VerifSudha

High-level specification model based Functional coverage generation

(With case study of USB Power delivery protocol layer coverage)

Author: Anand Shirahatti
(anand@verifsudha.com)

Intent is the seed of manifestation

1 Introduction

We were presenting our whitebox functional and statistical coverage generation solution, one of the engineer asked, can it take standard specifications as input and generate the functional coverage from it?

Figure 1: *Specification to functional coverage magic possible?*

I replied "No". It cannot.

But then after the presentation, questioned myself as to, why not?

No, no still not brave enough to parse the standard specifications use natural language processing (NLP) to extract the requirements and generate the functional coverage from it. But we have taken first step in this direction. It's a baby step. May be some of you might laugh at it.

We are calling it as high level specification model based functional coverage generation. It has some remarkable advantages. As every time, I felt this is "the" way to write functional coverage from now on ☺

Idea is very simple. I am sure some of you might have already doing it as well. Capture the specification in form of data structures. Define bunch of APIs to filter, transform, query and traverse the data structures. Combine these executable specifications with our python APIs for SystmVerilog functional coverage generation. Voila, poor man's specification to functional coverage generation is ready.

Yes, you need to learn scripting language (python in this case) and re-implement some of the specification information in it. That's because SystemVerilog by itself does not have necessary firepower to get it all done. Scared? Turned off? No problem. Nothing much is lost. Please stop reading from here and save your time.

Adventurers and explorers surviving this hard fact please hop on. I am sure you will fall in love with at least one thing during this ride.

2 How is this approach different?

How is this approach different from manually writing coverage model? This is a very important question and was raised by Faisal Haque.

There are multiple advantages, which we will discuss later in the article. In my view single biggest advantage is making the coverage intent executable by truly connecting the high-level model of specifications to functional coverage. No we are not talking about just putting specification section numbers in coverage plan we are talking about really capturing the specification and using it for generation of functional coverage.

Let me set the expectations right, this approach will not figure out your intent. The idea is about capturing and preserving human thought process behind the functional coverage creation in executable form. So that it can be easily repeated when things change. That's all. It's a start and first step towards specifications to functional coverage generation.

Typically functional coverage is implemented as set of discrete independent items. The intent and its connection to specifications are weak to non-existent in this type of implementation. Most of the intent gets either left behind in the word of excel plan where it was written or in the form of comments in the code, which cannot execute.

3 Making intent executable

Why capturing intent in executable form is important?

We respect and value the human intelligence. Why? Is it only for this emotion reason? No. Making human intelligence executable is first step to artificial intelligence.

Ability to translate the requirements specification into coverage plan is highly dependent on the experiences and depth of specification understanding of the engineer at the moment of writing it. If its not captured in the coverage plan it's lost. Even the engineer who wrote the functional coverage plan may find it difficult to remember why exactly certain cross was defined after 6 months.

Now this can become real challenge during the evolution and maintenance of the functional coverage plan as the requirements specifications evolve. Engineer doing incremental updates may not have luxury of the time as the earlier one had. Unless the intent is executable the quality of the functional coverage will degrade over period of time.

Now if you are doing this design IP for only one chip and after that if you are throwing it away this functional coverage quality degradation may not be such a big concern.

Let's understand this little further with example. In the USB Power delivery case study you will be able to find more details about this example. USB power delivery supports multiple specification revisions. Let's say, we want to cover all transmitted packets for revision x.

In manual approach we will discretely list protocol data units valid for revision x. For this listing you scan the specifications, identify them and list them. Only way to identify them in code as belonging to revision x is either through covergroup name or comment in the code.

In the new approach you will be able to operate on all the protocol data units supported by revision x as a unit through APIs. This is much more meaningful to readers and makes your intent executable. As we called out, our idea is to make coverage intent executable to make it adaptable. Let's contrast both approaches with another example.

For example, let's say you want to cover two items:

- All packet transmitted by device supporting revision 2.0
- Intermediate reset while all packet transmitted by device supporting revision 2.0

If you were to write discrete coverage, you would have sampled packet type and listed all the valid packet types of revision 2.0 as bins. Since bins are not reusable in SystemVerilog you would do copy and paste them across these two covergorups.

Now imagine, if you missed a packet type during initial specification scan or errata containing one more packet type came out later, you need to go back and add this new type at two different places.

But with this new approach, as soon as you update the specification data structure with new type you are done. All the queries requesting revision x will automatically get updated information. Hence all the functional coverage targeted to revision x will be automatically updated.

Remember initially it may be easy to spot two places where the change is required. But when you have hundreds of covergroups it will be difficult to reflect the incremental changes to all the discrete covergroups. It will be even more difficult when new engineer has to do the update without sufficient background on the initial implementation.

4 Benefits

What are the benefits of this approach?

With high-level specification model based functional coverage the abstraction of thought process of writing coverage moves up and it frees up brain bandwidth to

identify more items. This additional brain bandwidth can significantly help improve the quality of functional coverage plan and hence the overall quality of functional verification.

Benefits of high-level model based functional coverage generation:

- Intent gets captured in executable form. Makes it easy to maintain, update and review the functional coverage
- Executable intent makes your coverage truly traceable to specification. Its much better than just including the specification section numbers which leads to more overhead than benefit
- Its easy to map the coverage from single specification from different components points of view (Ex: USB device or host point of view or PCIe root complex or endpoint or USB Power delivery source or sink point of view) from single specification model
- Easy to define and control the quality of coverage controlled by the level of details in the coverage required for each feature (Ex: Cover any category, cover all categories or cover all items in each category)
- Easy to support and maintain multiple versions of the specifications
- Dynamically switch the view of the coverage implemented based on the parameters to ease the analysis (Ex: Per speed, per revision or for specific mode)

5 Architecture

How to go about building high-level specification model based functional coverage?

First let's understand the major components. Following is the block diagram of the high-level specification model based functional coverage. We will briefly describe role and functionality of each of these blocks. This diagram only shows basic building blocks.

Later we will look at the case studies where we will see these blocks in action making their explanations more clear. It will also guide how to implement these blocks for your project as well.

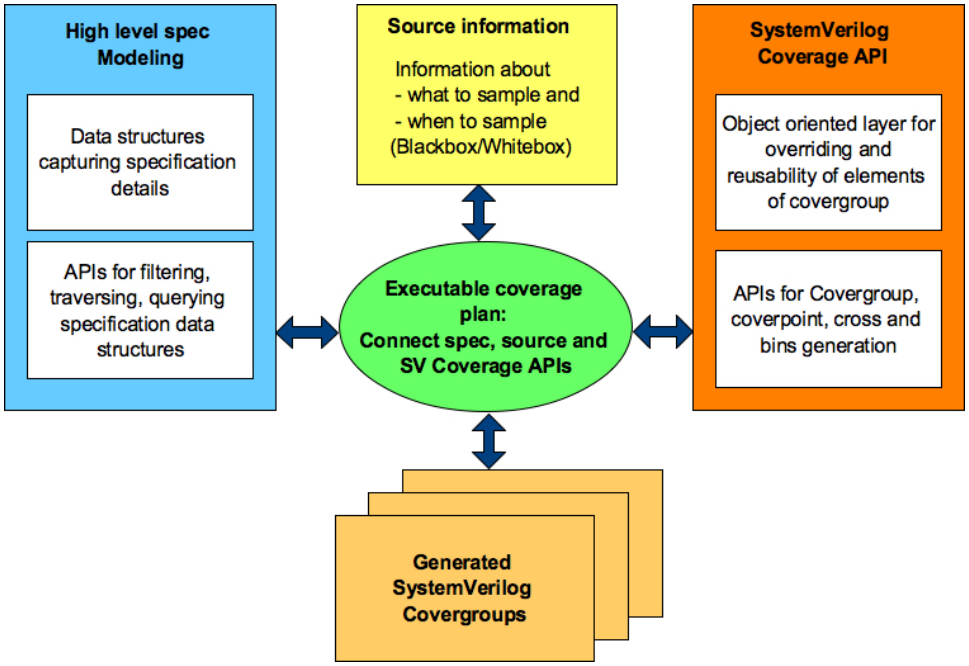


Figure 2: *Block diagram of high-level specification model based functional coverage generation*

5.1 Executable coverage plan

Executable coverage plan is the block that actually hosts all the functional coverage items. It's coverage plan and its implementation together.

It does the implementation of functional coverage items by connecting the high-level specification model, source of information and SV coverage APIs. The APIs utilized, specification information accessed and relations of various items utilized preserves the intent in executable form.

User still specifies the intent of what to cover.

It won't read your mind but you will be able to express your thoughts at higher level of abstractions and more closer or specifications and in highly programmable environment that is much more powerful that SystemVerilog alone.

5.2 High-level specification modeling

This block is combination of set of data structures and APIs.

Data structures capture high-level information from the specifications. These data structures can be capturing information about properties of different operations, state transition tables representing the state machines, information

about timers as to when they start, stop, timeout or graphs capturing various forms of sequences. Idea here is capture the relevant information about the specification that is required for the definition and implementation of the functional coverage. Choose the right form of data structures that fit the purpose. These data structures will vary from domain to domain.

APIs on the other hand process the data structures to generate different views of the information. APIs can be doing filtering, combinations, permutations or just ease access to the information by hiding the complexity of data structures. There is some level of reuse possible for these APIs across various domains.

Using these set of data structures and APIs now we are ready to translate the coverage plan to implementation.

5.3 Information source

Specification data structures may define the structure of operations but to cover it, we need to know how to identify the completion of operation, what is the type operation of operation completed and current values of its properties etc.

Information source provides the abstraction to bind the specification information to either test bench or design RTL to extract the actual values of these specification structures. This abstraction provides the flexibility to easily switch the source of coverage information.

Bottom line stores information about sources that are either sampled for information or provides triggers to help decide when to sample.

5.4 SystemVerilog Coverage API in Python

Why do we need these APIs, why can't we just directly write it in SystemVerilog itself?

That's because SystemVerilog covergroup has some limitations, which prevent the ease of reuse.

5.4.1 Limitations of SystemVerilog Covergroup

SystemVerilog functional covergroup construct has some limitations, which prevents its effective reuse. Some of the key limitations are following:

- Covergroup construct is not completely object oriented. It does not support inheritance. What it means is you cannot write a covergroup in base class and add, update or modify its behavior through derived class. This type of feature is very important when you want to share common functional coverage models across multiple configurations of DUT verified in different test benches and to share the common functional coverage knowledge

- Without right bins definitions the coverpoints don't do much useful job. The bins part of the coverpoint construct cannot be reused across multiple coverpoints either within the same covergroup or in different covergroup
- Key configurations are defined as crosses. In some cases you would like to see different scenarios taking place in all key configurations. But there is no clean way to reuse the crosses across covergroups
- Transition bin of coverpoints to get hit are expected to complete defined sequence on successive sampling events. There is no [!:$] type of support where the transition at any point is considered as acceptable. This makes transition bin implementation difficult on relaxed sequences

5.4.2 Coverage API Layering

At VerifSudha, we have implemented a Python layer that makes the SystemVerilog covergroup construct object oriented and addresses all of the above limitations to make the coverage writing process more productive. Also the power of python language itself opens up lot more configurability and programmability.

Based on this reusable coverage foundation we have also built many reusable high level coverage models bundled which make the coverage writing easier and faster. Great part is you can build library of high-level coverage models based on best-known verification practices of your organization.

These APIs allows highly programmable and configurable SystemVerilog functional coverage code generation.

Fundamental idea behind all these APIs is very simple.

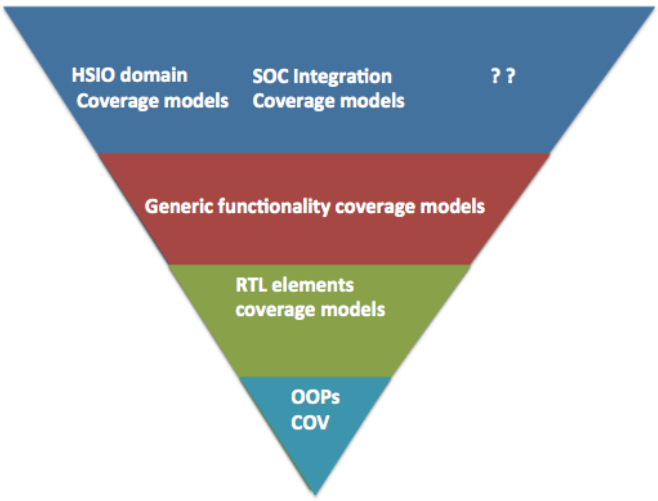


Figure 3: *SV Coverage API layering*

We have implemented these APIs as multiple layers in python.

Bottom most layer is basic python wrappers through which you can generate the functional coverage along with the support for object orientation. This provides the foundation for building easy to reuse and customize high-level functional coverage models. This is sufficient for the current case study.

