_get_line ... 171
10.5.2 scl_get_lines ... 171
10.5.3 scl_head ... 171
10.5.4 scl_tail ... 171
10.5.5 scl_put_lines ... 171
10.6 クロック生成 ... 171
10.7 メッセージプリント機能 ... 172
10.7.1 scl_set_print_file ... 174
10.7.2 scl_flush_print_file ... 174
10.7.3 scl_print_message ... 174
10.7.4 scl_info ... 174
10.7.5 scl_warning ... 175
10.7.6 scl_error ... 175
10.7.7 scl_fatal ... 175
10.7.8 scl_system ... 176
10.7.9 scl_system_wonl ... 176
10.7.10 scl_set_message_level ... 176
10.7.11 scl_set_info_message_level ... 176
10.7.12 scl_set_warning_message_level ... 176
10.7.13 scl_set_error_message_level ... 177
10.7.14 scl_set_system_message_level ... 177
10.7.15 scl_make_format ... 177
10.7.16 scl_sprint_left ... 177
10.7.17 scl_sprint_right ... 177
10.7.18 scl_sprint_center ... 177
10.7.19 scl_sprint_string ... 177
10.7.20 scl_breakup ... 178
10.7.21 scl_print_header ... 178
10.7.22 scl_sprintf_header ... 178
10.7.23 scl_print_footer ... 178
10.7.24 scl_sprintf_footer ... 178
10.7.25 scl_print_data ... 178
10.7.26 scl_sprintf_data ... 179
10.8 プリントオプションの操作 ... 179
10.8.1 scl_set_print_option ... 179
10.8.2 scl_clear_print_option ... 179
10.8.3 scl_is_print_option ... 179
10.8.4 scl_set_head_elements ... 179
10.8.5 scl_get_head_elements ... 180
10.8.6 scl_set_tail_elements ... 180
10.8.7 scl_get_tail_elements ... 180
10.9 プロセス生成 ... 180
10.10 ユーティリティ ... 180
10.10.1 scl_cmd ... 180
10.10.2 scl_cmd_output ... 181
10.10.3 scl_date ... 181
10.10.4 scl_cwd ... 181
10.10.5 scl_getenv ... 181
10.11 汎用マクロ ... 181
10.11.1 `scl_info_m ... 182
10.11.2 `scl_warning_m ... 182
10.11.3 `scl_error_m ... 182
10.11.4 `scl_hex_digits_m ... 183
10.11.5 `scl_short_bin_digits_m ... 183
10.11.6 `scl_short_hex_digits_m ... 183
10.11.7 `scl_full_bin_digits_m ... 183
10.11.8 `scl_full_hex_digits_m ... 183

10.11.9 `scl_init_print_row_m ... 183
10.11.10 `scl_add_print_column_m ... 183
10.11.11 `scl_print_row_m ... 183
10.11.12 `scl_sprintf_row_m ... 184
10.12 プリント書式マクロ ... 184
10.12.1 `scl_name_m ... 184
10.12.2 `scl_sformatfb_m ... 185
10.12.3 `scl_sformatfo_m ... 185
10.12.4 `scl_sformatfh_m ... 185
10.12.5 `scl_sformatfd_m ... 185
10.12.6 `scl_sformatfs_m ... 185
10.12.7 `scl_breakupb_m ... 185
10.12.8 `scl_breakupo_m ... 186
10.12.9 `scl_breakuph_m ... 186
10.12.10 `scl_breakupd_m ... 186
10.12.11 `scl_sprintfb_m ... 187
10.12.12 `scl_sprintfo_m ... 187
10.12.13 `scl_sprintfh_m ... 187
10.12.14 `scl_nsprintfb_m ... 187
10.12.15 `scl_nsprintfo_m ... 188
10.12.16 `scl_nsprintfh_m ... 188
10.12.17 `scl_nsprintfd_m ... 188
10.13 使用例 ... 189
10.13.1 scl_print_header / scl_print_data / scl_print_footer ... 189
10.13.2 ヒープ（scl_heap_t / scl_max_heap_t / scl_min_heap） ... 191
10.13.3 パラレルアレイ（scl_parallel_array_t） ... 195
10.13.4 プロセス生成（scl_process_t） ... 196
10.13.5 文字列処理 ... 200
10.13.6 ランダム文字列 ... 205
10.13.7 正則表現 ... 206
10.13.8 ファイル入出力 ... 210
10.13.9 クロック生成 ... 212
10.13.10 メッセージのプリント ... 212
10.13.11 プリント書式 ... 212
10.13.12 アレイのプリント制御 ... 214
10.13.13 ユーティリティ ... 216

11 補足 ... 218

11.1 ソースコードの構成 ... 218
11.2 SCL 起動メッセージの抑止 ... 218
11.3 検証法 ... 218
11.3.1 検証におけるタイミング ... 218
11.3.2 組み合わせ回路の検証法 ... 220
11.3.3 シーケンシャル回路の検証法 ... 221

12 参考文献 ... 225

1　概要

SCL は TLM（Transaction Level Modeling）による検証環境を構築するための SystemVerilog クラスライブラリーです。このライブラリーには、検証環境を構築するために必要な検証コンポーネントのベースクラス、トランザクションを定義するためのベースクラス、トランザクションを操作するためのメソッド、プリント機能、文字列操作機能、ファイル入出力機能、パターンマッチング機能、ユーティリティ等が含まれています。本章では、SCL の概要を解説します。

1.1　SCL の意義と目的

検証を行う際には、検証の対象である DUT に対して、検証環境の構築、検証用のデータ設計、検証結果の集計とレポート方式等を決定しなければなりません。このうち、検証用のデータ設計と検証結果の集計法は、明らかに、DUT に依存しますが、検証環境の構築には DUT の機能にかかわらずほぼ普遍的な機能が必要になります。

例えば、対象の DUT がどのような機能を持つにしても、検証環境にはドライバーの機能が必要になります。同様に、どのような検証においても、DUT からのレスポンスをサンプリングする機能を持つコレクター機能が必要になります。更に、ドライバーに検証用のデータを供給するジェネレータ、およびコレクターが収集したデータを解析するためのモニター、スコアボード等の機能も必要になります。しかも、これらの機能は DUT の種類にかかわらず、殆ど全ての検証環境で必要になります。したがって、これらの不可欠な機能を DUT に依存しない形式で備えておけば、検証環境構築作業の生産性を著しく向上できます。SCL は、その様な普遍的な機能を供給する SystemVerilog クラスライブラリーです。

1.2　検証環境を構成する概念

SCL では、検証に用いるデータをトランザクションと呼びます。トランザクションは、RTL の信号（すなわち、ネット）よりも高位な概念で、タスクやファンクション等のメソッドを使用してトランザクションを操作します。一方、DUT は RTL 信号の流れで機能が表現されています。トランザクションと信号では、全く異なる次元のデータ表現であるため、相互変換が必要になります。それが、ドライバーとコレクターの役目です。

ドライバーは、SCL の検証コンポーネントにより生成されたトランザクションを RTL 信号に変換して DUT をドライブします。一方、コレクターは、DUT からサンプリングしたレスポンスをトランザクションに逆変換をして他の検証コンポーネントに転送します。ドライバーが DUT をドライブする時、およびコレクターが DUT からレスポンスをサンプリングする時に、SystemVerilog の virtual インターフェース（以降、変数名で表現する際には `vif` と略称します）が使用されます（図 1-1）。

検証過程はトランザクションを使用して行われるので TLM と呼ばれ RTL と明確に区別されます。DUT はインターフェースを使用する必要はありませんが、インターフェースには DUT のポートに対応する信号が定義されています。したがって、DUT は仮想的にインターフェースを使用していると考えられます。図 1-1 は、その様に拡張解釈した概念を基にしています。

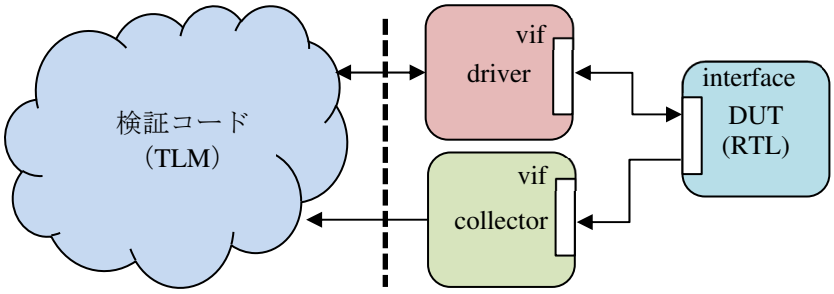


図 1-1　検証環境でのドライバー、コレクターと DUT の関連

1.3 インターフェースと virtual インターフェース

インターフェースは、通常、トップモジュールに配置されて以下のように DUT と接続されています。

```
interface simple_if(input logic clk);
...
endinterface

module top;
bit    clk;

simple_if SIF(.*);
device DUT(.clk(SIF.clk),...);
...
endmodule
```

DUT はインターフェースに定義されている信号に接続されている

DUT はインターフェースを使用していなくても、インターフェース内の信号に接続されていれば、DUT は仮想的にインターフェースとつながれていると考えられます。即ち、どのような DUT に対しても virtual インターフェースを適用できる機会が存在します。

そして、トップモジュールはドライバーやコレクターに定義されている `vif` にインターフェースのインスタンスを割り当てる準備をします。こうして、検証環境では virtual インターフェースを使用できるようになります（図 1-2）。

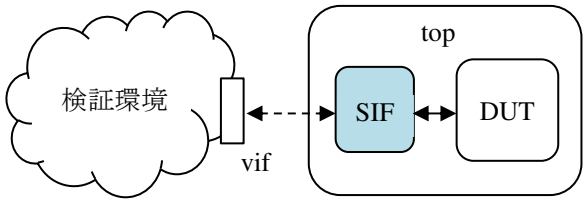


図 1-2　トップモジュールに配置されたインターフェース、DUT、検証環境の関連

`vif` はインターフェースのインスタンスへのポインターを表現しているので、ドライバーやコレクターがあたかも DUT と直接接続されているような状態を表現できます（図 1-1）。つまり、ドライバーが `vif` を介して信号を操作すると、DUT の対応する信号が反応します。同様に、DUT の信号に変化が起これば、コレクターは `vif` を介してその変化を知る事ができます。

1.4 virtual インターフェースの操作

virtual インターフェースは、SystemVerilog が備えている機能です。SystemVerilog のクラス内にはインターフェースのインスタンスを配置できないため、virtual インターフェースを使用しなければなりません。virtual インターフェースとは、インターフェースのインスタンスへのポインターを示す変数で、その変数にインターフェースのインスタンスが割り当てられなければなりません。

virtual インターフェースをクラス内に定義する際には、インターフェース名称が必要になりますが、インターフェース名称を検証環境で直接参照すると、検証環境自身の汎用性が失われる恐れがあります。そのため、SCL ではインターフェースを定義する際に virtual インターフェースを操作するためのクラスを定義する仕組みになっています。この仕組みによりクラス内でインターフェース名を参照する事を回避できます。例えば、インターフェースを定義した際に、以下のようにして virtual インターフェースを操作するためのクラス `vif_vonfig` も定義しておきます。

```
interface simple_if(input logic clk);
...
endinterface

`scl_vif_config_m(virtual simple_if,vif_config)
```

こうすると、トップモジュールで virtual インターフェースの設定準備が可能になります。また、ドライバーやコレクターは、トップモジュールにより設定された virtual インターフェースを vif_config クラスにより取得できます。

