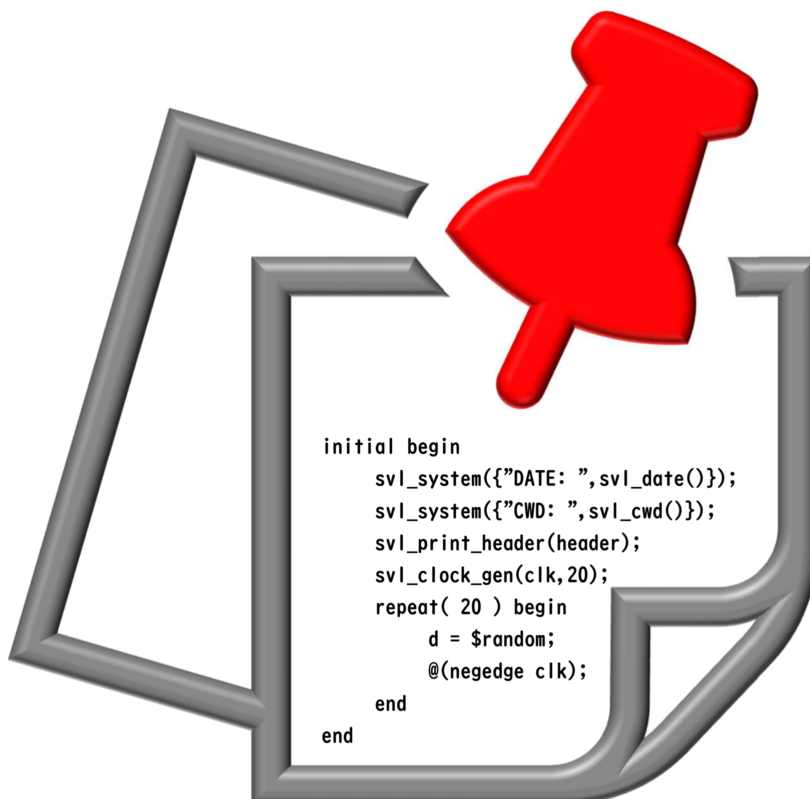
Artgraphics

SVL Premium

検証のための SystemVerilog ライブラリー

生産性向上のための SystemVerilog パッケージ

Document Identification Number: ARTG-TD-006-2024
Document Revision: 1.0, 2024.11.25
アートグラフィックス



SVL Premium
検証のための SystemVerilog ライブラリー

© 2024 アートグラフィックス
〒124-0012 東京都葛飾区立石 8-14-1
www.artgraphics.co.jp

SystemVerilog Library (SVL) for Verification

© 2024 Artgraphics. All rights reserved.
8-14-1, Tateishi, Katsushika-ku, Tokyo, 124-0012 Japan
www.artgraphics.co.jp

注意事項

- **SystemVerilog Library** （以下、**SVL** と略称）は、そのままの形で提供されるものであり、特別な技術サポートは含まれていません。
- **SVL** のソースコードにあるコピーライトは、常に、存在するようにして下さい。
- **SVL** および本解説書により得られた知識・情報の使用から生じるいかなる損害についても、弊社および本書の著者は責任を負わないものとします。
- 弊社の技術資料の内容の一部、あるいは全部を無断で複写、複製、転載する事は、禁じます。
- 弊社の技術資料の譲渡、転売、模倣、または、改造等の行為を禁止致します。

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、検証のための SystemVerilog ライブラリー（SVL）の解説書です。SVL は、SystemVerilog による検証作業で必要となる基本機能を含む SystemVerilog パッケージです。SVL には SVL Standard と SVL Premium の二種類があります。SVL Standard は検証作業で必要となる基本機能で構成され、SVL Premium は基本機能に加えて、検証環境構築時に必要となる機能で構成されています。本書は SVL Premium の解説書です。

SystemVerilog は豊富な機能を備えていますが、プログラミングに適した機能を備えているわけではありません。例えば、十進数を 123,456 のようにプリントする機能はありません。しかし、検証結果を見易く記録するためには、そのような機能が必要になる時があります。同様に、SystemVerilog には 16 進数を `32'h010a_ff34` のようにプリントする機能はありませんが、検証結果を見易くするためには必要な機能です。これらの一例が示すように、SystemVerilog には検証で必要とされる汎用的なプログラミング機能が不足しています。SVL はその欠乏した機能を補完する役目を持っています。

SVL ではマクロが使用されてプリント書式の自動生成が行われています。その結果、マクロにより汎用的なプログラミング記述が可能となっています。以下に、簡単なマクロ使用例を紹介します。

記述法	生成された書式の実行結果例
`` `svl_sformatfb_m(v) ``	10101011110011010000000100100011
`` `svl_sformatfo_m(v) ``	25363200443
`` `svl_sformatfh_m(v) ``	abcd0123
`` `svl_pbreakupb_m(v) ``	'b1010_1011_1100_1101_0000_0001_0010_0011
`` `svl_pbreakupo_m(v) ``	'o253_6320_0443
`` `svl_pbreakuph_m(v) ``	'habcd_0123
`` `svl_sprintfb_m(v) ``	32'b1010_1011_1100_1101_0000_0001_0010_0011
`` `svl_sprintfo_m(v) ``	32'o253_6320_0443
`` `svl_sprintfh_m(v) ``	32'habcd_0123
`` `svl_breakupd_m(n) ``	123,456,789

このように、使用するマクロを変えるだけで様々なプリント書式を生成する事ができます。これらのマクロはプリント機能のビルディングブロックであり、SVL ではそれらを組み合わせて更に高度なマクロ機能が構築されています。

SVL では、プリントするためのレイアウトを定義し、そのレイアウトをモデルにしてプリントします。したがって、モデルが変化すると自動的にプリント処理も変更されます。例えば、ビット数が変化すると、モデルが更新されるので、プリント処理も自動的に更新されます。以下に簡単なモデル例を紹介します。

```
svl_print_header_s header[] = '{
    '{"time",5,SVL_HEADER_DEFAULT},
    '{"reset",5,SVL_HEADER_DEFAULT},
    '{"d",`svl_full_hex_digits_m(WIDTH),SVL_HEADER_DEFAULT},
    '{"q",`svl_full_hex_digits_m(WIDTH),SVL_HEADER_DEFAULT}
    };
```

この例では DUT を検証する際の信号値に対応してモデルを定義しています。検証結果をプリントする際には、このモデルを指定してプリントします。こうすると、カラム情報や WIDTH が変化しても自動的にカラム調節が行われます。カラムを入れ替えてもプリント処理を変更する必要はありません。以下は、プリント結果を示しています。

Copyright 2024 © Artgraphics. All rights reserved.

```
最初の仕様（WIDTH==16）           カラム d と q を入れ替えた仕様
=========================   =========================
time   reset d       q       time   reset q       d
-------------------------   -------------------------
@ 10: 0     'hc778 'hc778   @ 10: 0     'hc778 'hc778
@ 30: 0     'hc8d0 'hc8d0   @ 30: 0     'hc8d0 'hc8d0
@ 50: 0     'h1471 'h1471   @ 50: 0     'h1471 'h1471
@ 70: 0     'h2004 'h2004   @ 70: 0     'h2004 'h2004
@ 90: 0     'hea38 'hea38   @ 90: 0     'hea38 'hea38
@100: 1     'h17ef 'h0000   @100: 1     'h0000 'h17ef
@110: 0     'h17ef 'h17ef   @110: 0     'h17ef 'h17ef
@130: 0     'h8562 'h8562   @130: 0     'h8562 'h8562
@150: 0     'hfc26 'hfc26   @150: 0     'hfc26 'hfc26
@170: 0     'h3b02 'h3b02   @170: 0     'h3b02 'h3b02
@190: 0     'h02a2 'h02a2   @190: 0     'h02a2 'h02a2
=========================   =========================
```

カラム名称の左揃え、右揃え、中央揃え等の指定もできます。以下に実行例を紹介します。

```
WIDTH==32 でカラム名称と値を右に揃えた仕様
===================================
time   reset           d           q
-----------------------------------
@ 10:      0 'he093_c778 'he093_c778
@ 30:      0 'h060c_c8d0 'h060c_c8d0
@ 50:      0 'h3154_1471 'h3154_1471
@ 70:      0 'h870a_2004 'h870a_2004
@ 90:      0 'ha9bb_ea38 'ha9bb_ea38
@100:      1 'h700e_17ef 'h0000_0000
@110:      0 'h700e_17ef 'h700e_17ef
@130:      0 'had45_8562 'had45_8562
@150:      0 'haf37_fc26 'haf37_fc26
@170:      0 'h848b_3b02 'h848b_3b02
@190:      0 'hd0cf_02a2 'hd0cf_02a2
===================================
```

検証環境が複雑な検証階層で構成される事は珍しくありませんが、階層の中から特定の検証コンポーネントを検索する際にはワイルドカードによるパターン検索が必要になります。SVL は、その機能を標準的に備えています。以下のように簡単にパターンを指定できます。階層名（pathname）がパターンにマッチすれば if 文の条件は真となります。

```
if( svl_pattern_matching("top.*.driver",pathname) )
```

SVL は、これらのプログラミング機能に加えて検証環境を構築するための重要な機能を含んでいます。例えば、以下のような機能を備えています。

- トランザクションのベースクラス
- メソドロジークラス（ドライバー、ジェネレータ、コレクター、モニター、エージェント、エンバイロンメント、スコアボード、テスト等）
- TLM ポート
- トランザクション生成機能
- テストを実行するためのシナリオ
- 直感的でわかり易い virtual インターフェースの設定と取得機能
- ダイナミックに検証環境を変更する機能

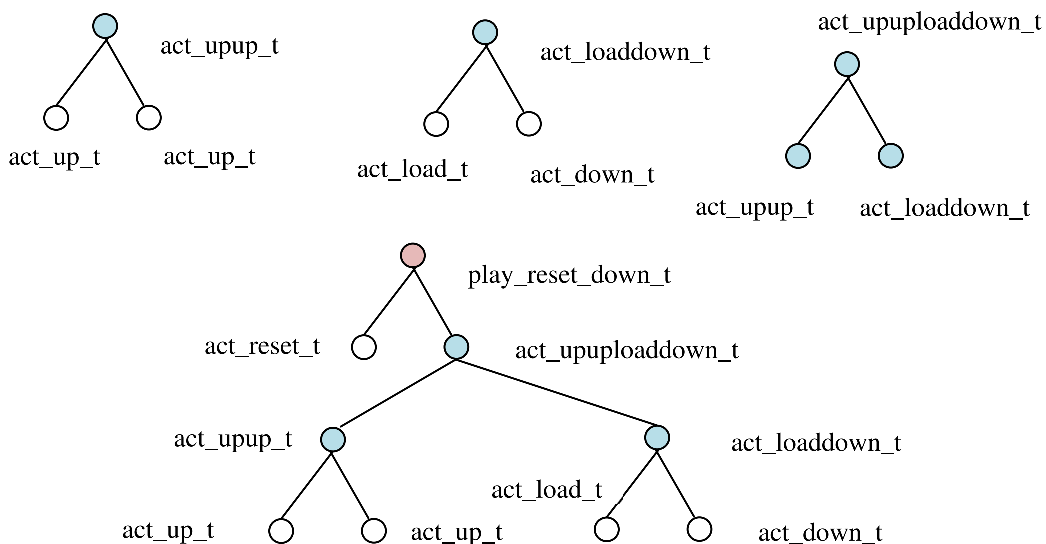
SVL でテストケースを構築する方法は、直感的でわかり易い特長があります。例えば、下記に示すような簡単なシーケンシャル回路を例にとりテストケースの概要を紹介します。

Copyright 2024 © Artgraphics. All rights reserved.

```
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
```

この回路は、RESET、LOAD、UP、DOWN の機能を持ちます。したがって、それらの機能に対応するテストデータを準備し、それらのテストデータを任意の順序で組み合わせれば、様々なテストケースを実現できます。

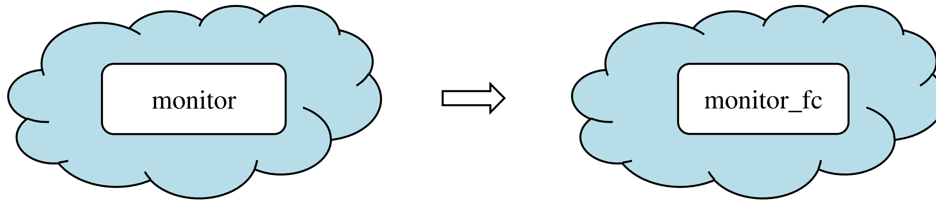
SVL では、テストデータをアクトと呼び、階層的に配置して複雑なテストケースを実現しています。階層構造のルートにはプレイと呼ばれるデータタイプが使用されますが、階層構造の内部ノードもアクトが担当します。内部ノードでは、SystemVerilog のプログラミング機能を使用できるので、下位の部分ツリーを複数回繰り返す事ができます。したがって、実質無数の組み合わせを実現できます。下図は、代表的な組み合わせを示しています。



SVL のジェネレータは階層構造のルートから順に走査してリーフノードを処理します。リーフノードは、割り当てられているトランザクション生成命令を実行してジェネレータに戻します。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。

また、SVL はダイナミックに検証環境を変更する機能を備えています。下図はモニターをカバレッジ機能搭載したモニターに入れ替える状態を示しています。ただし、ここでは `monitor_fc_t` は `monitor_t` のサブクラスであると仮定します。

Copyright 2024 © Artgraphics. All rights reserved.