RTL elements coverage models cover various standard RTL logic elements from simple expressions, CDC, interrupts to APPs for the standard RTL element such as FIFOs, arbiters, register interfaces, low power logic, clocks, sidebands.

Generic functionality coverage models are structured around some of the standard high-level logic structures. For example did interrupt trigger when it was masked for all possible interrupts before aggregation. Some times this type of coverage may not be clear from the code coverage. Some of these are also based on the typical bugs found in different standard logic structures.

At highest-level are domain specific overage model. For example many high-speed serial IOs have some common problems being solved especially at physical and link layers. These coverage models attempt to model those common features.

All these coverage models are easy to extend and customize as they are built on object oriented paradigm. That's the only reason they are useful. If they were not easy to extend and customize they would have been almost useless.

5.4.3 Implementation

- Backbone of these APIs is data structure for the SystemVerilog covergroups modeled as list of dictionaries. Each of the covergroup being a dictionary made up of list of coverpoint dictionaries and list of cross dictionaries. Each of the coverpoint and cross dictionaries contain list of bin dictionaries
- These data structures are combined with simple template design pattern to generate the final coverage code
- Using layer of APIs on these data structure additional features and limitations of SystemVerilog covergroup are addressed
- Set of APIs provided to generate the reusable bin types. For example if you want to divide an address range between N equal parts, you can do it through these APIs by just providing the start address, end address and number of ranges
- There are also bunch of object types representing generic coverage models. By defining the required properties for these object types covergroups can be generated
- Using python context managers the covegroup modeling is eased off for the user

Any user defined SystemVerilog code can co-exist with these APIs. This enables easy mix of generated and manually written code where APIs fall short.

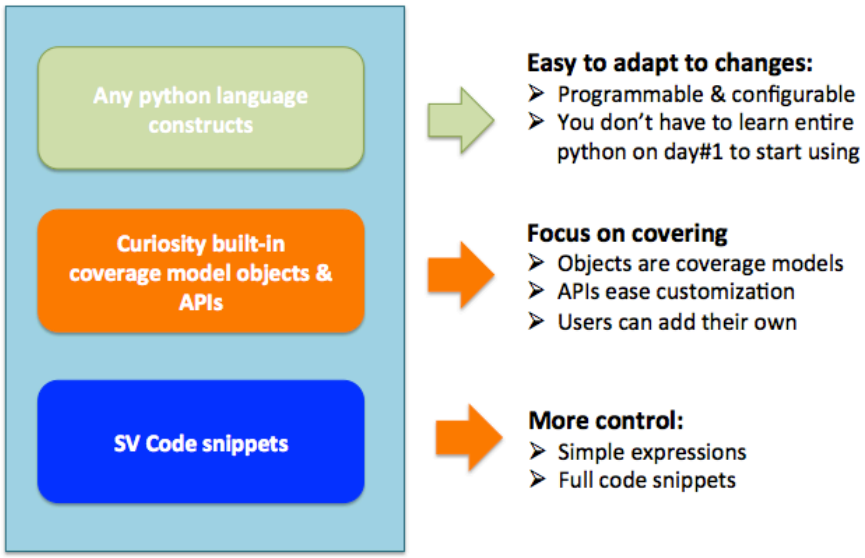


Figure 4: *What to expect from APIs*

5.4.4 Structure of user interface

All the APIs essentially work on the object. Global attributes can be thought of as applicable to entire covergroup. For example if you specified bins at the global level it would apply to all the coverpoints of the covergroup. Not only the information required for coverage generation but also description and tracking information can be stored in the corresponding object.

This additional information can be back annotated to simulator generated coverage results helping you correlate your high-level python descriptions to final coverage results from regressions easily.

Also the APIs support mindmaps and Excel file generations to make it easy to visualize the coverage plan for reviews.

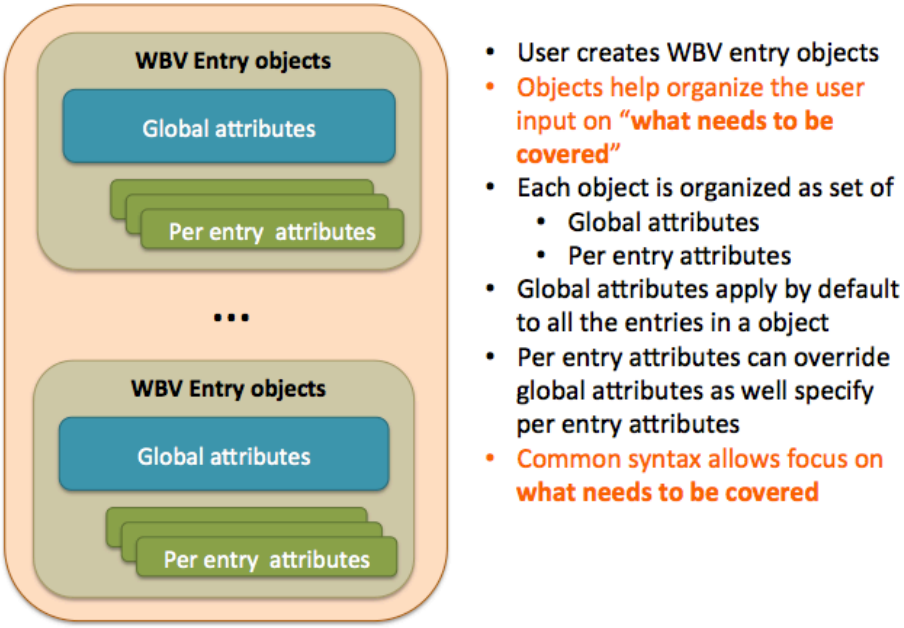


Figure 5: *Structure of user interface for objects*

5.5 Source information

Covergroups require what to sample and when to sample.

This is the block where you capture the sources of information for what to sample and when to sample. It's based on very simple concept like Verilog macros. All the coverage implementation will use these macros, so that it abstracts the coverage from statically binding to source of the information.

Later these macros can be initialized with the appropriate source information.

```
__USB_PD_RX_HARD_RESET_PULSE__              = 'tb_top.dut_top.prtocol_layer.rx.phy_hard_rst_rcvd',
__USB_PD_TX_HARD_RESET_PULSE__              = 'tb_top.dut_top.protocol_layer.tx.phy_hard_rst_sent',
__USB_PD_PRL_SUCCESSFULLY_MSG_SENT__        = 'tb_top.dut_top.policy_engine_layer.tx_success',
__USB_PD_DISCARD__                          = 'tb_top.dut_top.policy_engine_layer.tx_discard',
__USB_PD_PRL_TX_FAIL__                      = 'tb_top.dut_top.policy_engine_layer.tx_fail',
```

Snippet 1: *Specifying source information*

This flexibility allows using information source from either between the RTL and test bench. Easily be able to switch between them based on need.

Following code snippets showcase how covergroup implementation for simple read/write and address can be done using either RTL design or test bench transactions.

```
covergroup  bb_reg_rd_wr_cg;
  cp_rd_wr : coverpooint reg_rd_wr_tr_obj.cmd_type {
             bins b_read = {1};
             bins b_write = {0};
       }
  cp_addr: coverpoint reg_rd_wr_tr_obj.cmd_addr  {
             bins b_addr[] = {0, 1, 2};
       }
endgroup
```

Snippet 2: *Coverage generated using testbench transaction*

Coverpoints in snippet 2 are sampling the register read write transaction object (reg_rd_wr_tr_obj). Sampling is called on every new transaction

```
covergroup  wb_reg_rd_wr_cg @(dut.clk);
   cp_rd_wr : coverpooint dut.reg.valid_rd_wr
                          iff(!dut.rstn) {
              bins b_read  = {1};
              bins b_write = {0};
       }
      cp_addr: coverpoint dut.reg.addr
                     iff(dut.reg.valid) {
              bins b_addr[] = {0, 1, 2};
       }
endgroup
```

Snippet 3: *Coverage generated using DUT signals*

Coverpoints in snippet 3 are sampling the RTL signals to extract the read/write operation and address. Sampling is called on every new clock qualified by appropriate signals.

6 Summary:

Functional coverage is one of the last lines of defense for verification quality. Being able to repeatedly do a good job and do it productively will have significant impact on your quality of verification.

Initially it may seem like lot of work, you need to learn a scripting language and learn different techniques of modeling. But pay off will not only for the current project but throughout the lifetime of your project by easing the maintenance and allowing you to deliver the higher quality consistently.

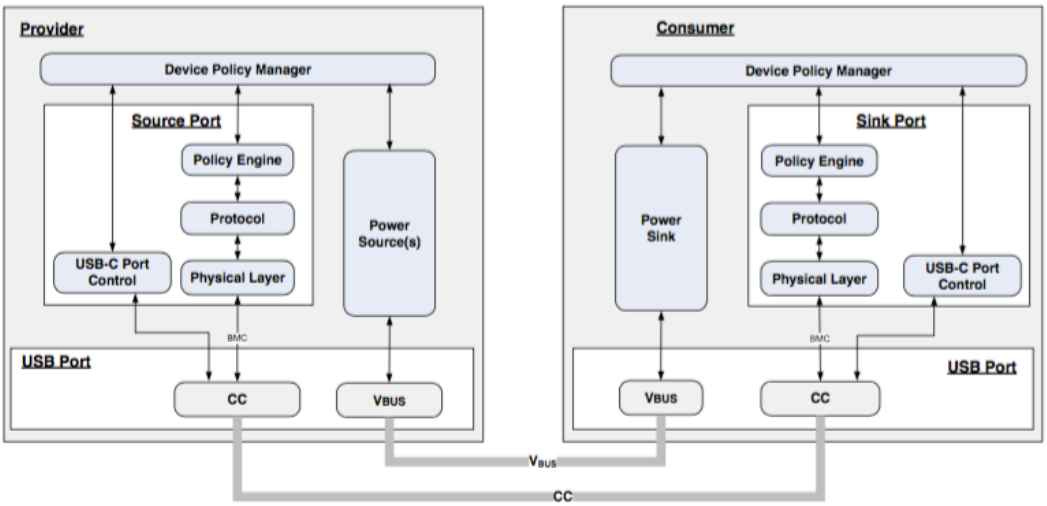
7 USB Power delivery protocol layer – Case study

Before we jump into implementation details, here is a very short refresher with only limited the details relevant for this case study.

7.1 Refresher

Dynamic negotiable power is basic idea behind the USB –power delivery. Obviously for negotiation to take place we need two entities. They are source and sink.

Figure 2-5 High Level Architecture View



Provider (SOURCE) supplies power and Consumer (SINK), uses the supplied power. A simple provider could be wall outlet or laptop and simple sink could be mobile or tablet.

Power negotiated can be up to 100Watts. The cables have to be designed to support that. Cables are electronically marked as well to specify their capabilities.

Refined USB connector called type-c is gaining popularity, used for the USB power delivery support. There two additional wires added in the connector to support the protocol.

On these two additional wires USB power delivery runs a protocol stack made up of following three layers:

- Physical layer
- Protocol layer
- Policy engine.

We will look in to briefly protocol layer only.

Now protocol itself is very simple. It's half duplex. Half, yes, that's right.

It's made up of bunch of messages. There can be only one outstanding message between the pair of communicating devices. Every message has to be acknowledged for successful reception. Only after that next message is transmitted. There are three categories of messages. They are CONTROL, DATA and EXTENDED.

Table 6-5 Control Message Types

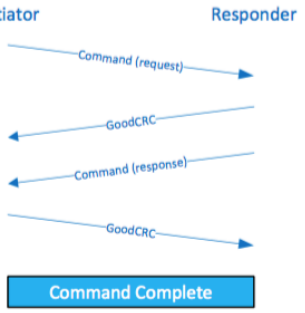
Bits 4...0	Message Type	Sent by	Description	Valid Start of Packet
0 0000	***Reserved***	N/A	All values not explicitly defined are ***Reserved*** and ***Shall Not*** be used.	
0 0001	*GoodCRC*	Source, Sink or Cable Plug	See Section 6.3.1.	SOP*
0 0010	*GotoMin*	Source only	See Section 6.3.2.	SOP only
0 0011	*Accept*	Source, Sink or Cable Plug	See Section 6.3.3.	SOP*
0 0100	*Reject*	Source or Sink	See Section 6.3.4.	SOP only
0 0101	*Ping*	Source only	See Section 6.3.5.	SOP only
0 0110	*PS_RDY*	Source or Sink	See Section 6.3.6.	SOP only
0 0111	*Get_Source_Cap*	Sink or DRP	See Section 6.3.7.	SOP only
0 1000	*Get_Sink_Cap*	Source or DRP	See Section 6.3.8.	SOP only
0 1001	*DR_Swap*	Source or Sink	See Section 6.3.9	SOP only

Let's look at the simple protocol sequence.