図 1-3 では vif_config を使用してトップモジュールが virtual インターフェースを設定している様子を示しています。そして、設定された virtual インターフェースをドライバーが取得しています。こうして、検証環境でインターフェース名をハードコーディングする状況から回避できます。

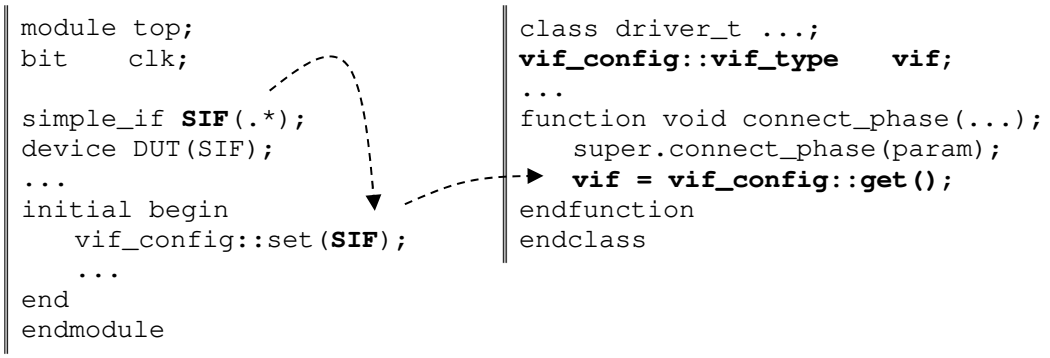


図 1-3　インターフェースのインスタンスと virtual インターフェース

SCL は virtual インターフェースを設定するための便利なメソッドを提供しています。第 4 章で、virtual インターフェースの設定法を詳しく解説します。この方法は、SCL に限らず一般的に適用できる便利な手法なので、確実に理解をする事をすすめます。

1.5　ダイナミックな検証環境

検証コンポーネントは virtual インターフェースを保有してさえいれば、DUT と通信を行えるので、検証コンポーネントは、スタテックなインスタンスとして存在する必要はありません。つまり、virtual インターフェースを使用する事により、ダイナミックに検証環境の構成を変更できる事になります。換言すれば、複数のテストケースを用意して、実行時にテストケースを決定できる事になり、検証作業の生産性が向上します。つまり、一度のコンパイルで複数のテストケースを実行できるようになります。このように、SCL は検証コンポーネントを記述するための機能を提供するだけでなく、検証作業全体の生産性を促進する効果をもたらします。

1.6　ドライバーとトランザクション

さて、ドライバーは DUT の機能に関わらず必要な存在ですが、ドライバーの処理内容は、DUT に依存します。したがって、ドライバーには DUT の機能に応じて変化する仕組みが必要になります。DUT をドライブするためには、以下の機能が必要になります。

- DUT の信号を操作するために必要なデータ
- DUT の信号を操作する手順

前者のデータはトランザクションと呼ばれ、後者の手順はメソッドと呼ばれます。メソッドは、ドライバイー内に SystemVerilog のタスク、およびファンクションとして定義されます。トランザクションには、一定の規則が必要であるため、SCL ではトランザクションのベースクラスを定義しています。そして、ユーザがトランザクションを定義する際には、そのベー

スクラスのサブクラスとして定義しなければなりません。同様に、SCL ではドライバーのベースクラスを定義しています。ユーザのドライバーは、直接または間接的にこのベースクラスのサブクラスとして定義されなければなりません。

ドライバーは、DUT の機能に応じて変化しなければならないので、ドライバーのベースクラスには DUT の機能を表現するパラメータが必要になります。そのパラメータは、トランザクションです。したがって、ドライバーのベースクラスは以下のように定義されています。ここで、scl_component_t は SCL における全ての検証コンポーネントのベースクラスです。

```
class scl_driver_t
    #(type TR=scl_transaction_t) extends scl_component_t;
...
endclass
```

ドライバーと同様にトランザクションのベースクラスが定義されていますが、以下に示すように、特別なパラメータを伴いません。この点は、ドライバーと大きく異なります。前述したように、ユーザがトランザクションを定義する際には、直接または間接的にこのベースクラスのサブクラスとして定義しなければなりません。ここで、scl_object_t は SCL における全てのオブジェクトのベースクラスです。

```
class scl_transaction_t extends scl_object_t;
...
endclass
```

1.7　トランザクションの操作

トランザクションの操作は、TLM の手法に従います。RTL では、モジュールのポート同士がネットまたは変数で接続されて信号値が常に流れている状態を表現します（図 1-4）。

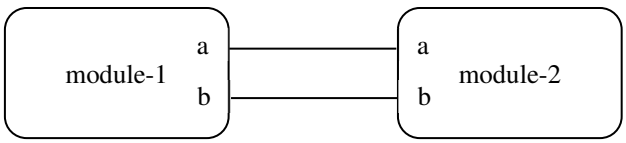


図 1-4　RTL による接続方式

一方、検証コンポーネントは DUT と物理的に接続されていないので、TLM では RTL とは全く異なる処理形態が必要になります。TLM はデータの送受信をメソッドの呼び出しで行います。例えば、トランザクションを取得するには get()メソッドを使用し、トランザクションを送るためには、put()メソッドを使用します。

1.7.1　トランザクションを取得する手順

トランザクションを get()メソッドで取得するためには、トランザクションを準備する側が get()メソッドの内容を実装しなければなりません。図 1-5 においては、get_server は get()メソッドの処理内容を定義しなければなりません。

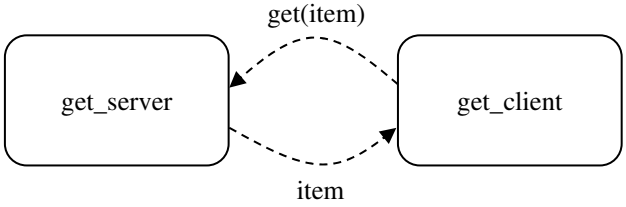


図 1-5　get()メソッドによるトランザクションの取得の流れ

通常はこの様なプロトコルにしたがってトランザクションを取得しなければなりませんが、SCL が提供するドライバーとジェネレータを使用してクラスを定義する事により、低レベルの get()や put()メソッドの使用から解放されます。

1.7.2 トランザクションを送信する手順

トランザクションの取得と同様に、トランザクションを put()メソッドで引き渡すためには、受け取り側は put()メソッドの内容を実装しなければなりません。図 1-6 においては、put_server は put()メソッドの処理内容を定義しなければなりません。

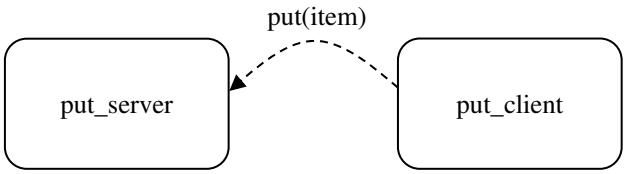


図 1-6 put()メソッドによるトランザクションの送信の流れ

1.7.3 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手順

これまでに紹介した put()と get()メソッドは、トランザクションを能動的に処理する場合に使用する TLM ですが、受動的（つまり、reactive）にトランザクションを処理する場合には write()メソッドを使用します。

例えば、DUT からのレスポンスをサンプリングする事により発生するトランザクション処理は受動的なので、write()メソッドを使用します。図 1-7 は、コレクターがモニターに write()メソッドでトランザクションを送信している状態を示しています。DUT からのレスポンスは、コレクターにより常に監視されなければならないため、コレクターは write()メソッドに時間を消費できません。このため、write()メソッドはファンクションとして実装されます。モニターは write()メソッドを実装しなければなりません。尚、write()メソッドは即座に戻るため、トランザクションを複数の検証コンポーネントに一斉に送信できる利点があります。

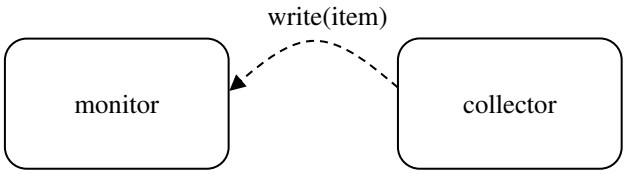


図 1-7 write()メソッドによるトランザクションの送信

以上述べたように、TLM ではトランザクション処理メソッドの呼ばれる側がトランザクションの処理方式を決定する事になります。更に詳しいトランザクション処理手順は第 3 章で解説します。

1.8 トランザクションの生成

トランザクションは scl_transation_t と呼ばれるクラスのサブクラスとして表現されますが、SCL ではトランザクションの生成および準備を scl_play_t と scl_act_t と呼ばれるクラスで行います。それらのクラスはトランザクションの生成手順をツリーで表現するために使用されます。

ツリーのルートを scl_play_t のオブジェクトで表現し、ツリー内の他のノードを

scl_act_t のオブジェクトで表現します（図 1-8）。そして、ツリーのリーフノードは一つのトランザクションを生成する役目を持ちます。他のノードは、下位のノードを使用してトランザクションの生成手順を制御します。ツリー構造には制限がないため複雑なトランザクション生成手順を表現できます。scl_act_t には get_item() と get_group() メソッドがあり、get_item() を呼ぶと一つのトランザクションを生成でき、get_group() を呼ぶと他の scl_act_t のオブジェクトを呼び出せます。つまり、階層を表現できます。

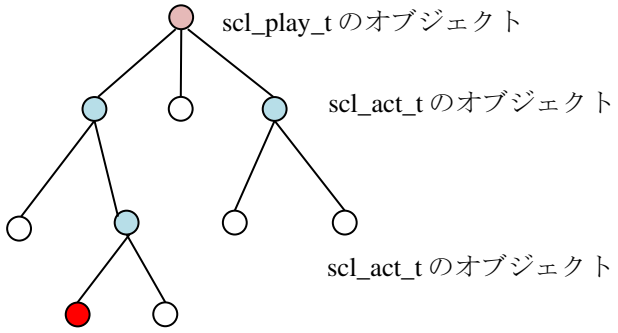


図 1-8　トランザクション生成ツリー

ドライバーがジェネレータにトランザクションを要求すると、ジェネレータは処理対象のリーフノード（図 1-8 における赤いノード）を実行してトランザクションを作りドライバーに引き渡します。赤いリーフノードの処理が終了すると右隣のリーフノードが次に処理される対象となり赤くなります。この操作を繰り返してツリーのリーフノードを左から順に処理して一連のトランザクションが生成されます。生成手順の中にループを記述できるので同じ手順を複数回繰り返す事もできます。したがって、簡潔にトランザクション生成手順を実現する事ができます。

1.9　テストとシナリオ

SCL では、テストがシナリオを割り当てる役目を持ちます。ここで、シナリオとはトランザクション生成手順を意味します。テストがシナリオを割り当てると、シナリオはジェネレータに scl_play_t のオブジェクトを割り当ててトランザクション生成ツリーを作る準備を開始させます。ただし、このツリーは最初から完成した構造ではなくドライバーがトランザクションを取得して行く過程において次第に構造を完成して行きます。

シミュレーションが開始すると、ジェネレータはトランザクション生成ツリーのルートノードを起動してトランザクション生成処理の初期化を行います。その後は、ドライバーからトランザクションの要求が来るのを待ちます。要求が来ると前節で解説したようにトランザクション生成手順が作動します。

1.10　検証環境の構成

テストベンチ[1]は、所謂、トップモジュールを指します。テストベンチには、DUT とインターフェースのインスタンスが配置されるだけですが、SCL のクラスを使用して開発されたテストがテストベンチにより選択されて実行されます。テストベンチの構成は検証内容に依存する事は明らかですが、一般的には、図 1-9 のような構成になります。それらの構成要素の解説は表 1-1 にあります。検証環境を構成するための SCL クラスはメソドロジークラスとも呼ばれます。

