"eXecutable Verification Plan (XVP)"

Joe McCann
Gaurav Brahmbhatt
Gaurang Chitroda
Pinal Patel
Manish Patel

Synopsys International Limited
Dublin, Ireland
www.synopsys.com

eInfochips Ltd.
Sunnyvale, USA
www.einfochips.com

ABSTRACT

*e**X**ecutable **V**erification **P**lan is an organized plan which describes all design features that need to be verified. XVP's parameterized plan, editor independence, Testbench independence and a degree of automation features helps to easily create and maintain an organized functional coverage report for any verification environment. It avoids the burden of maintaining large Word documents or Excel spreadsheets.*

Table of Contents

1. Introduction of XVP.......3
2. Need for XVP.......5
3. How to write XVP.......6
4. XVP Guidelines.......15
5. Conclusion.......27
6. Result.......28
7. Reference.......29

Table of Figures

Figure 1.1.1 XVP Flow Chart.......3
Figure 3.1 Inheritance Diagram for XvpGen<t_XvpItem> and Custom_Class.......6
Figure 4.4.1 Flow of XVP Development Guidelines.......15
Figure 4.4.2 Example of Hierarchy in Coverage Database.......18
Figure 4.4.3 Back Annotated Plan of Example.......26
Figure 4.4.4 Coverage Report with Issues.......26
Figure 7.6.1 Sample of XVP Coverage Report.......28

Table of Tables

Table 3.1 Naming Convention of the Functional Coverage Plan.......14

1. Introduction of XVP

XVP is a collection of SystemVerilog classes to encapsulate verification requirements. It is a method of creating and maintaining a hierarchical verification and functional coverage plan. It allows for editor independence, revision control and a degree of automation. It avoids the burden of maintaining a large Word documents or Excel spreadsheets. It uses VMM Planner as a tool to present the plan with annotated results from regressions.

The goal of creating XVP is to give the IP team confidence that the IP is bug free. Generate collateral to give customers visibility to our internal verification. Single XVP item is a verification requirement which is captured from protocol specifications (PCIe, USB, AMBA, PIPE etc.), product data book, Change order specifications or RTL implementation. XVP is Testbench independent. All changes made to the RTL code must contribute to the XVP plan.

The following figure 1.1 outlines the XVP flow, from specification in SystemVerilog to the generation of a plan annotated with results from the coverage database.

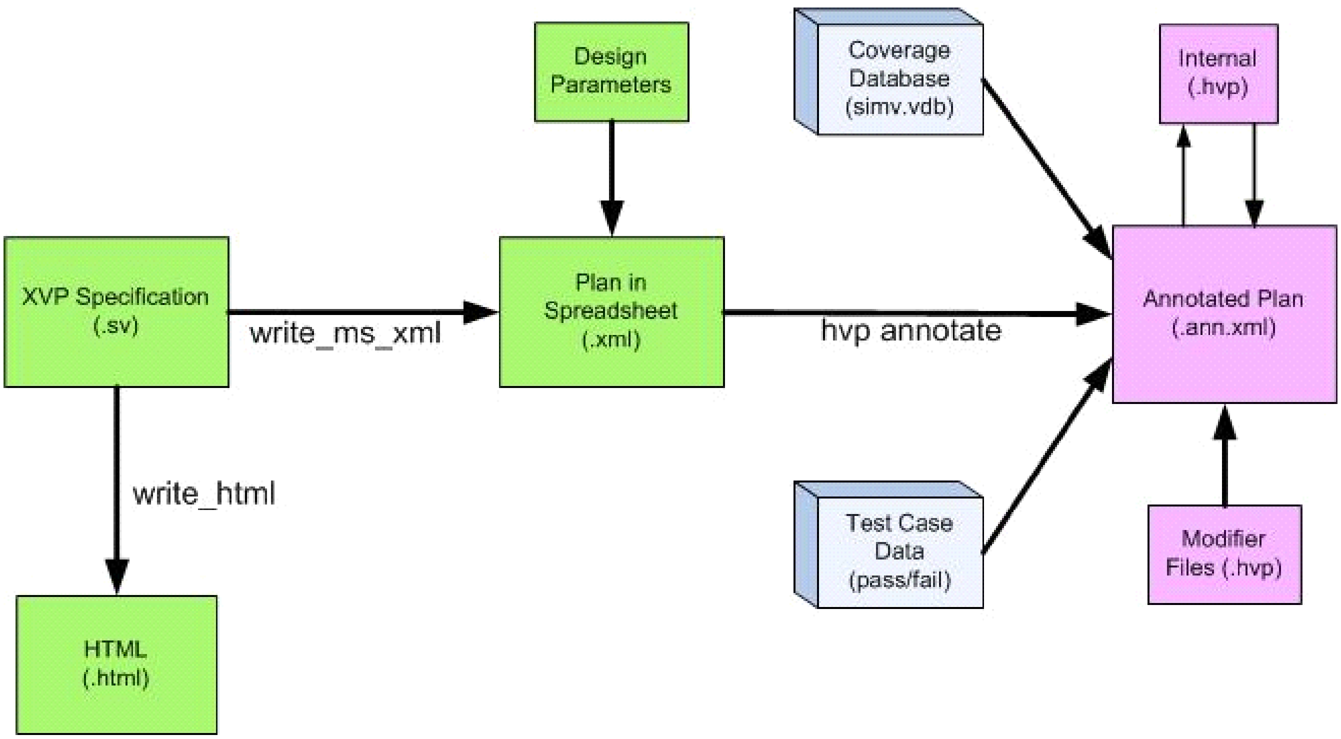


Figure 1.1.1 XVP Flow Chart

Green: XVP domain
Purple/pink: Verification planner domain
Blue: Simulation data inputs to annotator tool

Verification Planner is a Synopsys tool that presents a verification plan in a hierarchical manner. A verification plan can be described using spreadsheets formatted with special meta-tags. Results from regressions are then back-annotated using the Verification Planner Spreadsheet Annotator tool.

In the XVP flow, the plan is described in SystemVerilog, and an XML spreadsheet is generated with the required meta-tags.

The use of SystemVerilog as the entry point in the XVP flow shields the user from the underlying HVP (Hierarchical Verification Plan) language.

2. Need for XVP.

Currently many tools support their own coverage managers. These are used to represent the given functional or code coverage in either metric form. Traditionally, this coverage planner gives us the full picture after a regression on how much coverage is achieved or what are the missing coverage points. This generally works fine typically where you have a single DUT only. But when you are verifying a very complex IP where various combinations are possible of DUT based on customer requirements. You need the coverage planner to be intelligent so that it can auto scale based on which feature you select for a particular configuration of DUT.

Another approach is you can ifdef in your full Testbench for writing coverage but this looks messy and can become very difficult to manage when there are more and more features or changes in existing features. So we need a coverage planner which should be intelligent that based on the selected configuration it should automatically scale your coverage metrics so that we get the correct result at the end. XVP has this feature and that is why is much powerful and compared to other vplan manager. Even we will be writing the xvp by our-self so no license or tool dependency is involved. Some other feature XVP like we can add a Plan to the coverage metrics so that if some of the features is not part of current release (design) then we can simply add a phase against that XVP item and XVP item would simply be ignored. Because of auto scale feature mentioned above, XVP is best while verifying IP.