In very simple terms, transmitter sends request, wait for acknowledgement. Receiver sends acknowledgement.

Followed by that receiver sends the response and waits for the acknowledgement from request transmitter. That's it.

Figure 6-25 Command request/response sequence



Different timers govern maximum wait time for the various acknowledgement and responses.

Table 6-54 Timers

Timer	Parameter	Used By	Reference
BISTContModeTimer	*tBISTContMode*	Policy Engine	Section 6.6.7.2
ChunkingNotSupportedTimer	*tChunkingNotSupported*	Policy Engine	Section 6.6.17.1
ChunkSenderRequestTimer	*tChunkSenderRequest*	Protocol	Section 6.6.17.2
ChunkSenderResponseTimer	*tChunkSenderResponse*	Protocol	Section 6.6.17.3
CRCReceiveTimer	*tReceive*	Protocol	Section 6.6.1
DiscoverIdentityTimer	*tDiscoverIdentity*	Policy Engine	Section 6.6.14
HardResetCompleteTimer	*tHardResetComplete*	Protocol	Section 6.6.9
NoResponseTimer	*tNoResponse*	Policy Engine	Section 6.6.6
PSHardResetTimer	*tPSHardReset*	Policy Engine	Section 6.6.10.2
PSSourceOffTimer	*tPSSourceOff*	Policy Engine	Section 6.6.5.2

7.2 Protocol layer functional coverage challenges

There are three challenges here:

- Multiple specification revision support (Rev 2.0 and Rev 3.0)
- Protocol has multiple participants: Source, Sink, Cable plug, Source/Sink (DRP)

There are significant differences between revision 2.0 and 3.0. EXTENDED category of messages is added only in revision 3.0.

When you write coverage model SOURCE and SINK are like mirror images. What is covered as transmit for one needs to covered as receive for the other.

7.3 Functional coverage items for case study

Now let's pick some items to cover and see how they would look like in high-level modeling based functional coverage.

Picking rainbow of simple atomic items, simple sequence and complex sequences:

Sl. No.	Coverage item description
1	All transmitted protocol messages
2	All received protocol messages
3	Some key events during message transmission
4	For all transmitted messages getting all possible valid responses
5	For all transmitted messages unexpected acknowledgement received
6	Power negotiation sequence
7	Timeout error injection within power negotiation sequence

Table 1: *Selected functional coverage items for case study*

For all these items let's demonstrate how all the benefits of high-level model based functional coverage promised can be achieved.

Now let's jump into details of implementation starting with the specification modeling of the USB power delivery first.

7.4 USB power delivery specification information modeling

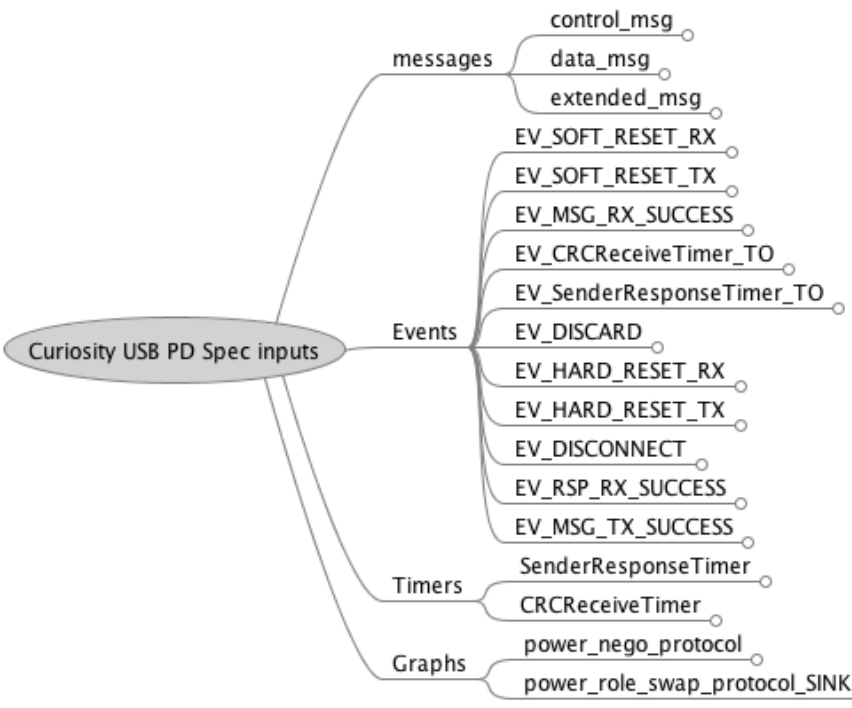
First thing we need to do is model the specification details.

We are showcasing limited items modeling here to balance between the details and ease of demonstrating concepts.

We will focus only on the details of the specification required for implementing the items we have picked in section 7.3.

Here is how we will go about describing the implementation of specification modeling. We will first show some mindmaps for high-level view of information captured and then show the corresponding implementation in python for reference.

High-level specification information modeled looks as following.



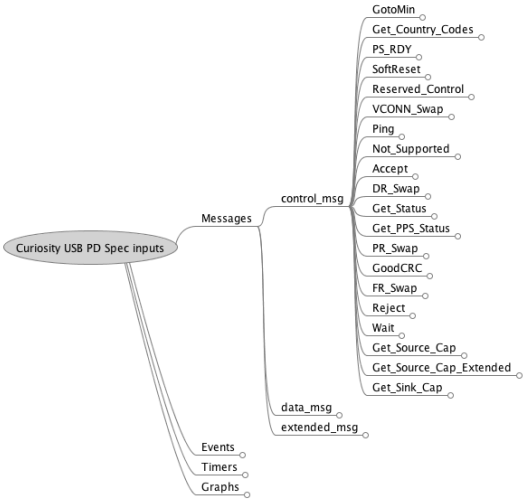
Mindmap 1: *Top level view of high-level specification information items captured*

We have captured the all the three categories of messages, some important events, information about timers and protocol sequences are captured as graphs. We will keep going deeper in each of these nodes to see how they are implemented.

7.4.1 Message modeling

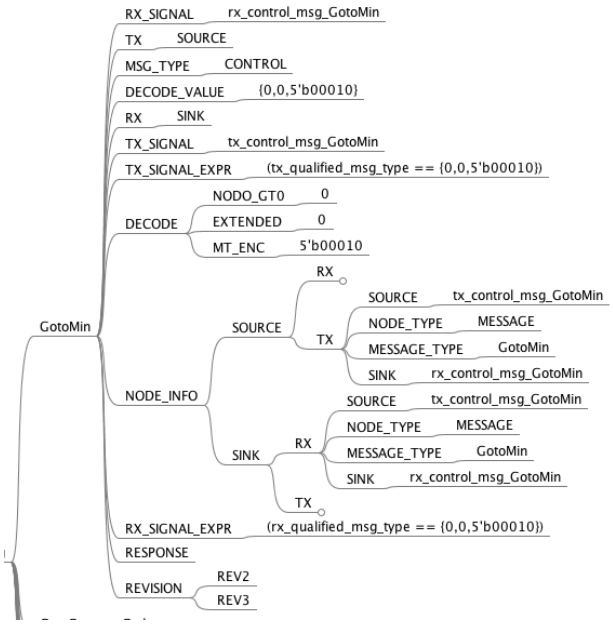
Let's go first level deep in to the message modeling. There are three categories of messages modeled: control, data and extended.

Each category is modeled separately to maintain the classifications even in the generated code. For example when we need to cover all the message types we can have either single coverpoint for all messages or three coverpoint covering each category of messages. Later is easy when we analyze the results. Although it may be additional effort to write it but let's not forget we write once but analyze results multiple times.



Mindmap 2: *All three types of message types and Control messages types expanded*

Lets expand one level further into one of control message type called GotoMin. Not all the information shown below is manually filled. Some of it is manually filled and some of it is automatically generated.



Mindmap 3: *GotoMin messages types expanded with different fields*

For example, for each message types decoding information is manually filled in "DECODE" but the TX_SIGNAL_EXPR and RX_SIGNAL_EXPR, which indicate how to decode the each message type, are created automatically for all the message types. Since it's used across multiple coverage items instead of creating every time, it's created once stored in the data structure.

Now let's get into details of how to implement in python. It's a simple dictionary.

Every message type is key. It's pointing to another dictionary containing following key-value pairs. This information has to be filled manually referring to the specifications.

Key	Description about value stored
DECODE	Show to how to decode this message type
TX	List of which DUT types can transmit this type of message
RX	List of which DUT types can receive this type of message
RESPONSE	If its request type, list of all possible legal responses
REVISION	List of specification revisions which support this message type

Table 2: *Fields of the basic message field modeling*

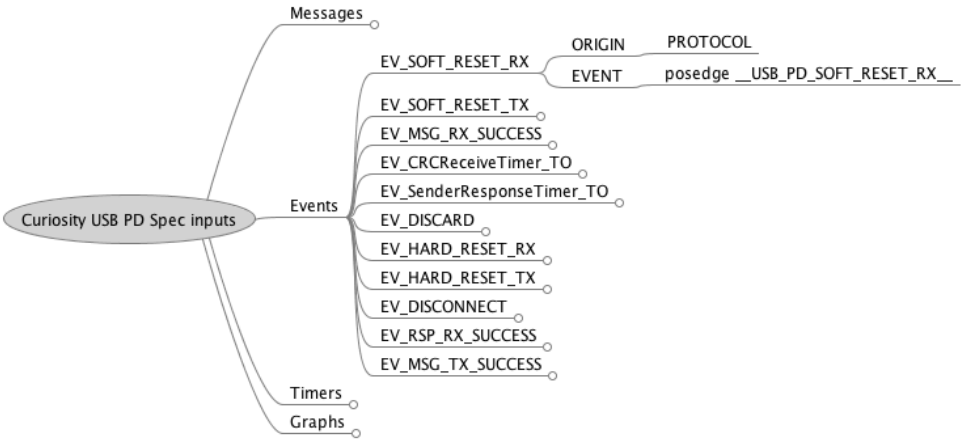
```
# Setup USB PD specification information
# Process it to create any additional information for ease of the coverage or analytics generation
def set_usb_pd_spec_msg_type_details(self):

    self.control_msg_info_dict = {
     "GoodCRC"                  : {
          'DECODE'   : { 'MT_ENC': "5'b00001", 'NODO_GT0' : '0', 'EXTENDED':'0'},
          'TX'       : [ 'SOURCE', 'SINK', 'CABLE_PLUG'],
          'RX'       : [ 'SOURCE', 'SINK', 'CABLE_PLUG'],
          'RESPONSE' : [],
          'REVISION' : [ 'REV2', 'REV3'],
      },

     "GotoMin"                  : {
          'DECODE'   : { 'MT_ENC': "5'b00010", 'NODO_GT0' : '0', 'EXTENDED':'0'},
          'TX'       : [ 'SOURCE'] ,
          'RX'       : [ 'SINK'] ,
          'RESPONSE' : [ ],
          'REVISION' : [ 'REV2', 'REV3'],
      },

     "Accept"                   : {
          'DECODE'   : { 'MT_ENC': "5'b00011", 'NODO_GT0' : '0', 'EXTENDED':'0'},
          'TX'       : [ 'SOURCE', 'SINK', 'CABLE_PLUG'] ,
```

Snippet 4: *Python modeling for the messages*

7.4.2 Event modeling

All specifications identify set of key events. It's interesting to cover, these key events taking place at different states.

Let's look at the events. Here is high-level view of some of the events captured.



Mindmap 4: *Events implementation expanded*

It's another simple dictionary. Keys being user defined event names. It points to another dictionary containing the information about events.

Key	Description about value stored
ORIGIN	Layer from which event is emitted
EVENT	Actual event. Think of those weird looking string names starting and ending with double underscores like Verilog macro

Table 3: *Fields of the basic event field modeling*

```
# Information about events
# - Can be statically initialized by the user
# - Derived from other information and updated in to the dictionary
def set_usb_pd_spec_event_details(self):
    self.event_info_dict = {
        'EV_MSG_TX_SUCCESS'           : {'ORIGIN' : 'PROTOCOL', 'EVENT' : 'posedge __USB_PD_MSG_TX_SUCCESS__'},
        'EV_MSG_RX_SUCCESS'           : {'ORIGIN' : 'PROTOCOL', 'EVENT' : 'posedge __USB_PD_MSG_RX_SUCCESS__'},

        'EV_RSP_RX_SUCCESS'           : {'ORIGIN' : 'PROTOCOL', 'EVENT' : 'posedge __USB_PD_RSP_RX_SUCCESS__'},

        'EV_HARD_RESET_TX'            : {'ORIGIN' : 'PROTOCOL', 'EVENT' : 'posedge __USB_PD_HARD_RESET_TX__'},
        'EV_HARD_RESET_RX'            : {'ORIGIN' : 'PROTOCOL', 'EVENT' : 'posedge __USB_PD_HARD_RESET_RX__'},

        'EV_SOFT_RESET_TX'            : {'ORIGIN' : 'PROTOCOL', 'EVENT' : 'posedge __USB_PD_SOFT_RESET_TX__'},
        'EV_SOFT_RESET_RX'            : {'ORIGIN' : 'PROTOCOL', 'EVENT' : 'posedge __USB_PD_SOFT_RESET_RX__'},

        'EV_DISCONNECT'               : {'ORIGIN' : 'PHY',      'EVENT' : 'posedge __USB_PD_DISCONNECT__'},
        'EV_DISCARD'                  : {'ORIGIN' : 'PHY',      'EVENT' : 'posedge __USB_PD_DISCARD__'},
```

Snippet 5: *Python modeling for the events*

Those events (EVENT field) can be pointed to RTL design signals or test bench events.