テストは、virtual インターフェースを介して DUT をドライブする事、および DUT からのレスポンスを収集して DUT の動作を検証および解析する役目を担当します。テストベンチには幾つかのテストが含まれますが、その中の一つが選択されて実行します。

[1] テストベンチの定義は人により異なりますが、ここではトップモジュールと同意語とします。

エージェントは最も下位にある階層コンポーネントで DUT をドライブする機能と DUT からのレスポンスをサンプリングする機能を持ちます。エージェントにはジェネレータ、ドライバー、コレクター、モニター等のコンポーネントが配置されます。

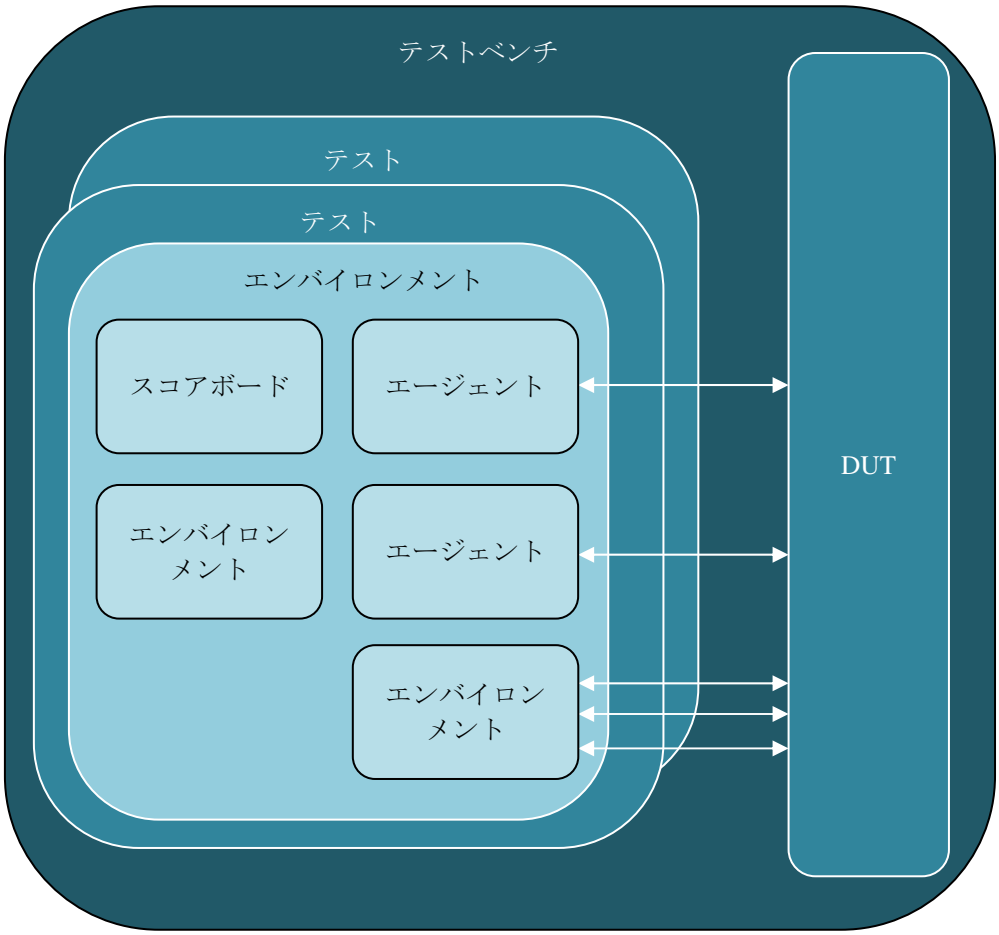


図 1-9　テストベンチの構成

表 1-1　テストベンチを構成する要素

階層を構成する要素	意味
テストベンチ	これは、いわゆる、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 およびインターフェースをインスタンスとして保有します。検証環境を決定して起動する役目も持ちます。
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。
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。
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。すなわち、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。
エージェント	基本的な検証コンポーネントで、ジェネレータ、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。
ジェネレータ	トランザクションを生成する制御を司るコンポーネントです。ジェネレータはアクトを作り実行する事によりトランザクションを生成します。

ドライバー	ドライバーはジェネレータ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。ジェネレータとドライバーの通信には TLM が使用されます。一方、DUT をドライブするためには、virtual インターフェースが使用されます。
コレクター	DUT からのレスポンスをサンプリングしてトランザクションに変換し、他の検証コンポーネントに送信する役目を持ちます。つまり、コレクターは RTL より TLM への変換機能です。
モニター	モニターはコレクターから受信したトランザクションを他の検証コンポーネントに一斉に送信します。モニター自身もカバレッジ計算、およびレスポンスに関する簡単なチェックも行います。
アクト	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。アクトは他のアクトを呼び出す事ができるので複雑なテストシナリオを定義する事ができます。
プレイ	トランザクションを生成するための手順はツリー状に配置されたアクトのオブジェクトで表現されます。ツリーのルートはプレイのオブジェクトで他のツリーノードと区別されます。プレイはジェネレータに割り当てられるとツリーを構築する準備を開始します。
シナリオ	検証環境にトランザクション生成手順を割り当てる働きをします。

1.11 ダイナミックなコンフィギュレーション設定変更

検証中にはコンフィギュレーションを一時的に変更する必要性が出てきます。SCL では、該当するクラスを修正せずにクラスのプロパーティを変更できます。また、クラス自身を他のサブクラスに置き換える事さえ可能です。例えば、検証環境にドライバー（driver_t）のインスタンスが使用されているとすると、検証環境を修正せずに driver_t と互換性のあるクラス new_driver_t のインスタンスで置き換える事ができます（図 1-10）。

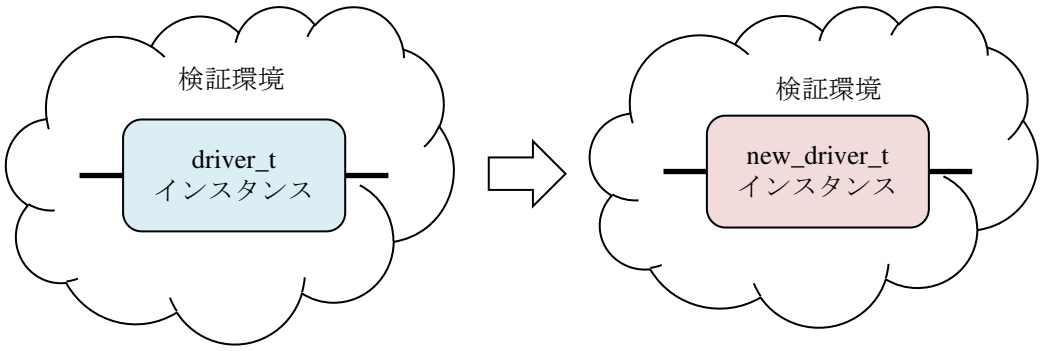


図 1-10　実行時のコンフィギュレーション設定変更

同様に、検証環境の外部からクラスのプロパーティの設定変更ができます。したがって、検証環境を修正せずに一時的に新しい機能の確認をする事ができます。この便利な機能は第 2 章で解説されます。

1.12 SCL マクロ

SCL では、同じ内容を持つ命令を頻繁に記述する場合、マクロを使用するようにしています。基本的には、マクロはタイプ入力の負荷を減少させる効果を持ちますが、多くの複雑な命令群を生成するためにも使用されます。マクロの簡単な使用例としてドライバーの定義を以下に紹介します。

2 SCL の構成

SCL のクラスは、オブジェクト系と検証コンポーネント系に分類されますが、scl_object_t と scl_component_t がそれぞれのベースクラスとなっています。つまり、全てのオブジェクトは、直接または間接的に scl_object_t のサブクラスで、全ての検証コンポーネントは、直接または間接的に scl_component_t のサブクラスです。scl_object_t と scl_component_t クラスをユーザが直接使用する機会は少ないと思いますが、これらのクラスが備えている機能を理解しておく必要があります。何故なら、それらの機能の多くは virtual メソッドとして定義されているからです。

2.1 クラス階層

SCL では、ごく自然で、かつ現代的な概念に基づいてクラスの設計がされています。SCL のベースクラスは、scl_thing_t であり、全てがそこから始まります。トランザクションもポートもコンポーネントも「物」であるので、その概念は適合します。scl_thing_t から派生したクラスは、物を具体的に表現し始める目的を持つため、SCL ではアプリケーションに適合する命名法を使用しています。

例えば、シナリオはテストケースに対応し、シナリオはプレイを実行させます。プレイは、トランザクションを作るためにアクトを使います。図 2-1 は、SCL のクラス階層の一部です。クラス階層に現れるクラスの概要を表 2-1 に示します。SCL のクラスは、検証コンポーネント系とオブジェクト系に分類され、オブジェクト系のクラスは検証コンポーネントが使用する機能を提供します。

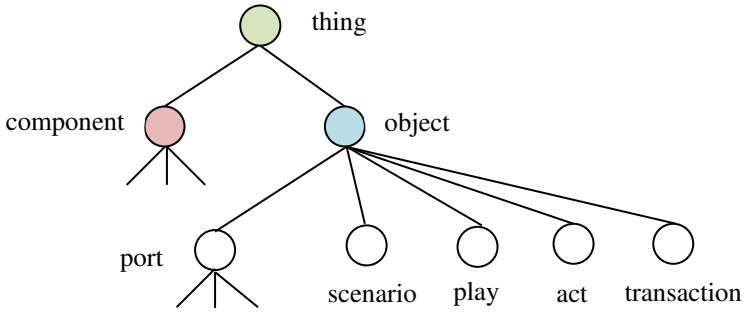


図 2-1 SCL クラス階層（抜粋）

表 2-1 主な SCL クラスの概要

SCL クラス	機能
scl_thing_t	他の全ての SCL クラスのベースクラスで、アブストラクトクラスとして定義されています。
scl_component_t	全ての検証コンポーネントのベースクラスです。
scl_object_t	全てのオブジェクトのベースクラスです。
scl_port_t	TLM ポートのベースクラスです。
scl_scenario_t	シナリオのベースクラスです。
scl_play_t	トランザクションを作る手順のベースクラスです。
scl_act_t	トランザクションを作るためのベースクラスです。
scl_transaction_t	トランザクションを定義するためのベースクラスです。

これらのクラスの機能は、第 3 章以降で詳しく解説されます。

2.2 scl_void_t

実は、SCL にはもう一つ別のクラス scl_void_t が定義されていて scl_thing_t のベースクラスになっていますが、通常使用される事はないので scl_thing_t が全てのベースクラスであるという表現をしています。

次節に scl_thing_t の解説がありますが、コンストラクタはパラメータとして name を必要とします。一方、一般的な記述用としてパラメータを必要としないクラスも必要になるため、scl_void_t が用意されています。 このクラスの存在により、どのようなクラスも scl_void_t クラスのサブクラスになるように定義できます。要約すれば、scl_void_t クラスは SCL の完全性を保証する目的に存在し、scl_thing_t は SCL の実質的なベースクラスです。因みに、scl_void_t は以下のように定義されています。

```
virtual class scl_void_t;
endclass
```