このダイナミックな変更機能を利用するためには、`svl_auto_component_m マクロでモニターを宣言しなければなりません。こうすると、monitor_t のインスタンスを monitor_fc_t のインスタンスで置き換える事が可能になります。例えば、以下のようにトップモジュールでタイプ変換を指定しておけば十分です。

```
module top;
import svl_pkg::*;
import pkg::*;
test_t test;

initial begin
    `svl_change_type_m(monitor_t,monitor_fc_t)
    ...
end
endmodule
```

置き換えるクラスタイプを指定する

このように準備しておくと、monitor には monitor_t のインスタンスの代わりに monitor_fc_t のインスタンスが作られます。

```
class agent_t extends svl_agent_t;
driver_t      driver;
generator_t   generator;
collector_t   collector;
monitor_t     monitor;
`svl_auto_component_m(agent_t)
`svl_component_new_m
`svl_extern_build_step_m
`svl_extern_connect_step_m
endclass
function void agent_t::build_step(svl_run_param_t param);
    super.build_step(param);
    monitor = `svl_create_component_m(monitor_t,"monitor",this);
    ...
endfunction
```

monitor には monitor_t の代わりに monitor_fc_t のインスタンスが割り当てられます

勿論、SVL にはこの他にも便利な機能が多く備えられています。特に、文字列処理機能が豊富です。本書は、SVL が提供する全ての機能を使用例と共に詳しく解説しています。

アートグラフィックス

Copyright 2024 © Artgraphics. All rights reserved.

変更履歴

日付	Revision	変更点
2024.11.25	1.0	初版。

Copyright 2024 © Artgraphics. All rights reserved.

目次

1 概要 1
1.1 SVL の意義と目的 1
1.2 SVL の機能概要 4
1.3 SVL の使用手順 4
1.4 SVL の記法 5
1.5 本書の記法 5
2 SVL の構成 6
2.1 クラス階層 6
2.2 SVL_VOID_T 6
2.3 SVL_THING_T 7
2.3.1 new 8
2.3.2 get_name 8
2.3.3 get_id 8
2.3.4 is_object 8
2.3.5 is_component 8
2.4 SVL_OBJECT_T 8
2.4.1 new 9
2.4.2 is_object 9
2.5 SVL_COMPONENT_T 9
2.5.1 new 11
2.5.2 is_component 11
2.5.3 is_top 11
2.5.4 add_child 11
2.5.5 add_object 12
2.5.6 get_objects 12
2.5.7 get_ordered_children 12
2.5.8 get_parent 12
2.5.9 get_child 12
2.5.10 get_first_child 12
2.5.11 get_last_child 12
2.5.12 get_next_child 12
2.5.13 get_prev_child 13
2.5.14 get_full_name 13
2.5.15 build_step 13
2.5.16 connect_step 13
2.5.17 setup_step 13
2.5.18 run_step 13
2.5.19 collect_step 13
2.5.20 check_step 13
2.5.21 conclude_step 14
2.6 オブジェクトとコンポーネントの生成 14
2.6.1 `svl_create_object_m 14
2.6.2 `svl_create_component_m 14
2.7 コンフィギュレーション設定変更 15
2.7.1 クラスタイプの変更 15
2.7.2 クラスプロパティの変更 16
2.8 シミュレーション 18
2.8.1 実行制御 18
2.8.2 svl_run_test 19
2.8.3 svl_set_timeout 20
2.8.4 svl_set_max_simulation_time 21
2.8.5 svl_get_test_name 21

Copyright 2024 © Artgraphics. All rights reserved.

2.9 SVL マクロ 21
2.9.1 `svl_auto_object_m 21
2.9.2 `svl_object_new_m 22
2.9.3 `svl_object_new_default_m 22
2.9.4 `svl_auto_component_m 22
2.9.5 `svl_component_new_m 23
2.9.6 `svl_extern_build_step_m 23
2.9.7 `svl_extern_connect_step_m 23
2.9.8 `svl_extern_setup_step_m 23
2.9.9 `svl_extern_run_step_m 24
2.9.10 `svl_extern_collect_step_m 24
2.9.11 `svl_extern_check_step_m 24
2.9.12 `svl_extern_conclude_step_m 24
2.9.13 `svl_generator_m 24
3 TLM 25
3.1 ポート 25
3.1.1 svl_port_t 26
3.1.2 svl_io_port_t 27
3.1.3 svl_nbio_port_t 28
3.1.4 svl_pass_port_t 30
3.2 GET 30
3.2.1 svl_get_port_t 31
3.2.2 svl_get_server_t 31
3.2.3 get の使用例 32
3.3 PUT 33
3.3.1 svl_put_port_t 34
3.3.2 svl_put_server_t 35
3.3.3 put の使用例 35
3.4 WRITE 36
3.4.1 svl_send_port_t 37
3.4.2 svl_receive_port_t 38
3.4.3 write の使用例 39
3.5 TRY_GET 40
3.5.1 svl_nbget_port_t 41
3.5.2 svl_nbget_server_t 42
3.5.3 try_get の使用例 42
3.6 TRY_PUT 43
3.6.1 svl_nbput_port_t 43
3.6.2 svl_nbput_server_t 44
3.6.3 try_put の使用例 45
3.7 メソドロジークラスと TLM ポート 45
4 VIRTUAL インターフェースの設定と取得 46
4.1 VIRTUAL インターフェースの使用準備 46
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 47
4.3 VIRTUAL インターフェースの取得 47
5 検証コンポーネント 49
5.1 ドライバー 49
5.1.1 ドライバーのベースクラス 49
5.1.2 ドライバーの定義法 50
5.2 ジェネレータ 51
5.2.1 ジェネレータのベースクラス 51
5.2.2 ジェネレータの定義法 52
5.3 コレクター 52

Copyright 2024 © Artgraphics. All rights reserved.

5.3.1 コレクターのベースクラス 52
5.3.2 コレクターの定義法 53
5.4 モニター 54
5.4.1 モニターのベースクラス 54
5.4.2 モニターの定義法 55
5.5 エージェント 56
5.5.1 エージェントのベースクラス 56
5.5.2 エージェントの定義法 57
5.6 エンバイロンメント 58
5.6.1 エンバイロンメントのベースクラス 58
5.6.2 エンバイロンメントの定義法 58
5.7 スコアボード 59
5.7.1 スコアボードのベースクラス 59
5.7.2 スコアボードの定義法 60
5.8 テスト 60
5.8.1 テストのベースクラス 60
5.8.2 テストの定義法 61

6 検証手順とシナリオ 63

6.1 シナリオ 63
6.1.1 シナリオのベースクラス 63
6.1.2 シナリオの使用法 64
6.2 プレイ 65
6.2.1 プレイのベースクラス 65
6.2.2 プレイの定義法 67
6.3 アクト 68
6.3.1 アクトのベースクラス 68
6.3.2 アクトの定義法 70
6.3.3 アクトを生成するマクロ 71
6.4 ドライバー、ジェネレータ、プレイによるハンドシェーク 72
6.4.1 テストとシナリオ 72
6.4.2 シミュレーション実行開始の動作 72
6.4.3 ジェネレータとアクトのハンドシェーク 73

7 検証支援機能 74

7.1 データタイプと定数 74
7.1.1 enum タイプ 74
7.1.2 クラスとストラクチャ 78
7.2 文字列処理 87
7.2.1 svl_max_len 89
7.2.2 svl_is_alnum 89
7.2.3 svl_is_alpha 89
7.2.4 svl_is_digit 89
7.2.5 svl_is_lower 89
7.2.6 svl_is_upper 89
7.2.7 svl_is_space 90
7.2.8 svl_tolower 90
7.2.9 svl_toupper 90
7.2.10 svl_reverse_string 90
7.2.11 svl_starts_with 90
7.2.12 svl_ends_with 90
7.2.13 svl_first_index 90
7.2.14 svl_first_index_from 90
7.2.15 svl_last_index 91
7.2.16 svl_last_index_from 91

Copyright 2024 © Artgraphics. All rights reserved.

7.2.17 svl_find_first_substr 91
7.2.18 svl_find_last_substr 91
7.2.19 svl_trim 91
7.2.20 svl_replace_head_ws 91
7.2.21 svl_replace_tail_ws 91
7.2.22 svl_replace_first 92
7.2.23 svl_replace_last 92
7.2.24 svl_replace_all 92
7.2.25 svl_replace_substr 92
7.2.26 svl_separate_string 92
7.2.27 svl_strncmp 93
7.2.28 svl_strnicmp 93
7.2.29 svl_strnset 93
7.2.30 svl_ascii_to_hex 93
7.2.31 svl_ascii_to_bin 93
7.2.32 svl_ascii_to_oct 93
7.3 ランダム文字列の生成 94
7.4 正則表現 94
7.4.1 svl_reg_expr_t 94
7.4.2 クラスを使用しない正則表現 96
7.5 ファイル入出力 97
7.5.1 svl_get_line 97
7.5.2 svl_get_lines 97
7.5.3 svl_head 97
7.5.4 svl_tail 98
7.5.5 svl_put_lines 98
7.6 クロック生成 98
7.7 メッセージプリント機能 99
7.7.1 svl_set_print_file 100
7.7.2 svl_suspend_print_file 101
7.7.3 svl_resume_print_file 101
7.7.4 svl_reset_print_file 101
7.7.5 svl_flush_print_file 101
7.7.6 svl_print_message 101
7.7.7 svl_info 102
7.7.8 svl_warning 102
7.7.9 svl_error 102
7.7.10 svl_fatal 103
7.7.11 svl_system 103
7.7.12 svl_system_wonl 103
7.7.13 svl_change_message_prefix 104
7.7.14 svl_set_message_level 104
7.7.15 svl_set_info_message_level 104
7.7.16 svl_set_warning_message_level 104
7.7.17 svl_set_error_message_level 104
7.7.18 svl_set_system_message_level 104
7.7.19 svl_make_format 105
7.7.20 svl_sprint_left 105
7.7.21 svl_sprint_right 105
7.7.22 svl_sprint_center 105
7.7.23 svl_sprint_string 105
7.7.24 svl_breakup 105
7.7.25 svl_print_header 106
7.7.26 svl_sprint_header 106
7.7.27 svl_print_footer 106
7.7.28 svl_sprint_footer 106
7.7.29 svl_print_data 106

Copyright 2024 © Artgraphics. All rights reserved.

7.7.30 svl_sprint_data ... 106
7.8 プロセス生成 ... 106
7.9 ユーティリティ ... 107
7.9.1 svl_cmd ... 107
7.9.2 svl_cmd_output ... 107
7.9.3 svl_date ... 107
7.9.4 svl_cwd ... 107
7.9.5 svl_getenv ... 107
7.10 汎用マクロ ... 108
7.10.1 `svl_info_m ... 109
7.10.2 `svl_warning_m ... 109
7.10.3 `svl_error_m ... 109
7.10.4 `svl_hex_digits_m ... 109
7.10.5 `svl_short_bin_digits_m ... 109
7.10.6 `svl_short_hex_digits_m ... 109
7.10.7 `svl_full_bin_digits_m ... 110
7.10.8 `svl_full_hex_digits_m ... 110
7.10.9 `svl_init_print_row_m ... 110
7.10.10 `svl_add_print_column_m ... 110
7.10.11 `svl_print_row_m ... 110
7.10.12 `svl_sprint_row_m ... 110
7.10.13 `svl_foreach_enum_m ... 110
7.11 プリント書式マクロ ... 111
7.11.1 `svl_name_m ... 112
7.11.2 `svl_sformatfb_m ... 112
7.11.3 `svl_sformatfo_m ... 112
7.11.4 `svl_sformatfh_m ... 112
7.11.5 `svl_sformatfd_m ... 112
7.11.6 `svl_sformatfs_m ... 112
7.11.7 `svl_breakupb_m ... 113
7.11.8 `svl_breakupo_m ... 113
7.11.9 `svl_breakuph_m ... 113
7.11.10 `svl_breakupd_m ... 114
7.11.11 `svl_pbreakupb_m ... 114
7.11.12 `svl_pbreakupo_m ... 114
7.11.13 `svl_pbreakuph_m ... 114
7.11.14 `svl_sprintfb_m ... 114
7.11.15 `svl_sprintfo_m ... 114
7.11.16 `svl_sprintfh_m ... 115
7.11.17 `svl_nsprintfb_m ... 115
7.11.18 `svl_nsprintfo_m ... 115
7.11.19 `svl_nsprintfh_m ... 116
7.11.20 `svl_nsprintfd_m ... 116
7.11.21 `svl_nsformatfb_m ... 116
7.11.22 `svl_nsformatfo_m ... 116
7.11.23 `svl_nsformatfh_m ... 117
7.11.24 `svl_nsformatfd_m ... 117
7.11.25 `svl_nsformatfs_m ... 117

8 使用例 ... 118

8.1 SVL_PRINT_HEADER / SVL_PRINT_DATA / SVL_PRINT_FOOTER ... 118
8.2 ヒープ（SVL_HEAP_T / SVL_MAX_HEAP_T / SVL_MIN_HEAP） ... 120
8.3 パラレルアレイ（SVL_PARALLEL_ARRAY_T） ... 124
8.4 プロセス生成（SVL_PROCESS_T） ... 126
8.5 文字列処理 ... 130
8.6 ランダム文字列 ... 136
8.7 正則表現 ... 138

Copyright 2024 © Artgraphics. All rights reserved.