3. How to write XVP

XVP model uses a recursive class structure to build the plan hierarchy. The base class (Xvp_Item) is a parameterized recursive class. It must be extended to create a new plan. The extended class sets the input parameter type t_XvpItem and this sets the type for the nested items. The XVP base class library consists of a set of SystemVerilog classes. XVP is compiled, a hierarchical database of coverage items is built up that can be converted in different formats including XML and HTML.

XVP Generator class is used for writing out the verification plan in Microsoft XML, Microsoft XML (Custom Format), HVP or HTML Table form. XVP_Gen class is extended from the XvpItem. The input parameter type t_XvpItem is set by the extended class.

XvpCustom is a XVP container class. XvpCustom class Extends XvpGen class. We can set the parameter to generate nested items of the custom class type. XvpCustom class declare feature tasks and add to feature list.

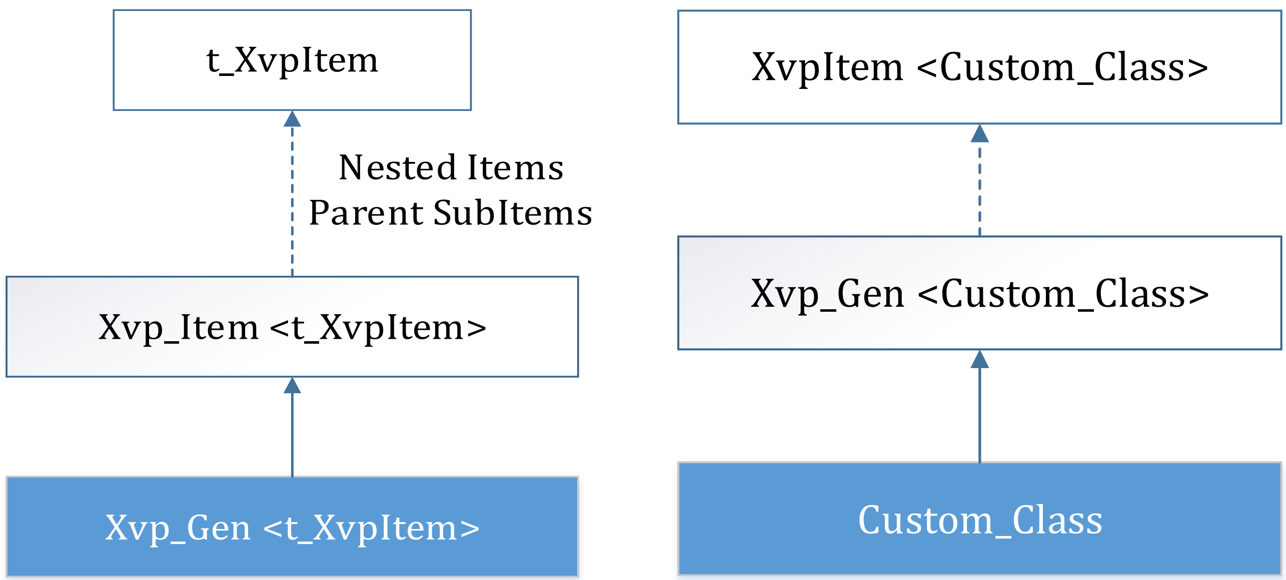


Figure 3.1 Inheritance Diagram for XvpGen<t_XvpItem> and Custom_Class

Below is the code snippet of base class (XvpItem).

<u>Code Snippet:</u>

<u>XvpEnums:</u>

```
// This file has all the enums which are used in XVP
typedef enum { RX_DATAPATH,
               TX_DATAPATH,
               AMBA,
               LTSSM,
               …
             } e_eng_grp;
```

XvpItem:

```
// Base class for defining all XVP coverage types.
// The base class is a parameterized recursive class.
// This base class contains a number of utility functions used to add verification requirement to the // hierarchy while creating the plan hierarchy.

`include "XvpEnums.sv"
class XvpItem # (type t_XvpItem = XvpItem)

      // -------------------------------------------------------------
      function new();
         this.plan_name = "";
         this.name = "";
         this.description = "";
         this.eng_grp = "";
         …
       endfunction : new
      // --------------------------------------------------------------

   // we have different method as mentioned below to print the XVP
   // cover group:
   // check the description string for special characters.
   // Print plan to screen
   // Print a string to the output file pointer
   // propagate the file pointer down through the hierarchy
   // propagate the max index value down through the hierarchy
   // -----------------------------------------------------------------

   // -----------------------------------------------------------------
   // Task: begin_feature
   // XvpItem::begin_feature
   // Used to create a new feature and added new level of hierarchy.
   // Pass different XVP items in to the argument
   // i.e. name          Name of the coverage item
   //      description   Brief description of the coverage item
   //      item_type     Coverage item type
   //      exclude       Exclude item from plan
   //      check_status  Check status (used when item_type==CHECK_ITEM)
   // -----------------------------------------------------------------

  virtual task begin_feature(string name = "",
                             string description = "",
                             e_item_type   item_type = FUNC_FEATURE,
                             e_dev_status dev_status = NOT_IMPLEMENTED,
                             bit exclude = 0,
                             …
                             );
                            t_XvpItem si;
```

```
// Set members passed in via task arguments, and also Inherit
// arguments (i.e. top_inst, engineering group) if not passed from
// task argument
    si.name = name;
    si.description = description;
    …
    // Inherit exclude if set in parent feature
    if (item_index == 0) begin
      if (this.exclude == 1) begin
        si.exclude = 1;
      end else begin
        si.exclude = exclude; // Use task argument
      end
    end else begin
      if (this.nestedItems[this.item_index-1].exclude == 1) begin
        si.exclude = 1;
      end else begin
        si.exclude = exclude; // Use task argument
      end
    end
    …

    // Store pointer to parent item and evaluate unique ID
    if (item_index==0) begin
      // Need cast for container pointer
      $cast(si.parent, this);
      si.item_id = $psprintf("%0s%0s", this.name, name);
    end else begin
      si.parent = this.nestedItems[this.item_index-1];
      si.item_id = $psprintf("%0s%0s",
this.nestedItems[this.item_index-1].item_id, name);
    end

    // Insert feature object into nestedItems queue
    this.nestedItems.insert(this.item_index, si);

endtask : begin_feature

// -----------------------------------------------------------------
// Task: add_sub_item
// XvpItem::add_sub_item
// Use to add requirement into current level of hierarchy. It enters
// a new item to the plan.
// creates a new item in the plan which adds a leaf to the hierarchy
// tree.
// i.e. name            Name of the coverage item
//    description       Brief description of the coverage item
//    item_type         Coverage item type
//    exclude           Exclude item from plan
//    check_status Check status (used when item_type == CHECK_ITEM)
// -----------------------------------------------------------------
```

```
virtual task add_sub_item (string        name = "",
                           string        description = "",
                           e_item_type   item_type = FUNC_FEATURE,
                           e_check_status check_status = NOT_VERIFIED,
                           bit           exclude = 0,
                           …
                           ); t_XvpItem si;

       // Set members passed in via task arguments
       si.name = name;
       si.description = description;
       si.item_type = item_type;
       …

       // Inherit exclude if set in parent feature
       if(this.nestedItems[this.item_index-1].exclude == 1) begin
         si.exclude = 1;
       end else begin
         si.exclude = exclude;
       end
       …

     // Count number of Check Item and store sub item in nestedItems
     // array
     this.nestedItems[item_index-1].subItems.push_back(si);
  endtask : add_sub_item

  // -------------------------------------------------------------------
  // Task: add_feature_description
  // XvpItem::add_feature_description
  // Use to add the description about the added new feature
  // we can also append feature description after begin_feature has
  // been called.
  // -------------------------------------------------------------------

  virtual task add_feature_description (string description="");
     this.nestedItems[this.item_index-1].add_description(description);
  endtask : add_feature_description

  // -------------------------------------------------------------------
  // Task: end_feature
  // XvpItem::end_feature
  // Use to step back up a level of hierarchy.
  // All begin_feature() entries must have corresponding end_feature()
  // Compile error if not obeyed.
  // End of feature:
  //         - Push all sub items into parent item
  //         - Flag error if task called when item_index==0
  // -------------------------------------------------------------------

  virtual task end_feature();
        t_XvpItem si;
```

```
       if (this.item_index==0) begin
      $display("ERROR:    Extra    end_feature()    call    detected
(item_index=%0d), number of      {begin/end}_feature calls should
match.", this.item_index);
      $finish();
    end else begin
      if (this.item_index==1)
        this.subItems.push_back (this.nestedItems.pop_back());
      else begin
        si=this.nestedItems.pop_back();
        this.nestedItems[item_index-2].subItems.push_back(si);

        // Push all sub items into parent subItems queue
        this.nestedItems[item_index-2].ff_num_ci+=si.ff_num_ci;
        this.nestedItems[item_index-2].ff_ci_completed+=
                                        si.ff_ci_completed;
      end
      this.item_index--;
    end
  endtask : end_feature

   // Check plan is generated correctly
   virtual task check_plan();
    if(item_index > 0) begin
      $display("ERROR:  Plan  not  generated  correctly,  number  of
{begin/end}_feature  calls  should  match  (Actual  item_index=%0d,
Expected item_index=0).", this.item_index);
    end else begin
      $display("Plan   generated   correctly   (item_index=%0d).",
this.item_index);
    end
  endtask : check_plan

// ------------------------------------------------------------------
// Methods to generate XVP in different formats i.e. XML, HVP and
// Html form
// ------------------------------------------------------------------

  //Write plan in Microsoft XML format
  virtual task write_ms_xml();
    // User fills implementation details for this task in their
    // extended class.
  endtask : write_ms_xml

  // ----------------------------------------------------------------
  // Write plan as a html table
  virtual task write_html_table();
    // User fills implementation details for this task in their
    // extended class.
  endtask : write_html_table

  // ----------------------------------------------------------------
```

```
     // Write out plan in hvp format
      virtual task write_hvp (integer subf=0);
       // User fills implementation details for this task in their
       // extended class.
     endtask : write_hvp

endclass : XvpItem
```

