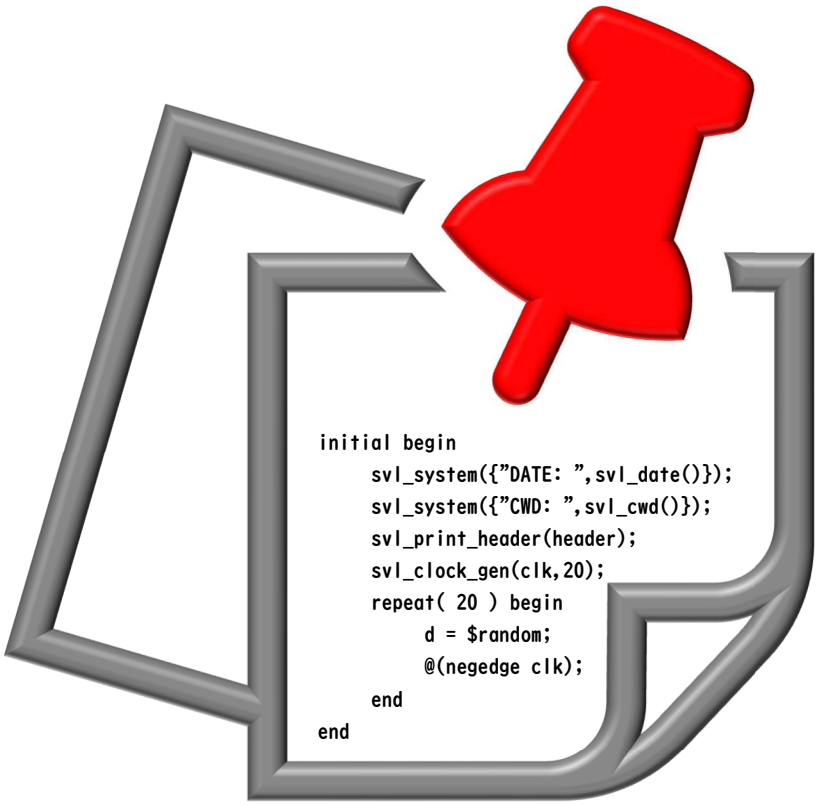
Artgraphics

SVL

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-005-2024
Document Revision: 1.0, 2024.10.31
アートグラフィックス
篠塚一也



SVL
検証のための SystemVerilog ライブラリー

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

SystemVerilog Library (SVL) 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、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。

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 は非常に強力なプリント機能を備えています。その機能は、柔軟性があり、しかも汎用的です。それらの機能を使用する事により、ビット数の変化に影響を受けない汎用的なプリント機能を開発できます。

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、WIDTH が変化しても自動的にカルム調節が行われます。以下に実行例を紹介します。

```
          WIDTH==16                          WIDTH==32
=========================          ====================================
time  reset d      q               time  reset d           q
-------------------------          ------------------------------------
@ 10: 0     'h9d66 'h9d66          @ 10: 0     'h8f17_9d66 'h8f17_9d66
@ 30: 0     'h9b47 'h9b47          @ 30: 0     'h575e_9b47 'h575e_9b47
@ 50: 0     'h7f20 'h7f20          @ 50: 0     'h0424_7f20 'h0424_7f20
@ 70: 0     'h51a6 'h51a6          @ 70: 0     'hf816_51a6 'hf816_51a6
@ 90: 0     'ha6e2 'ha6e2          @ 90: 0     'h814a_a6e2 'h814a_a6e2
@100: 1     'h4017 'h0000          @100: 1     'h4c12_4017 'h0000_0000
@110: 0     'h4017 'h4017          @110: 0     'h4c12_4017 'h4c12_4017
@130: 0     'hcd05 'hcd05          @130: 0     'h422d_cd05 'h422d_cd05
@150: 0     'h2534 'h2534          @150: 0     'hccef_2534 'hccef_2534
@170: 0     'ha061 'ha061          @170: 0     'h6873_a061 'h6873_a061
@190: 0     'h75d9 'h75d9          @190: 0     'h7f26_75d9 'h7f26_75d9
=========================          ====================================
```

モデルで指定された名称が指定された順に、指定されたカラム幅でプリントされている事を確認できます。カラムの順序を変更するには、モデルを変更し、値の設定順序を変更すれば済むので複雑な作業を伴いません。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。この例が示すように、SVL を使用する事により生産性が著しく向上する事は容易に想像できます。