2.3 scl_thing_t

このクラスは、全ての SCL クラスのベースクラスで、このクラス自身のインスタンスを作る意義がないので、アブストラクトクラスとして定義されています。scl_thing_t の全容は、以下のようになります。

```
virtual class scl_thing_t extends scl_void_t;
// members
static int    m_count;  // # of instances created
string        m_name;
local int     m_id;     // unique number in entire environment
const static string m_type_name = "scl_thing_t";

// methods
extern function new(string name);
extern virtual function string get_name();
extern virtual function int get_id();
extern virtual function string get_type_name();
extern virtual function bit is_object();
extern virtual function bit is_component();
// the following function is implemented by macros
extern virtual function void
    scl_field_processor(input scl_thing_t source,string tabs="",
        scl_field_command_e command=SCL_FIELD_PRINT);
pure virtual function void print();
pure virtual function void copy(scl_thing_t source);
endclass
```

このクラスに定義されているプロパーティを表 2-2 に紹介します。

表 2-2　scl_thing_t のプロパーティ

プロパーティ	説明
m_count	生成されたインスタンスの総数を管理します。このプロパーティを利用して、全てのインスタンスにユニークな番号（ID）を割り付けます。
m_name	scl_thing_t のインスタンスに付けられた名称を示します。
m_id	scl_thing_t のインスタンスに割り当てられたユニークな ID です。
m_type_name	クラスのタイプを示す文字列です。サブクラスにより再定義されます。

このクラスに定義されているメソッドの機能概要を表 2-3 に示します。

3 TLM

SCL では、トランザクションの取得に get()、送信[2]に put()、転送に write()メソッドを使用します。何れのメソッドを使用する際にも、クライアントとサーバーがあります。クライアントは、命令を発行する側で、その命令を処理する側がサーバーです。そして、サーバーが処理手順をメソッドとして定義します。送信と転送命令は、ポートを介して行われます。SCL では、この関係を図 3-1 のように表現します。サーバー側は、メソッドの内容を埋めるという意味でポートを黒く塗りつぶします。

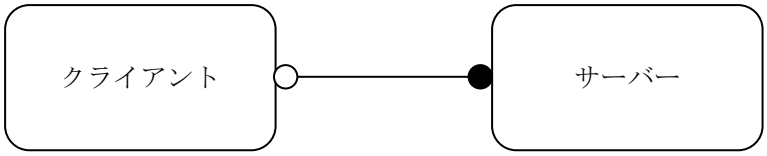


図 3-1　SCL でのクライアントとサーバーの表現法

クライアントとサーバーにはポートが配置されますが、それらのポートの接続は、クライアントとサーバーのインスタンスを保有する検証コンポーネントが行います。したがって、クライアントは特別なサーバーを仮定していません。同様に、サーバーは特別なクライアントを前提としていません。メソッドで指定するプロトコルに互換性がある限り、クライアントとサーバーの通信は正しく動作します。

本章では、以下のように定義されたトランザクションを使用して、トランザクションを操作するメソッドの定義法と使用法を順に解説します。

```
class simple_item_t extends scl_transaction_t;
rand logic [3:0] a,
                 b;

function new(string name="simple_item");
   super.new(name);
endfunction

endclass
```

本章では、トランザクションの送受信方式を解説しますが、メソドロジークラスを使用すれば、ここで解説されている細かい物理的な処理手順から解放されます。したがって、メソドロジークラスを使えば本章で解説する内容の詳細を理解する必要はありませんが、TLM ポートの概念は重要なので理解しておくと便利です。例えば、ユーザ自身で機能を拡張する際には重要な知識となります。

3.1 ポート

SCL では、ポートを介してトランザクションの授受を行います。一対一のポート授受には、scl_io_port_t または scl_nbio_port_t を使用し、一対 N の送信には scl_pass_port_t を使用します。ポートはクラスとして表現され、図 3-2 に示すようなクラス階層を構成します。それぞれのクラスの機能概要は表 3-1 に示されています。図中に現れる接頭辞 nb は non-blocking の略称です。つまり、ノンブロッキング方式で送受信します。

[2] 通信の対象が一つである場合には送信、複数であれば転送と表現しています。

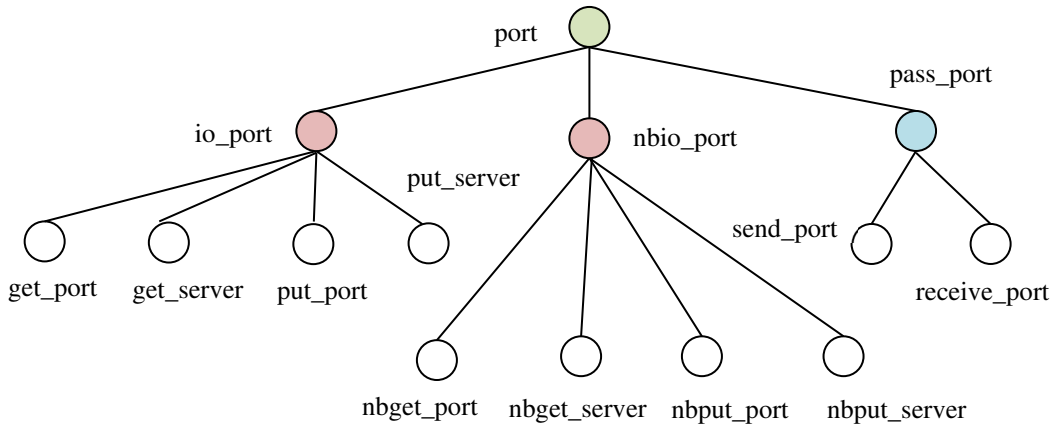


図 3-2　SCL ポートクラスの階層構造

表 3-1　SCL のポートクラス

ポートクラス	機能
scl_port_t	全ての SCL ポートのベースクラスです。全てのポートに共通する情報を定義しています。
scl_io_port_t	一対一のトランザクション処理をするポートのベースクラスです。
scl_get_port_t	トランザクションを取得するポートのベースクラスです。
scl_get_server_t	トランザクションを準備するポートのベースクラスです。
scl_put_port_t	トランザクションを送信するポートのベースクラスです。
scl_put_server_t	送信されたトランザクションを処理するポートのベースクラスです。
scl_nbio_port_t	一対一のトランザクション処理をするポートのベースクラスですが、ノンブロッキング方式を用います。
scl_nbget_port_t	ノンブロッキング方式でトランザクションを取得するポートのベースクラスです。
scl_nbget_server_t	ノンブロッキング方式でトランザクションを準備するポートのベースクラスです。
scl_nbput_port_t	ノンブロッキング方式でトランザクションを送信するポートのベースクラスです。
scl_nbput_server_t	ノンブロッキング方式で送信されたトランザクションを処理するポートのベースクラスです。
scl_pass_port_t	一対 N のトランザクション処理をするポートのベースクラスです。
scl_send_port_t	一対 N のトランザクションを転送するポートのベースクラスです。
scl_receive_port_t	一対 N のトランザクションを受信するポートのベースクラスです。

3.1.1　scl_port_t

scl_port_t は以下のように定義されています。

```
class scl_port_t #(type PARENT=scl_component_t)
    extends scl_object_t;
`scl_decl_object_m(scl_port_t#(PARENT))
// members
PARENT              m_parent;
scl_port_option_e   m_option;

// methods
extern function new(string name,PARENT parent);
```

nbput_server を接続する。
- クライアントがトランザクションをノンブロッキング方式で出力するためには、put()メソッドの代わりに try_put()メソッドを使用する。

put()メソッドと殆ど同様に使用できるので、使用例を省略します。

3.7 FIFO

トランザクションを必要とする consumer とトランザクションを生成する producer が独立プロセスとして実行するためには、トランザクションをプーリングして置く必要があります。その際に使用できる FIFO キューが scl_fifo_t です。FIFO を使用すると、FIFO がトランザクションの操作を行うのでサーバーになり、producer と consumer の何れもクライアントになります（図 3-13）。トランザクションの書き込みと取得に get()、try_get()、put()、try_put()等を使用できます。つまり、ブロッキング方式およびノンブロッキング方式の何れの方式でも FIFO にアクセスできます。

図 3-13　FIFO を使用したトランザクションの授受

このトランザクション処理法において、FIFO キューにトランザクションが存在すれば、get()は直ぐに戻りますが、キューが空であれば、トランザクションの準備ができるまで get()はブロックします。同様に、キューに空きがあれば、put()は直ぐに完了しますが、キューが一杯であれば、キューに空きができるまで put()はブロックします。これらの一連の制御を scl_fifo_t が行います。ここで、もし FIFO キューのサイズを無制限に設定しておくと、常に put()は直ぐに戻ります。scl_fifo_t クラスのコンストラクタには FIFO キューのサイズを指定する事ができるので、ユーザは FIFO キューの管理に柔軟性を持たせる事ができます。

なお、get()の代わりに try_get()を使用すれば、FIFO キューにトランザクションがあってもなくてもすぐに戻ります。put()の代わりに try_put()を使用しても同様な事が言えます。

3.7.1 scl_fifo_t

FIFO キューは以下のように定義されています。FIFO を使用するためには、トランザクションのタイプを TR に指定しなければなりません。

```
class scl_fifo_t
    #(type TR=scl_transaction_t) extends scl_fifo_base_t#(TR);
`scl_decl_component_m(scl_fifo_t#(TR))
// members

// methods
extern function
    new(string name,scl_component_t parent=null,int size=0);
extern virtual task put(input TR item);
extern virtual function bit try_put(input TR item);
extern virtual task get(output TR item);
extern virtual function bit try_get(output TR item);
extern virtual function int fifo_size();
extern virtual function int num_items();
extern virtual function bit is_empty();
extern virtual function bit is_full();
```

```
extern virtual function bit can_put();
extern virtual function bit can_get();
extern virtual function void empty();
endclass
```

FIFO で使用する TLM ポートやキューの定義は、ベースクラス scl_fifo_base_t で以下のように行われています。

```
class scl_fifo_base_t
    #(type TR=scl_transaction_t) extends scl_component_t;
`scl_decl_component_m(scl_fifo_base_t#(TR))
// members
scl_get_server_t#(TR,scl_fifo_base_t)   m_get_server;
scl_nbget_server_t#(TR,scl_fifo_base_t) m_nbget_server;
scl_put_server_t#(TR,scl_fifo_base_t)   m_put_server;
scl_nbput_server_t#(TR,scl_fifo_base_t) m_nbput_server;
mailbox          m_fifo;
int              m_size;
int              m_pending_requests;

// methods
extern function new(string name,scl_component_t parent=null,int
size=0);
extern virtual task put(input TR item);
extern virtual function bit try_put(input TR item);
extern virtual task get(output TR item);
extern virtual function bit try_get(output TR item);
endclass
```

ベースクラスに定義されている重要なプロパティを表 3-30 に示します。

表 3-30　scl_fifo_base_t のプロパティ

プロパティ	説明
m_get_server	ブロッキング方式でトランザクションを取得する場合に使用されるポートです。scl_get_port_t タイプのポートをこのポートに接続しなければなりません。
m_nbget_server	ノンブロッキング方式でトランザクションを取得する場合に使用されるポートです。scl_nbget_port_t タイプのポートをこのポートに接続しなければなりません。
m_put_server	ブロッキング方式でトランザクションを出力する場合に使用されるポートです。scl_put_port_t タイプのポートをこのポートに接続しなければなりません。
m_nbput_server	ノンブロッキング方式でトランザクションを出力する場合に使用されるポートです。scl_nbput_port_t タイプのポートをこのポートに接続しなければなりません。
m_fifo	SystemVerilog の mailbox で実現されているキューです。トランザクションはこのキューに登録されて管理されます。
m_size	FIFO キューのサイズを示します。m_size==0 であれば、キューは無制限の大きさを持ちます。
m_pending_requests	ブロッキング方式で get()を使用する場合、現在待ち状態にあるプロセスの数を示しています。つまり、get()メソッドの処理が未完了のプロセス数を示しています。