Each plan entered in using begin_feature(), add_sub_item() and end_feature() tasks. Each item has the following attributes: Container, FuncFeature, CheckItem, TestCase, CovGroup, CovPoint, CrossCov, Assertion, or GoldenRef.

Each item type is set using the item_type (enum e_item_type) attribute. Note there is one and only one Container per plan. This is the root item, which contains the entire plan.

1. Container: Used if the class object is used as a container for the plan. There is one and only one CONTAINER per plan. This is the root item which contains the entire plan.
2. FuncFeature: Used where the type is just a functional feature; a functional feature is not a check or coverage item. It is used to implement hierarchy in the plan.
3. CheckItem: A check item derived from protocol/functional specification. A checkitem is verified either by an agent in the Testbench (BFM, protocol checker) or by a directed testcase.
4. CovGroup: A functional coverage group
5. CovPoint: A functional coverage point. A member of a functional coverage group
6. CrossCov: Define a cross coverage between two or more cover points/variables
7. TestCase: A specified directed test case description.
8. GoldenRef: A golden reference is used to verify the DUT response.
9. CodeCoverage: Used for specifying code coverage
10. Assertion: A SystemVerilog assertion.

An additional class XvpChkLst is used to create an object in the test environment which tallies the checkitem as coverage points. Each checkItem has an equivalent enumerated value. The XvpChkLst creates an associative array of this item whose value is recorded as e_check_status.

```
typedef enum {
  NOT_VERIFIED,       /*< A verification hole */
  VERIFIED,           /*< Stimulus for checkitem has been injected to DUT with correct response */
  VERIFIED_BY_BFM,    /*< Check is automatically covered by env or some agent in Testbench */
  NOT_APPLICABLE,     /*< Check does not apply to current DUT configuration */
  UNIT_LEVEL,         /*< Verified by standalone or unit level Testbench */
```

FAILED /*< Checkitem simulation has failed and associated coverpoint will remain unchanged */

} e_check_status;

XVP can write out the following formats: html, HVP, Microsoft XML, SystemVerilog. URG exclude file.
Utility tasks that should be extended by user when implementing their plan:

1. XvpItem::hvp_dump Write out plan in hvp format
2. XvpItem::write_svh_code. Write out systemverilog header file to be included in test environment.
3. XvpItem::write_sv_code Write out systemverilog file to be included in test environment
4. XvpItem::write_doxy Write out Doxygen marked up file.
5. XvpItem::write_xvp_chk_lst Write out XvpChkLst and enumerated types.
6. XvpItem::write_el Write out URG exclude file.
7. XvpItem::write_html_table Write out plan as a html table
8. XvpItem::write_ms_xml Write out plan as MS Excel XML file. Note this is limited to adding rows to an existing XML file rather than creating a customizable XML file.

To back annotate the results of a full covergroup to the plan, specify the covergroup in XVP without any coverpoints. In this case the plan score will match the covergroup score in the URG. Otherwise, the plan score will be calculated from the specified coverpoints.

Annotate Coverage in XVP:

1) COV_POINT:

```
add_sub_item (
                .name ("cp_vc_enable"),
                .description ("Coverage of VC Enable"),
                .snps_src ( ),
                .item_type (COV_POINT)
              );
```

2) COV_POINT_BIN:

```
add_sub_item (
                .name ("cov_rspeed_rx_ele_idle_inferred_succesful_speed_negotiation_0_GEN3"),
                .description ("When successful_speed_negotiation is 0 then Electrical Idle condition is
        inferred if EIEOS is not received within 16000UI for Gen3 on any configured lanes"),
                .exclude (!this.params.get_val ("CX_GEN3_SPEED")),
```

```
            .snps_src
("*::*::**.cp_chks.*COV_RSPEED_RX_ELE_IDLE_INFERRED_SUCCESFUL_SPEED_NEGOTIATION_0_GEN3"),
            .item_type (COV_POINT_BIN)
            );
```

3) ASSERTION:

```
add_sub_item(
            .name         ("a_supporetd_data_rate_assertion_ap"),
            .exclude      (!this.params.get_val("CX_GEN2_SPEED") ),
            .description  ("xmlh_ltssm: The supported data rate must not change in recovery.rcvLock, recovery.rcvrCfg, recovery.rcvreq unless ......"),
            .item_type    (ASSERTION)
            );
```

4) COV_GROUP:

```
begin_feature (
            .name         ("cg_rcvry_lock_stimulii"),
            .description  ("Coverage of how the state has been stimulated, especially where no transition is expected"),
            .exclude      (! this.params.get_val ("CX_GEN2_SPEED")),
            .item_type    (COV_GROUP)
            );
end_feature ();
```

5) COV_GROUP_INST:

It should be used if a cover group is being defined inside a class and that class being instantiated multiple times and the cover group name is being passed as an input to the new() function.

```
add_sub_item (
            .name         ("OS error injection"),
            .description  ("Coverage of ordered set error injection implemented by RxPipeAgent manipulating OS received on the PIPE before the DUT receives them."),
            .snps_src     ("**.USD.CG_PIPE_RX_AGENT_CB_S_CFG_LINKWD_START"),
            .item_type    (COV_GROUP_INST)
            );
```

To create an XVP for any protocol, we need to create XVP plan for the same. This XVP plan specifies the target cover bins that need to be covered in the given regression.