8.8　ファイル入出力 142
8.9　クロック生成 143
8.10　メッセージのプリント 144
8.11　プリント書式 144
8.12　ユーティリティ 144

9　検証環境構築例 146

9.1　検証環境（モニターによる検証） 146
9.1.1　up_down_counter 147
9.1.2　pkg_definitions 148
9.1.3　simple_if 148
9.1.4　simple_item_t 148
9.1.5　act_down_t 149
9.1.6　act_load_t 149
9.1.7　act_loaddown_t 149
9.1.8　act_reset_t 150
9.1.9　act_up_t 150
9.1.10　act_upup_t 150
9.1.11　act_upuploaddown_t 151
9.1.12　play_t 152
9.1.13　play_reset_down_t 152
9.1.14　play_reset_up_t 152
9.1.15　driver_t 153
9.1.16　generator_t 154
9.1.17　collector_t 154
9.1.18　monitor_t 155
9.1.19　agent_t 156
9.1.20　env_t 157
9.1.21　test_base_t 157
9.1.22　test1_t 158
9.1.23　test2_t 158
9.1.24　pkg 158
9.1.25　top 159
9.1.26　テストの実行 159
9.2　検証環境（スコアボードによる検証） 160
9.2.1　パッケージ 161
9.2.2　モニター 162
9.2.3　エンバイロンメント 162
9.2.4　スコアボード 163
9.2.5 165
9.2.6　トップモジュール 165
9.2.7　テストの実行 165

10　補足 167

10.1　ソースコードの構成 167
10.2　SVL 起動メッセージの抑止 167

11　参考文献 168

Copyright 2024 © Artgraphics. All rights reserved.

1 概要

SVL は、検証作業で必要となる機能を汎用的に開発した SystemVerilog パッケージです。汎用的であるため、実装に依存する内容はパラメータとして指定される仕組みになっています。したがって、パラメータに割り当てられた値が変化すれば、実装された内容も自動的に変化するように構築されています。例えば、ビット幅が変化しても検証結果のレポート処理を変更する必要はありません。本章では、SVL の概要を解説します。

1.1 SVL の意義と目的

検証環境を構築するためのパッケージとしては UVM が良く知られています。UVM には、検証コンポーネントやトランザクションの定義を容易にする機能が含まれていますが、検証コード記述のための生産性向上技術は含まれていません。SVL は、UVM に不足している生産性向上技術を提供します。勿論、SVL は UVM とは直接的な関連を持たないため、SVL を単独に生産性向上のためのパッケージとして使用する事ができます。寧ろ、これが SVL の目指す本来の目的です。

SVL の意義は、記述されたコードの柔軟性と汎用性を促進する機能を提供する事にあります。次に簡単な例を用いて SVL には柔軟性のある記述法を導く働きがある事を解説します。下記のような検証結果をプリントする処理を考察します。

```
=========================
time  reset d      q
-------------------------
@ 10: 0     'h9d66 'h9d66
@ 30: 0     'h9b47 'h9b47
@ 50: 0     'h7f20 'h7f20
@ 70: 0     'h51a6 'h51a6
@ 90: 0     'ha6e2 'ha6e2
@100: 1     'h4017 'h0000
@110: 0     'h4017 'h4017
@130: 0     'hcd05 'hcd05
@150: 0     'h2534 'h2534
@170: 0     'ha061 'ha061
@190: 0     'h75d9 'h75d9
=========================
```

ここで、`reset` は 1 ビット、`d` と `q` は 16 ビットであると仮定します。ヘッダーをプリントするためには、以下のようにするのが一般的です。

```
$display("%s",{25{"="}});
$display("%s  %s %-6s %-6s","time","reset","d","q");
$display("%s",{25{"-"}})
```

データをプリントするためには、一般的には、以下のように記述します。

```
$display("@%3t: %b     'h%h 'h%h",$time,reset,d,q);
```

また、フッターをプリントするためには、以下のようにします。

```
$display("%s",{25{"="}});
```

確かに、上記の記述は正しいのですが幾つかの問題が出てきます。例えば、以下のような問題は簡単に思いつきます。

Copyright 2024 © Artgraphics. All rights reserved.

- d と q のビット数に変更があると、多くの変更を発生する。例えば、全ての$display 文の変更が必要になります。
- reset 信号の表示位置を変更すると、多くの変更が生じる。
- time の表示桁数を変更すると、多くの変更が生じる。
- クロック信号を reset 信号の前に追加表示しようとすると、多くの変更が生じる。

このような問題を未然に防ぐための工夫が SVL には取り込まれています。具体的に言えば、SVL によるレポート処理ではプリントレイアウトをデザインするためのモデルを定義し、モデルを使用してプリント処理を記述します。こうすると、モデルに定義されている情報を変更すればプリント処理も自動的に変化します。例えば、上記の例の場合には以下のようなモデルを定義します。

```
svl_print_header_s header[] = '{
    '{"time",5,SVL_HEADER_DEFAULT},
    '{"reset",5,SVL_HEADER_DEFAULT},
    '{"d",`svl_full_hex_digits_m(WIDTH),SVL_HEADER_DEFAULT},
    '{"q",`svl_full_hex_digits_m(WIDTH),SVL_HEADER_DEFAULT}
    };
```

ここで、WIDTH は 16 に設定されています。そして、定義されたモデルを使用してプリント処理を行います。SVL のプリント機能は、モデル情報を基にプリントするので、カラム位置を正しく算出できるため、前記のようなプリント結果を得られます。そして、WIDTH を 32 ビットに変更しても、モデルを基にしているプリント処理には影響がありません。したがって、WIDTH==32 の場合には以下のようにプリントされます。

```
===================================
time  reset d           q
-----------------------------------
@ 10: 0     'h8f17_9d66 'h8f17_9d66
@ 30: 0     'h575e_9b47 'h575e_9b47
@ 50: 0     'h0424_7f20 'h0424_7f20
@ 70: 0     'hf816_51a6 'hf816_51a6
@ 90: 0     'h814a_a6e2 'h814a_a6e2
@100: 1     'h4c12_4017 'h0000_0000
@110: 0     'h4c12_4017 'h4c12_4017
@130: 0     'h422d_cd05 'h422d_cd05
@150: 0     'hccef_2534 'hccef_2534
@170: 0     'h6873_a061 'h6873_a061
@190: 0     'h7f26_75d9 'h7f26_75d9
===================================
```

time の表示桁数を 5 から 7 に変更するには、モデル内で 5 を 7 に変更するだけで済みます。他の項目に関する変更も同様に対処できます。このように、SVL を使用する事により多くの問題を自然に解消できる事がわかります。この例が示すように、SVL には生産性を向上させるための意義があります。

SVL には、この他にも重要な機能が装備されています。それは、検証環境構築機能です。メソドロジークラスを使用する事により、検証環境構築の作業が省力化されます。例えば、ドライバーのベースクラスには TLM ポートが定義されているので、ユーザのドライバーでは特別な準備をしなくても TLM ポートを使用できます。例えば、一般的なユーザ定義のドライバーの記述は以下のようになります。

```
class driver_t extends svl_driver_t#(simple_item_t);
vif_config::vif_type    vif;
`svl_auto_component_m(driver_t)
`svl_component_new_m
```


Copyright 2024 © Artgraphics. All rights reserved.

```
`svl_extern_connect_step_m
`svl_extern_run_step_m
extern task drive_dut(TR item);
endclass
```

run_step()では、トランザクションをジェネレータから取得しますが、TLM ポートの使用準備ができているので get()を呼ぶだけでトランザクションを取得できます。

```
task driver_t::run_step(svl_run_param_t param);
TR item;
    forever begin
        m_get_port.get(item);
        drive_dut(item);
        @(negedge vif.clk);
    end
endtask
```

ベースクラスで TLM ポートが準備されるので、ユーザ定義のドライバーでは直ぐに使用できます

一方、ジェネレータの定義は、以下の一行で済みます。

```
`svl_generator_m(generator_t,simple_item_t)
```

SVL では、ベースクラスのジェネレータ（svl_generator_t）がトランザクション取得処理の全てを司るので、ユーザはコンストラクタを定義するだけです。したがい、一行のマクロで事足ります。他の検証コンポーネントも同様に定義できます。

UVM と異なり、コレクターやモニターのベースクラスには TLM ポートが標準的に装備されているので検証環境構築の生産性が向上します。次に、検証環境を実行するために必要なテストケースの準備を解説します。

トランザクションは svl_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、SVL ではトランザクションの生成および準備を svl_play_t と svl_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 svl_play_t のオブジェクトで表現し、ツリー内の他のノードを svl_act_t のオブジェクトで表現します（図 1-1）。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。svl_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 svl_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。

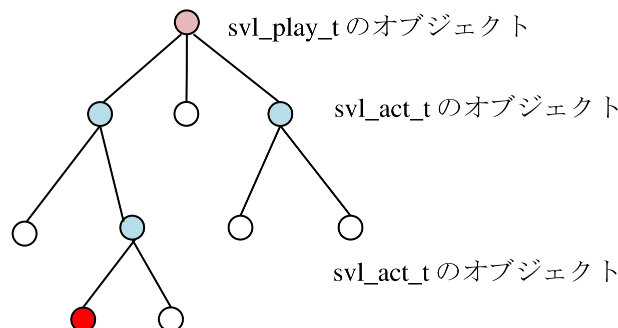


図 1-1　トランザクション生成ツリー

Copyright 2024 © Artgraphics. All rights reserved.

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 1-1 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。UVM と異なり、よりプログラミングに近い形でテストデータを生成する事ができます。

このように、SVL は汎用的な機能と定型的処理機能を備えて、SystemVerilog による検証環境構築の生産性向上と品質向上を目指しています。

1.2 SVL の機能概要

検証作業には多くの機能が必要になります。SVL には、検証環境で必要な以下に示すような機能が備えられています。

- 検証環境を構築するためのメソドロジークラス（ドライバー、ジェネレータ、コレクター、モニター、エージェント、エンバイロンメント、スコアボード、テスト等）
- トランザクションのベースクラス
- TLM を支援する機能
- テストケースを実行制御するシナリオ機能
- トランザクション生成機能
- 検証環境をダイナミックに変更する機能
- virtual インターフェース操作機能
- クロック生成機能
- タイムアウト機能
- テストケースを実行時に指定する機能（この機能により、一度のコンパイルで複数のテストケースを実行できます）
- 記述作業の効率化を促進するマクロ機能
- プロセスのスケジューリングや待ち行列の管理に必要なプライオリティキューを実装するためのデータ構造
- 文字列をパターンで検索する事を可能にする正則表現機能
- 文字列内での検索機能
- 文字列の比較と変換機能
- ファイル入出力機能
- 検証結果をプリントするための汎用的なプリント書式生成機能
- メッセージレベルによるメッセージプリント機能
- プリント出力のターミナルとファイルへの切り替え機能
- プロセス生成機能
- 現在時刻や作業ディレクトリの取得機能
- 環境変数の内容取得

これらの機能を使用する事により、定型的な作業から解放されて本質的な作業への専念へと移行し、より優れた性能と機能を持つ検証環境を構築できるようになります。

1.3 SVL の使用手順

SVL の機能は、`svl_pkg.sv` ファイルに定義されているので、そのファイルをインクルードする必要があります。まとめると以下のような手順となります。

- `svl_pkg.sv` ファイルをインクルードする。
- ユーザが定義したファイルをインクルードする。
- SVL を使用するスコープ内で `svl_pkg` パッケージをインポートする。ユーザが定義したパッケージも同時にインポートする。

Copyright 2024 © Artgraphics. All rights reserved.

例えば、以下のような使用手順になります。通常、インターフェースの定義が必要になるので、インターフェースの定義を含んだファイルも以下のようにインクルードします。

```
`include "svl_pkg.sv"
`include "simple_if.sv"
`include "pkg.sv"

module test;
import svl_pkg::*;
import pkg::*;
logic     clk;

simple_if SIF(.clk(clk));
...
endmodule
```

SVLを使用するために必須のファイルをインクルードする

インターフェースの定義をインクルードする

ユーザが定義したデータタイプやクラスをインクルードする

SVL パッケージをインポートする

ユーザが定義したパッケージをインポートする

このように準備すると、モジュール test 内で SVL が提供する機能を自由に使用できます。

1.4 SVL の記法

既に気づいたと思いますが、SVL を構成する要素には一定の命名法が適用されています。命名法のルールを表 1-1 にまとめておきます。

表 1-1　SVL の命名法

命名対象	意味
svl_pkg	SVL パッケージを意味します。
svl_*_m	SVL マクロを意味します。
svl_*_t	SVL クラスのタイプを意味します。
svl_*_s	SVL で定義されたストラクチャを意味します。
svl_*_e	SVL で定義された enum を意味します。
m_*	SVL のグローバル変数とクラス内の変数には、接頭辞 m_が使用されています。

例えば、SVLにはメソッドの終了状態を示すコードが enumとして定義されていますが、そのデータタイプ名には以下のように _e が付けられています。

```
typedef enum { SVL_RETURN_SUCCESS, SVL_RETURN_WARNING,
        SVL_RETURN_ERROR, SVL_RETURN_FATAL,
        SVL_RETURN_SYSTEM } svl_return_e;
```

1.5 本書の記法

本書で示す説明図では、読み易さのためにクラス名の接頭辞（svl_）と接尾辞（_t）を省略する事があります。例えば、port と書くと svl_port_t を意味します。

Copyright 2024 © Artgraphics. All rights reserved.