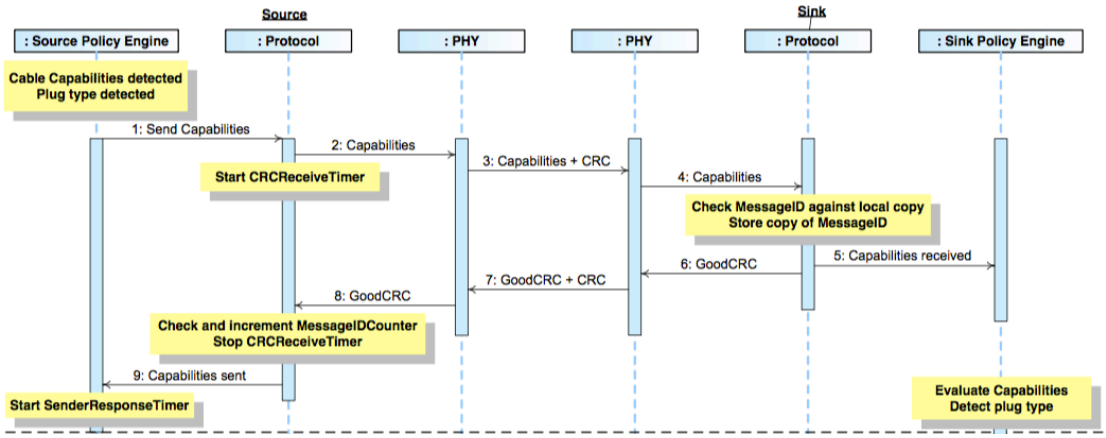
7.4.3 Protocol sequence modeling

Here comes a juicy part.

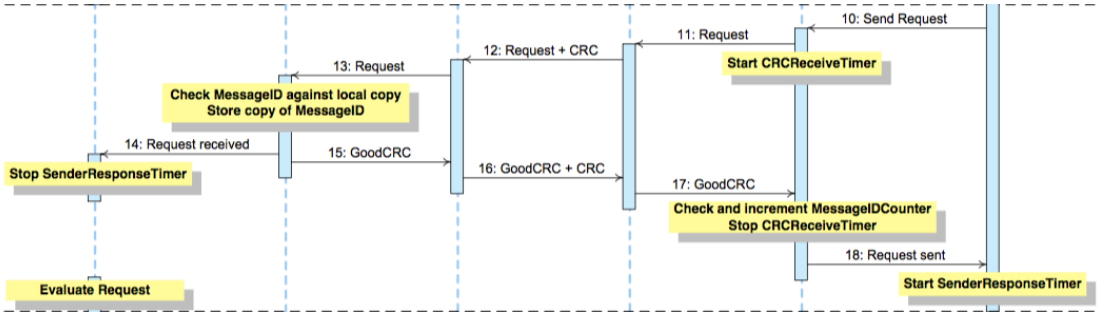
Simple tree data structure is used to represent the protocol sequence as a graph. Before we can get to that, let's briefly familiarize ourselves with the power negotiation sequence.

Following diagram from specification shows the steps of power negotiation sequence.

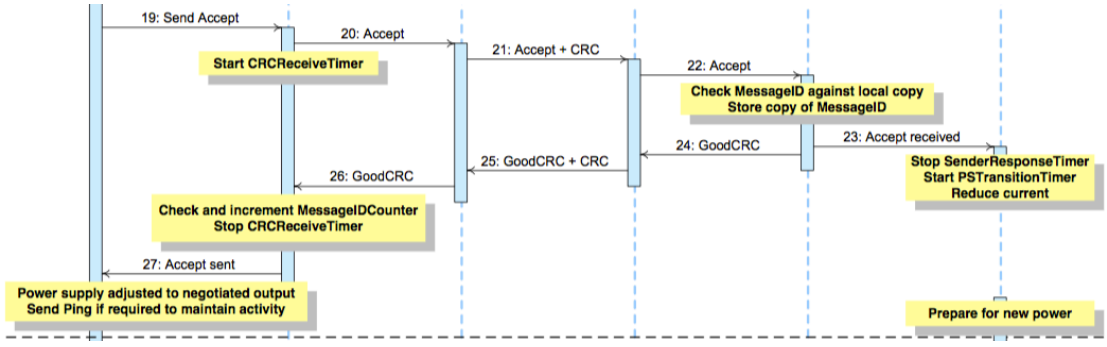
Step1: SOURCE sends its capabilities to SINK



Step 2: SINK selects the power offering that suits it



Step 3: SOURCE accepts the SINK request for specific offering if it likes it



Step 4: SOURCE sends new power offering is ready. Deal is done

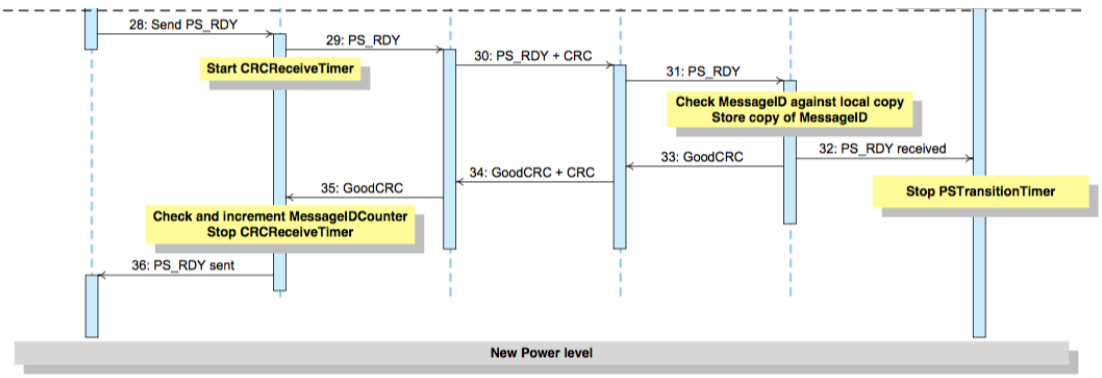


Figure: *4 steps successful power negotiation sequence*

At Step3 above shows Accept scenario, there are multiple possible responses. If we go back to our specification-modeling table, we can see, RESPONSE indicates multiple possibilities to this sequence. Above sequence shows only one of them. But to completely model it we need to capture all the possibilities and generate the functional coverage for all possible sequences.

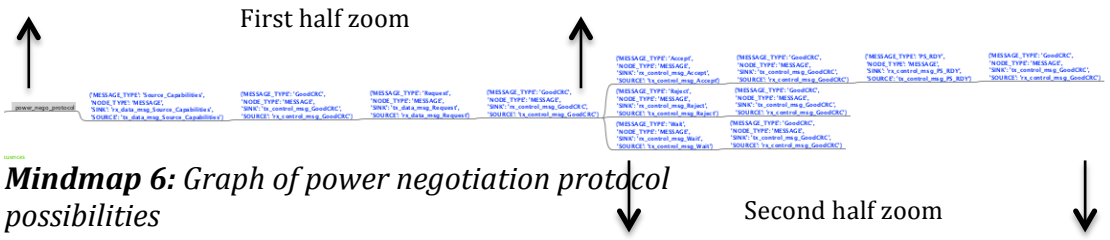
```
        "Request"                   : {
            'DECODE'    : { 'MT_ENC': "5'b00010", 'NODO_GT0' : '1', 'EXTENDED':'0',
                            'MAX_NODO' : 1, },
            'TX'        : [ 'SINK'],
            'RX'        : [ 'SOURCE'],
            'RESPONSE'  : [ 'Accept', 'Reject', 'Wait', 'Not_Supported'],
            'REVISION'  : [ 'REV2', 'REV3'],
        },
```

Snippet 6: *Python modeling for all possible RESPONSE to Request message*

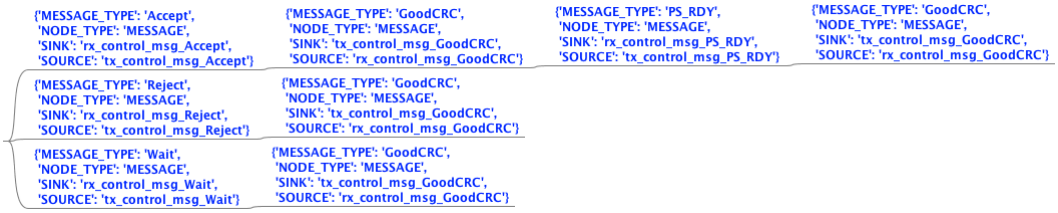
Lets look at how the simple power negotiation sequence with all these possibilities can be captured in the tree data structure.

This is how the tree of the protocol sequences would look like visualized in Mindmap. Look at the ordering of MESSAGE_TYPE fields in the node first. The one in middle (Mindmap 6) is the complete diagram. The one in top (Mindmap 5) in zoom of first half (2/4 steps) and one in bottom (Mindmap 7) is the zoom of second half(4/4 steps) of the graph.

Mindmap 5: *Zoom of first half of power negotiation sequence (See below for full graph)*



Mindmap 6: *Graph of power negotiation protocol possibilities*

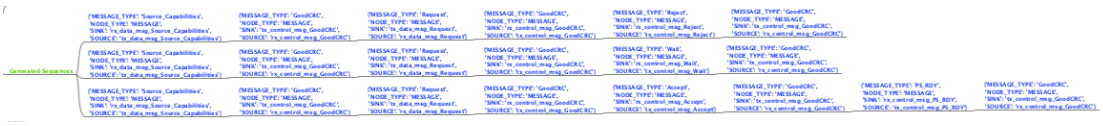


***Mindmap 7**: Zoom of second half of power negotiation sequence (see above for full graph)*

Complete graph may not be clearly visible. Use that only for getting the big picture. For details, I have zoomed them in two parts. First part is from start to the node where it branches into three parts (first two steps). Second one contains the details about all the three possible options.

Now if we expand this graph into sequences it will results in three unique sequences with the first two steps (Step1, Step2 from Figure 4)) being common.

Just catch the visual view with the following picture. I will explain bit of details below.



***Mindmap 8**: Possible 3 sequences from the power negotiation graph*

Three sequences possible are:

- **Sequence 1:** Source Capabilities => GoodCRC => Request => GoodCRC => Reject => GoodCRC (6 nodes)
- **Sequence 2:** Source Capabilities => GoodCRC => Request => GoodCRC => Wait => GoodCRC (6 nodes)
- **Sequence 3:** Source Capabilities => GoodCRC => Request => GoodCRC => Accept => GoodCRC => PS_RDY => GoodCRC (8 nodes)

Implementation of these sequences requires DUT type (SOURCE, SINK) to be defined. Based on the DUT type the direction of each of these messages gets decided. Using the DUT type appropriate event is picked from the node information.

Key	Description about value stored
NODE_TYPE	Type of node: It can be MESSAGE, EVENT
MESSAGE_TYPE	If the node type is MESSAGE what is the type of MESSAGE
SINK	For SINK type of device, the event to be used
SOURCE	For SOURCE type of device, the event to be used

***Table 4:** Fields used in the nodes of the graphs*

Ordering of steps of sequence is extracted from graphs and event information for implementing each step is extracted from the node information for functional coverage sequences generation.

Interesting part of these graphs is not just above sequence generation, but expanding these graphs to cover many more cases algorithmically.

For example we saw that every message transmitted requires a confirmation message called GoodCRC. Now we know wherever there is wait there is generally a timer associated to prevent infinite waiting. For GoodCRC there is CRCReceiveTimer.

For important sequences we can automatically add a timeout node wherever there is GoodCRC message is involved. This allows controlling the quality of verification on specific sequences by controlling the verbosity of coverage generated. Its demonstrated in section 7.5.8.

7.5 Executable coverage plan

Now that we have captured all the specification information, let's jump into the how to use the specification model for functional coverage generation. We will implement all the 7 coverage items selected in Section 7.3.

In early items, we will put in little bit more explanation. But as we go along, we will reduce it as you start getting familiar.