Using a consistent naming convention will further enhance the readability and consistency of the functional coverage plan. Here is the naming convention recommended when specifying functional coverage items in XVP:

Prefix	Description
feature_*	Task containing a list of XVP coverage items and assertions relating to a particular functional feature.
cg_<feature>_*	A SystemVerilog coverage group linked to a particular feature.
cp_*	A SystemVerilog coverage point, a member of a coverage group.
cc_*	A SystemVerilog cross coverage point, a member of a coverage group.
cb_*	A SystemVerilog coverage bin, a member of a coverage point.
ap_<feature>_*	A SystemVerilog assert property used as a checker for a particular feature.
cp_<feature>_*	A SystemVerilog cover property used for coverage of a particular feature.

Table 3.1 Naming Convention of the Functional Coverage Plan

XVP is intended to be used in planning verification for configurable IP products. Therefore, by having a verification plan written in SV as source it is then possible to configure the plan by reading parameter files (<xx>_cc_constants.v) and adjusting the plan accordingly.

4. XVP Guidelines

While developing XVP we have to follow few guidelines. As shown in the below mentioned figure 4.1, To develop XVP for any DUT, initially we have to extract requirements from the design specification of the design and as mentioned in the above mentioned section add features and items in to the corresponding XVP hierarchy.

After added xvp item for new feature, we have to implement the verification code to perform/ check the coverage of the feature. Update xvp database by adding new xvp item which is related to the added new code reference. After update xvp database compile XVP structure so that because of any typo XVP structure is not break.

After compilation generate the XVP and first check the hierarchy which is same as expected or not. Check the cover bins whether all are hit or not if any bin not hit then debug it and if require write a testcase or develop a code for that and again check whether it hit or not or cover or not. After achieved 95-100% coverage it is reviewed by design and verification team and if it is achieved all then submit that XVP to customer.

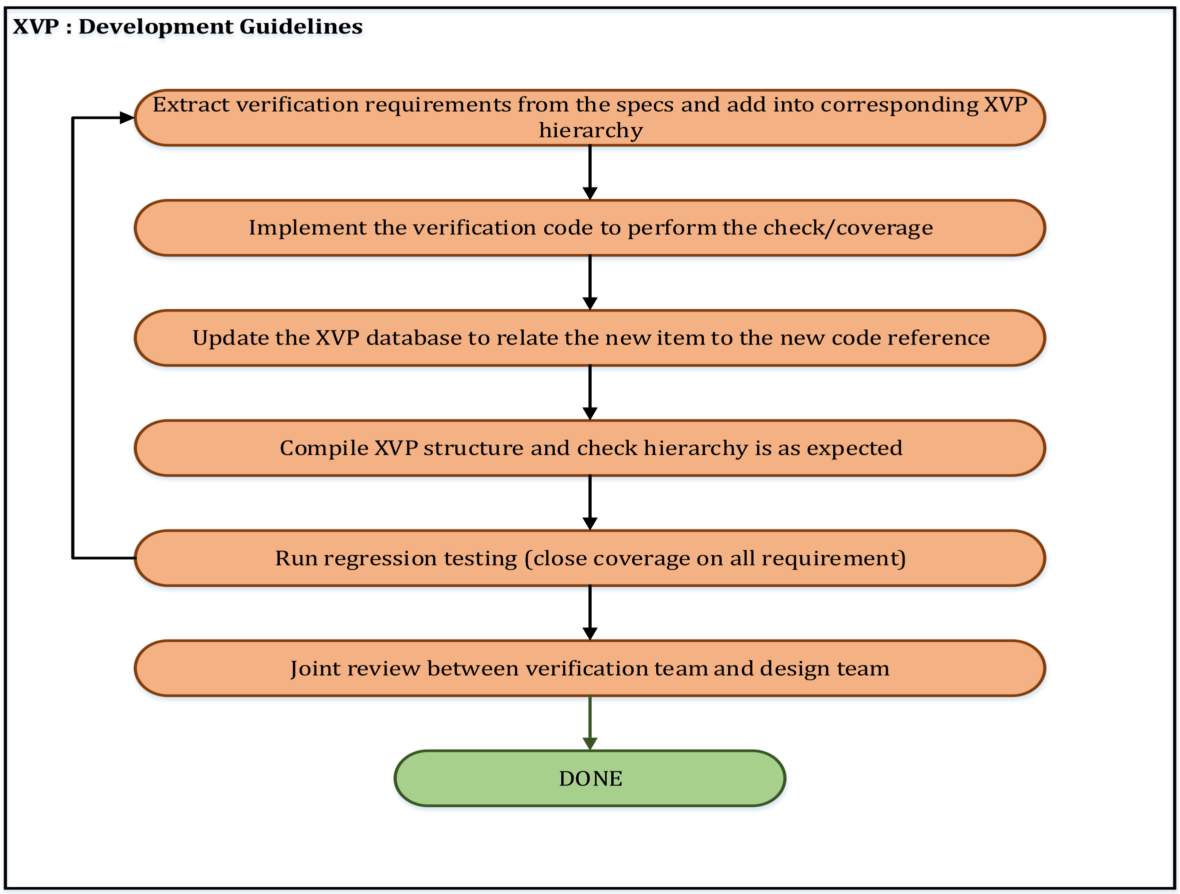


Figure 4.4.1 Flow of XVP Development Guidelines

This Testbench contains nested list of features (to implement hierarchy) and sub items (leaf cell for annotation) in a hierarchical manner. Two API tasks provided as part of the XVP library. Arguments for name, description, item_type, phase... specified within a task, which is included in a class of the XVP library.

The following members are inherited from the parent feature if not provided as an argument to the begin_feature task:

- phase (can also be provided as argument to add_sub_item task)
- exclude
- eng_grp
- spec_ref
- top_inst
- inst

Phase filtering:

"phase" argument is use in begin _feature () and add_sub_item () methods. We can specifying any number as an input to the phase. If the phase value is same as user define number (which user can select any number during development cycle) then item will not appear in the spreadsheet. If we do not need filter than do not supply any argument. Phase filtering is hierarchical. If item at higher nesting level is filtered, your item is filtered. If we are not able to see expected entry in the spreadsheet then check, the higher level phase values. If a phase value is provided as argument to the begin_feature or add_sub_item tasks that is lower than the parent feature phase, then the phase argument is ignored. Here, in the below mentioned example we define phase value 1.

```
begin_feature (
        .name          ("state transitions"),
        .description   ("Cover state transitions from Configuration.Idle (any conditions)"),
       .spec_ref       (),
       .item_type      (FUNC_FEATURE),
       .phase          (1)
      );
```

Exclusions:

If the verification requirement is configuration dependent, it must have correct exclusion applied. Controlled via the "exclude" argument in begin_feature () and add_sub_item () methods. Use this.params.get_value ("PARAM_NAME") to apply the exclusion. We cannot use core parameters directly because XVP structure must compile without cc_file. If the exclude member of a feature is set to 1, then the exclude member of all sub features and sub items is also set.