FIFO で使用できるメソッドを表 3-31 に紹介します。

4　virtual インターフェースの設定と取得

virtual インターフェースは、検証コンポーネントが DUT をドライブする時や DUT からのレスポンスをサンプリングする際に使用される重要な手段です（図 4-1）。

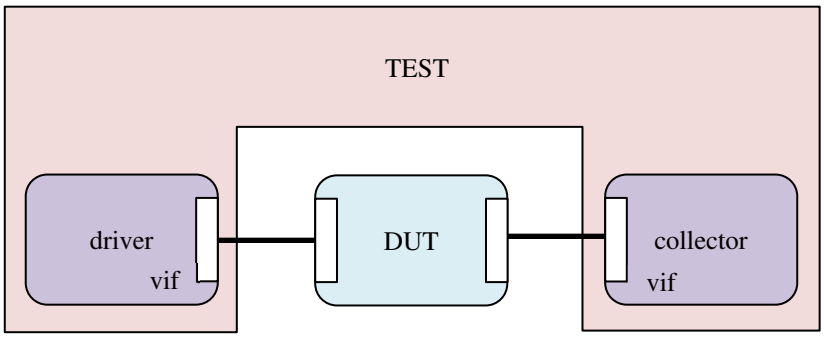


図 4-1　検証環境と virtual インターフェース

DUT ではインターフェースを使用していなくても、virtual インターフェースに使用されてい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、ドライバーやコレクターは恰も DUT と直接接続されているような状況となります。つまり、DUT には恰もインターフェースが使用されているような状態となります。図 4-1 はその様な拡張解釈を基にしています。

一般的には、柔軟性のある virtual インターフェースの設定手順は複雑になる傾向がありますが、SCL ではマクロを使用して使用手順の簡素化を実現しています。手順は以下のようになります。

- `scl_vif_config_m マクロを使用して virtual インターフェースを操作するためのクラスを定義する。
- 定義したクラスの set() メソッドを使用して virtual インターフェースを設定する。
- 定義したクラスの get() メソッドを使用して virtual インターフェースを取得する。

本章では、以下のようなインターフェースが定義されていると仮定して、virtual インターフェースの設定法と取得法を解説します。ここでは、インターフェースの内容は重要ではないので、省略しています。

```
interface simple_if;
...
endinterface
```

4.1　virtual インターフェースの使用準備

以下に示すマクロを使用して virtual インターフェースを操作するためのクラスを定義します。マクロに指定するパラメータの意味は、
表 4-1 の通りです。

```
`scl_vif_config_m(VIF_TYPE,VIF_CONFIG_NAME)
```

表 4-1　`scl_vif_config_m マクロに指定するパラメータ

パラメータ	意味
VIF_TYPE	定義すべき virtual インターフェースを指定します。このパラメータはユーザが使用するインターフェースの名称を指定します。ただし、キーワード virtual を添えなければなりません。

5　トランザクション

本章では、トランザクションの定義法を解説します。トランザクションはオブジェクトであり、検証コンポーネントと多くの点で定義法が異なります。例えば、トランザクションは、一般的に、階層を作らないので定義項目は DUT に関連するデータ項目に限定されます。

5.1　scl_transaction_t

トランザクションは、直接または間接的に scl_transaction_t のサブクラスとして定義しなければなりません。以下のように、トランザクションのベースクラスが定義されています。

```
class scl_transaction_t extends scl_object_t;
`scl_decl_object_m(scl_transaction_t)

extern function new(string name="transation");
extern function string get_type_name();
endclass
```

`scl_decl_object_m マクロは、scl_transaction_t のクラスのタイプ情報を登録するマクロで、以下の情報を生成します。

```
static string m_type_name = "scl_transaction_t";
function string get_type_name();
   get_type_name = m_type_name;
endfunction
```

ベースクラスに定義されている主なプロパーティは、表 5-1 のようになります。

表 5-1　scl_transaction_t クラスの主なプロパーティ

パラメータ	意味
m_type_name	トランザクションタイプを示す文字列です。

参考 5-1

ベースクラスの scl_transaction_t にはトランザクションのデータを表現する項目が定義されていないので、ユーザはサブクラス内に必要なデータ項目を定義しなければなりません。つまり、scl_transaction_t をそのまま使用しても意味のあるトランザクション処理を実現できません。
□

ベースクラスに定義されているメソッドを以下に紹介します。

5.1.1　new

```
function new(string name="transation");
```

このメソッドは、トランザクションのインスタンスを作るコンストラクタを意味します。このコンストラクタには、インスタンスを命名するための文字列を name に指定しなければなりません。

5.1.2　get_type_name

```
function string get_type_name();
```

このトランザクションタイプの名称を文字列として戻します。サブクラスでは、このメソッドを再定義しなければなりません。

5.2 トランザクションの定義

ユーザがトランザクションを定義するためには、以下に示すルールに従う必要があります。

- トランザクションは scl_transaction_t またはそのサブクラスから定義されなければなりません。
- コンストラクタを必ず指定します。便利なマクロが準備されているので、それを使用するとコンストラクタの定義は簡単になります。
- オブジェクトである事を示すためのマクロ `scl_object_m を指定します。
- トランザクションにデータ項目がある場合には、フィールドマクロを指定してデータ項目の属性を宣言する。このマクロを使用しないと、プリント処理等での対象には含まれまれなくなります。

次に、トランザクションの簡単な定義例を紹介します。以下の例ではフィールドマクロを指定していますが、第 6 章で解説されます。

例 5-1　トランザクションの定義例

トランザクションは、ベースクラスの scl_transaction_t を指定して以下のように定義します。定義したトランザクションのクラスを SCL に登録するために `scl_object_m マクロを指定しなければなりません。そして、トランザクション内に定義されているプロパティに対しては、フィールドマクロを指定しなければなりません。

```
class simple_item_t extends scl_transaction_t;
logic [UP_DOWN_WIDTH-1:0]  d, q, qn;
logic             reset, load, up_down;


`scl_object_field_begin_m(simple_item_t)
   `scl_field_integral_m(reset,SCL_DEFAULT)
   `scl_field_integral_m(load,SCL_DEFAULT)
   `scl_field_integral_m(up_down,SCL_DEFAULT)
   `scl_field_integral_m(d,SCL_DEFAULT)
   `scl_field_integral_m(q,SCL_DEFAULT)
   `scl_field_integral_m(qn,SCL_DEFAULT)
`scl_field_end_m

function new(string name="simple_item");
   super.new(name);
endfunction

endclass
```

トランザクションのベースクラスを指定する

フィールドマクロ

コンストラクタの定義

このように定義された simple_item_t のオブジェクトを print()メソッドでプリントすると以下のような書式でプリントされます。

```
=========================================
Name          Type              Size    Value
-----------------------------------------
item          simple_item_t     -       @16
  reset       integral          1       'h0
  load        integral          1       'h0
  up_down     integral          1       'h0
  d           integral          8       'h00
  q           integral          8       'ha3
  qn          integral          8       'h5c
=========================================
```

6　フィールドマクロ

トランザクションのデータ項目にはフィールドマクロを使用して属性を指定しなければなりません。フィールドマクロを指定するとトランザクションをプリントする機能が自動生成されます。

トランザクションにデータ項目が存在する事を SCL に知らせるためには、以下のように`scl_field_begin_m と`scl_field_end_m マクロで囲みます。その間に、`scl_field_*_m マクロを使用してデータ項目を指定します。

```
`scl_field_begin_m(CLASS_TYPE)
...
`scl_field_end_m
```

参考 6-1

トランザクションにデータ項目が存在しない場合には、`scl_field_begin_m と`scl_field_end_m マクロを指定する必要はありません。
□

フィールドマクロの一般形式は、以下のようになります。ここで、VAR はデータ項目を示す変数名で、OPTION はデータ項目に関する取扱い方法を示します。

```
`scl_field_type_m(VAR,OPTION)
```

データ項目が enum 型である場合には、VAR の直前に enum 型を指定しなければなりません。OPTION としては、現在はプリント処理機能のみが実装されているため、SCL_DEFAULT を指定しておけば十分です。現在実装されている OPTION を表 6-1 に示します。

表 6-1　OPTION の指定項目

OPTION	機能
SCL_UNSIGNED	byte、shortint、int、longint、time、integer 型の値を符号なしとして扱うためのオプションです。
SCL_BREAKUP	数値の桁を考慮して見易くプリントする機能です。
SCL_COPYABLE	該当するフィールドをコピー処理の対象にするオプションです。ブロッキング代入文で割り当て可能なフィールドに対して、このオプションを指定して下さい。クラスのハンドルに関しては、shallow copy が適用されます。deep copy をする場合には、ユーザ自身でコピー処理を準備して下さい。
SCL_DEFAULT	標準値を示します。SCL_COPYABLE は標準値に含まれます。

参考 6-2

SCL_DEFAULT からオプションを外すためには、 SCL_DEFAULT&~option として下さい。例えば、以下のようにすると、SCL_DEFAULT から SCL_COPYABLE が外されます。

```
`scl_field_integral_m(d,SCL_DEFAULT&~SCL_COPYABLE)
```

□

フィールドマクロの開始と終了の意味は、表 6-2 の通りです。

表 6-2　フィールドマクロの開始と終了

フィールドマクロ名	機能
`scl_field_begin_m(CLASS_TYPE)	フィールドマクロの開始を示します。 CLASS_TYPE にはクラス名を指定して下さい。
`scl_field_end_m	フィールドマクロの終了を示します。

オブジェクトやコンポーネントの定義をする際に指定するマクロとフィールド開始を宣言するマクロを一つにまとめるマクロもあります（表 6-3）。

表 6-3　短縮形のフィールド開始マクロ

基本的な使用法	短縮形マクロ
`scl_object_m(TYPE)** **`scl_field_begin_m(TYPE) … `scl_field_end_m	**`scl_object_field_begin_m(TYPE)** … `scl_field_end_m
`scl_component_m(TYPE)** **`scl_field_begin_m(TYPE) … `scl_field_end_m	**`scl_component_field_begin_m(TYPE)** … `scl_field_end_m

例えば、オブジェクトを定義する際には以下のようにマクロを使用します。

```
`scl_object_m(simple_item_t)
`scl_field_begin_m(simple_item_t)
   `scl_field_integral_m(a,SCL_DEFAULT)
   ...
`scl_field_end_m
```

こうする代わりに、以下のように短縮形を指定できます。

```
`scl_object_field_begin_m(simple_item_t)
   `scl_field_integral_m(a,SCL_DEFAULT)
   ...
`scl_field_end_m
```

使用するフィールドマクロは、変数が単数形であるか、あるいはアレイであるかで大きく異なります。以下では、それぞれを区別してマクロの種類と意味を解説します。

6.1　単数形変数

変数が単数形である場合には、SystemVerilog のデータタイプに対応したフィールドマクロが定義されています。

6.1.1　`scl_field_integral_m

```
` scl_field_integral_m(VAR,OPTION)
```

logic、bit、reg 型の変数を指定する場合に使用します。

6.1.2　` scl_field_byte_m

```
` scl_field_byte_m(VAR,OPTION)
```

byte 型の変数に使用します。

7　検証コンポーネント

検証コンポーネントを開発するためのベースクラスとしては、表 7-1 に示すクラスがあります。これらのクラスは、メソドロジークラスと呼ばれ、scl_component_t クラスを基にして作られています。本章では、これらの検証コンポーネントを解説します。