For each item we will briefly explain what the item is suppose to cover, then show the input python code snippet for coverage item implementation and generated functional coverage. We will briefly explain the APIs used in the input and point to anything specific to be observed in the covergroups.

Idea here is not to explain everything detail of the code but at the same time stay grounded to give a flavor of code to make concepts concrete by showing how you can go about implementations.

Covergroups and coverpoints need additionally need what to sample and when to sample that needs to be provided as separate input. This can either be tapped from test bench variables or from RTL. This part is skipped as we have already explained it in section 5.5.

7.5.1 Global configuration

To keep it simple the coverage model is made configurable at global level across all covergroups for some parameters but it can be done easily at per item level as well.

For all the example items we are using following configuration. What it means?

Covergroups will be generated for

- DUT supporting only Revision 2.0 of specification (Selected this because number of messages are lesser and its easy show snippets)
- DUT Type of SINK
- Quality of coverage set to highest level (This concept will be demonstrated in the section 7.5.4)

```
# Setup the type of coverage you need
# Which Revisions you care
# Which type of DUT are you verifying: SOURCE, SINK, DRP, CABLE_PLUG
# What is the quality level desired:
# - Overall (Level1: Basic, Level2: Medium, Level 3: Comprehensive)
# - Per feature level
curiosity_usb_pd_hlm_cov_obj.set_coverage_requirements(
   DUT_REV_SUPPORT       = ['REV2'],
   DUT_TYPE              = [ 'SINK'],
   GLOBAL_QUALITY_LEVEL  = 'LEVEL3',
)
```

Snippet 7: *Global configuration*

7.5.2 Item 1: Cover all transmitted protocol messages

Here we want to cover all transmitted messages types possible across all three categories for specified DUT_TYPE and DUT_REV_SUPPORT.

Input: Python code for coverage item

```
# All message types
with add_wbv_entry(self,
    WBV_NAME            = 'basic_all_msg_types_tx',
    WBV_CLOCK           = self.event_info_dict['EV_MSG_TX_SUCCESS']['EVENT'],
    WBV_TYPE            = 'WBV_SIGNAL_VALUE',
    WBV_FUNCTIONAL_COV  = '1',
    WBV_STATUS          = 'DONE',
    WBV_CATEGORY        = 'Protocol',
    WBV_DESCRIPTION     = 'COV: All control/data/extended message types transmitted to peer and cableplug') as wbv_entry:

    bins_per_category_dict = self.get_bins_for_matching_msgs(["CONTROL", "DATA", "EXTENDED"], "TX")
    self.add_cp_for_msg_types(wbv_entry, 'tx_qualified_msg_type', 'to_peer', bins_per_category_dict)
```

Snippet 8: *Python implementation to cover all transmitted protocol messages*

So, What's happening?

The API **get_bins_for_matching_msgs** returns the bins per category of messages listed (argument: ["CONTROL", "DATA", "EXTENDED"]) that can be transmitted(argument: "TX") by DUT of the type SINK(Global setting : DUT_TYPE) and supporting the USB power delivery revision 2.0(Global setting DUT_REV_SUPPORT).

The API **add_cp_for_msg_types** crates and adds the coverpoints and bins to the covergroup. Along with bins per category, it takes in what to be sampled and when to be sampled information as well.

Both of these APIs internally use the object oriented functional and statistical coverage code generation APIs that are part of curiosity framework.

If you see the generated output, there is one covergroup with two coverpoints. There is coveporint each for CONTROL and DATA message categories. Encodings

and names of bins in the generated output are derived from the specification information captured in the snippet 4.

Output: Covergroup generated

```
covergroup basic_all_msg_types_tx() @(posedge usb_pd_msg_tx_success);
        option.comment = "COV: All control/data/extended message types transmitted to peer and cableplug";
        option.name = "basic_all_msg_types_tx";
        option.per_instance = 1;

        // Cover points
        cp_CONTROL_msg_types : coverpoint tx_qualified_msg_type iff (to_peer){

                bins b_PR_Swap = {{0,0,5'b01010}};
                bins b_SoftReset = {{0,0,5'b01101}};
                bins b_VCONN_Swap = {{0,0,5'b01011}};
                bins b_Accept = {{0,0,5'b00011}};
                bins b_DR_Swap = {{0,0,5'b01001}};
                bins b_PS_RDY = {{0,0,5'b00110}};
                bins b_GoodCRC = {{0,0,5'b00001}};
                bins b_Reject = {{0,0,5'b00100}};
                bins b_Get_Source_Cap = {{0,0,5'b00111}};
                bins b_Wait = {{0,0,5'b01100}};
        }

        cp_DATA_msg_types : coverpoint tx_qualified_msg_type iff (to_peer){

                bins b_BIST = {{0,1,5'b00011}};
                bins b_Request = {{0,1,5'b00010}};
                bins b_Vendor_Defined = {{0,1,5'b01000}};
                bins b_Sink_Capabilities = {{0,1,5'b00100}};
        }

        // Cross coverage

endgroup : basic_all_msg_types_tx
```

Snippet 9: *Generated SystemVerilog Covergroup for all transmitted protocol messages*

Did you notice? Message categories passed includes “CONTROL”, “DATA” and “EXTENDED”. But in the generated output we don’t see EXTENDED messages – Why?

That’s because the USB Power delivery revision 2.0 does not support the EXTENDED message types. If we switch the supported revision type to 3.0 then the EXTENDED message types will be added as another coverpoint.

Based on the DUT type and supported revisions the coverpoints and bins will automatically change for each message categories. Isn’t that something?

7.5.3 Item 2: All received protocol messages

This item selected to demonstrate how easy it is to switch direction in the coverage implementation from all transmitted to all received messages coverage.

Input: Python code for coverage item

```
with add_wbv_entry(self,
    WBV_NAME            = 'basic_all_msg_types_rx',
    WBV_CLOCK           = self.event_info_dict['EV_MSG_RX_SUCCESS']['EVENT'],
    WBV_TYPE            = 'WBV_SIGNAL_VALUE',
    WBV_FUNCTIONAL_COV  = '1',
    WBV_STATUS          = 'DONE',
    WBV_CATEGORY        = 'Protocol',
    WBV_DESCRIPTION     = 'COV: All control/data/extended message types received from peer and cableplug') as wbv_entry:

    bins_per_category_dict = self.get_bins_for_matching_msgs(["CONTROL", "DATA", "EXTENDED"], "RX")
    self.add_cp_for_msg_types(wbv_entry, 'rx_qualified_msg_type', 'to_peer', bins_per_category_dict)
```

Snippet 10: *Python implementation to cover all received protocol messages*

Everything is same as covering all transmitted message types, except direction sensitive information changed to receive.

When to sample changes to receive message event.

To API **get_bins_for_matching_msgs** we change the direction as RX

The API **add_cp_for_msg_types** changes the information as to be sampled to qualified received message.

Output: Covergroup generated

```
covergroup basic_all_msg_types_rx() @(posedge usb_pd_msg_rx_success);
        option.comment = "COV: All control/data/extended message types received from peer and cableplug";
        option.name = "basic_all_msg_types_rx";
        option.per_instance = 1;

        // Cover points
        cp_CONTROL_msg_types : coverpoint rx_qualified_msg_type iff (to_peer){

                bins b_GotoMin = {{0,0,5'b00010}};
                bins b_PS_RDY = {{0,0,5'b00110}};
                bins b_SoftReset = {{0,0,5'b01101}};
                bins b_VCONN_Swap = {{0,0,5'b01011}};
                bins b_Accept = {{0,0,5'b00011}};
                bins b_DR_Swap = {{0,0,5'b01001}};
                bins b_PR_Swap = {{0,0,5'b01010}};
                bins b_GoodCRC = {{0,0,5'b00001}};
                bins b_Reject = {{0,0,5'b00100}};
                bins b_Get_Sink_Cap = {{0,0,5'b01000}};
                bins b_Wait = {{0,0,5'b01100}};
        }

        cp_DATA_msg_types : coverpoint rx_qualified_msg_type iff (to_peer){

                bins b_BIST = {{0,1,5'b00011}};
                bins b_Source_Capabilities = {{0,1,5'b00001}};
                bins b_Vendor_Defined = {{0,1,5'b01000}};
        }

        // Cross coverage

endgroup : basic_all_msg_types_rx
```

Snippet 11: *Generated SystemVerilog Covergroup for all received protocol messages*

7.5.4 Item 3: Some key events during message transmission

In this item we want to cover some key events taking place while different messages types are being transmitted. Many of these events lead to some form abrupt termination of message being transmitted followed by recovery. It's interesting to see if DUT can recover correctly and operate normally again.

We want to introduce you to an interesting concept of quality of functional coverage with this item.

Basic idea is depth and width of functional coverage can be guided by the importance of particular feature to current project.

Now you can cover this item at three levels of details depending on importance of this feature for your application:

- LEVEL 3: For each of the message in all three categories of messages. Results in very comprehensive coverage

- LEVEL 2: Any one message in each category. Here we are relaxing the coverage but still covering each of category
- LEVEL 1: Any one message across all categories. Completely relaxed.

This type of configurability can also help easily change your mind while closing coverage. One could close on LEVEL 1 for initial milestone and then gradually reach LEVEL 3 based on schedule and priorities.

Input: Python code for coverage item

```
# Different key events while transmitting different message types
for event_str in ['EV_HARD_RESET_RX', 'EV_SOFT_RESET_RX', 'EV_DISCONNECT', 'EV_DISCARD']:

   wbv_name_str = event_str + '_' + 'message_types'
   with add_wbv_entry(self,
        WBV_NAME             =  wbv_name_str,
        WBV_TYPE             = 'WBV_SIGNAL_VALUE',
        WBV_FUNCTIONAL_COV   = '1',
        WBV_CATEGORY         = 'Phy/Prot Interaction',
        WBV_STATUS           = 'DONE',
        WBV_CLOCK            = self.event_info_dict[event_str]['EVENT'],
        WBV_DESCRIPTION      = 'COV: Key events messages types '+ event_str) as wbv_entry:

        if self.global_quality_level is "LEVEL3": # Bin per value
          bins_per_category_dict = self.get_bins_for_matching_msgs(["CONTROL", "DATA", "EXTENDED"], "TX")
          self.add_cp_for_msg_types(wbv_entry, 'tx_qualified_msg_type', 'to_peer', bins_per_category_dict)
        elif self.global_quality_level is "LEVEL2": # Bin per category
          bins_per_category_dict = self.get_bins_for_matching_msgs(["CONTROL", "DATA", "EXTENDED"], "TX", 0, [], "all_msg_types")
          self.add_cp_for_msg_types(wbv_entry, 'tx_qualified_msg_type', 'to_peer', bins_per_category_dict )
        else: # Single bin for all categories
          bins_per_category_dict = self.get_bins_for_matching_msgs(["ALL"], "TX", 0, [], "all_msg_types")
          self.add_cp_for_msg_types(wbv_entry, 'tx_qualified_msg_type', 'to_peer', bins_per_category_dict)
```

Snippet 12: *Python implementation to cover key events during message transmission*

Output: With quality LEVEL3

Highest quality. Cover all the messages in all the categories.

```
covergroup EV_DISCONNECT_message_types() @(posedge __USB_PD_DISCONNECT__);
        option.comment = "COV: Key events messages types EV_DISCONNECT";
        option.name = "EV_DISCONNECT_message_types";
        option.per_instance = 1;

        // Cover points
        cp_CONTROL_msg_types : coverpoint tx_qualified_msg_type iff (to_peer){

                bins b_PR_Swap = {{0,0,5'b01010}};
                bins b_SoftReset = {{0,0,5'b01101}};
                bins b_VCONN_Swap = {{0,0,5'b01011}};
                bins b_Accept = {{0,0,5'b00011}};
                bins b_DR_Swap = {{0,0,5'b01001}};
                bins b_PS_RDY = {{0,0,5'b00110}};
                bins b_GoodCRC = {{0,0,5'b00001}};
                bins b_Reject = {{0,0,5'b00100}};
                bins b_Get_Source_Cap = {{0,0,5'b00111}};
                bins b_Wait = {{0,0,5'b01100}};
        }

        cp_DATA_msg_types : coverpoint tx_qualified_msg_type iff (to_peer){

                bins b_BIST = {{0,1,5'b00011}};
                bins b_Request = {{0,1,5'b00010}};
                bins b_Vendor_Defined = {{0,1,5'b01000}};
                bins b_Sink_Capabilities = {{0,1,5'b00100}};
        }

        // Cross coverage

endgroup : EV_DISCONNECT_message_types
```