XVP's real power is .exclude () variable because with the help of .exclude () variable we can exclude cover properties based on the configuration of the IP. i.e. few properties only covered @Gen3 rate in PCIe so we can add exclude in begin_feature() and add_sub_item() method so these properties are only covers when data rate is Gen3 else those properties are excluded in the XVP file.

- begin_feature(.exclude (!this.params.get_val("CX_GEN3_SPEED")));

Similarly, xvp also take care to also add Up Stream Port / Down Stream Port exclusions.

- begin_feature(.exclude (`ifdef DOWNSTREAM_PORT 1 `else 0 `endif)

```
add_sub_item (
      .name        ("cov_rspeed_rx_ele_idle_inferred_succesful_speed_negotiation_0_GEN3"),
      .description ("When successful_speed_negotiation is 0 then Electrical Idle condition is
```

```
    Inferred if EIEOS is not received within 16000UI for Gen3 on any configured lanes"),
   .exclude   (! this.params.get_val ("CX_GEN3_SPEED")),
   .snps_src   ("*::*::**.cp_chks.*COV_RSPEED_RX_ELE_IDLE_INFERRED_
                    SUCCESFUL_SPEED_NEGOTIATION_0_GEN3"),
   .item_type (COV_POINT_BIN)
  );
```

Engineering Group:

Engineering Group argument use to give a logical grouping to requirements that span the specification or span the datapath. Controlled via the "eng_grp" argument in begin_feature () and add_sub_item () methods. In order to focus in on requirements for a particular project it is useful to give engineering group that can be filtered in the spreadsheet. This argument can be string type or enumerated type.

```
add_sub_item (
   .name ("cfglwstart_noupcfg_init_upcapable1_send_ts1_link_num_inactive_lanes"),
   .description ("Cover: If Transmitter has upconfigure_capable=1, and the LTSSM is not initiating upconfig of the Link Width, the LTSSM sends TS1 OSs with Link number selected from CDM and Lane number set to PAD, on each inactive lane after it detected an exit from Electrical Idle since entering Recovery and has subsequently received two consecutive TS1 OSs with the Link and Lane numbers each set to PAD."),
   .eng_grp   ("UPCFG"),
   .snps_src   ("**.cp_chks.*CFGLWSTART_DSP_NOUPCFG_INIT_UPCAPABLE1_SEND
                        _TS1_LINK_NUM_INACTIVE_LANES"),
   .item_type (COV_POINT_BIN)
  );
```

Spec Reference:

It should be used in the vplan_dwc_pcie_core hierarchy. Not required in vplan_*spec hierarchy. Controlled via the "spec_ref" argument in begin_feature () and add_sub_item () methods. If an item in the core hierarchy can be referenced back to the data book we must add databook.section to the spec_ref argument. Avoid using spec section numbers as these are subject to change.

Input to the spec_ref argument can be from

- Data book
- Any design Change Order Specification
- RTL Implementation
- Design specification
- Protocol specification

```
add_sub_item (.name          ("cp_received_filter_reserved"),
              .description   ("Determine that TLP fields that are marked reserved are not
                               checked at the receiver"),
              .spec_ref      ("PCIe Spec Figure 2-31"),
              .item_type     (COV_PROPERTY)
             );
```

Coverage for checks:

All checks performed anywhere in the environment must be included in XVP structure. If you have a verification component that is performing checks (emulation model, monitor, etc.), guideline is as follows.

- Create a coverage group within the component (i.e. cg_<compName>_check_coverage)
  - Add a coverage point into the group for each check performed by the model
  - Sample the group as the check is performed (if check passes)
  - In XVP list only the group and not each individual coverage point

Same guidelines above for checks embedded in tests. If the check is implemented in an Assertion, add to XVP as normal.

```
add_sub_item (
        .name           ("cp_lpbk_active_unaligned"),
        .description    ("smlh_ltssm: cover entry to loopback active when
                          blockalignment has been lost."),
        .exclude        (! this.params.get_val ("CX_GEN3_SPEED")),
        .item_type      (ASSERTION)
    );
```

XVP Generation:

Below is an example of how to generate a functional coverage plan using XVP is provided with the class library. This consists of a basic Testbench containing a random packet generator driving a simple DUT. Coverage points and assertions are implemented in the DUT and Testbench. The corresponding XVP specification is provided in code snippet, which is included as a task of the extended class XvpCustom. This is a useful reference for understanding how to specify a coverage plan in XVP, and how to link it back to the verification environment so that coverage results can be annotated to the plan.

Note the example Testbench is not integrated to a product development area and can only be run from workspace of the XVP library.