2 SVL の構成

SVL 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、svl_object_t と svl_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 svl_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 svl_component_t のサブクラスです。svl_object_t と svl_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

SVL 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。SVL のベースクラスは、svl_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。svl_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、SVL 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。

図 2-1 は、SVL 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。SVL 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。

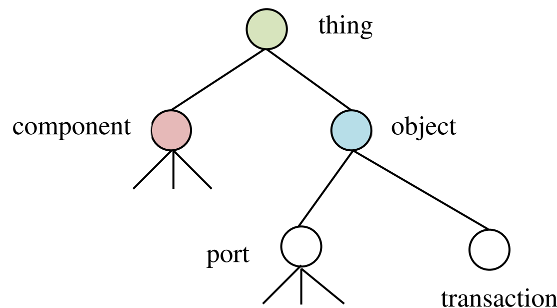


図 2-1 SVL クラス階層（抜粋）

表 2-1 主な SVL クラスの概要

SVL クラス	機能
svl_thing_t	他の全ての SVL 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。
svl_component_t	全ての検証コンポーネントのベースクラスです。
svl_object_t	全てのオブジェクトのベースクラスです。
svl_port_t	TLM ポートのベースクラスです。
svl_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 svl_void_t

実は、SVL にはもう一つ別のクラス svl_void_t が定義されていて svl_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 svl_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 svl_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、svl_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 svl_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、svl_void_t ク

Copyright 2024 © Artgraphics. All rights reserved.

ラスは SVL の完全性を保証する目的に存在し、svl_thing_t は SVL の実質的なベースクラスです。因みに、svl_void_t は以下のように定義されています。

```
virtual class svl_void_t;
endclass
```

2.3 svl_thing_t

このクラスは、全ての SVL 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はありません。svl_thing_t の全容は、以下のようになります。

```
class svl_thing_t extends svl_void_t;
//
// members
//
static int   m_count = 0;  // # of instances created
string       m_name;       // name given to this instance
int          m_id;         // unique number in entire environment

//
// methods
//

extern function new(string name);
extern virtual function string get_name();
extern virtual function int get_id();
virtual function bit is_object(); return 0; endfunction
virtual function bit is_component(); return 0; endfunction
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　svl_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	svl_thing_t のインスタンスに付けられた名称を示します。
m_id	svl_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。

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。

表 2-3　svl_thing_t に定義されているメソッドの機能概要

メソッド	機能概要
new(name)	コンストラクタです。インスタンスに割り当てる名称を name に指定します。
get_name()	このインスタンスに与えられている名称（name）を戻します。
get_id()	このインスタンスに与えられているユニーク ID を戻します。
is_object()	このインスタンスがオブジェクトであれば 1 を戻します。サブクラスはこの機能を定義しなければなりません。
is_component()	このインスタンスがコンポーネントであれば 1 を戻します。サブクラスはこの機能を定義しなければなりません。

Copyright 2024 © Artgraphics. All rights reserved.

次に、これらのメソッドを以下に解説します。

2.3.1 new

```
function new(string name);
```

このメソッドはコンストラクタです。svl_thing_t のインスタンス名を name に指定できます。指定された名称は m_name に設定されます。また、このインスタンスに対してユニークな ID も設定されます。

2.3.2 get_name

```
virtual function string get_name();
```

インスタンスに割り当てられている名称を戻します。

2.3.3 get_id

```
virtual function int get_id();
```

このインスタンスに割り当てられたユニークな ID を戻します。

2.3.4 is_object

```
virtual function bit is_object();
```

svl_object_t のサブクラスであれば 1、そうでなければ 0 を戻します。サブクラスはこのファンクションを再定義しなければなりません。

2.3.5 is_component

```
virtual function bit is_component();
```

svl_component_t のサブクラスであれば 1、そうでなければ 0 を戻します。サブクラスはこのファンクションを再定義しなければなりません。

2.4 svl_object_t

このクラスは、オブジェクトに関するベースクラスの役割を果たします。svl_object_t の全容は、以下のようになります。

```
class svl_object_t extends svl_thing_t;
//
// members
//

//
// methods
//

extern function new(string name="object");
extern function bit is_object();
endclass
```

このクラスに定義されているメソッドの機能概要を表 2-4 に示します。

表 2-4　svl_object_t に定義されているメソッドの機能概要

Copyright 2024 © Artgraphics. All rights reserved.

メソッド	機能概要
new(name)	コンストラクタです。インスタンスに割り当てる名称をname に指定します。
is_object()	このインスタンスはオブジェクトなので、1 を戻します。

次に、これらのメソッドを以下に紹介します。

2.4.1 new

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

このメソッドはコンストラクタです。オブジェクトのインスタンス名を name に指定できます。

2.4.2 is_object

```
function bit is_object();
```

このクラスはオブジェクトなので 1 を戻すように定義しておきます。サブクラスでは、このメソッドを再定義する必要はなくなります。

2.5 svl_component_t

このクラスは、コンポーネントに関するベースクラスの役割を果たします。svl_component_t の全容は、以下のようになります。オブジェクトと異なり、コンポーネントには階層を管理するための情報、およびシミュレーション制御をするための情報が必要なので定義は少し複雑になります。

```
class svl_component_t extends svl_thing_t;
//
// members...
//
svl_component_t     m_parent;
svl_component_t     m_sorted_children[string];
svl_component_t     m_ordered_children[$];
protected svl_object_t m_object_list[$];   // list of ports, etc

//
// methods...
//

extern function new(string name,svl_component_t parent);
extern function bit is_component();
extern function int is_top();
extern virtual function int add_child(svl_component_t child);
extern function void add_object(svl_object_t obj);
extern function void get_objects(output svl_object_t objects[$]);
extern function void
   get_ordered_children(output svl_component_t children[$]);
extern function svl_component_t get_parent();
extern virtual function svl_component_t get_child(string name);
extern virtual function int get_first_child(ref string name);
extern virtual function int get_last_child(ref string name);
extern virtual function int get_next_child(ref string name);
extern virtual function int get_prev_child(ref string name);
extern virtual function string get_full_name();
// step methods
extern virtual function void build_step(svl_run_param_t param);
extern virtual function void connect_step(svl_run_param_t param);
```


Copyright 2024 © Artgraphics. All rights reserved.

```
extern virtual function void setup_step(svl_run_param_t param);
extern virtual task run_step(svl_run_param_t param);
extern virtual function void collect_step(svl_run_param_t param);
extern virtual function void check_step(svl_run_param_t param);
extern virtual function void conclude_step(svl_run_param_t param);
endclass
```

このクラスには、他の全てのコンポーネントクラスに共通なプロパーティ、および機能が定義されています。共通プロパーティを表 2-5 に紹介します。

表 2-5 svl_component_t のプロパーティ

プロパーティ	説明
m_parent	このインスタンスの親コンポーネントを示します。m_parent==null であれば、このインスタンスはコンポーネント階層のトップに位置します。
m_sorted_children[string]	このインスタンスが保有するサブコンポーネントを記録する associative アレイです。サブコンポーネントのインスタンス名称でソートされています。
m_ordered_children[$]	サブコンポーネントが登録された順序で記録されています。
m_object_list[$]	このインスタンスが保有するオブジェクトを管理するためのキューです。

このクラスに定義されているメソッドの機能概要を表 2-6 に示します。

表 2-6 svl_component_t に定義されているメソッドの機能概要

メソッド	機能概要
new(name,parent)	コンストラクタです。インスタンスに割り当てる名称を name に指定します。
is_component()	このインスタンスはコンポーネントなので、1 を戻します。
is_top();	このインスタンスが階層上のトップであれば 1、そうでなければ 0 を戻します。
add_child(child)	指定されたコンポーネント（child）を下位階層にサブコンポーネントとして追加します。全てのサブコンポーネントは、一定の順序に配置されます。
add_object(obj)	このコンポーネントにオブジェクトを追加します。このメソッドを使用してオブジェクトを登録すると、コンポーネントに配置されているオブジェクトを取り出す事ができます。
get_objects(objects)	このコンポーネントに配置されている全てのオブジェクトを取り出します。
get_ordered_children(children)	このコンポーネント内に登録されているコンポーネントのインスタンスを登録順にキューに取り出します。
get_parent()	このコンポーネントの階層上の親コンポーネントを戻します。
get_child(name)	指定した名称を持つサブコンポーネントを戻します。
get_first_child(name)	名称順にソートされているサブコンポーネントのリストから最初のサブコンポーネントの名称を取り出して戻します。
get_last_child(name)	名称順にソートされているサブコンポーネントのリストから最後のサブコンポーネントの名称を取り出して戻し

Copyright 2024 © Artgraphics. All rights reserved.

	ます。
get_next_child(name)	指定したサブコンポーネント名称の次の名称を取り出して戻します。
get_prev_child(name)	指定したサブコンポーネント名称の直前の名称を取り出して戻します。
get_full_name()	このコンポーネントの階層名を戻します。
build_step(param)	シミュレーション用のステップです。
connect_step(param)	
setup_step(param)	
run_step(param)	
collect_step(param)	
check_step(param)	
conclude_step(param)	

検証環境を構成する検証コンポーネントは、svl_component_t クラスのサブクラスとして定義されます。検証コンポーネントには、それぞれの特徴があるとともに共通の属性・機能も保有します。SVL にはそれらの属性と機能を持つ汎用的な検証コンポーネントがメソドロジークラスとして準備されています。メソドロジークラスについては、第 4 章で解説します。このクラスに定義されているメソッドを以下に紹介します。

2.5.1 new

```
function new(string name,svl_component_t parent);
```

このメソッドはコンストラクタです。コンポーネントは階層を構成するので、親コンポーネントのインスタンスを parent に指定しなければなりません。親コンポーネントが存在しない場合には、parent として null を指定します。そして、インスタンス名を name に指定できます。

参考 2-1

同じ階層レベルには同じインスタンス名が存在しないように命名しなければなりません。
□

2.5.2 is_component

```
function bit is_component();
```

このクラスは、コンポーネントなので 1 を戻すように定義しておきます。サブクラスでは、このメソッドを再定義する必要はありません。

2.5.3 is_top

```
function int is_top();
```

階層のトップであれば、1 を戻します。トップでなければ、0 を戻します。

2.5.4 add_child

```
virtual function int add_child(svl_component_t child);
```

コンポーネントのインスタンスを下位階層のインスタンスとして追加します。このようなイ

Copyright 2024 © Artgraphics. All rights reserved.

ンスタンスをサブコンポーネントと呼びます。

2.5.5 add_object

```
function void add_object(svl_object_t obj);
```

このインスタンス内にオブジェクトを追加します。

2.5.6 get_objects

```
function void get_objects(output svl_object_t objects[$]);
```

このクラスのインスタンスが保有するオブジェクトをキューに戻します。

2.5.7 get_ordered_children

```
function void
   get_ordered_children(output svl_component_t children[$]);
```

このコンポーネント内に登録されているコンポーネントのインスタンスを登録順にキューに取り出します。

2.5.8 get_parent

```
virtual function svl_component_t get_parent();
```

コンポーネント階層における親コンポーネントを戻します。このコンポーネントが階層上のトップに配置されている場合には、null が戻されます。

2.5.9 get_child

```
virtual function svl_component_t get_child(string name);
```

指定した名称を持つサブコンポーネントを戻します。存在しなければ null が戻されます。

2.5.10 get_first_child

```
virtual function int get_first_child(ref string name);
```

名称順にソートされているサブコンポーネントのリストから最初のサブコンポーネントの名称を取り出します。このメソッドは、全てのサブコンポーネントを順に取り出すための初期化処理として役立ちます。サブコンポーネントが存在すれば 1、存在しなければ 0 を戻します。

2.5.11 get_last_child

```
virtual function int get_last_child(ref string name);
```

名称順にソートされているサブコンポーネントのリストから最後のサブコンポーネントの名称を取り出します。サブコンポーネントが存在すれば 1、存在しなければ 0 を戻します。

2.5.12 get_next_child

```
virtual function int get_next_child(ref string name);
```


Copyright 2024 © Artgraphics. All rights reserved.

指定したサブコンポーネント名称の次のサブコンポーネント名称を取り出します。次のサブコンポーネントが存在すれば 1、存在しなければ 0 を戻します。

2.5.13 get_prev_child

```
virtual function int get_prev_child(ref string name);
```

指定したサブコンポーネント名称の直前のサブコンポーネント名称を取り出します。直前のサブコンポーネントが存在すれば 1、存在しなければ 0 を戻します。

2.5.14 get_full_name

```
virtual function string get_full_name();
```

このコンポーネントが属する階層の名称を戻します。名称はトップレベルのインスタンス名称から始まり、このインスタンスの名称で終了します。階層名の区切りにはドット（.）が使用されます。

2.5.15 build_step

```
virtual function void build_step(svl_run_param_t param);
```

シミュレーションを開始すると、最初に呼ばれる s 轍鮒です。このメソッドには、サブコンポーネントを作成する処理を記述します。

2.5.16 connect_step

```
virtual function void connect_step(svl_run_param_t param);
```

`build_step()`の次に呼び出されるステップで、サブコンポーネント間の接続関係を完了するための処理を実装します。

2.5.17 setup_step

```
virtual function void setup_step(svl_run_param_t param);
```

シミュレーションステップの `run_step()`を実行する直前に呼ばれるフェーズで、実行に必要な初期化処理を記述できます。

2.5.18 run_step

```
virtual task run_step(svl_run_param_t param);
```

実際のシミュレーションをするステップです。このステップは長い時間実行するので、タスクとして定義されています。

2.5.19 collect_step

```
virtual function void collect_step(svl_run_param_t param);
```

全ての `run_step()`が終了すると、このステップが呼び出されます。

2.5.20 check_step