Snippet 13: *Generated SystemVerilog Covergroup for quality LEVEL3*

Medium quality. Cover at least one message per category.

```
covergroup EV_DISCONNECT_message_types() @(posedge __USB_PD_DISCONNECT__);
        option.comment = "COV: Key events messages types EV_DISCONNECT";
        option.name = "EV_DISCONNECT_message_types";
        option.per_instance = 1;

        // Cover points
        cp_CONTROL_msg_types : coverpoint tx_qualified_msg_type iff (to_peer){

                bins b_all_msg_types = {{0,0,5'b01010},{0,0,5'b01101},{0,0,5'b01011},{0,0,5'b00011},
                                        {0,0,5'b01001},{0,0,5'b00110},{0,0,5'b00001},{0,0,5'b00100},
                                        {0,0,5'b00111},{0,0,5'b01100}};
        }

        cp_DATA_msg_types : coverpoint tx_qualified_msg_type iff (to_peer){

                bins b_all_msg_types = {{0,1,5'b00011},{0,1,5'b00010},{0,1,5'b01000},{0,1,5'b00100}};
        }

        // Cross coverage

endgroup : EV_DISCONNECT_message_types
```

Snippet 14: *Generated SystemVerilog Covergroup for quality LEVEL2*

Lowest quality. Cover at least one message across all three categories.

```
covergroup EV_DISCARD_message_types() @(posedge __USB_PD_DISCARD__);
        option.comment = "COV: Key events messages types EV_DISCARD";
        option.name = "EV_DISCARD_message_types";
        option.per_instance = 1;

        // Cover points
        cp_ALL_msg_types : coverpoint tx_qualified_msg_type iff (to_peer){

                bins b_all_msg_types = {{0,0,5'b00110},{0,0,5'b01101},{0,1,5'b00100},{0,0,5'b01011},
                                        {0,1,5'b00011},{0,1,5'b00010},{0,0,5'b00011},{0,0,5'b01001},
                                        {0,0,5'b01010},{0,0,5'b00001},{0,0,5'b00100},{0,0,5'b00111},
                                        {0,1,5'b01000},{0,0,5'b01100}};
        }

        // Cross coverage

endgroup : EV_DISCARD_message_types
```

Snippet 15: *Generated SystemVerilog Covergroup for quality LEVEL1*

7.5.5 Item 4: For all transmitted messages getting all possible valid responses

In the specification table we had captured for all the messages what are the possible valid responses. For example check the Snippet 6 for information captured for the Request message type in field RESPONSE. This information is utilized in the APIs below to generate all the request and response pairs.

Input: Python code for coverage item

```
# For all the transmitted messages all possible valid responses are received
with add_wbv_entry(self,
     WBV_NAME            = 'tx_all_msg_requiring_resp_rx_getting_all_valid_responses_cov',
     WBV_TYPE            = 'WBV_SIGNAL_VALUE',
     WBV_FUNCTIONAL_COV  = '1',
     WBV_CATEGORY        = 'Protocol',
     WBV_STATUS          = 'DONE',
     WBV_CLOCK           = self.event_info_dict['EV_RSP_RX_SUCCESS']['EVENT'],
     WBV_DESCRIPTION     = 'COV: All messages requiring responses getting all possible responses') as wbv_entry:

    bins_per_category_dict = self.get_bins_req_resp_msgs(["CONTROL", "DATA", "EXTENDED"], "TX", 1, "VALID_RESPONSE")
    self.add_cp_for_msg_types(wbv_entry, '{tx_qualified_msg_type, rx_qualified_msg_type}', 'to_peer', bins_per_category_dict)
```

Snippet 16: *Python implementation to cover all transmitted messages getting all possible valid responses*

Output: Covergroup generated

```
covergroup tx_all_msg_requiring_resp_rx_getting_all_valid_responses_cov() @(posedge usb_pd_rsp_rx_success);
        option.comment = "COV: All messages requiring responses getting all possible responses";
        option.name = "tx_all_msg_requiring_resp_rx_getting_all_valid_responses_cov";
        option.per_instance = 1;

        // Cover points
        cp_CONTROL_msg_types : coverpoint {tx_qualified_msg_type, rx_qualified_msg_type} iff (to_peer){

                bins b_TX_DR_Swap_Expected_Response_RX_Accept = {{{0,0,5'b01001},{0,0,5'b00011}}};
                bins b_TX_DR_Swap_Expected_Response_RX_Not_Supported = {{{0,0,5'b01001},{0,0,5'b10000}}};
                bins b_TX_DR_Swap_Expected_Response_RX_Reject = {{{0,0,5'b01001},{0,0,5'b00100}}};
                bins b_TX_DR_Swap_Expected_Response_RX_Wait = {{{0,0,5'b01001},{0,0,5'b01100}}};
                bins b_TX_Get_Source_Cap_Expected_Response_RX_Not_Supported = {{{0,0,5'b00111},{0,0,5'b10000}}};
                bins b_TX_Get_Source_Cap_Expected_Response_RX_Source_Capabilities = {{{0,0,5'b00111},{0,1,5'b00001}}};
                bins b_TX_PR_Swap_Expected_Response_RX_Accept = {{{0,0,5'b01010},{0,0,5'b00011}}};
                bins b_TX_PR_Swap_Expected_Response_RX_Not_Supported = {{{0,0,5'b01010},{0,0,5'b10000}}};
                bins b_TX_PR_Swap_Expected_Response_RX_Reject = {{{0,0,5'b01010},{0,0,5'b00100}}};
                bins b_TX_PR_Swap_Expected_Response_RX_Wait = {{{0,0,5'b01010},{0,0,5'b01100}}};
                bins b_TX_SoftReset_Expected_Response_RX_Accept = {{{0,0,5'b01101},{0,0,5'b00011}}};
                bins b_TX_SoftReset_Expected_Response_RX_Not_Supported = {{{0,0,5'b01101},{0,0,5'b10000}}};
                bins b_TX_VCONN_Swap_Expected_Response_RX_Accept = {{{0,0,5'b01011},{0,0,5'b00011}}};
                bins b_TX_VCONN_Swap_Expected_Response_RX_Not_Supported = {{{0,0,5'b01011},{0,0,5'b10000}}};
                bins b_TX_VCONN_Swap_Expected_Response_RX_Reject = {{{0,0,5'b01011},{0,0,5'b00100}}};
                bins b_TX_VCONN_Swap_Expected_Response_RX_Wait = {{{0,0,5'b01011},{0,0,5'b01100}}};
        }

        cp_DATA_msg_types : coverpoint {tx_qualified_msg_type, rx_qualified_msg_type} iff (to_peer){

                bins b_TX_Request_Expected_Response_RX_Accept = {{{0,1,5'b00010},{0,0,5'b00011}}};
                bins b_TX_Request_Expected_Response_RX_Not_Supported = {{{0,1,5'b00010},{0,0,5'b10000}}};
                bins b_TX_Request_Expected_Response_RX_Reject = {{{0,1,5'b00010},{0,0,5'b00100}}};
                bins b_TX_Request_Expected_Response_RX_Wait = {{{0,1,5'b00010},{0,0,5'b01100}}};
        }

endgroup : tx_all_msg_requiring_resp_rx_getting_all_valid_responses_cov
```

Snippet 17: *Generated SystemVerilog Covergroup for all transmitted messages getting all possible valid responses*

7.5.6 Item 5: For all transmitted messages unexpected acknowledgement

The output coverage shown here is of LEVEL3 quality. Meaning, all possible unexpected responses are considered.

One could debate that crosses with some complex exclusion bins could also be used. Remember we write coverage once but analyze results multiple times. So writing coverage which might be elaborate to write but easier to analyze the results the faster will be well worth the effort. However please note elaborate work if you are manually writing it but if you are generating you can make it easily suit your preferences.

Also any items missed or specifications misinterpreted are easy to fix in the specification tables. It automatically reflects in generated code rather than have to painfully reanalyze the intent to fix the bins and exclusion bins of the crosses.

Input: Python code for coverage item

```
# For all the transmitted messages unexpected ack
with add_wbv_entry(self,
     WBV_NAME             = 'tx_all_msg_unexpected_ack_cov',
     WBV_TYPE             = 'WBV_SIGNAL_VALUE',
     WBV_FUNCTIONAL_COV   = '1',
     WBV_CATEGORY         = 'Protocol',
     WBV_STATUS           = 'DONE',
     WBV_CLOCK            = self.event_info_dict['EV_MSG_RX_SUCCESS']['EVENT'],
     WBV_DESCRIPTION      = 'COV: All messages receiving unexpected acknowledgement') as wbv_entry:

    bins_per_category_dict = self.get_bins_req_resp_msgs(["CONTROL", "DATA", "EXTENDED"], "TX", 1, "UNEXP_ACK")
    self.add_cp_for_msg_types(wbv_entry, '{tx_qualified_msg_type, rx_qualified_msg_type}', 'to_peer', bins_per_category_dict)
```

Snippet 18: *Python implementation to cover all transmitted messages getting unexpected acknowledgement*

Output: Covergroup generated

```
55 covergroup tx_all_msg_unexpected_ack_cov() @(posedge usb_pd_msg_rx_success);
56         option.comment = "COV: All messages receiving unexpected acknowledgement";
57         option.name = "tx_all_msg_unexpected_ack_cov";
58         option.per_instance = 1;
59
60         // Cover points
61         cp_CONTROL_msg_types : coverpoint {tx_qualified_msg_type, rx_qualified_msg_type} iff (to_peer){
62
63                 bins b_TX_Accept_Instead_of_GoodCRC_RX_Accept = {{{0,0,5'b00011},{0,0,5'b00011}}};
64                 bins b_TX_Accept_Instead_of_GoodCRC_RX_BIST = {{{0,0,5'b00011},{0,1,5'b00011}}};
65                 bins b_TX_Accept_Instead_of_GoodCRC_RX_DR_Swap = {{{0,0,5'b00011},{0,0,5'b01001}}};
66                 bins b_TX_Accept_Instead_of_GoodCRC_RX_Get_Sink_Cap = {{{0,0,5'b00011},{0,0,5'b01000}}};
67                 bins b_TX_Accept_Instead_of_GoodCRC_RX_GotoMin = {{{0,0,5'b00011},{0,0,5'b00010}}};
68                 bins b_TX_Accept_Instead_of_GoodCRC_RX_PR_Swap = {{{0,0,5'b00011},{0,0,5'b01010}}};
69                 bins b_TX_Accept_Instead_of_GoodCRC_RX_PS_RDY = {{{0,0,5'b00011},{0,0,5'b00110}}};
70                 bins b_TX_Accept_Instead_of_GoodCRC_RX_Reject = {{{0,0,5'b00011},{0,0,5'b00100}}};
71                 bins b_TX_Accept_Instead_of_GoodCRC_RX_SoftReset = {{{0,0,5'b00011},{0,0,5'b01101}}};
72                 bins b_TX_Accept_Instead_of_GoodCRC_RX_Source_Capabilities = {{{0,0,5'b00011},{0,1,5'b00001}}};
73                 bins b_TX_Accept_Instead_of_GoodCRC_RX_VCONN_Swap = {{{0,0,5'b00011},{0,0,5'b01011}}};
74                 bins b_TX_Accept_Instead_of_GoodCRC_RX_Vendor_Defined = {{{0,0,5'b00011},{0,1,5'b01000}}};
75                 bins b_TX_Accept_Instead_of_GoodCRC_RX_Wait = {{{0,0,5'b00011},{0,0,5'b01100}}};
76                 bins b_TX_DR_Swap_Instead_of_GoodCRC_RX_Accept = {{{0,0,5'b01001},{0,0,5'b00011}}};
77                 bins b_TX_DR_Swap_Instead_of_GoodCRC_RX_BIST = {{{0,0,5'b01001},{0,1,5'b00011}}};
78                 bins b_TX_DR_Swap_Instead_of_GoodCRC_RX_DR_Swap = {{{0,0,5'b01001},{0,0,5'b01001}}};
79                 bins b_TX_DR_Swap_Instead_of_GoodCRC_RX_Get_Sink_Cap = {{{0,0,5'b01001},{0,0,5'b01000}}};
80                 bins b_TX_DR_Swap_Instead_of_GoodCRC_RX_GotoMin = {{{0,0,5'b01001},{0,0,5'b00010}}};
81                 bins b_TX_DR_Swap_Instead_of_GoodCRC_RX_PR_Swap = {{{0,0,5'b01001},{0,0,5'b01010}}};
82                 bins b_TX_DR_Swap_Instead_of_GoodCRC_RX_PS_RDY = {{{0,0,5'b01001},{0,0,5'b00110}}};
83                 bins b_TX_DR_Swap_Instead_of_GoodCRC_RX_Reject = {{{0,0,5'b01001},{0,0,5'b00100}}};
84                 bins b_TX_DR_Swap_Instead_of_GoodCRC_RX_SoftReset = {{{0,0,5'b01001},{0,0,5'b01101}}};
85                 bins b_TX_DR_Swap_Instead_of_GoodCRC_RX_Source_Capabilities = {{{0,0,5'b01001},{0,1,5'b00001}}};
86                 bins b_TX_DR_Swap_Instead_of_GoodCRC_RX_VCONN_Swap = {{{0,0,5'b01001},{0,0,5'b01011}}};
87                 bins b_TX_DR_Swap_Instead_of_GoodCRC_RX_Vendor_Defined = {{{0,0,5'b01001},{0,1,5'b01000}}};
88                 bins b_TX_DR_Swap_Instead_of_GoodCRC_RX_Wait = {{{0,0,5'b01001},{0,0,5'b01100}}};
89                 bins b_TX_Get_Source_Cap_Instead_of_GoodCRC_RX_Accept = {{{0,0,5'b00111},{0,0,5'b00011}}};
90                 bins b_TX_Get_Source_Cap_Instead_of_GoodCRC_RX_BIST = {{{0,0,5'b00111},{0,1,5'b00011}}};
91                 bins b_TX_Get_Source_Cap_Instead_of_GoodCRC_RX_DR_Swap = {{{0,0,5'b00111},{0,0,5'b01001}}};
92                 bins b_TX_Get_Source_Cap_Instead_of_GoodCRC_RX_Get_Sink_Cap = {{{0,0,5'b00111},{0,0,5'b01000}}};
93                 bins b_TX_Get_Source_Cap_Instead_of_GoodCRC_RX_GotoMin = {{{0,0,5'b00111},{0,0,5'b00010}}};
94                 bins b_TX_Get_Source_Cap_Instead_of_GoodCRC_RX_PR_Swap = {{{0,0,5'b00111},{0,0,5'b01010}}};
95                 bins b_TX_Get_Source_Cap_Instead_of_GoodCRC_RX_PS_RDY = {{{0,0,5'b00111},{0,0,5'b00110}}};
```