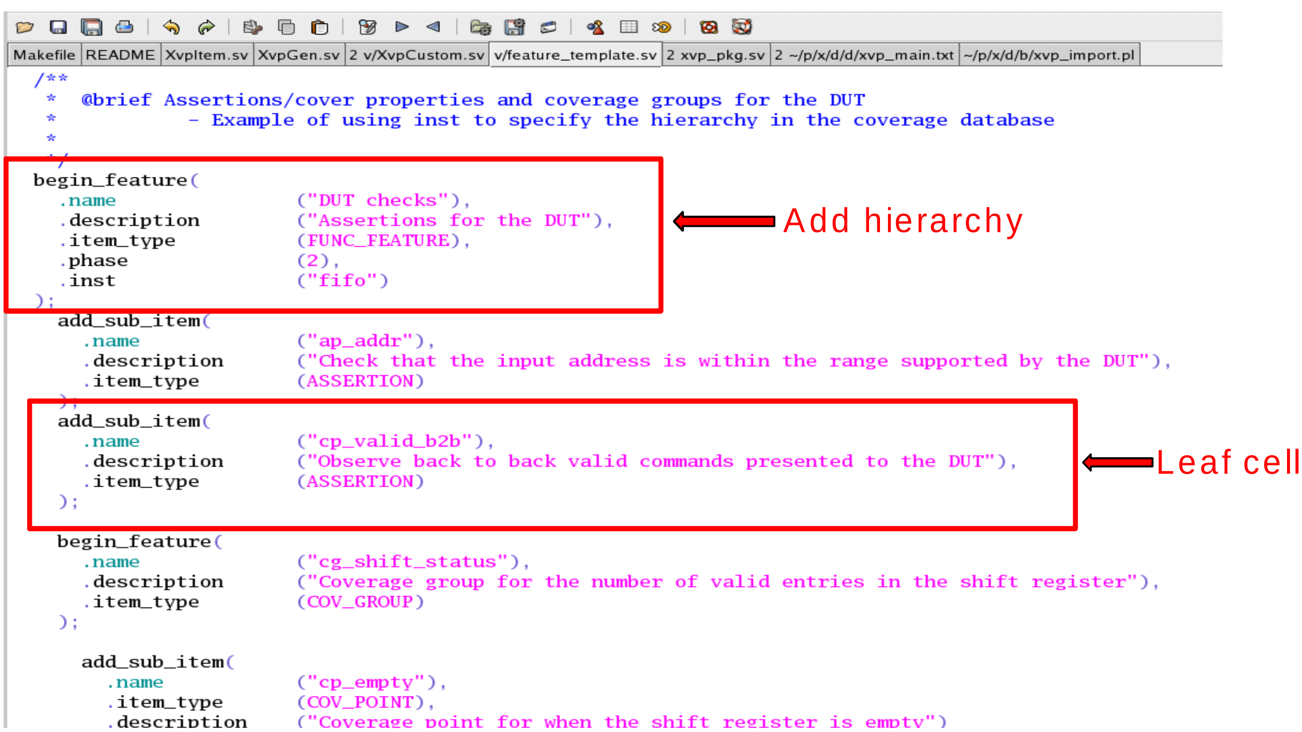


Figure 4.4.2 Example of Hierarchy in Coverage Database

Code Snippet:

```
//==============================================================
// @brief Feature specification template:
// - Specify coverage items for the example Testbench.
//   - Use the following tasks defined in the base class XvpItem:
//       - begin_feature ()
//       - add_sub_item ()
//       - end_feature ()
//   - Implement hierarchy by specifying nested features.
// - The number of {begin/end} feature calls should match.
// - Not all arguments have to be specified for each feature or sub
// item.
// - The following members are inherited from the parent feature if
//      not provided as an argument to the begin_feature task:
//      - top_inst
//      - inst
//      - phase(can also be provided as argument to add_sub_item task)
//      - exclude
//      - no_cg_name
//==============================================================

task `CUSTOM_CLASS:: feature_template (input bit exclude_decode=0);

 //==============================================================
 // Functional coverage group (without associated coverage points)
 // - Source string specified for the coverage group in the XML file.
 // - Coverage group score is for all coverage points implemented in
 // the group.
 //==============================================================

   begin_feature(
     .name          ("Address coverage (full)"),
     .description   ("Coverage for the address ranges (without
associated coverage points)"),
     .spec_ref      ($psprintf("%s - Section 1.0", this.spec_ref)),
     .item_type     (FUNC_FEATURE),
     .phase         (1)
   );

     add_sub_item (
       .name         ("cg_addr"),
       .description  ("Coverage group for the address ranges (without
associated coverage points)"),
       .item_type    (COV_GROUP)
   );

   end_feature (); // Address coverage

   //==============================================================
   // Functional coverage group (with associated coverage points)
   // - Source string not specified for the coverage group in the XML
```

```
   // file.
   // Coverage group score is only for the specified coverage points.
   //============================================================

   begin_feature (
     .name               ("Address coverage (selected)"),
     .description         ("Coverage   for   the   address   ranges   (with
associated coverage points)"),
     .item_type          (FUNC_FEATURE),
     .phase              (1)
   );

     begin_feature (
       .name            ("cg_addr"),
       .description     ("Coverage group for the address ranges (with
associated coverage points)"),
       .item_type       (COV_GROUP)
     );

       add_sub_item (
         .name           ("cp_addr_low"),
         .item_type      (COV_POINT),
         .description    ("Coverage point for low address ranges")
       );
       add_sub_item (
         .name           ("cp_addr_mid"),
         .item_type      (COV_POINT),
         .description    ("Coverage point for mid address ranges")
       );
       add_sub_item (
         .name           ("cp_addr_high"),
         .item_type      (COV_POINT),
         .description    ("Coverage point for high address ranges")
       );

     end_feature (); // cg_addr

//============================================================
// Example of a covergroup that contains bins.
//============================================================

     begin_feature (
       .name            ("cg_addr"),
       .description      ("Coverage group for the address ranges (with
associated  coverage  points)  but  with  a  coverpoint  with  specified
bins"),
       .item_type       (COV_GROUP)
     );

       begin_feature (
         .name           ("cp_addr_mid"),
         .item_type      (COV_POINT),
```

```
        .description      ("Coverage point for all address ranges but
with conditional bins")
       );
         add_sub_item (
           .name          ("mid_low_range"),
           .item_type     (COV_POINT_BIN),
           .description   ("Coverage point for all address ranges but
with conditional bins")
         );

      end_feature (); // cg_addr
    end_feature (); // cg_addr
  end_feature (); // Address coverage

//==============================================================
// Functional coverage group (with associated coverage points)
// Example of COV_GROUP_INST usage for covergroups that are
// instantiated several times.
//==============================================================

   begin_feature (
     .name                ("Data coverage"),
     .description         ("Coverage for the data ranges"),
     .item_type           (FUNC_FEATURE),
     .top_inst            ("*"),
     .phase               (1)
   );

     begin_feature (
       .name             ("cg_example"),
       .description       ("Coverage group data for all packets (sum of
all cg instances)"),
       .item_type        (COV_GROUP)
     );
     end_feature (); // cg_example

     begin_feature (
       .name              ("**.cg_data_0"),
       .description       ("Coverage group data for first packet"),
       .item_type         (COV_GROUP_INST)
     );
       add_sub_item (
         .name            ("cp_access"),
         .item_type       (COV_POINT),
         .description     ("Coverage point for data in first packet")
       );
     end_feature (); // cg_data_0

     begin_feature (
       .name              ("**.cg_data_1"),
       .description       ("Coverage group data for second packet"),
       .item_type         (COV_GROUP_INST)
```

```
      );
        add_sub_item (
          .name           ("cp_access"),
          .item_type      (COV_POINT),
          .description    ("Coverage point for data in second packet")
        );
      end_feature (); // cg_data_1

      begin_feature (
        .name             ("**.cg_data_2"),
        .description      ("Coverage group data for third packet"),
        .item_type        (COV_GROUP_INST)
      );
        add_sub_item (
          .name           ("cp_access"),
          .item_type      (COV_POINT),
          .description    ("Coverage point for data in third packet")
        );
      end_feature (); // cg_data_2
   end_feature (); // Data coverage

//=============================================================
// Assertions/cover properties and coverage groups for the DUT
// Example of using inst to specify the hierarchy in the coverage
// database
//=============================================================
   begin_feature (
     .name               ("DUT checks"),
     .description        ("Assertions for the DUT"),
     .item_type          (FUNC_FEATURE),
     .phase              (2),
     .inst               ("fifo")
   );
     add_sub_item (
       .name            ("ap_addr"),
       .description      ("Check that the input address is within the
range supported by the DUT"),
       .item_type       (ASSERTION)
     );
     add_sub_item (
       .name           ("cp_valid_b2b"),
       .description    ("Observe back to back valid commands presented
to the DUT"),
       .item_type      (ASSERTION)
     );

     begin_feature (
       .name             ("cg_shift_status"),
       .description       ("Coverage group for the number of valid
entries in the shift register"),
       .item_type        (COV_GROUP)
     );
```

```
       add_sub_item (
         .name           ("cp_empty"),
         .item_type      (COV_POINT),
         .description     ("Coverage point for when the shift register
is empty")
       );
       add_sub_item (
         .name           ("cp_half"),
         .item_type      (COV_POINT),
         .description     ("Coverage point for when the shift register
is half full")
       );
       add_sub_item (
         .name           ("cp_full"),
         .item_type      (COV_POINT),
         .description     ("Coverage point for when the shift register
is full")
       );

     end_feature (); // cg_shift_status

     add_sub_item (
       .name             ("cp_shift_addr_even"),
       .description       ("Observe a valid even address at the end of
the shift array"),
       .exclude          (exclude_decode),
       .item_type        (ASSERTION)
     );
 end_feature (); // DUT checks

//==============================================================
// Assertions/cover properties for the packet interface
//==============================================================
 begin_feature (
     .name               ("Interface checks"),
     .description        ("Assertions for the packet interface"),
     .item_type          (FUNC_FEATURE),
     .phase              (2)
   );
     add_sub_item (
       .name           ("ap_addr_x"),
       .description    ("Check address not X when valid is asserted"),
       .item_type      (ASSERTION)
     );

   end_feature (); // Interface checks

//==============================================================
// Code coverage for the DUT
// Top level instance coverage
//==============================================================
```

```
   begin_feature (
     .name           ("Code Coverage"),
     .description    ("Code coverage for the DUT"),
     .item_type      (FUNC_FEATURE),
     // Indicates we are collecting instance (not module) coverage
     .inst           ("dut"),
     .phase          (3)
   );

     add_sub_item (
       .name           ("DUT top level"),
       .description    ("Code coverage for the DUT top level (instance
coverage)"),
       .item_type      (CODE_COVERAGE)
     );
   end_feature (); // Code Coverage

//============================================================
// Test cases
//   - Example of linking test case results to the plan.
//   - A HVP file containing a list of test case results for the
// example Testbench is generated in the Makefile.
//============================================================

   begin_feature (
     .name           ("Test cases"),
     .description    ("List of test cases"),
     .item_type      (FUNC_FEATURE),
     .phase          (4)
   );

     add_sub_item (
       .name          ("Demo test case"),
       .description   ("Demonstrate the generation of packets in the
example Testbench"),
       .item_type     (TEST_CASE),
       .testcase_name ("test_demo")
     );

   end_feature (); // Test cases

//============================================================
// Sub plan
// - Example of including a sub plan.
//  - The name of the sub plan and the location are provided as
// arguments.
// - Sub plan appears as an extra tab in the annotated plan.
//============================================================
   begin_feature (
     .name           ("AMBA"),
     .description    ("Functional coverage for the AMBA AXI and AHB
interfaces"),
```

```
    .item_type      (FUNC_FEATURE),
    .phase          (5)
  );

    add_sub_item (
      .name           ("vplan_amba"), // Name of the sub plan
      .description     ("Sub   plan   for   the   AMBA   AXI   and   AHB
interfaces."),
      .item_type       (SUB_PLAN),
      .include_file_name  ("../vplan_amba/vplan_amba.xml") // relative
to xvp/vplan_template directory
    );

  end_feature (); // AMBA

 endtask: feature_template

 //============================================================
```