表 7-1　主要な検証コンポーネントのクラス

SCL クラス	機能
scl_driver_t	ドライバーのベースクラスです。ユーザはドライバーをこのクラスのサブクラスとして定義しなければなりません。ドライバーは、ジェネレータからトランザクションを取得する機能を備えています。
scl_generator_t	ジェネレータのベースクラスです。ジェネレータは、ドライバーの要求に応じてトランザクションを生成して戻します。実際には、シナリオにより割り当てられたプレイを介して、トランザクションを生成して貰います。
scl_collector_t	コレクターのベースクラスです。コレクターは、DUT からのレスポンスをサンプリングしてトランザクションに変換して、モニターにトランザクションを送信する役目を持ちます。一般的には、コレクターは検証に関わる作業を担当しません。
scl_monitor_t	モニターのベースクラスです。モニターは、コレクターから受信したトランザクションを他の検証コンポーネントに一斉に転送します。モニター自身も簡単な検証作業を行います。
scl_agent_t	エージェントのベースクラスです。エージェントは、ドライバー、ジェネレータ、コレクター、モニターから構成される最小単位の階層的検証コンポーネントです。
scl_env_t	エンバイロンメントのベースクラスです。エンバイロンメントは、エージェントや他のエンバイロンメントから構成される階層的な検証コンポーネントで、検証内容に応じて様々な規模のエンバイロンメントが開発されます
scl_scoreboard_t	スコアボードのベースクラスです。スコアボードは、DUT からのレスポンスを検証する役目を持ちます。
scl_test_t	テストのベースクラスです。テストは、一般に、複数のエンバイロンメントから構成されますが、テスト同士が共有する資源をベースクラスとして定義するのが一般的です。

以下では、検証コンポーネントのベースクラスの仕様と機能を解説し、ユーザの検証コンポーネントを定義する手順を紹介します。

7.1　ドライバー

ドバイバーは、get ポートを使用してジェネレータからトランザクションを取得しますが、scl_driver_t クラスはトランザクションを操作するための基本的機能を提供します。ユーザは、このクラスからサブクラスを定義する事により、定型的な準備作業から解放されます。

7.1.1　ドライバーのベースクラス

ドライバーのベースクラス scl_driver_t は以下のように定義されています。

```
class scl_driver_t #(type TR=scl_transaction_t)
    extends scl_component_t;
`scl_decl_component_m(scl_driver_t#(TR))
// members
scl_get_port_t#(TR,scl_driver_t)     m_get_port;

// methods
extern function new(string name,scl_component_t parent);
```

8　検証手順とシナリオ

シナリオとは、テストケースを実行するための手順で、バッチ処理のスクリプトの役割に相当します。テストケースがシナリオを決定すると、ジェネレータにプレイ（脚本）が割り当てられ、トランザクションを生成するための脚本が確定します。選択された脚本は、幾つかのアクト（幕）から構成されて、脚本に記されている順に幕が進行します。このようにして、検証手順が遂行されて行きます（図 8-1）。本章では、テストを構成するジェネレータ、シナリオ、プレイ、アクトの関係を詳しく解説します。

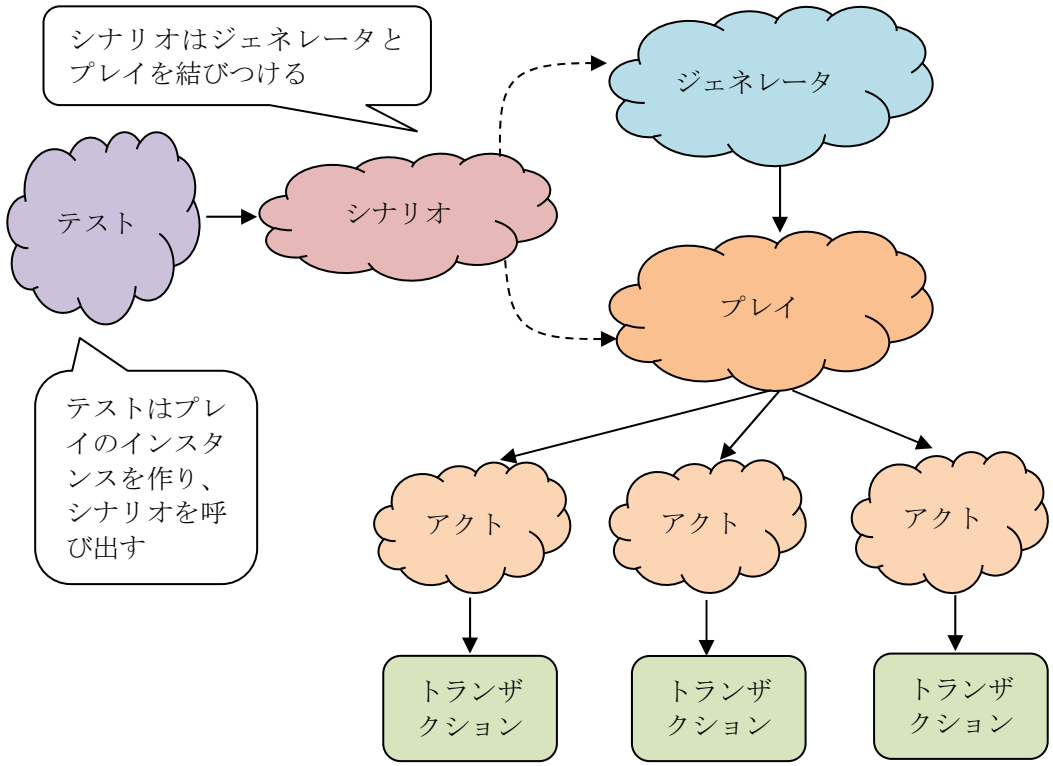


図 8-1　シナリオの役割とトランザクション生成処理

8.1　シナリオ

シナリオはオブジェクトですが、特別なインスタンスを作らずに使用できます。したがって、ユーザはシナリオのサブクラスを定義する必要がないので、使用法は非常に簡単です。

8.1.1　シナリオのベースクラス

シナリオのベースクラスは、以下のように定義されています。

```
class scl_scenario_t extends scl_object_t;
typedef   scl_config_t#(scl_play_t)  CONFIG_TYPE;
// members
const static string m_type_name = "scl_scenario_t";

// methods
extern function new(string name="scenario");
extern function string get_type_name();
extern static function void
   allocate(input string path,scl_play_t play);
extern static function bit
   get(input string path,output scl_play_t play);
endclass
```

主なプロパーティは、表 8-1 にまとめられています。

```
      `scl_create_object_m(play_reset_down_t,"play"));
endfunction
endclass
```

■

例 8-2　ジェネレータに割り当てられているプレイを取得する例

ジェネレータのベースクラスは connect_phase()でシナリオを利用して割り当てられているプレイを取得します。

```
class scl_generator_t #(type TR=scl_transaction_t)
   extends scl_component_t;
...
function void connect_phase(scl_run_param_t param);
   if( !scl_scenario_t::get(get_full_name(),m_play) )
      scl_warning({get_full_name(),
         ": ",scl_global_t::m_msg_couldnt_get_default_scenario});
endfunction
endclass
```

■

参考 8-1

scl_scenario_t::get()は検証コンポーネントに割り当てられたプレイを取得しますが、この取得処理は scl_generator_t クラスが行うため、ユーザがこのメソッドを使用する必要はありません。しかし、既に解説したように、ユーザがシナリオをテストケースに定義するためには scl_scenario_t:: allocate()メソッドを使用しなければなりません。
□

8.2　プレイ

プレイはオブジェクトであり、トランザクションを作る手順を定義する機能を持ちます。プレイは、仮想的にアクトのリストを保有しています。シミュレーション開始時には、リストの最初のアクトがアクティブでありそれが実行されると、リスト上にある次のアクトがアクティブになります。こうして、リストの最初から最後までのアクトが順に実行されるようになります。

ドライバーがジェネレータにトランザクションを要求すると、ジェネレータは現在アクティブなアクトを実行してトランザクションを生成して貰います。その後、ジェネレータは生成されたトランザクションをドライバーに戻します。

8.2.1　プレイのベースクラス

プレイのベースクラスは、以下のように定義されています。パラメータとしてトランザクションのタイプを指定しなければなりません。

```
class scl_play_t #(type TR=scl_transaction_t) extends scl_object_t;
`scl_decl_object_m(scl_play_t#(TR))
// members
TR                  m_item;
scl_item_status_e   m_item_status;
bit                 m_reuse_transaction;

// methods
extern function new(string name);
```

ここで使用したマクロ`scl_act_get_item_m と`scl_act_get_group_m は 8.3.3 項で解説されます。
■

8.3 アクト

アクトは、トランザクションを作る機能を持ちます。原則として、一つのアクトは一つのトランザクションを作ると役目を終わりますが、アクトに階層を設けて複数のアクトを生成する事もできます。

8.3.1 アクトのベースクラス

アクトのベースクラスは、以下のように定義されています。パラメータとしてトランザクションのタイプを指定しなければなりません。

```
class scl_act_t #(type TR=scl_transaction_t) extends scl_object_t;
`scl_decl_object_m(scl_act_t#(TR))
typedef   scl_play_t#(TR)     PLAY_TYPE;
// members
PLAY_TYPE    m_play;

// methods
extern function new(string name);
extern function void set_play(PLAY_TYPE play);
extern virtual task get_item();
extern virtual task get_group();
extern virtual function void make_item();
endclass
```

`scl_decl_object_m マクロは、scl_act_t のクラスのタイプ情報を登録するマクロで、以下の情報を生成します。

```
static string m_type_name = "scl_act_t#(TR)";
function string get_type_name();
   get_type_name = m_type_name;
endfunction
```

主なプロパーティは、表 8-4 にまとめられています。

表 8-4　scl_act_t の主なプロパーティ

プロパーティ	説明
m_type_name	このクラスタイプを示す文字列です。サブクラスにより再定義されます。
m_play	このアクトが属するプレイを示すハンドルです。

参考 8-2

アクトを定義する際、get_group()メソッドか make_item()メソッドのいずれかを選択する事になります。両方のメソッドを定義する必要はありません。アクトがツリーのリーフノードを表現するのであれば、make_item()メソッドを定義し、アクトがツリーの内部ノードを表現するのであれば、get_group()メソッドを定義すれば良いです。
□

このクラスに定義されている主なメソッドを以下にまとめます。

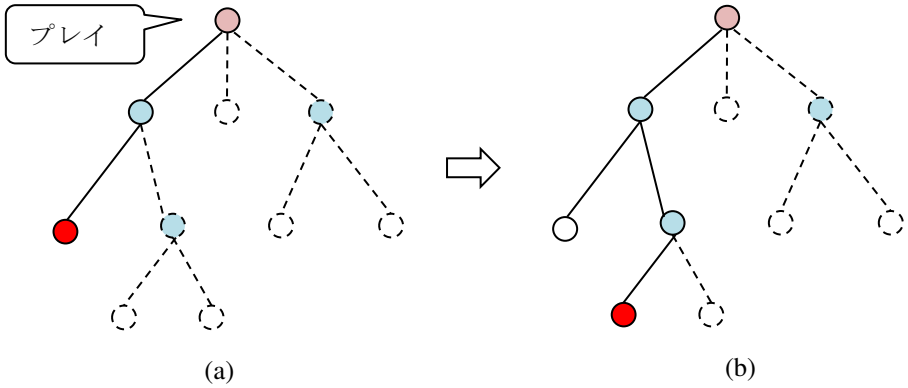


図 8-4　シミュレーション実行開始直後のツリーの状態

ドライバーがトランザクションの要求をすると、ジェネレータはアクティブなアクトからトランザクションを受け取りドライバーに戻します。同時に、プレイの `get_item()` は次のアクティブノードを作ります（図 8-4 (b)）。このようにして、トランザクションのツリーが順に完成して行きます。シミュレーションが終了した時にはツリーが完成した事になります。