(…) Line 95 to 180: Bins of cp_CONTROL_msg_types

```
180                 bins b_TX_Wait_Instead_of_GoodCRC_RX_Accept = {{{0,0,5'b01100},{0,0,5'b00011}}};
181                 bins b_TX_Wait_Instead_of_GoodCRC_RX_BIST = {{{0,0,5'b01100},{0,1,5'b00011}}};
182                 bins b_TX_Wait_Instead_of_GoodCRC_RX_DR_Swap = {{{0,0,5'b01100},{0,0,5'b01001}}};
183                 bins b_TX_Wait_Instead_of_GoodCRC_RX_Get_Sink_Cap = {{{0,0,5'b01100},{0,0,5'b01000}}};
184                 bins b_TX_Wait_Instead_of_GoodCRC_RX_GotoMin = {{{0,0,5'b01100},{0,0,5'b00010}}};
185                 bins b_TX_Wait_Instead_of_GoodCRC_RX_PR_Swap = {{{0,0,5'b01100},{0,0,5'b01010}}};
186                 bins b_TX_Wait_Instead_of_GoodCRC_RX_PS_RDY = {{{0,0,5'b01100},{0,0,5'b00110}}};
187                 bins b_TX_Wait_Instead_of_GoodCRC_RX_Reject = {{{0,0,5'b01100},{0,0,5'b00100}}};
188                 bins b_TX_Wait_Instead_of_GoodCRC_RX_SoftReset = {{{0,0,5'b01100},{0,0,5'b01101}}};
189                 bins b_TX_Wait_Instead_of_GoodCRC_RX_Source_Capabilities = {{{0,0,5'b01100},{0,1,5'b00001}}};
190                 bins b_TX_Wait_Instead_of_GoodCRC_RX_VCONN_Swap = {{{0,0,5'b01100},{0,0,5'b01011}}};
191                 bins b_TX_Wait_Instead_of_GoodCRC_RX_Vendor_Defined = {{{0,0,5'b01100},{0,1,5'b01000}}};
192                 bins b_TX_Wait_Instead_of_GoodCRC_RX_Wait = {{{0,0,5'b01100},{0,0,5'b01100}}};
193         }
194
195         cp_DATA_msg_types : coverpoint {tx_qualified_msg_type, rx_qualified_msg_type} iff (to_peer){
196
197                 bins b_TX_BIST_Instead_of_GoodCRC_RX_Accept = {{{0,1,5'b00011},{0,0,5'b00011}}};
198                 bins b_TX_BIST_Instead_of_GoodCRC_RX_BIST = {{{0,1,5'b00011},{0,1,5'b00011}}};
199                 bins b_TX_BIST_Instead_of_GoodCRC_RX_DR_Swap = {{{0,1,5'b00011},{0,0,5'b01001}}};
200                 bins b_TX_BIST_Instead_of_GoodCRC_RX_Get_Sink_Cap = {{{0,1,5'b00011},{0,0,5'b01000}}};
201                 bins b_TX_BIST_Instead_of_GoodCRC_RX_GotoMin = {{{0,1,5'b00011},{0,0,5'b00010}}};
202                 bins b_TX_BIST_Instead_of_GoodCRC_RX_PR_Swap = {{{0,1,5'b00011},{0,0,5'b01010}}};
203                 bins b_TX_BIST_Instead_of_GoodCRC_RX_PS_RDY = {{{0,1,5'b00011},{0,0,5'b00110}}};
204                 bins b_TX_BIST_Instead_of_GoodCRC_RX_Reject = {{{0,1,5'b00011},{0,0,5'b00100}}};
205                 bins b_TX_BIST_Instead_of_GoodCRC_RX_SoftReset = {{{0,1,5'b00011},{0,0,5'b01101}}};
206                 bins b_TX_BIST_Instead_of_GoodCRC_RX_Source_Capabilities = {{{0,1,5'b00011},{0,1,5'b00001}}};
207                 bins b_TX_BIST_Instead_of_GoodCRC_RX_VCONN_Swap = {{{0,1,5'b00011},{0,0,5'b01011}}};
208                 bins b_TX_BIST_Instead_of_GoodCRC_RX_Vendor_Defined = {{{0,1,5'b00011},{0,1,5'b01000}}};
209                 bins b_TX_BIST_Instead_of_GoodCRC_RX_Wait = {{{0,1,5'b00011},{0,0,5'b01100}}};
210                 bins b_TX_Request_Instead_of_GoodCRC_RX_Accept = {{{0,1,5'b00010},{0,0,5'b00011}}};
211                 bins b_TX_Request_Instead_of_GoodCRC_RX_BIST = {{{0,1,5'b00010},{0,1,5'b00011}}};
212                 bins b_TX_Request_Instead_of_GoodCRC_RX_DR_Swap = {{{0,1,5'b00010},{0,0,5'b01001}}};
213                 bins b_TX_Request_Instead_of_GoodCRC_RX_Get_Sink_Cap = {{{0,1,5'b00010},{0,0,5'b01000}}};
214                 bins b_TX_Request_Instead_of_GoodCRC_RX_GotoMin = {{{0,1,5'b00010},{0,0,5'b00010}}};
215                 bins b_TX_Request_Instead_of_GoodCRC_RX_PR_Swap = {{{0,1,5'b00010},{0,0,5'b01010}}};
216                 bins b_TX_Request_Instead_of_GoodCRC_RX_PS_RDY = {{{0,1,5'b00010},{0,0,5'b00110}}};
217                 bins b_TX_Request_Instead_of_GoodCRC_RX_Reject = {{{0,1,5'b00010},{0,0,5'b00100}}};
218                 bins b_TX_Request_Instead_of_GoodCRC_RX_SoftReset = {{{0,1,5'b00010},{0,0,5'b01101}}};
219                 bins b_TX_Request_Instead_of_GoodCRC_RX_Source_Capabilities = {{{0,1,5'b00010},{0,1,5'b00001}}};
220                 bins b_TX_Request_Instead_of_GoodCRC_RX_VCONN_Swap = {{{0,1,5'b00010},{0,0,5'b01011}}};
221                 bins b_TX_Request_Instead_of_GoodCRC_RX_Vendor_Defined = {{{0,1,5'b00010},{0,1,5'b01000}}};
222                 bins b_TX_Request_Instead_of_GoodCRC_RX_Wait = {{{0,1,5'b00010},{0,0,5'b01100}}};
223                 bins b_TX_Sink_Capabilities_Instead_of_GoodCRC_RX_Accept = {{{0,1,5'b00100},{0,0,5'b00011}}};
224                 bins b_TX_Sink_Capabilities_Instead_of_GoodCRC_RX_BIST = {{{0,1,5'b00100},{0,1,5'b00011}}};
225                 bins b_TX_Sink_Capabilities_Instead_of_GoodCRC_RX_DR_Swap = {{{0,1,5'b00100},{0,0,5'b01001}}};
226                 bins b_TX_Sink_Capabilities_Instead_of_GoodCRC_RX_Get_Sink_Cap = {{{0,1,5'b00100},{0,0,5'b01000}}};
227                 bins b_TX_Sink_Capabilities_Instead_of_GoodCRC_RX_GotoMin = {{{0,1,5'b00100},{0,0,5'b00010}}};
228                 bins b_TX_Sink_Capabilities_Instead_of_GoodCRC_RX_PR_Swap = {{{0,1,5'b00100},{0,0,5'b01010}}};
229                 bins b_TX_Sink_Capabilities_Instead_of_GoodCRC_RX_PS_RDY = {{{0,1,5'b00100},{0,0,5'b00110}}};
230                 bins b_TX_Sink_Capabilities_Instead_of_GoodCRC_RX_Reject = {{{0,1,5'b00100},{0,0,5'b00100}}};
231                 bins b_TX_Sink_Capabilities_Instead_of_GoodCRC_RX_SoftReset = {{{0,1,5'b00100},{0,0,5'b01101}}};
232                 bins b_TX_Sink_Capabilities_Instead_of_GoodCRC_RX_Source_Capabilities = {{{0,1,5'b00100},{0,1,5'b00001}}};
233                 bins b_TX_Sink_Capabilities_Instead_of_GoodCRC_RX_VCONN_Swap = {{{0,1,5'b00100},{0,0,5'b01011}}};
234                 bins b_TX_Sink_Capabilities_Instead_of_GoodCRC_RX_Vendor_Defined = {{{0,1,5'b00100},{0,1,5'b01000}}};
235                 bins b_TX_Sink_Capabilities_Instead_of_GoodCRC_RX_Wait = {{{0,1,5'b00100},{0,0,5'b01100}}};
236                 bins b_TX_Vendor_Defined_Instead_of_GoodCRC_RX_Accept = {{{0,1,5'b01000},{0,0,5'b00011}}};
237                 bins b_TX_Vendor_Defined_Instead_of_GoodCRC_RX_BIST = {{{0,1,5'b01000},{0,1,5'b00011}}};
238                 bins b_TX_Vendor_Defined_Instead_of_GoodCRC_RX_DR_Swap = {{{0,1,5'b01000},{0,0,5'b01001}}};
239                 bins b_TX_Vendor_Defined_Instead_of_GoodCRC_RX_Get_Sink_Cap = {{{0,1,5'b01000},{0,0,5'b01000}}};
240                 bins b_TX_Vendor_Defined_Instead_of_GoodCRC_RX_GotoMin = {{{0,1,5'b01000},{0,0,5'b00010}}};
241                 bins b_TX_Vendor_Defined_Instead_of_GoodCRC_RX_PR_Swap = {{{0,1,5'b01000},{0,0,5'b01010}}};
242                 bins b_TX_Vendor_Defined_Instead_of_GoodCRC_RX_PS_RDY = {{{0,1,5'b01000},{0,0,5'b00110}}};
243                 bins b_TX_Vendor_Defined_Instead_of_GoodCRC_RX_Reject = {{{0,1,5'b01000},{0,0,5'b00100}}};
244                 bins b_TX_Vendor_Defined_Instead_of_GoodCRC_RX_SoftReset = {{{0,1,5'b01000},{0,0,5'b01101}}};
245                 bins b_TX_Vendor_Defined_Instead_of_GoodCRC_RX_Source_Capabilities = {{{0,1,5'b01000},{0,1,5'b00001}}};
246                 bins b_TX_Vendor_Defined_Instead_of_GoodCRC_RX_VCONN_Swap = {{{0,1,5'b01000},{0,0,5'b01011}}};
247                 bins b_TX_Vendor_Defined_Instead_of_GoodCRC_RX_Vendor_Defined = {{{0,1,5'b01000},{0,1,5'b01000}}};
248                 bins b_TX_Vendor_Defined_Instead_of_GoodCRC_RX_Wait = {{{0,1,5'b01000},{0,0,5'b01100}}};
249         }
250
251         // Cross coverage
252
253 endgroup : tx_all_msg_unexpected_ack_cov
```

Snippet 19: *Generated SystemVerilog Covergroup for all transmitted messages getting unexpected acknowledgement*

7.5.7 Item 6: Power negotiation sequence

We have already seen the power negotiation protocol sequence and its graph in the Section 7.4.3.

Here we will look at its python implementation. We are using simple tree data structure to capture the graph.