SVL は、プリント処理等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。

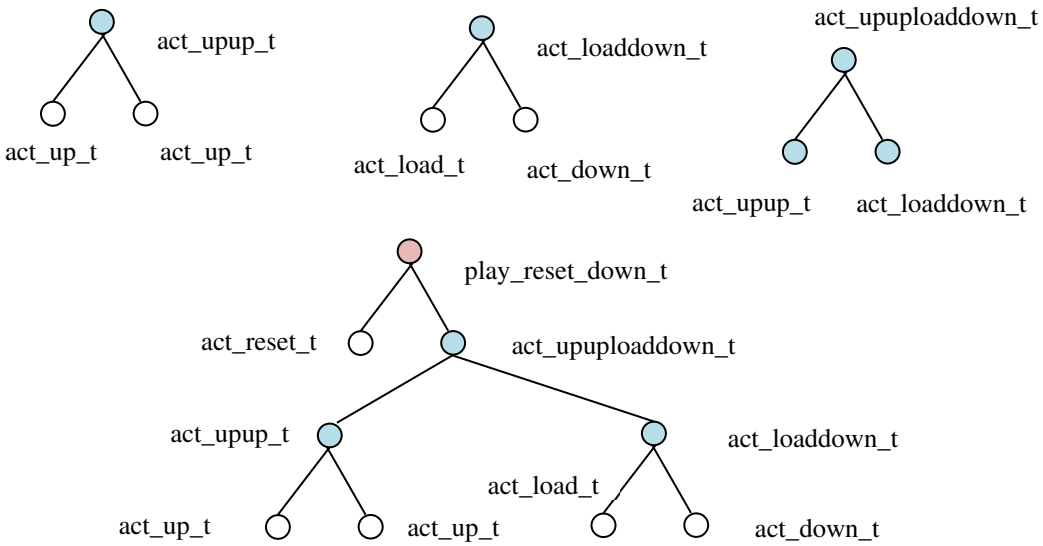
- トランザクションのベースクラス
- メソドロジークラス（ドライバー、ジェネレータ、コレクター、モニター、エージェント、テスト等）
- TLM ポート
- テストを行うためのシナリオ
- 直感的でわかり易い virtual インターフェースの設定と取得法
- ダイナミックに検証環境を変更する機能

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。

```
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
```

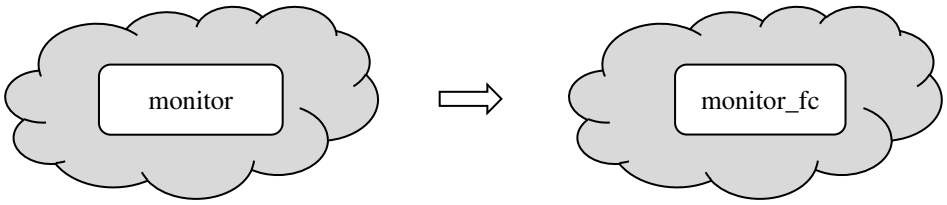
この回路は、RESET、LOAD、UP、DOWN の機能を持ちます。したがって、それらの機能に対応するテストデータを準備し、それらのテストデータを任意の順序で組み合わせれば、様々なテストケースを実現できます。

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。シミュレーションの実行は `run_step()` で行われますが、それ以前のステップにおいて初期値の変更が可能です。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。



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。

```
class monitor_t extends svl_monitor_t#(simple_item_t);
`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
```

置き換え可能な自動コンポーネントである事を宣言する

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。実際に置き換えるためには、インスタンスを作る命令が実行する前に、変換タイプを設定しておきます。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)
    ...
    svl_run_test();
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、検証環境内において以下のようにモニターのインスタンスを作っていても、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。つまり、検証環境のソースコードを変更せずに検証コンポーネントを入れ替える事ができます。

```
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

// build_step
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。また、正則表現機能や便利で使い易いファイル入出力機能も備えられています。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。

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
篠塚一也

変更履歴

日付	Revision	変更点
2024.10.31	1.0	初版。

目次

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。

- 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
```

```
`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。

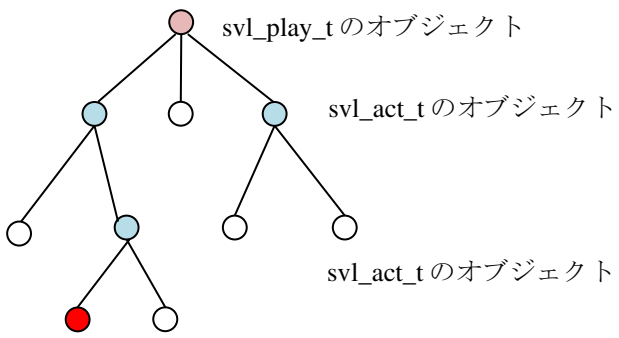


図 1-1　トランザクション生成ツリー

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。

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。