```
virtual function void check_step(svl_run_param_t param);
```


Copyright 2024 © Artgraphics. All rights reserved.

`collect_step()`が終了すると、このステップが呼ばれます。

2.5.21 conclude_step

```
virtual function void conclude_step(svl_run_param_t param);
```

シミュレーションの最後に呼ばれるステップです。

2.6 オブジェクトとコンポーネントの生成

既に紹介したように全てのオブジェクトは直接または間接的に`svl_object_t`クラスから定義されます。同様に全ての検証コンポーネントは直接または間接的に`svl_component_t`クラスから定義されます。定義されたクラスは、そのインスタンスを作られて使用されますが、ベースクラスとサブクラスの関連を活用して柔軟性のあるインスタンス作成をしなければなりません。SVL ではインスタンス作成用に以下の二つのマクロを備えています。

- `` `svl_create_object_m ``
- `` `svl_create_component_m ``

これらのマクロにより作成されたインスタンスは、実行時に互換性のあるクラスタイプのインスタンスで置き換える事ができます。

2.6.1 `svl_create_object_m

オブジェクトを作るためには、以下のマクロを使用します。クラス名を CLASS_TYPE に指定し、インスタンス名を NAME に指定します。

```
`svl_create_object_m(CLASS_TYPE,NAME)
```

例えば、`play_reset_down_t`と呼ばれるオブジェクトのクラスが定義されていると、そのインスタンスを生成するためには以下のようにして`` `svl_create_object_m ``マクロを使用します。

```
svl_scenario_t::allocate("*generator",
    `svl_create_object_m(play_reset_down_t,"play"));
```

このマクロを使用してオブジェクトを作る場合には、対象となるクラスでは`` `svl_auto_object_m ``マクロが使用されなければなりません。

```
class play_reset_down_t extends play_t;
`svl_auto_object_m(play_reset_down_t)
...
enclass
```

2.6.2 `svl_create_component_m

コンポーネントのインスタンスを作るためには、以下のマクロを使用します。クラス名を CLASS_TYPE に指定し、インスタンス名を NAME に指定します。インスタンスがコンポーネント階層の一部である場合には、階層上の親コンポーネントを PARENT に指定します。階層のルートである場合には、PARENT に`null`を指定します。

```
`svl_create_component_m(CLASS_TYPE,NAME,PARENT)
```

例えば、コンポーネントのインスタンスを作るためには、以下のように`` `svl_create_component_m ``マクロを使用します。

Copyright 2024 © Artgraphics. All rights reserved.

```
collector = `svl_create_component_m(collector_t,"collector",this);
```

このマクロを使用してインスタンスを作る場合には、対象となるクラスでは`svl_auto_component_m マクロが使用されなければなりません。

```
class collector_t extends svl_collector_t#(simple_item_t);
vif_config::vif_type    vif;
simple_item_t    item;
`svl_auto_component_m(collector_t)
...
enclass
```

2.7 コンフィギュレーション設定変更

既に紹介したように、SVL では検証環境をダイナミックに変更する事ができます。設定変更には、次の二種類があります。

- クラスタイプの変更
- クラスプロパーティの変更

以下では、これらの便利な機能の使用法を解説します。

2.7.1 クラスタイプの変更

クラスを互換性のある他のクラス（つまりサブクラス）で置き換えるためには、以下の仕様を持つ`svl_change_type_m マクロを使用します。

```
`svl_change_type_m(curr_type,new_type)
```

ここで、curr_type は現在使用しているクラスタイプで、new_type は置き換えるべき新しいクラスのタイプです。なお、クラスのインスタンスは `svl_create_component_m マクロで作られていなければなりません。

svl_run_test()が実行を開始する前に、トップモジュール内でこのマクロを使用すればクラスタイプが置き換わります。厳密に言えば、クラスのインスタンスを作るまでに、このマクロが使用されれば十分です。

例 2-1　クラスタイプの変更例

クラス subvcomp_t は vcomp_t のサブクラスで、以下のように定義されているとします。

```
class vcomp_t extends svl_component_t;
...
endclass

class subvcomp_t extends vcomp_t;
...
endclass
```

以下の例では、クラス vcomp_t のインスタンスを subvcomp_t のインスタンスに置き換える指示をしています。

```
module top;
import svl_pkg::*;
import pkg::*;
test_t test;
```


Copyright 2024 © Artgraphics. All rights reserved.

```
initial begin
    `svl_change_type_m(vcomp_t,subvcomp_t)
    ...
    svl_run_test();
end

endmodule
```

vcomp_t のインスタンスを subvcomp_t のインスタンスで置き換える

検証環境内で `svl_create_component_m マクロで vcomp_t のインスタンスを作っていれば、vcomp_t は subvcomp_t に置き換わります。例えば、以下のようにして vcomp_t のインスタンスを vcomp に設定していても、vcomp には subvcomp_t のインスタンスが割り当てられます。

```
vcomp_t       vcomp;
vcomp = `svl_create_component_m(vcomp_t,"vcomp",this);
```

■

2.7.2 クラスプロパーティの変更

クラス内に定義されているプロパーティの値をクラスの修正をせずに置き換えるためには、以下の仕様を持つ svl_roster_t#(T)::set()を使用します。ここで、T はプロパーティのタイプを示します。

```
function void svl_roster_t#(T)::set(
    input svl_component_t scope,string path,string field_name,
    input T value);
```

scope は検証コンポーネントを意味しますが、現在では null を指定して下さい。path はクラスのインスタンスを示すパターンです。例えば、*.driver のように path を指定できます。設定変更の対象となるプロパーティ名称を field_name で指定し、その値を value で指定します。

設定された値を取り出すためには以下の仕様を持つ svl_roster_t#(T)::get()を使用して準備しておかなければなりません。使用するパラメータの意味は set()メソッドと同じですが、value に値が取り出されます。このメソッドは、値が正しく取り出されれば 1、取り出されなければ0を戻します。svl_roster_t#(T)::set()で値が設定されていない時にも0が戻されます。必須なパラメータ値を実行時に取得する場合に有効です。

```
function bit svl_roster_t#(T)::get(
    input svl_component_t scope,string path,string field_name,
    output T value);
```

必須なパラメータではなくオプショナルなパラメータの場合には、以下の仕様を持つ svl_roster_t#(T)::apply()メソッドの方が適切です。

```
function void svl_roster_t#(T)::apply(
    input svl_component_t scope,string path,string field_name,
    output T value);
```

使用方法は get()と同じですが、戻り値を持ちません。つまり、パラメータが設定されていれば、指定したフィールドに値を取り出しますが、設定されていなければ何もしません。

Copyright 2024 © Artgraphics. All rights reserved.

参考 2-2

#(T)で指定する T は、svl_roster_t#(T)::set()、svl_roster_t#(T)::get()、svl_roster_t#(T)::apply()で一致しなければなりません。
□

例 2-2　クラスプロパーティの設定変更例

検証コンポーネントとして vcomp_t を定義して、クラス内のプロパーティ（m_function と m_number）を変更する例を紹介します。先ず、vcomp_t の概要は以下のようになります。

```
class vcomp_t extends svl_component_t;
string        m_function;
logic [3:0]   m_number;
`svl_auto_component_m(vcomp_t)
`svl_component_new_m
`svl_extern_connect_step_m
`svl_extern_run_step_m
endclass
```

vcomp_t 内では、変更された値を取り出す仕組みを以下のように定義しておかなければなりません。

```
function void vcomp_t::connect_step(svl_run_param_t param);
   if( svl_roster_t#(string)::get(null,get_full_name(),
         "m_function",m_function) )
      $display("m_function = %s",m_function);
   if( svl_roster_t#(logic [3:0])::get(null,get_full_name(),
         "m_number",m_number) )
      $display("m_number = %0d",m_number);
endfunction
```

あるいは、以下のように apply()メソッドを使用できます。この場合には、m_function や m_number に値が設定されていれば自動的に取り出しますが、設定されていなければ現在の値を保持します。

```
function void vcomp_t::connect_step(svl_run_param_t param);
   svl_roster_t#(string)::apply(null,get_full_name(),
         "m_function",m_function);
   svl_roster_t#(logic [3:0])::apply(null,get_full_name(),
         "m_number",m_number);
endfunction
```

例えば、設定変更をトップモジュールで以下のように行えます。

```
module top;
import svl_pkg::*;
import pkg::*;
test_t test;

initial begin
   svl_roster_t#(string)::set(null,
         "test*vcomp","m_function","check");
   svl_roster_t#(logic [3:0])::set(null,"*vcomp","m_number",10);
   ...
   test = `svl_create_component_m(test_t,"test",null);
   svl_run_test();
```


Copyright 2024 © Artgraphics. All rights reserved.

```
end

endmodule
```

参考 2-3

検証環境のコンフィギュレーションを決定する重要なプロパーティに対しては、上記の例の様にして予め `svl_roster_t#(T)::get()`または `svl_roster_t#(T)::apply()`メソッドで値を取り出す命令を実装しておく必要があります。そうすれば、ソースコードを変更せずに、いつでも標準値の設定変更が可能になります。

必須なパラメータには`get()`を、オプショナルなパラメータに対しては`apply()`を適用すると柔軟性があります。

2.8 シミュレーション

本節では、ユーザが構築した検証環境がどのように実行されるかを解説します。検証環境を SVL のメソドロジークラスで実装する事により、シミュレーションは以下に紹介するように自動的に進行します。

2.8.1 実行制御

SVL のシミュレーションは、`svl_run_test()`メソッドにより起動されます。そして、シミュレーションが開始すると SVL が実行制御を握り、適切なタイミングで検証コンポーネントに定義されているシミュレーションステップを呼び出します（図 2-2）。

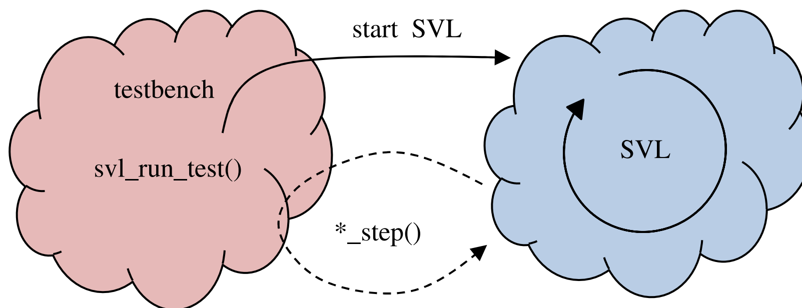


図 2-2　SVL を使用した環境での実行の流れ

シミュレーションステップには、表 2-7 に示すような種類があります。それぞれのシミュレーションステップは、virtual メソッドとして定義されており、ユーザは必要なステップだけを記述すれば良い事になっています。例えば、ドライバーには、`connect_step()`と`run_step()`の記述が必要ですが、エージェントはドライバー、ジェネレータ、コレクター、モニターのコンテイナーに過ぎないので、`run_step()`の記述は必要ありません。また、シミュレーションステップを定義する場合には、ベースクラスのシミュレーションステップを呼び出す必要がある場合もあります。

Copyright 2024 © Artgraphics. All rights reserved.

表 2-7　シミュレーションステップ

実行順序	フェーズ	説明
1	build_step	コンポーネントの階層を構築するステップです。階層のトップから順に呼ばれて行きます。通常、サブコンポーネントをこのステップで作成します。サブコンポーネントを作ると、そのサブコンポーネントのbuild_step()が次に呼ばれるので、階層構造がトップダウンで構築されます。トップダウンで呼び出されるので、このステップ内では最初にベースクラスのbuild_step()を呼び出す事が必要になります。
2	connect_step	コンポーネント間の接続を完成するステップです。例えば、TLM ポートの接続を完了します。あるいは、virtual インターフェースの設定等も行います。このステップはボトムアップの順序で呼ばれます。
3	setup_step	シミュレーションが開始する直前にこのステップが呼ばれます。初期化処理等を行えます。
4	run_step	シミュレーションを行うためのステップです。
5	collect_step	run_step()の実行が終了すると、このステップに制御が移ります。
6	check_step	collect_step()が終了すると、このステップが呼ばれます。
7	conclude_step	最後に呼ばれるステップです。

参考 2-4

run_step() はシミュレーションを行うので、いつ終了するか分かりません。したがって、run_step() はタスクとして定義されていますが、他のステップは全てファンクションとして定義されています。
□

参考 2-5

SVL は、UVM が備えている `raise_objection()`と `drop_objection()`に対応する機能を装備していません。検証コンポーネントに特別な記述がなくてもシミュレーションが実行します。
□

2.8.2　svl_run_test

シミュレーションを開始するためには、`svl_run_test()`メソッドを呼び出さなければなりません。仕様は以下のようになります。

```
task svl_run_test(string test_name="");
```

引数 `test_name` は、表 2-8 のような役割を果たします。

表 2-8　svl_run_test() の引数 test_name の役割

test_name	説明
指定あり	test_name に指定された名称を持つテストクラスを検索して実行します。該当するクラスが存在しなければ、エラーになります。
指定なし	この場合には、ユーザはコマンドラインからテストクラス名を指定しなければなりません。名称が正しくないとエラーになります。

Copyright 2024 © Artgraphics. All rights reserved.

例 2-3　クラス名を指定して実行する例

クラス名を指定して、`svl_run_test()`を呼ぶ例を以下に紹介します。`svl_run_test()`を呼ぶ際に `test_t` を指定しているので、`svl_run_test()`はこのクラスのインスタンスを作成して実行します。

```
module top;
import svl_pkg::*;
import pkg_definitions::*;
import pkg::*;
logic     clk;

simple_if SIF(.clk(clk));
...
initial begin
   svl_clock_gen(clk,20,100);
   vif_config::set(SIF);
   svl_run_test("test_t");
end
endmodule
```

■

例 2-4　名称を指定せずに テストを実行する例

クラス名を省略して、`svl_run_test()`を呼ぶ例を以下に紹介します。この場合には、コマンドラインで実行するコンポーネント名を指定しなければなりません。

```
module top;
import svl_pkg::*;
import pkg_definitions::*;
import pkg::*;
logic     clk;

simple_if SIF(.clk(clk));
...
initial begin
   svl_clock_gen(clk,20);
   svl_set_timeout(10,100);
   vif_config::set(SIF);
   svl_run_test();
end

endmodule
```

■

2.8.3　svl_set_timeout

シミュレーションが永遠に続く事を避けるためにタイムアウトを設定できます。タイムアウト設定は以下の仕様に従います。時間経過を `time_interval` 毎に計測して、`timeout` を経過するとシミュレーションが終了するようになります。

```
function void svl_set_timeout(longint interval, longint timeout);
```

例えば以下のように設定すると、#10 毎に経過時間を計測し、経過時間が#1000 を超えるとシミュレーションは終了します。

Copyright 2024 © Artgraphics. All rights reserved.