Now, If we want to run the XVP flow in the Testbench environment than first of all we have to add the package XVP library create a ccustomized script to generate the URG reports and run XVP after simulation regressions have completed then generate a HTML report with links to the URG and XVP reports for each configuration of the DUT.

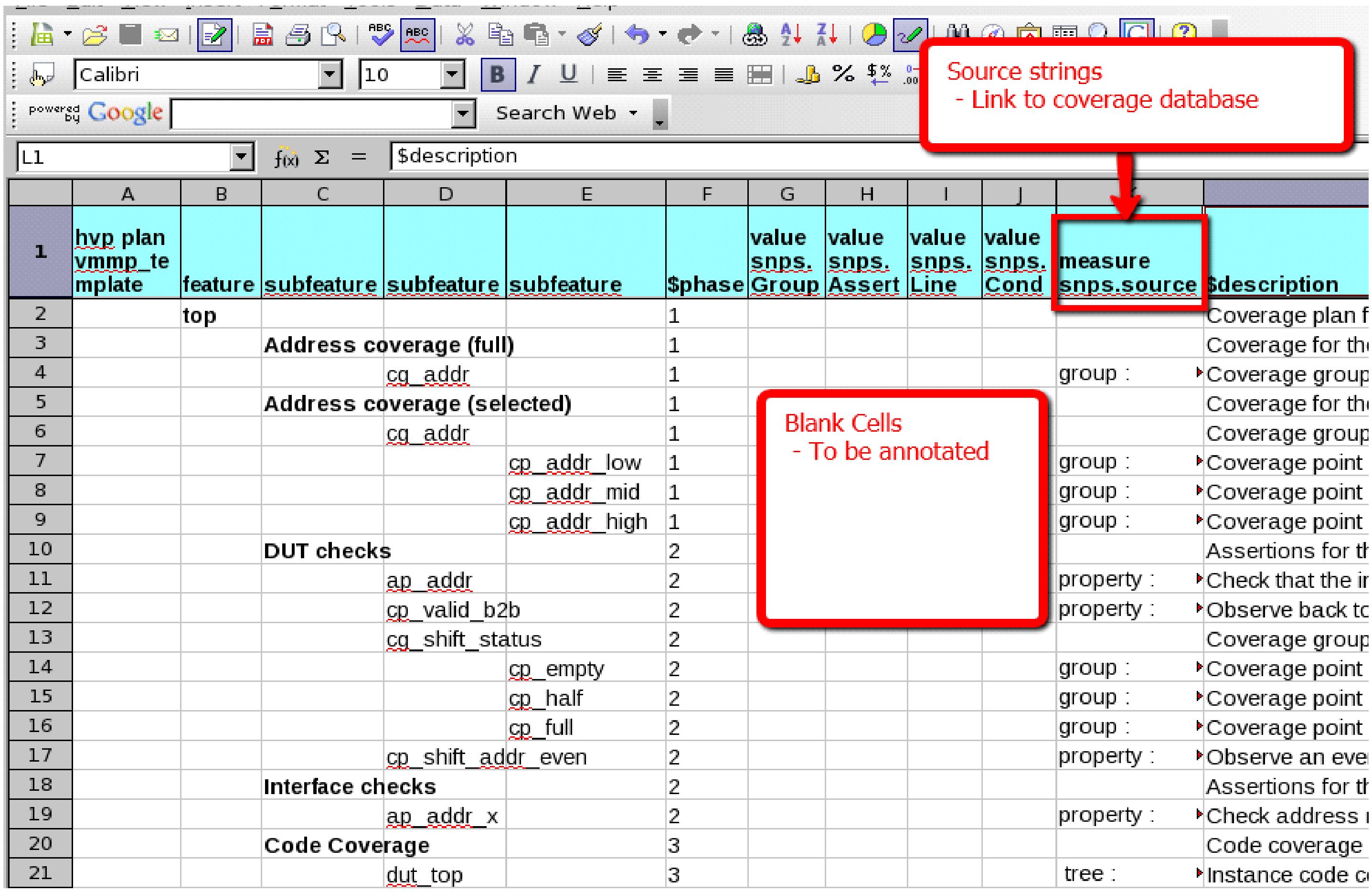
As mentioned below, the first row in the generated table contains tags to identify columns for the Verification Planner tool. Generation of source strings automated in XVP based on information provided in specification.

XVP is compiled using Make file with different arguments as mentioned below,

- To generate the plan in XML format:
  - make xml TAG=template
- To generate a filtered plan based on the configuration:
  - make xml TAG=template FILTER_TYPE=CONFIG
- To generate a filtered plan based on the phase:
  - make xml TAG=template FILTER_TYPE=PHASE PHASE_VALUE=2
- To generate a back annotated plan:
  - make hvp TAG=template COVDB_DIR=example_tb/simv.vdb

Testing & Debugging of XVP

After XVP structure generated by scripts we have to ensure that all entries in to the XVP structure are covered or not. We have to verify the whole structure, we have to make sure that we do not break the compilation of XVP, also have to check the feature name mentioned in feature is match with the name, which mentioned in the environment file so that there is no blue cell in the XVP report. If there is any blue cell in the XVP file then there is must be no link between the cover point between the XVP item and cover point, so to remove that we have to check exclusions and also check annotation works across all the configs.