Input: Graph of power negotiation scenarios

This is the tree data structure representation of the protocol sequence graph shown in mindmap 6.

```
# Power negotiation protocol : Always initiated by SOURCE
self.usb_pd_protocol_sequence_dict["power_nego_protocol"] = {}

self.usb_pd_protocol_sequence_dict["power_nego_protocol"]['GRAPH'] = Graph("power_nego_protocol")
self.usb_pd_protocol_sequence_dict["power_nego_protocol"]['GRAPH'].create_node(self.all_msg_info_dict["Source_Capabilities"]["NODE_INFO"]["SOURCE"]["TX"], "Source_Capabilities")  # root node
self.usb_pd_protocol_sequence_dict["power_nego_protocol"]['GRAPH'].create_node(self.all_msg_info_dict["GoodCRC"]["NODE_INFO"]["SOURCE"]["RX"], "GoodCRC_Source_Capabilities", parent = "Source_Capabilities")

self.usb_pd_protocol_sequence_dict["power_nego_protocol"]['GRAPH'].create_node(self.all_msg_info_dict["Request"]["NODE_INFO"]["SINK"]["TX"],   "Request",                                parent = "GoodCRC_Source_Capabilities")
self.usb_pd_protocol_sequence_dict["power_nego_protocol"]['GRAPH'].create_node(self.all_msg_info_dict["GoodCRC"]["NODE_INFO"]["SINK"]["RX"],   "GoodCRC_Request",                 parent = "Request")

self.usb_pd_protocol_sequence_dict["power_nego_protocol"]['GRAPH'].create_node(self.all_msg_info_dict["Accept"]["NODE_INFO"]["SOURCE"]["TX"],  "Accept",                                parent = "GoodCRC_Request")
self.usb_pd_protocol_sequence_dict["power_nego_protocol"]['GRAPH'].create_node(self.all_msg_info_dict["GoodCRC"]["NODE_INFO"]["SOURCE"]["RX"], "GoodCRC_Accept",                  parent = "Accept")

self.usb_pd_protocol_sequence_dict["power_nego_protocol"]['GRAPH'].create_node(self.all_msg_info_dict["Reject"]["NODE_INFO"]["SOURCE"]["TX"],  "Reject",                                parent = "GoodCRC_Request")
self.usb_pd_protocol_sequence_dict["power_nego_protocol"]['GRAPH'].create_node(self.all_msg_info_dict["GoodCRC"]["NODE_INFO"]["SOURCE"]["RX"], "GoodCRC_Reject",                  parent = "Reject")

self.usb_pd_protocol_sequence_dict["power_nego_protocol"]['GRAPH'].create_node(self.all_msg_info_dict["Wait"]["NODE_INFO"]["SOURCE"]["TX"],    "Wait",                                  parent = "GoodCRC_Request")
self.usb_pd_protocol_sequence_dict["power_nego_protocol"]['GRAPH'].create_node(self.all_msg_info_dict["GoodCRC"]["NODE_INFO"]["SOURCE"]["RX"], "GoodCRC_Wait",                    parent = "Wait")

self.usb_pd_protocol_sequence_dict["power_nego_protocol"]['GRAPH'].create_node(self.all_msg_info_dict["PS_RDY"]["NODE_INFO"]["SOURCE"]["TX"],  "PS_RDY",                                parent = "GoodCRC_Accept")
self.usb_pd_protocol_sequence_dict["power_nego_protocol"]['GRAPH'].create_node(self.all_msg_info_dict["GoodCRC"]["NODE_INFO"]["SOURCE"]["RX"], "GoodCRC_PS_RDY",               parent = "PS_RDY")
```

Snippet 20: *Python implementation for graph of power negotiation protocol possibilities*

Output: Covergroups per sequence

Sequences can be covered with the transition bins but it's very restrictive in terms of sampling. We cover the sequence with the additional state machine like glue logic in-order to use the covergroup construct. SystemVerilog assertions do provide some level flexibility for sequence coverage but taxing while debugging.

A simple state machine is generated as glue logic to track each steps of the sequence. This piece of generated code is not included to keep focus on the coverage.

Bin per step allows very clear idea of how far the sequence had progressed in regression. This type of information is difficult to find out with the SystemVerilog assertion based coverage. Also the state of steps tracking state machine generated can be added to the waves along with the RTL making the coverage holes debug easier as well.

Three sequences generated are:

- **power_nego_protocol_SINK_0 (bins for 8 steps):** Source Capabilities => GoodCRC => Request => GoodCRC => Accept => GoodCRC => PS_RDY => GoodCRC (8 steps)
- **power_nego_protocol_SINK_1 (bins for 6 steps):** Source Capabilities => GoodCRC => Request => GoodCRC => Wait => GoodCRC (6 steps)
- **power_nego_protocol_SINK_2 (bins for 6 steps):** Source Capabilities => GoodCRC => Request => GoodCRC => Reject => GoodCRC

```
covergroup power_nego_protocol_SINK_0() @(posedge dut.clk);
        option.comment = "power_nego_protocol initiated sequence for SINK";
        option.name = "power_nego_protocol_SINK_0";
        option.per_instance = 1;

        // Cover points
        cp_sig_val_power_nego_protocol_SINK_0_0 : coverpoint seq_steps_l_power_nego_protocol_SINK_0_0 iff (1){

                bins sequence_step_0 = {8'b00000001};
                bins sequence_step_1 = {8'b00000011};
                bins sequence_step_2 = {8'b00000111};
                bins sequence_step_3 = {8'b00001111};
                bins sequence_step_4 = {8'b00011111};
                bins sequence_step_5 = {8'b00111111};
                bins sequence_step_6 = {8'b01111111};
                bins sequence_step_7 = {8'b11111111};
        }
endgroup : power_nego_protocol_SINK_0

covergroup power_nego_protocol_SINK_1() @(posedge dut.clk);
        option.comment = "power_nego_protocol initiated sequence for SINK";
        option.name = "power_nego_protocol_SINK_1";
        option.per_instance = 1;

        // Cover points
        cp_sig_val_power_nego_protocol_SINK_1_0 : coverpoint seq_steps_l_power_nego_protocol_SINK_1_0 iff (1){

                bins sequence_step_0 = {6'b000001};
                bins sequence_step_1 = {6'b000011};
                bins sequence_step_2 = {6'b000111};
                bins sequence_step_3 = {6'b001111};
                bins sequence_step_4 = {6'b011111};
                bins sequence_step_5 = {6'b111111};
        }
endgroup : power_nego_protocol_SINK_1

covergroup power_nego_protocol_SINK_2() @(posedge dut.clk);
        option.comment = "power_nego_protocol initiated sequence for SINK";
        option.name = "power_nego_protocol_SINK_2";
        option.per_instance = 1;

        // Cover points
        cp_sig_val_power_nego_protocol_SINK_2_0 : coverpoint seq_steps_l_power_nego_protocol_SINK_2_0 iff (1){

                bins sequence_step_0 = {6'b000001};
                bins sequence_step_1 = {6'b000011};
                bins sequence_step_2 = {6'b000111};
                bins sequence_step_3 = {6'b001111};
                bins sequence_step_4 = {6'b011111};
                bins sequence_step_5 = {6'b111111};
        }
endgroup : power_nego_protocol_SINK_2
```

Snippet 21: *Generated SystemVerilog Covergroup for 3 generated sequence of power negotiation protocol possibilities (observe number of steps to correlate)*

7.5.8 Item 7: Timeout error injection within power negotiation sequence

As we had seen, for every request or response messages the GoodCRC is expected to confirm its successful reception by peer. There is CRCReceiveTimer defined by the specifications which is started when request or response are transmitted and stopped when the corresponding GoodCRC acknowledgement is received.

Power negotiation sequence is one of the very important sequences of the USB power delivery protocol. So let's say we decide to cover CRCReceiveTimer timeout at all possible points in this sequence to ensure this sequence is thoroughly verified.

Now this type of comprehensive coverage might be very effort intensive to write but with graphs it can be created algorithmically with the following approach.

Input: Algorithm

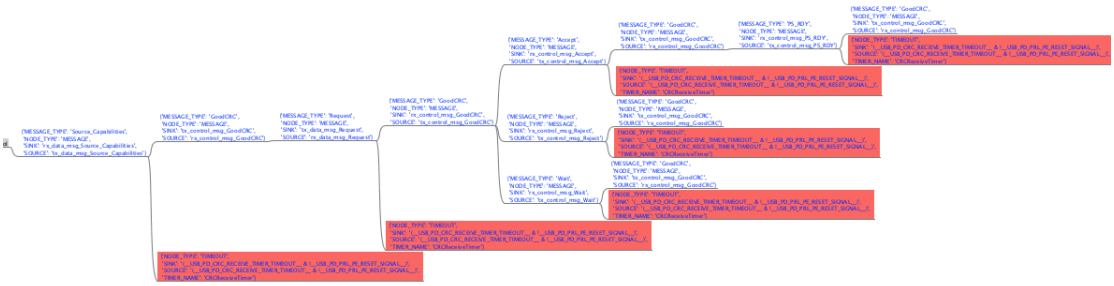
In the following snippet we are traversing the graph of the power negotiation sequence and wherever the GoodCRC is node is found we are adding an additional timeout node to parent node of GoodCRC node.

```
def add_crc_receive_timeout_nodes(self, graph):
    curr_graph_root_node =  graph.get_root_node_id()

    for node in graph.expand_tree(curr_graph_root_node, mode=_DEPTH):
        parent_node_id = graph.get_parent_id(node)
        node_info =  graph.get_node_info(node)
        if node_info['NODE_TYPE'] is "MESSAGE" and node_info['MESSAGE_TYPE'] is "GoodCRC":
            timeout_node_id_str = "Timeout_" + node
            graph.create_node(self.timer_info_dict["CRCReceiveTimer"]["NODE_INFO"]["SOURCE"]["TIMEOUT"], timeout_node_id_str,  parent = parent_node_id)
```

Snippet 22: *Python implementation to algorithmically add CRCReceiveTimeout for all GoodCRC message waits*

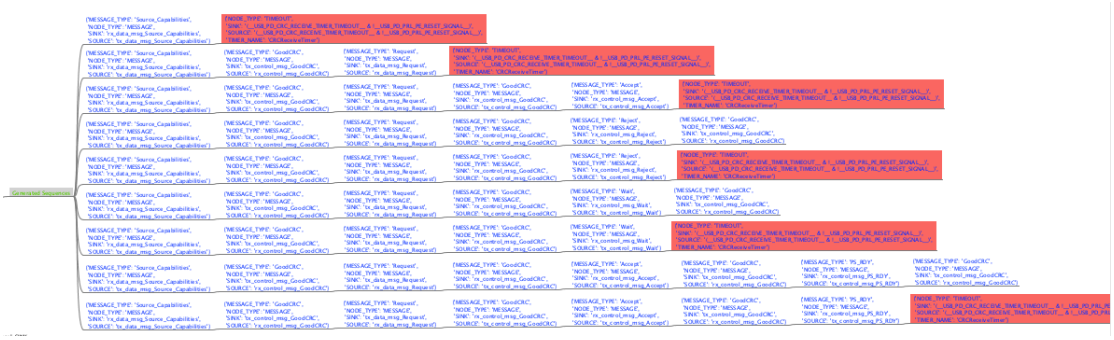
Output: Mindmap of modified graph

Now you can see with every GoodCRC node, a parallel node of CRCReceiveTimer timeout node (RED) has been added.



Mindmap 9*: After adding CRCReceiveTimer timeout node to power negotiation graph*

From this new graph automatically new functional coverage sequence can be generated.



Mindmap 10*: Sequences generated after adding CRCReceiveTimer timeout node to power negotiation graph*

We are just showing the mindmaps but code on the similar lines as simple power negotiation sequence will get generated. You can see in the timeout sequence graphs there are three sequences not ending in RED block are normal power

negotiation sequence scenarios without timeout (which we had seen for previously for power negotiation sequence item). The ones ending with the red block are new ones added algorithmically for covering timeout at each stages of the power negotiation sequence.

Now of-course based on your quality goals you can decide to add timeout to only certain branches or after certain depth in the protocol. Possibilities are unlimited. For complex protocols this provides a lot of flexibility to dynamically control the quality of coverage generated.

8 Conclusion

USB power delivery protocol provides a time capsule. Its overall complexity is lesser than standard USB 2.0 but still relevant in the current times. We choose it for its simplicity and relevance. USB power delivery does exemplify the problems of similar nature faced at higher complexity in other high-speed serial IO protocols.

Specification to functional coverage possibility has been demonstrated with this case study. For complex designs this approach can make writing comprehensive and maintainable functional coverage models easier and faster.

Also being able to dynamically define level of quality of coverage can help tune it to your project priorities. In general by making coverage intent executable, it will help you churn higher quality of design IP revisions throughout its life cycle.