```
svl_set_timeout(10,1000);
```

2.8.4 svl_set_max_simulation_time

一度設定したタイムアウトを変更する事ができます。以下に示すメソッドを使用して、タイムアウトを変更します。max_time で指定した値が、新たなタイムアウトになります。

```
function void svl_set_max_simulation_time(longint max_time);
```

2.8.5 svl_get_test_name

svl_run_test()で実行されたテスト名を以下のメソッドで取り出す事ができます。シミュレーション結果をファイルに出力する時には便利な機能です。

```
function string svl_get_test_name();
```

参考 2-6

以下のメソッドは、svl_run_test()でシミュレーションを実行する時にだけ有効です。

- svl_set_timeout()
- svl_set_max_simulation_time ()
- svl_get_test_name ()

□

2.9 SVL マクロ

SVL には多くのマクロが使用されていますが、本節では検証環境を構築する際に必要になる以下のようなマクロを解説します。これらのマクロは、トランザクションや検証コンポーネントを定義する際に使用します。

- `svl_auto_object_m
- `svl_object_new_m
- `svl_object_new_default_m
- `svl_auto_component_m
- `svl_component_new_m
- `svl_extern_build_step_m
- `svl_extern_connect_step_m
- `svl_extern_setup_step_m
- `svl_extern_run_step_m
- `svl_extern_collect_step_m
- `svl_extern_check_step_m
- `svl_extern_conclude_step_m
- `svl_generator_m

2.9.1 `svl_auto_object_m

```
`svl_auto_object_m(T)
```

T には、クラス名を指定します。このマクロは、T で指定されたクラスのオブジェクトを `svl_create_object_m マクロで生成する場合に必要です。決して生成しない場合には、このマクロを使用する必要はありません。このマクロは、以下のように使用できます。

```
class simple_item_t extends svl_transaction_t;
```


Copyright 2024 © Artgraphics. All rights reserved.

```
logic [UP_DOWN_WIDTH-1:0]  d, q, qn;
logic                      reset, load, up_down;
`svl_auto_object_m(simple_item_t)
`svl_object_new_m
extern function up_down_op_e make_op();
endclass
```

2.9.2 `svl_object_new_m

```
`svl_object_new_m
```

このマクロは、以下のようにオブジェクトのコンストラクタを生成します。

```
function new(string name);
   super.new(name);
endfunction
```

このマクロを使用するとコンストラクタの指定が簡素化されます。以下に示すように、一行のマクロでコンストラクトの指定が完了します。

```
class act_down_t extends svl_act_t#(simple_item_t);
`svl_object_new_m
extern function void make_item();
endclass
```

2.9.3 `svl_object_new_default_m

```
`svl_object_new_default_m(NAME)
```

NAME には、オブジェクトに与える名称を指定します。この名称はオブジェクトの初期値になります。例えば、以下のように指定します。

```
`svl_object_new_default_m("item")
```

こうすると、以下のようなコンストラクタが生成されます。

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

2.9.4 `svl_auto_component_m

```
`svl_auto_component_m(T)
```

T には、クラス名を指定します。このマクロは、T で指定されたクラスのインスタンスを `` `svl_create_component_m `` マクロで生成する場合に必要です。決して生成しない場合には、このマクロを使用する必要はありません。このマクロは、以下のように使用できます。

```
class driver_t extends svl_driver_t#(simple_item_t);
vif_config::vif_type   vif;
`svl_auto_component_m(driver_t)
`svl_component_new_m
```


Copyright 2024 © Artgraphics. All rights reserved.

```
`svl_extern_connect_step_m
`svl_extern_run_step_m
extern task drive_dut(TR item);
endclass
```

ドライバーは、エージェントにより生成されるため、このマクロを指定する必要があります。

2.9.5 `svl_component_new_m

```
`svl_component_new_m
```

このマクロは、以下のようにコンポーネントのコンストラクタを生成します。

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

このマクロを使用するとコンストラクタの指定が簡素化されます。以下に示すように、一行のマクロでコンストラクトの指定が完了します。

```
class agent_t extends svl_agent_t;
driver_t  driver;
generator_t  generator;
collector_t  collector;
monitor_t monitor;
`svl_auto_component_m(agent_t)
`svl_component_new_m
`svl_extern_build_step_m
`svl_extern_connect_step_m
endclass
```

2.9.6 `svl_extern_build_step_m

```
`svl_extern_build_step_m
```

build_step()の extern 宣言を生成するマクロです。以下の宣言が生成されます。

```
extern function void build_step(svl_run_param_t param);
```

2.9.7 `svl_extern_connect_step_m

```
`svl_extern_connect_step_m
```

connect_step()の extern 宣言を生成するマクロです。以下の宣言が生成されます。

```
extern function void connect_step(svl_run_param_t param);
```

2.9.8 `svl_extern_setup_step_m

```
` svl_extern_setup_step_m
```

setup_step()の extern 宣言を生成するマクロです。以下の宣言が生成されます。

Copyright 2024 © Artgraphics. All rights reserved.

```
extern function void setup_step(svl_run_param_t param);
```

2.9.9 `svl_extern_run_step_m

```
` svl_extern_run_step_m
```

run_step()の extern 宣言を生成するマクロです。以下の宣言が生成されます。

```
extern function void run_step(svl_run_param_t param);
```

2.9.10 `svl_extern_collect_step_m

```
` svl_extern_collect_step_m
```

collect_step()の extern 宣言を生成するマクロです。以下の宣言が生成されます。

```
extern function void collect_step(svl_run_param_t param);
```

2.9.11 `svl_extern_check_step_m

```
` svl_extern_check_step_m
```

check_step()の extern 宣言を生成するマクロです。以下の宣言が生成されます。

```
extern function void check_step(svl_run_param_t param);
```

2.9.12 `svl_extern_conclude_step_m

```
` svl_extern_conclude_step_m
```

conclude_step()の extern 宣言を生成するマクロです。以下の宣言が生成されます。

```
extern function void conclude_step(svl_run_param_t param);
```

2.9.13 `svl_generator_m

```
` svl_generator_m(GENERATOR_TYPE,ITEM_TYPE)
```

このマクロは、ジェネレータを生成するために使用されます。ベースクラスの svl_generator_t がトランザクション処理を遂行するので、ユーザのジェネレータはコンストラクタを定義するだけで済みます。したがって、ジェネレータの指定は簡単なので、このマクロで生成できます。GENERATOR_TYPE にはジェネレータのクラス名、ITEM_TYPE にはトランザクションのクラス名を指定して下さい。

例えば、以下のように使用します。こうすると、generator_t クラスがジェネレータとして生成されます。トランザクションの型は simple_item_t クラスです。

```
`svl_generator_m(generator_t,simple_item_t)
```


Copyright 2024 © Artgraphics. All rights reserved.

9　検証環境構築例

本章では検証環境に SVL 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。

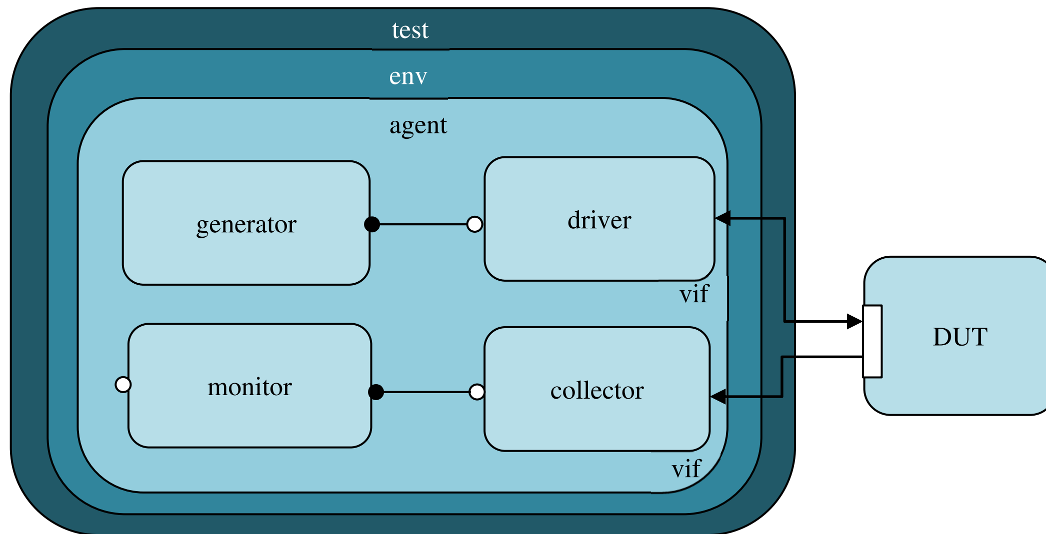


図 9-1　SVL による検証環境構築例

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	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。
act_load_t	
act_loaddown_t	
act_reset_t	
act_up_t	
act_upup_t	
act_upuploaddown_t	
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_step()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

Copyright 2024 © Artgraphics. All rights reserved.

	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
```


Copyright 2024 © Artgraphics. All rights reserved.

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

`svl_vif_config_m(virtual simple_if,vif_config)
```

virtual インターフェースを操作するためのクラスを生成する

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 svl_transaction_t;
logic [UP_DOWN_WIDTH-1:0]  d, q, qn;
logic           reset, load, up_down;
`svl_auto_object_m(simple_item_t)
`svl_object_new_m
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。

Copyright 2024 © Artgraphics. All rights reserved.

```
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 svl_act_t#(simple_item_t);
`svl_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 svl_act_t#(simple_item_t);
`svl_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 svl_act_t#(simple_item_t);
//
// members
//
act_load_t       act_load;
act_down_t       act_down;

`svl_object_new_m

//
```


Copyright 2024 © Artgraphics. All rights reserved.

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

endclass
```

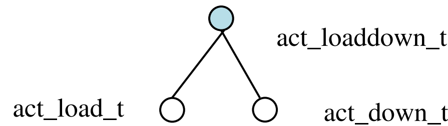


図 9-2　act_loaddown_t が表現する部分ツリー

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 svl_act_t#(simple_item_t);
`svl_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 svl_act_t#(simple_item_t);
`svl_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。

Copyright 2024 © Artgraphics. All rights reserved.

```
class act_upup_t extends svl_act_t#(simple_item_t);
//
// members
//
act_up_t      act_up;
`svl_object_new_m

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

endclass
```

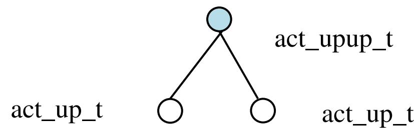


図 9-3　act_upup_t が表現する部分ツリー

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 svl_act_t#(simple_item_t);
//
// members
//
act_upup_t       act_upup;
act_loaddown_t      act_loaddown;

`svl_object_new_m

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

endclass
```

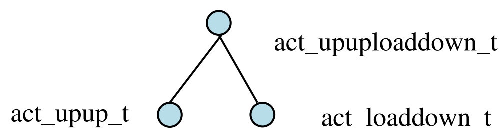


図 9-4　act_upuploaddown_t が表現する部分ツリー

Copyright 2024 © Artgraphics. All rights reserved.

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 svl_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;
`svl_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;
`svl_auto_object_m(play_reset_down_t)
//
// members
//

//
// methods
//

function new(string name); super.new("play_reset_down"); endfunction

task get_item();
    `svl_act_get_item_m(act_reset,this)
    `svl_act_get_group_m(act_upuploaddown,this)
endtask
endclass
```

階層的なアクト

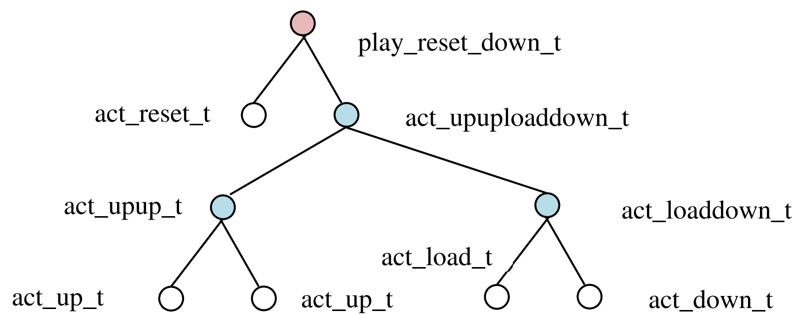


図 9-5　テストケース 1 のツリー

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;
`svl_auto_object_m(play_reset_up_t)
//
// members
//
```


Copyright 2024 © Artgraphics. All rights reserved.