ツリーのルートノードはプレイで、リーフノードがトランザクションを作り、内部ノードは階層を作るための役割を持ちます。リーフノードは左から順に実行され、そのうちの一つだけがアクティブなノードで、アクティブなノードがトランザクションを生成します。内部ノードは、アクトの `get_group()` メソッドに対応し、リーフノードはアクトの `make_item()` メソッドに対応します。最終的には、図 8-5 に示すようなツリーが完成します。

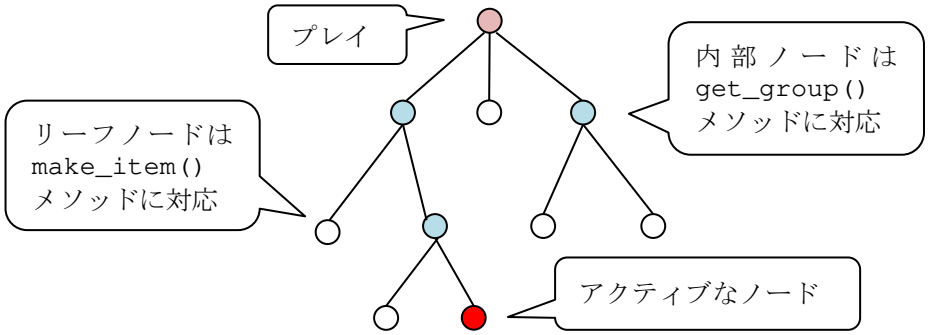


図 8-5　トランザクション作成用のツリー

参考 8-3

本書でトランザクション生成ツリーを図示する時には完成したツリーで表現しますが、実際には一部のノードや枝は未完成の状態にあると理解して下さい。
□

8.4.3　ジェネレータとアクトのハンドシェーク

ドライバーが、ジェネレータにトランザクションを要求すると、ジェネレータがアクトに対して REQUEST_ITEM イベントを起こします。すると、現在アクティブなアクトがトランザクションを作り ITEM_READY イベント起こします。そして、ジェネレータは作られたトランザクションをドライバーに戻します。同時に、現在アクティブであったノードの次に並んでいるリーフノードが次のアクティブなノードになります。全てのリーフノードの処理が終了するとトランザクション処理の終了となります。これらの流れを図 8-6 に示します。尚、既に紹介したように内部ノードを利用するとリーフノードを何回か繰り返す事もできます。

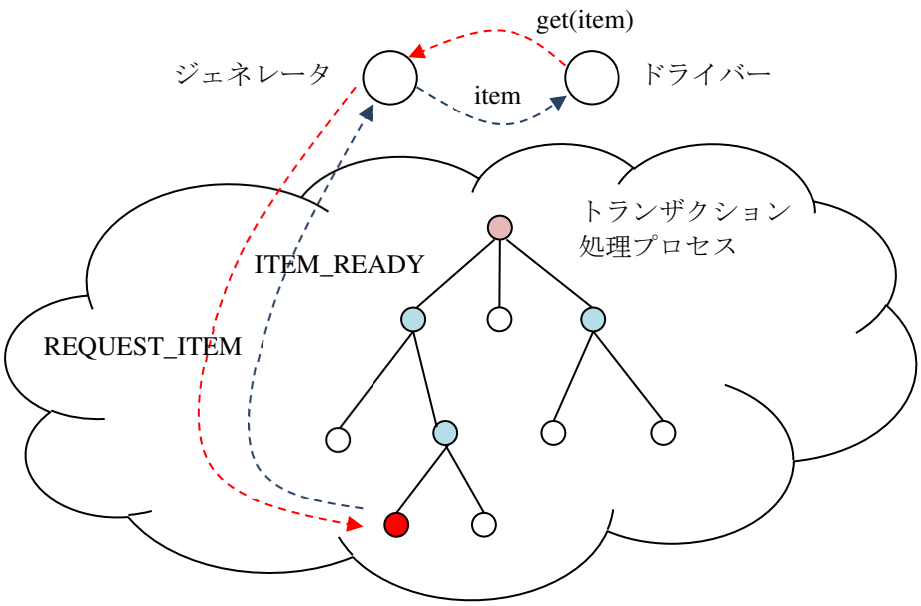


図 8-6　トランザクション生成処理の流れ

9　検証環境構築例

本章では検証環境に SCL を適用する例を紹介します。簡単なシーケンシャル回路（非同期リセット信号を持つアップダウンカウンター）を DUT として検証環境を構築します。以下では、二種類の検証環境構築法を紹介します。最初の検証環境では、モニターが検証結果をプリントしますが、二番目の検証環境では、スコアボードを使用して検証結果を確認します。

9.1　検証環境（モニターによる検証）

図 9-1 に示すような検証環境を構築します。また、表 9-1 は検証環境の主な構成要素の機能を示しています。

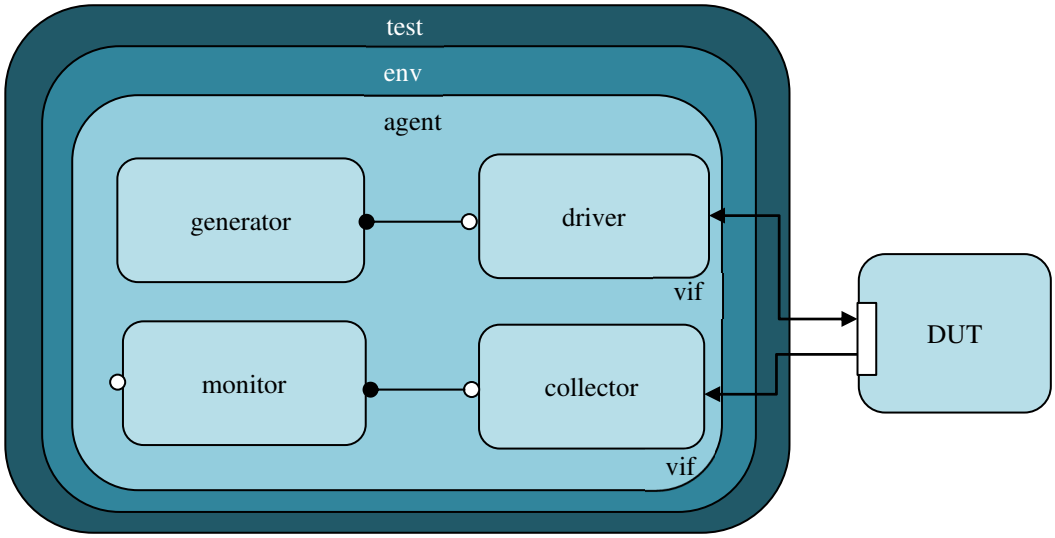


図 9-1　SCL による検証環境構築例

表 9-1　検証環境を構成する主な要素

構成要素	機能
up_down_counter	DUT のアップダウンカウンターです。
pkg_definitions	検証環境で共有する情報を定義しておきます。
simple_if	インターフェースを示します。
simple_item_t	トランザクションを示すクラスです。
act_down_t act_load_t act_loaddown_t act_reset_t act_up_t act_upup_t act_upuploaddown_t	トランザクションは RESET、LOAD、UP、DOWN 等のコマンド機能を備えているので、それらのコマンドに対応するアクトを定義しておきます。
play_t	テストケースのベースクラスです。
play_reset_down_t	テストケース 1 のシナリオを生成するクラスです。
play_reset_up_t	テストケース 2 のシナリオを生成するクラスです。
driver_t	DUT をドライブするドライバーのクラスです。
generator_t	ドライバーの要求によりトランザクションを生成するクラスです。このクラスの run_phase()には、テストケースのシナリオが割り当てられます。
collector_t	DUT からのレスポンスをサンプリングするコレクターのクラスです。
monitor_t	コレクターから受信したトランザクションの処理をします。この

	例では、トランザクションをプリントするだけの簡単な処理内容になっています。
agent_t	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るエージェントのクラスです。
env	エージェントから構成されるトップレベルのエンバイロンメントクラスです。
test_base_t	テストのベースクラスを示します。二つのテストに共通する機能をベースクラスに吸収し、テストの記述を簡略化します。
test1_t	テストケース 1 を実行するクラスです。
test2_t	テストケース 2 を実行するクラスです。
pkg	検証環境に必要な資源をパッケージに集めておきます。
top	トップモジュールを示します。

検証環境の構成要素を順に解説します。

9.1.1 up_down_counter

アップダウンカウンターは、非同期なリセット信号を持ち、通常の動作時には load 信号と up_down 信号の値により機能が決定されます（表 9-2）。load 信号の方が up_down 信号よりも優先順位が高く設定されています。

表 9-2　アップダウンカウンターの仕様

ポート	意味
clk	クロック信号です。
reset	非同期なリセット信号です。
load	load==1 であれば、データを外部からロードします。
up_down	load==0 の時に動作する機能で、up_down==1 であればカウンターをインクリメントし、up_down==0 であればカウンターをデクリメントします。
d	load==1 の時にレジスターにロードする値を意味します。
q	現在のカウンター値を示します。
qn	q の値をビット毎に反転した値を示します。

アップダウンカウンターの記述は以下のようになります。

```
module up_down_counter #(NBITS=4)
    (input clk,reset,load,up_down,logic [NBITS-1:0] d,
    output logic [NBITS-1:0] q,qn);
logic [NBITS-1:0]   counter;

assign q = counter;
assign qn = ~counter;

always @(posedge clk,posedge reset)
    if( reset )
        counter <= 0;
    else if( load )
        counter <= d;
    else if( up_down )
        counter <= counter + 1;
    else
        counter <= counter - 1;

endmodule
```

9.1.2 pkg_definitions

以下のように、検証環境で共有される情報を定義しておきます。

```
package pkg_definitions;
parameter UP_DOWN_WIDTH = 8;
typedef enum { RESET, LOAD, UP, DOWN }  up_down_op_e;
endpackage
```

DUT を操作するためのコマンドは up_down_op_e に定義されていますが、コマンドは表 9-3 のような意味を持ちます。

表 9-3　up_down_op_e の定義内容

シンボル	意味
RESET	DUT をリセットします。
LOAD	DUT に値をロードさせます。
UP	DUT のカウンターをインクリメントさせます。
DOWN	DUT のカウンターをデクリメントさせます。

9.1.3 simple_if

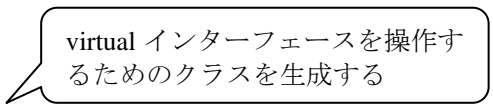
インターフェースには、DUT に接続する信号とクロッキングブロックを定義しておきます。そして、virtual インターフェースを操作するためのクラス（vif_config）を生成しておきます。

```
interface simple_if import pkg_definitions::*; (input logic clk);
logic [UP_DOWN_WIDTH-1:0]  d, q, qn;
logic                      reset, load, up_down;

clocking cb @(posedge clk); endclocking
clocking cbr @(posedge reset); endclocking

initial begin
   reset = 0;
   load = 0;
   up_down = 1;
   d = '0;
end
endinterface

`scl_vif_config_m(virtual simple_if,vif_config)
```