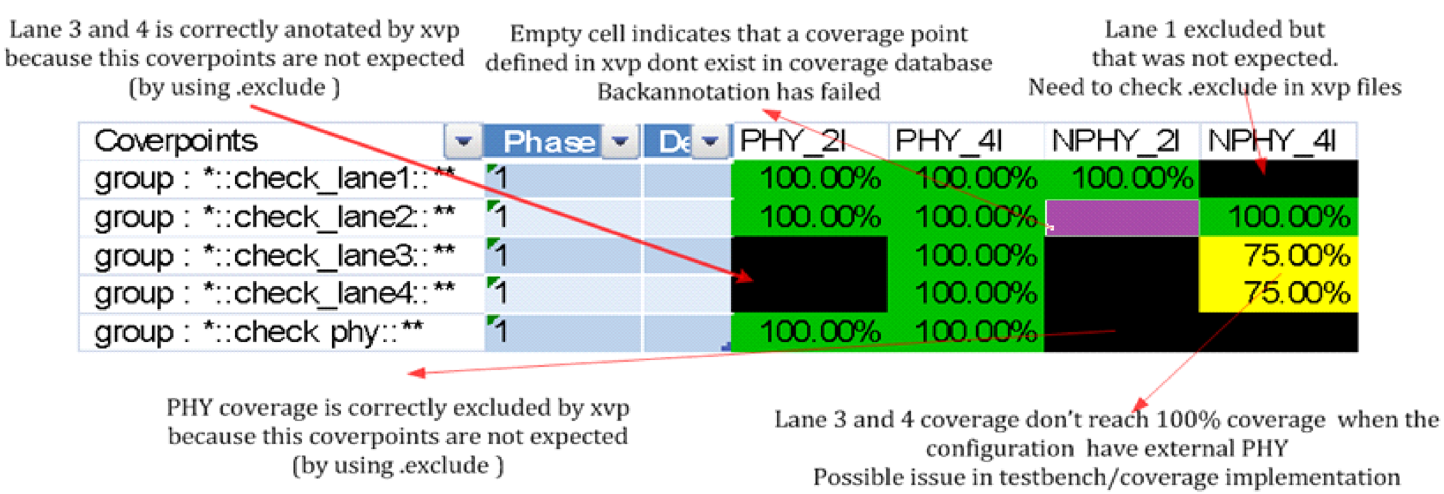


	A	B	C	D	E	F	G	H	I	J	K	
1	hvp plan vmmp_template	feature	subfeature	subfeature	subfeature	$phase	value snps. Group	value snps. Assert	value snps. Line	value snps. Cond	measure snps.source	$description
2		top				1						Coverage plan f
3			Address coverage (full)			1						Coverage for th
4				cg_addr		1					group :	Coverage group
5			Address coverage (selected)			1						Coverage for th
6				cg_addr		1						Coverage group
7					cp_addr_low	1					group :	Coverage point
8					cp_addr_mid	1					group :	Coverage point
9					cp_addr_high	1					group :	Coverage point
10			DUT checks			2						Assertions for th
11				ap_addr		2					property :	Check that the ir
12				cp_valid_b2b		2					property :	Observe back to
13				cg_shift_status		2						Coverage group
14					cp_empty	2					group :	Coverage point
15					cp_half	2					group :	Coverage point
16					cp_full	2					group :	Coverage point
17				cp_shift_addr_even		2					property :	Observe an eve
18			Interface checks			2						Assertions for th
19				ap_addr_x		2					property :	Check address i
20			Code Coverage			3						Code coverage
21				dut_top		3					tree :	Instance code c

Figure 4.4.3 Back Annotated Plan of Example

We have to make sure that we have to cover required targets in regression. If we need to write any testcase than we have to write required testcase to cover it and also have to ensure that unreachable targets are not left. We also have to add proper excludes for the cover points so that it run only for the required config and required feature.

A quick look at the results can point us to some issues. below is an axample:



Coverpoints	Phase	De	PHY_2l	PHY_4l	NPHY_2l	NPHY_4l
group : *::check_lane1::**	1		100.00%	100.00%	100.00%	
group : *::check_lane2::**	1		100.00%	100.00%		100.00%
group : *::check_lane3::**	1			100.00%		75.00%
group : *::check_lane4::**	1			100.00%		75.00%
group : *::check phy::**	1		100.00%	100.00%		

Figure 4.4.4 Coverage Report with Issues

5. Conclusion

XVP is an organized verification plan, which describes all of the design's features that need to verify. It allows features to be broken down in to sub features in a hierarchical fashion. With the XVP we can also accommodate other information like descriptions, diagrams, charts etc. XVP provide both global and target view of the plan. XVP is better utilized while verifying IP because of its auto scaling feature.

6. Result

Following figure shows the sample XVP coverage report. This report contains the coverage figures for the checker and cover properties under a specific section of the specification.

feature	value. Group	value .Assert	value tst.test	measure.source	$description
PCIe Base Specification	61.01%	87.18%	total=50 pass=48 fail=2		Coverage plan for the PCIe Base Specification
1. Physical Layer Specification	83.20%	76.19%	total=50 pass=48 fail=2		PCI EXPRESS Base Specification REV. 3.1
1.2.. Link Training and Status State Rules	90.15%	73.33%			PCI EXPRESS Base Specification REV. 3.1
1.2.1. Detect	75.00%				PCI EXPRESS Base Specification REV. 3.1
1.2.1.1. Detect.Quiet	100.00%				PCI EXPRESS Base Specification REV. 3.1
state transitions	100.00%				Cover state transitions from detect quiet (any conditions)
detect_quiet_to_detect_active	100.00%			group instance bin : **LINK_STATE_TRANSITIONS.cp_ltssm_state_transitions .detect_quiet_to_detect_active	Cover state transtion from detect quiet to detect active.
state transitions conditions	100.00%				Cover state transitions from detect quiet (all conditions explicitly covered)
detect_active_via_12ms_timeout	100.00%			group instance bin : **.LTSSM_WB_S_DETECT_QUIET_to_S_DETECT_ACT.cp_reason.valid_reason_TIMEOUT_12MS	Cover next state is detect active after a 12 ms timeout.
transmitter rules	50.00%				Rules that transmitter must obey in Detect.Quiet
check_detect_txdeem_half_swing_gen1	100.00%			group bin : *::*::**.cp_chks.*CHECK_DETECT_TXDEEMP_HALF_SWING_GEN1	If the negotiated data rate is 2.5 GT/s, and if operating in half swing mode, 2'b10(No de-emphasis) de-emphasis level must be selected for operation
check_detect_txdeem_full_swing_gen1	0.00%			group bin : *::*::**.cp_chks.*CHECK_DETECT_TXDEEMP_FULL_SWING_GEN1	If the negotiated data rate is 2.5 GT/s, and if operating in full swing mode, -3.5 dB de-emphasis level must be selected for operation

Figure 7.6.1 Sample of XVP Coverage Report

Coverage figure in red shows that the cover point is not covered, so coverage is 0.00% for the same. We can also see the overall coverage for a particular section and its subsection. So, XVP is an easy and useful visual representation of coverage plan which helps track design coverage in efficient and manageable way.

7. Reference

[1] Universal Verification Methodology (UVM) 1.1 Class Reference

[2] PCI Express ® Base Specification Revision 3.0 Version 1.0

[3] AMBA™ Specification (Rev 2.0)