```
//
// methods
//

`svl_object_new_m

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

endclass
```

階層を持たないアクト

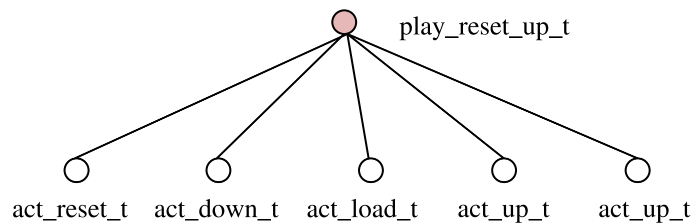


図 9-6　テストケース 2 のツリー

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 svl_driver_t#(simple_item_t);
vif_config::vif_type   vif;
`svl_auto_component_m(driver_t)
`svl_component_new_m
`svl_extern_connect_step_m
`svl_extern_run_step_m
extern task drive_dut(TR item);
endclass
```

virtual インターフェースの定義

`connect_step()`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。

```
function void driver_t::connect_step(svl_run_param_t param);
    super.connect_step(param);
    vif = vif_config::get();
endfunction
```

`run_step()`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。

```
task driver_t::run_step(svl_run_param_t param);
TR item;
    m_get_port.m_connected_port.set_option(
            SVL_PORT_REUSE_TRANSACTION);
    forever begin
        m_get_port.get(item);
        drive_dut(item);
        @(negedge vif.clk);
```


Copyright 2024 © Artgraphics. All rights reserved.

```
    end
endtask
```

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。

```
`svl_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 svl_collector_t#(simple_item_t);
vif_config::vif_type    vif;
simple_item_t    item;
`svl_auto_component_m(collector_t)
`svl_component_new_m
`svl_extern_connect_step_m
`svl_extern_run_step_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_step() 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。

```
function void collector_t::connect_step(svl_run_param_t param);
    vif = vif_config::get();
endfunction
```

run_step() 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。

```
task collector_t::run_step(svl_run_param_t param);
    item = `svl_create_object_m(simple_item_t,"item");
```


Copyright 2024 © Artgraphics. All rights reserved.

```
    fork
        collect_regular();
        collect_reset();
    join
endtask
```

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
```

リセット信号のイベントを collect_reset()メソッドで監視します。

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

DUT からのレスポンスをサンプリングした後、トランザクションに変換してモニターに送信します。

```
function void collector_t::send_item();
    item.reset = vif.reset;
    item.load = vif.load;
    item.up_down = vif.up_down;
    item.d = vif.d;
    item.q = vif.q;
    item.qn = vif.qn;
    m_send_port.write(item);
endfunction
```

トランザクションをモニターに送信する

9.1.18 monitor_t

モニターの全容は以下のようになります。このモニターは、受信したトランザクションをプリントするだけの簡単な処理になっています。通常は簡単なカバレッジ計算等を行います。以下の記述では、ビット幅の変更に対応するため自動カラム調節機能を備えたヘッダーデータ構造（svl_print_header_s）を使用しています。他のプロセスでもプリントしているとプリントされる情報が混ざり合って見難くなるので、プリントする内容を一時的にキューに保存し、処理終了後にプリントする事にします。

```
class monitor_t extends svl_monitor_t#(simple_item_t);
parameter HEX_DIGITS = `svl_hex_digits_m(UP_DOWN_WIDTH);
svl_print_header_s  header[] = '{
               '{"time",5,SVL_HEADER_DEFAULT},
               '{"reset",5,SVL_HEADER_DEFAULT},
               '{"load",4,SVL_HEADER_DEFAULT},
               '{"up_down",7,SVL_HEADER_DEFAULT},
               '{"d",HEX_DIGITS,SVL_HEADER_DEFAULT},
               '{"q",HEX_DIGITS,SVL_HEADER_DEFAULT},
               '{"qn",HEX_DIGITS,SVL_HEADER_DEFAULT},
               '{"op",5,SVL_HEADER_DEFAULT}
```


Copyright 2024 © Artgraphics. All rights reserved.

```
          };
up_down_op_e  operation;
string    lines[$];
`svl_auto_component_m(monitor_t)
`svl_component_new_m
`svl_extern_setup_step_m
`svl_extern_check_step_m
extern virtual function void write(TR item);
endclass
```

コレクターが送信したトランザクションを write()メソッドで受け取る事ができます。以下の write()メソッドでは、トランザクションの内容をプリントしているだけです。

```
function void monitor_t::write(TR item);
svl_print_data_s row[$], column;
   operation = item.make_op();
   `svl_init_print_row_m(row)
   foreach(header[i]) begin
      case (i)
      0: column.m_value = $sformatf("@%3t:",$time);
      1: column.m_value = `svl_sformatfb_m(item.reset);
      2: column.m_value = `svl_sformatfb_m(item.load);
      3: column.m_value = `svl_sformatfb_m(item.up_down);
      4: column.m_value = `svl_sformatfh_m(item.d);
      5: column.m_value = `svl_sformatfh_m(item.q);
      6: column.m_value = `svl_sformatfh_m(item.qn);
      7: column.m_value = operation.name;
      endcase
      `svl_add_print_column_m(row,column,header[i])
   end
   lines.push_back(`svl_sprint_row_m(row));
endfunction
```

setup_step()では、ヘッダー情報をキューにセットします。

```
function void monitor_t::setup_step(svl_run_param_t param);
   lines.push_back(svl_sprint_header(header));
endfunction
```

check_step()では、キューにセットされている検証結果をプリントします。

```
function void monitor_t::check_step(svl_run_param_t param);
   foreach(lines[i])
      svl_system(lines[i]);
   svl_print_footer(header);
endfunction
```

9.1.19 agent_t

エージェントの全容は以下のようになります。このエージェントは、ドライバー、ジェネレータレータ、コレクター、モニターの全てを配置しています。

```
class agent_t extends svl_agent_t;
driver_t      driver;
generator_t   generator;
collector_t   collector;
monitor_t     monitor;
`svl_auto_component_m(agent_t)
```


Copyright 2024 © Artgraphics. All rights reserved.

```
`svl_component_new_m
`svl_extern_build_step_m
`svl_extern_connect_step_m
endclass
```

build_step()で、ドライバー、ジェネレータ、コレクター、モニターのインスタンスを作ります。

```
function void agent_t::build_step(svl_run_param_t param);
    super.build_step(param);
    if( m_function == SVL_DRIVE_CHECK ) begin
        driver = `svl_create_component_m(driver_t,"driver",this);
        generator = `svl_create_component_m(generator_t,"generator",this);
    end
    collector = `svl_create_component_m(collector_t,"collector",this);
    monitor = `svl_create_component_m(monitor_t,"monitor",this);
endfunction
```

connect_step()では、インスタンス同士を接続します。

```
function void agent_t::connect_step(svl_run_param_t param);
    super.connect_step(param);
    if( m_function == SVL_DRIVE_CHECK )
        driver.m_get_port.connect(generator.m_get_server);
    collector.m_send_port.add_receiver(monitor.m_receive_port);
endfunction
```

9.1.20 env_t

エンバイロンメントは、エージェントだけから構成される簡単な環境になっています。全容は以下の通りです。

```
class env_t extends svl_env_t;
agent_t       agent;
`svl_auto_component_m(env_t)
`svl_component_new_m
`svl_extern_build_step_m
endclass
endclass
```

build_step()でエージェントのインスタンスを配置します。

```
function void env_t::build_step(svl_run_param_t param);
    super.build_step(param);
    agent = `svl_create_component_m(agent_t,"agent",this);
endfunction
```

9.1.21 test_base_t

この検証環境には二つのテストケースがありますが、何れも同じ階層構造を持ちます。したがって、階層を作るベースクラスを定義しておくと便利になります。

```
class test_base_t extends svl_test_t;
env_t  env0;
`svl_component_new_m

function void build_step(svl_run_param_t param);
    super.build_step(param);
    env0 = `svl_create_component_m(env_t,"env0",this);
```


Copyright 2024 © Artgraphics. All rights reserved.

```
endfunction

endclass
```

9.1.22 test1_t

テストケース 1 のシナリオを build_step()で設定します。

```
class test1_t extends test_base_t;
`svl_auto_component_m(test1_t)
function new(string name="test1",svl_component_t parent=null);
    super.new(name,parent);
endfunction

// build_step
function void build_step(svl_run_param_t param);
    super.build_step(param);
    svl_scenario_t::allocate("test1*generator",`svl_create_object_m(
play_reset_down_t,"play"));
endfunction
endclass
```

9.1.23 test2_t

テストケース 2 のシナリオを build_step()で設定します。

```
class test2_t extends test_base_t;
`svl_auto_component_m(test2_t)
function new(string name="test2",svl_component_t parent=null);
    super.new(name,parent);
endfunction

// build_step
function void build_step(svl_run_param_t param);
    super.build_step(param);
    svl_scenario_t::allocate("test2*generator",`svl_create_object_m(
play_reset_up_t,"play"));
endfunction
endclass
```

9.1.24 pkg

検証環境を以下の示すパッケージにまとめておきます。

```
package pkg;
import svl_pkg::*;
import pkg_definitions::*;
`include "simple_item.sv"
`include "act_reset.sv"
`include "act_down.sv"
`include "act_up.sv"
`include "act_load.sv"
`include "act_upup.sv"
`include "act_loaddown.sv"
`include "act_upuploaddown.sv"
`include "play.sv"
`include "play_reset_down.sv"
`include "play_reset_up.sv"
`include "driver.sv"
```


Copyright 2024 © Artgraphics. All rights reserved.

```
`include "generator.sv"
`include "collector.sv"
`include "monitor.sv"
`include "agent.sv"
`include "env.sv"
`include "test_base.sv"
`include "test1.sv"
`include "test2.sv"
endpackage
```

9.1.25 top

トップモジュールの構成は、以下のようになります。

```
`include "svl_pkg.sv"
`include "svl_macros.sv"
`include "simple_if.sv"
`include "pkg.sv"

module top;
import svl_pkg::*;
import pkg_definitions::*;
import pkg::*;
logic     clk;

simple_if SIF(.clk(clk));
up_down_counter #(.NBITS(UP_DOWN_WIDTH))
   DUT(.clk,.reset(SIF.reset),.load(SIF.load),
       .up_down(SIF.up_down),
       .d(SIF.d),.q(SIF.q),.qn(SIF.qn));

initial begin
   svl_clock_gen(clk,20);
   svl_set_timeout(10,100);
   vif_config::set(SIF);
   svl_run_test();
end

endmodule
```

virtual インターフェース用にインターフェースのインスタンスを取り出しておく

参考 9-2

top モジュールでは、svl_set_timeout()メソッドによりタイムアウトを設定しています。したがって、タイムアウトが発生すると run_step()は終了させられますが、それぞれの検証コンポーネントに定義されている check_step()、conclude_step()と conclude_step()は順に呼び出されます。
□

9.1.26 テストの実行

シナリオが二つ存在するため、実行時にコマンドラインでテストを指定しなければなりません。指定法は以下のようになります。ここで、test_name はtest1_t、またはtest2_t の何れかです。

```
svsim +svl_testname=test_name
```

実行するとエンバイロンメントの内容もプリントされますが、プリント結果が非常に横に長いので、以下では省略します。

Copyright 2024 © Artgraphics. All rights reserved.

9.1.26.1 +svl_testname=test1_t

以下の結果を得ます。

```
=======================================
time  reset load up_down d  q  qn op
---------------------------------------
@  0: 1     0    0       00 00 ff RESET
@ 10: 0     0    0       00 ff 00 DOWN
@ 30: 0     0    1       00 00 ff UP
@ 50: 0     0    1       00 01 fe UP
@ 70: 0     1    0       50 50 af LOAD
@ 90: 0     0    0       00 4f b0 DOWN
=======================================
```

9.1.26.2 +svl_testname=test2_t

以下の結果を得ます。

```
=======================================
time  reset load up_down d  q  qn op
---------------------------------------
@  0: 1     0    0       00 00 ff RESET
@ 10: 0     0    0       00 ff 00 DOWN
@ 30: 0     0    0       00 fe 01 DOWN
@ 50: 0     1    0       50 50 af LOAD
@ 70: 0     0    1       00 51 ae UP
@ 90: 0     0    1       00 52 ad UP
=======================================
```

9.2 検証環境（スコアボードによる検証）

本節では、スコアボードを使用して検証環境を構築しますが、前節で使用した検証部品をできるだけそのままの形で活用します。つまり、一部だけを置き換えるだけでスコアボードによる検証環境を構築できるようにします。

パッケージ pkg ではファイルをインクルードしているので、関連する検証コンポーネントのファイルを入れ替えるだけで済みます。以下のコンポーネントの定義変更を行います。その他の検証部品には影響がありません。

- pkg の変更
- モニターの変更
- エンバイロンメントの変更
- スコアボードの追加
- トップモジュールの変更

スコアボードによる検証環境を図 9-7 のように構築します。つまり、スコアボードをエンバイロンメント内に配置して検証を行います。なお、モニターでは簡単なカバレッジ計算を行うようにします。

Copyright 2024 © Artgraphics. All rights reserved.