9.1.4 simple_item_t

トランザクションには、DUT のポートに対応する変数を全て定義しておきます。

```
class simple_item_t extends scl_transaction_t;
logic [UP_DOWN_WIDTH-1:0]  d, q, qn;
logic                      reset, load, up_down;
`scl_object_field_begin_m(simple_item_t)
   `scl_field_integral_m(reset,SCL_DEFAULT)
   `scl_field_integral_m(load,SCL_DEFAULT)
   `scl_field_integral_m(up_down,SCL_DEFAULT)
   `scl_field_integral_m(d,SCL_DEFAULT)
   `scl_field_integral_m(q,SCL_DEFAULT)
   `scl_field_integral_m(qn,SCL_DEFAULT)
`scl_field_end_m
```

```
`scl_object_new_default_m("simple_item")
extern function up_down_op_e make_op();
endclass
```

make_op()メソッドは、reset、load、up_down からコマンド RESET、LOAD、UP、DOWN を作り出します。このメソッドは、以下のように定義されています。

```
function up_down_op_e simple_item_t::make_op();
   if( reset )
      make_op = RESET;
   else if( load )
      make_op = LOAD;
   else if( up_down )
      make_op = UP;
   else
      make_op = DOWN;
endfunction
```

9.1.5 act_down_t

DUT に DOWN コマンドを送るためのアクトです。

```
class act_down_t extends scl_act_t#(simple_item_t);
`scl_object_m(act_down_t)
`scl_object_new_m
extern function void make_item();
endclass

// make_item
function void act_down_t::make_item();
   m_play.m_item.reset = 0;
   m_play.m_item.d = 0;
   m_play.m_item.up_down = 0;
   m_play.m_item.load = 0;
endfunction
```

9.1.6 act_load_t

DUT に LOAD コマンドを送るためのアクトです。

```
class act_load_t extends scl_act_t#(simple_item_t);
`scl_object_m(act_load_t)
`scl_object_new_m
extern function void make_item();
endclass

// make_item
function void act_load_t::make_item();
   m_play.m_item.reset = 0;
   m_play.m_item.d = $random;
   m_play.m_item.up_down = 0;
   m_play.m_item.load = 1;
endfunction
```

9.1.7 act_loaddown_t

階層的なアクトを表現し、LOAD と DOWN コマンドを順に生成します（図 9-2）。

```
class act_loaddown_t extends scl_act_t#(simple_item_t);
// members
act_load_t       act_load;
act_down_t       act_down;
`scl_object_m(act_loaddown_t)
`scl_object_new_m

// methods
task get_group();
   `scl_act_get_item_m(act_load,m_play)
   `scl_act_get_item_m(act_down,m_play)
endtask
endclass
```

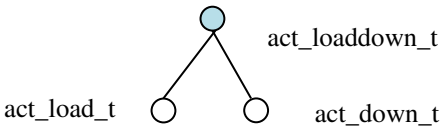


図 9-2　act_loaddown_t が表現する部分ツリー

9.1.8　act_reset_t

DUT に RESET コマンドを送るためのアクトです。

```
class act_reset_t extends scl_act_t#(simple_item_t);
`scl_object_m(act_reset_t)
`scl_object_new_m
extern function void make_item();
endclass

// make_item
function void act_reset_t::make_item();
   m_play.m_item.reset = 1;
   m_play.m_item.d = 0;
   m_play.m_item.up_down = 0;
   m_play.m_item.load = 0;
endfunction
```

9.1.9　act_up_t

DUT に UP コマンドを送るためのアクトです。

```
class act_up_t extends scl_act_t#(simple_item_t);
`scl_object_m(act_up_t)
`scl_object_new_m
extern function void make_item();
endclass

// make_item
function void act_up_t::make_item();
   m_play.m_item.reset = 0;
   m_play.m_item.d = 0;
   m_play.m_item.up_down = 1;
   m_play.m_item.load = 0;
endfunction
```

9.1.10 act_upup_t

このアクトは、階層的なコマンドを作るための例として導入しました。この場合には、UP コマンドを二回繰り返すように get_group()タスクを定義しています。アクトに複雑な処理がある場合には、このようにして処理の細分化を実現できます。このアクトは図 9-3 のような部分ツリーを表現します。

```
class act_upup_t extends scl_act_t#(simple_item_t);
// members
act_up_t      act_up;
`scl_object_m(act_upup_t)
`scl_object_new_m

// methods
task get_group();
    `scl_act_get_item_m(act_up,m_play);
    `scl_act_get_item_m(act_up,m_play);
endtask

endclass
```

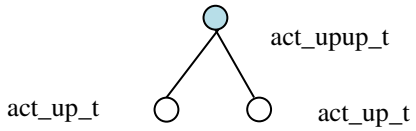


図 9-3　act_upup_t が表現する部分ツリー

9.1.11 act_upuploaddown_t

階層的なアクトを表現しています。階層的な act_upup_t のアクトと act_loaddown_t のアクトを順に呼び出します（図 9-4）。このように階層の深さに制限はありません。

```
class act_upuploaddown_t extends scl_act_t#(simple_item_t);
// members
act_upup_t       act_upup;
act_loaddown_t   act_loaddown;
`scl_object_m(act_upuploaddown_t)
`scl_object_new_m

// methods
task get_group();
    `scl_act_get_group_m(act_upup,m_play);
    `scl_act_get_group_m(act_loaddown,m_play);
endtask
endclass
```

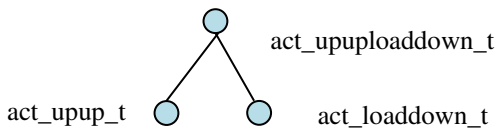


図 9-4　act_upuploaddown_t が表現する部分ツリー

9.1.12 play_t

テストケースのベースクラスを以下のように定義しておきます。役目は、それぞれのテストケースが使用する変数を定義するだけです。

```
class play_t extends scl_play_t#(simple_item_t);
act_reset_t   act_reset;
act_down_t    act_down;
act_up_t      act_up;
act_load_t    act_load;
act_upuploaddown_t  act_upuploaddown;
`scl_object_m(play_t)
`scl_object_new_m
endclass
```

9.1.13 play_reset_down_t

テストケース 1 のためのクラスです（図 9-5）。

```
class play_reset_down_t extends play_t;
// members
//
`scl_object_m(play_reset_down_t)
// methods
`scl_object_new_default_m("play_reset_down");

task get_item();
    `scl_act_get_item_m(act_reset,this)
    `scl_act_get_group_m(act_upuploaddown,this)
endtask
endclass
```

階層的なアクト

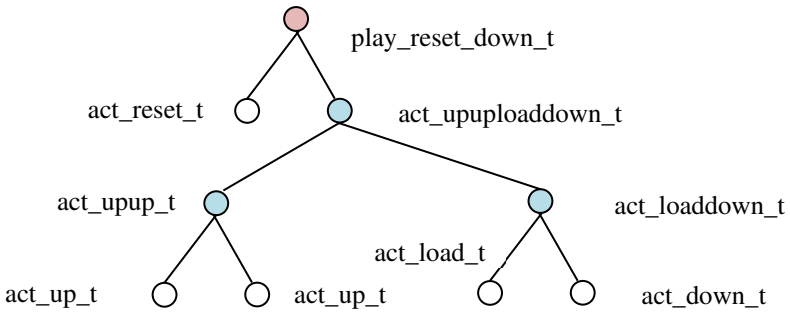


図 9-5　テストケース 1 のツリー

9.1.14 play_reset_up_t

テストケース 2 のクラスです。このテストケースでは階層を設けずにテストを実現しています（図 9-6）。

```
class play_reset_up_t extends play_t;
// members
`scl_object_m(play_reset_up_t)

// methods
`scl_object_new_default_m("play_reset_up")

task get_item();
    `scl_act_get_item_m(act_reset,this)
    `scl_act_get_item_m(act_down,this)
    `scl_act_get_item_m(act_load,this)
    `scl_act_get_item_m(act_up,this)
    `scl_act_get_item_m(act_up,this)
```

階層を持たないアクト

```
endtask
endclass
```

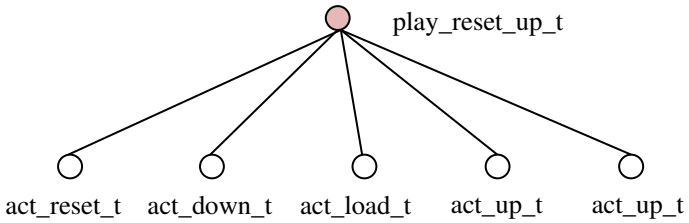


図 9-6　テストケース 2 のツリー

9.1.15　driver_t

ドライバーの全容は以下のようになります。必ず、virtual インターフェースを宣言しなければなりません。

```
class driver_t extends scl_driver_t#(simple_item_t);
vif_config::vif_type   vif;
`scl_component_m(driver_t)
`scl_component_new_m
`scl_extern_connect_phase_m
`scl_extern_run_phase_m
extern task drive_dut(TR item);
endclass
```

virtual インターフェースの定義

connect_phase()において、以下のようにして virtual インターフェースを設定します。

```
`scl_connect_phase_m(driver_t)
   `scl_super_connect_phase_m
   vif = vif_config::get();
`scl_end_connect_phase_m
```

run_phase()では以下のようにして、ジェネレータよりトランザクションを取得して、DUT をドライブします。

```
`scl_run_phase_m(driver_t)
TR  item;
   m_get_port.m_connected_port.set_option(
          SCL_PORT_REUSE_TRANSACTION);
   forever begin
      m_get_port.get(item);
      drive_dut(item);
      @(negedge vif.clk);
   end
`scl_end_run_phase_m
```

DUT をドライブする部分は以下のようになります。

```
task driver_t::drive_dut(TR item);
   vif.reset <= item.reset;
   vif.load <= item.load;
   vif.up_down <= item.up_down;
   vif.d <= item.d;
   if( item.reset )
      vif.reset = #1 0;
endtask
```

参考 9-1

非同期信号 reset に値を設定する場合には、ノンブロッキング代入文を使用すると安全です。一般的に、シーケンシャル回路を検証する際に DUT をドライブするためには、ノンブロッキング代入文を使用する方が安全です。
□

9.1.16 generator_t

ジェネレータは、ベースクラスが殆どの処理を担当するので、ユーザは以下のようにジェネレータを定義するだけで済みます。

```
`scl_generator_m(generator_t,simple_item_t)
```

9.1.17 collector_t

コレクターの全容は以下のようになります。ドライバーと同様に virtual インターフェースを宣言しなければなりません。

```
class collector_t extends scl_collector_t#(simple_item_t);
vif_config::vif_type   vif;
simple_item_t    item;
`scl_component_m(collector_t)
`scl_component_new_m
`scl_extern_connect_phase_m
`scl_extern_run_phase_m
extern task collect_regular();
extern task collect_reset();
extern function void send_item();
endclass
```

virtual インターフェースの定義

connect_phase()で virtual インターフェースの使用を以下のように準備します。

```
`scl_connect_phase_m(collector_t)
   vif = vif_config::get();
`scl_end_connect_phase_m
```

run_phase()では、DUT からのレスポンスを監視するために、クロッキングブロックを使用してイベントの発生を待ちます。

```
`scl_run_phase_m(collector_t)
   item = `scl_create_object_m(simple_item_t,"item");
   fork
      collect_regular();
      collect_reset();
   join
`scl_end_run_phase_m
```

クロックのイベントを collect_regular()メソッドで監視します。

```
task collector_t::collect_regular();
   forever begin
      @vif.cb;
      send_item();
   end
endtask
```