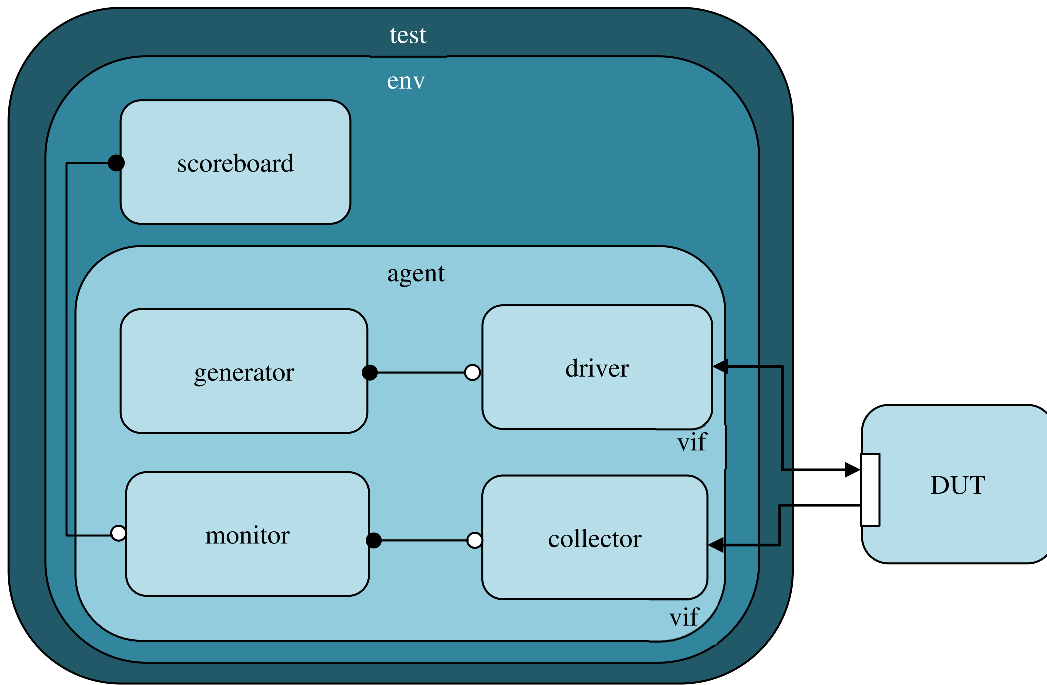


図 9-7　スコアボードによる検証環境

以下では、変更の影響を受ける検証コンポーネントを順に解説します。

9.2.1　パッケージ

パッケージ pkg_scb を pkg_scb.sv ファイルに以下の内容を定義します。

```
`ifndef   PKG_SCB_H        // scoreboard version
`define   PKG_SCB_H

`include "svl_pkg.sv"
`include "pkg_definitions.sv"

package pkg_scb;           // scoreboard version
import svl_pkg::*;
import pkg_definitions::*;
`include "simple_item.sv"
`include "act_reset.sv"
`include "act_down.sv"
`include "act_up.sv"
`include "act_load.sv"
`include "act_upup.sv"
`include "act_loaddown.sv"
`include "act_upuploaddown.sv"
`include "play.sv"
`include "play_reset_down.sv"
`include "play_reset_up.sv"
`include "driver.sv"
`include "generator.sv"
`include "collector.sv"
`include "monitor_scb.sv"  // scoreboard version
`include "agent.sv"
`include "scoreboard.sv"   // scoreboard version
`include "env_scb.sv"      // scoreboard version
`include "test_base.sv"
`include "test1.sv"
```


Copyright 2024 © Artgraphics. All rights reserved.

```
`include "test2.sv"
endpackage
```

9.2.2 モニター

スコアボード用のモニターを以下のように定義します。この定義を含むファイルは、monitor_scb.sv です。このモニターでは、簡単なカバレッジ計算を行います。

```
class monitor_t extends svl_monitor_t#(simple_item_t);
bit    perform_coverage = 1;
covergroup cg with function sample(up_down_op_e op);
   coverpoint op;
endgroup
`svl_auto_component_m(monitor_t)
extern function new(string name,svl_component_t parent);
extern virtual function void write(TR item);
endclass
```

モニターのコンストラクタでは、以下のようにカバーグループのインスタンスを作らなければなりません。

```
function monitor_t::new(string name,svl_component_t parent);
   super.new(name,parent);
   cg = new;
endfunction
```

write()メソッドでは、カバレッジ計算を行うと共にトランザクションを他の検証コンポーネントに送信します。カバレッジ計算としては、使用されたオペレーション（RESET、LOAD、UP、DOWN）の頻度を集計します。

```
function void monitor_t::write(TR item);
   if( m_perform_coverage )
      cg.sample(item.make_op());
   m_send_port.write(item);
endfunction
```

9.2.3 エンバイロンメント

エンバイロンメントにはスコアボードを配置するので、以下に示す変更を env_scb.sv ファイルに記述します。

```
class env_t extends svl_env_t;
agent_t       agent;
scoreboard_t  scoreboard;
`svl_auto_component_m(env_t)
`svl_component_new_m
`svl_extern_build_step_m
`svl_extern_connect_step_m
endclass
```

build_step()では、以下のようにエージェントに加えてスコアボードのインスタンスを作ります。

```
function void env_t::build_step(svl_run_param_t param);
   super.build_step(param);
   agent = `svl_create_component_m(agent_t,"agent",this);
```


Copyright 2024 © Artgraphics. All rights reserved.

```
    scoreboard = `svl_create_component_m(scoreboard_t,"scb",this);
endfunction
```

connect_step()では、以下のようにしてモニターとスコアボードを接続します。

```
function void env_t::connect_step(svl_run_param_t param);
    agent.monitor.m_send_port.add_receiver(scoreboard.m_receive_port);
endfunction
```

9.2.4 スコアボード

スコアボードを scoreboard.sv ファイルに定義します。スコアボードで検証を行うので、プリント用のヘッダーを定義する必要があります。その他、predict()メソッドを実装する必要があります。

スコアボードは DUT の出力が正しいか否かを検証するために、DUT の現在の状態に対応する状態を保存する変数が必要になります。この検証の場合には、DUT からの出力である q と qn に対応する値を保存しなければならないのですが、qn は~q として計算可能なので q だけで十分です。この値は、curr_q として定義されています。スコアボードの全容は以下のようになります。

```
class scoreboard_t extends svl_scoreboard_t#(simple_item_t);
parameter HEX_DIGITS = `svl_hex_digits_m(UP_DOWN_WIDTH);
svl_print_header_s  header[] = '{
              '{"time",5,SVL_HEADER_DEFAULT},
              '{"reset",5,SVL_HEADER_DEFAULT},
              '{"load",4,SVL_HEADER_DEFAULT},
              '{"up_down",7,SVL_HEADER_DEFAULT},
              '{"d",HEX_DIGITS,SVL_HEADER_DEFAULT},
              '{"q",HEX_DIGITS,SVL_HEADER_DEFAULT},
              '{"qn",HEX_DIGITS,SVL_HEADER_DEFAULT},
              '{"op",5,SVL_HEADER_DEFAULT},
              '{"results",7,SVL_HEADER_DEFAULT}
          };
up_down_op_e operation;
TR         expected_item;
logic [UP_DOWN_WIDTH-1:0]  curr_q;
string    lines[$];
`svl_auto_component_m(scoreboard_t)
extern function new(string name,svl_component_t parent);
`svl_extern_setup_step_m
`svl_extern_check_step_m
extern virtual function void write(TR item);
extern virtual function void predict(input TR item);
endclass
```

コンストラクタでは、検証結果を予測するために使用するトランザクションの入れ物（expeted_item）を定義しておきます。

```
function scoreboard_t::new(string name,svl_component_t parent);
    super.new(name,parent);
    expected_item = `svl_create_object_m(simple_item_t,"expected_item");
endfunction
```

setup_step()では、プリント用のヘッダー情報をキューにセットします。

```
function void scoreboard_t::setup_step(svl_run_param_t param);
```


Copyright 2024 © Artgraphics. All rights reserved.

```
    lines.push_back(svl_sprint_header(header));
endfunction
```

write()メソッドでは、predict()メソッドを呼び出して検証結果の予測を行い、DUT からの出力が予測された値と一致するか調べ、結果を入力値と共にプリントします。一致すれば SUCCESS、不一致であれば FAILURE がプリントされます。

```
function void scoreboard_t::write(TR item);
svl_print_data_s   row[$], column;
int         judge;
    predict(item);
    judge = (expected_item.q == item.q) && (expected_item.qn == item.qn);
    `svl_init_print_row_m(row)
    foreach(header[i]) begin
        case (i)
        0: column.m_value = $sformatf("@%3t:",$time);
        1: column.m_value = `svl_sformatfb_m(item.reset);
        2: column.m_value = `svl_sformatfb_m(item.load);
        3: column.m_value = `svl_sformatfb_m(item.up_down);
        4: column.m_value = `svl_sformatfh_m(item.d);
        5: column.m_value = `svl_sformatfh_m(item.q);
        6: column.m_value = `svl_sformatfh_m(item.qn);
        7: column.m_value = operation.name;
        8: column.m_value = judge ? "SUCCESS" : "FAILURE";
        endcase
        `svl_add_print_column_m(row,column,header[i])
    end
    lines.push_back(`svl_sprint_row_m(row));
endfunction
```

検証結果を予測するpredict()メソッドは以下のようになります。トランザクションに指定されている入力を基にして期待される出力を算出します。算出した値をトランザクションに設定すると共に、現在の状態も保存します。

```
function void scoreboard_t::predict(TR item);
    if( item.reset ) begin
        operation = RESET;
        expected_item.q = '0;
    end else if( item.load ) begin
        operation = LOAD;
        expected_item.q = item.d;
    end else if( item.up_down ) begin
        operation = UP;
        expected_item.q = curr_q+1;
    end else begin
        operation = DOWN;
        expected_item.q = curr_q-1;
    end
    expected_item.qn = ~expected_item.q;
    curr_q = expected_item.q;
endfunction
```

check_step()()では以下のようにキューに記録しておいた検証結果をプリントします。

```
function void scoreboard_t::check_step(svl_run_param_t param);
    foreach(lines[i])
        svl_system(lines[i]);
    svl_print_footer(header);
endfunction
```


Copyright 2024 © Artgraphics. All rights reserved.

9.2.5

9.2.6 トップモジュール

トップモジュールでは、インクルードファイルの変更と使用するパッケージを`pkg_scb`に変更するだけで良いです。

```
`include "svl_pkg.sv"
`include "svl_macros.sv"
`include "simple_if.sv"
`include "pkg_scb.sv"

module top;
import svl_pkg::*;
import pkg_definitions::*;
import pkg_scb::*;
logic      clk;

simple_if SIF(.clk(clk));
up_down_counter #(.NBITS(UP_DOWN_WIDTH))
    DUT(.clk,.reset(SIF.reset),.load(SIF.load),.up_down(SIF.up_down),
    .d(SIF.d),.q(SIF.q),.qn(SIF.qn));

initial begin
    svl_clock_gen(clk,20);
    svl_set_timeout(10,100);
    vif_config::set(SIF);
    svl_run_test();
end
endmodule
```

9.2.7 テストの実行

二つのテストケースを以前と同様に実行してみます。今回の実行では、検証結果の判定カラムが追加されています。

9.2.7.1 +svl_testname=test1_t

以下の結果を得ます。

```
===============================================
time   reset load up_down d  q  qn op    results
-----------------------------------------------
@  0: 1     0    0       00 00 ff RESET SUCCESS
@ 10: 0     0    0       00 ff 00 DOWN  SUCCESS
@ 30: 0     0    1       00 00 ff UP    SUCCESS
@ 50: 0     0    1       00 01 fe UP    SUCCESS
@ 70: 0     1    0       e8 e8 17 LOAD  SUCCESS
@ 90: 0     0    0       00 e7 18 DOWN  SUCCESS
===============================================
```

カバレッジ計算結果は以下のように 100%カバレッジです。

```
COVERPOINT "op";
    COVERAGE 100.00 4 4;
    GOAL 100;
    WEIGHT 1;
    COMMENT "";
    ATLEAST 1;
    EBIN "auto" "RESET" 1;
    EBIN "auto" "LOAD" 1;
    EBIN "auto" "UP" 2;
    EBIN "auto" "DOWN" 2;
```


Copyright 2024 © Artgraphics. All rights reserved.

```
ENDCOVERPOINT
```

9.2.7.2 +svl_testname=test2_t

以下の結果を得ます。

```
===============================================
time   reset load up_down d  q  qn op    results
-----------------------------------------------
@  0: 1     0    0       00 00 ff RESET SUCCESS
@ 10: 0     0    0       00 ff 00 DOWN  SUCCESS
@ 30: 0     0    0       00 fe 01 DOWN  SUCCESS
@ 50: 0     1    0       e8 e8 17 LOAD  SUCCESS
@ 70: 0     0    1       00 e9 16 UP    SUCCESS
@ 90: 0     0    1       00 ea 15 UP    SUCCESS
===============================================
```

カバレッジ計算結果は以下のように 100%カバレッジです。

```
COVERPOINT "op";
   COVERAGE 100.00 4 4;
   GOAL 100;
   WEIGHT 1;
   COMMENT "";
   ATLEAST 1;
   EBIN "auto" "RESET" 1;
   EBIN "auto" "LOAD" 1;
   EBIN "auto" "UP" 2;
   EBIN "auto" "DOWN" 2;
ENDCOVERPOINT
```


Copyright 2024 © Artgraphics. All rights reserved.