Advanced UVM command line processor for central maintenance and randomization of control knobs

Siddharth Krishna Kumar
Samsung Austin R&D Center
sidd.kk@samsung.com

***Abstract* - this paper describes an advanced command line processor, an extension of the uvm_cmdline_processor class which enables users to pass a combination of range or specific values and specify weights to enable randomization of control knobs. The original functionality of the base class is kept intact and additional features are developed on top of it. The source code is made available in the appendix section of the paper.**

I. INTRODUCTION

System Verilog UVM provides an *uvm_cmdline_processor* class, with methods to parse command line arguments. This allows bench developers to control and configure various aspects of the test bench by passing user values along with the test command. This class serves the purpose of passing int, string etc and parsing them, but does not provide additional smart capabilities like randomization. This is powerful by itself but lacks finer control since it is accepts one value per argument. By modifying the parsing a slight bit, this concept here tries to make it much more powerful and robust and portable to most UVM verification environments.

II. CONCEPT

The general thought process behind the advanced UVM command line processor is to have an enhanced yet easy way to pass not just values but provide a mix of values for a particular field and randomize it based on weights. Typical usage of control knob specified on the command line or as a run-time plus arg, is to provide a single value which is then used in a specific fashion in the code to shape traffic. This is definitely useful but typically tends to create scenarios where multiple values have to be specified to create a specific scenario.

Let us first try and explore some scenarios that will provide an idea of the motivation behind the concept:

1. Delay knobs – one of the common methods to stress the RTL is to cause a burst behavior on the various input interfaces. Typically, this is achieved by providing min-delay and max-delay values with the code randomizing the delay between these 2 values. Straight forward as it may be, it comes with its own drawbacks. If a user wants to provide an occasional large delay while maintaining a cadence of smaller delays lot of plumbing needs to be present in the underlying code.

2. Opcodes – in processor verification, a large list of opcodes are present in the instruction set and shaping traffic becomes priority. The opcodes are typically not consecutive and occasionally there are reserved opcodes in derivatives. Thus, using the above concept of min/max becomes tough. Also, plus-arg traditionally take in an integer or string value. Remembering the hex value for each opcode is not the ideal situation and strings need to be converted into enumerated values to get the desired result

3. Instance specific control – another common situation in modern day test benches. We come across cases, where we have to specify a value for a specific instance of a component while the other instances need a different value. UVM provides a set_config_db command line option but that again takes in a single value.

What if all three above scenarios could be solved in a single setup? Another equally important aspect is re-usability. Common scenarios translate to lot of similar code across test benches in the same project or chip. If the end processing is pretty much the same, would we rather not create a more central piece of code which everyone can re-use instead of developing their own? The advanced command line processor is an attempt to address and simplify most of these scenarios.

III. FEATURES

This section highlights some of the features and touches upon some example code. The idea was adapted from an internal C based system to avoid numerous DPI calls and provide support in native System Verilog.

The data structure class which is used to store all the information associated with a control knob is called *acmdline_opts*. This class contains fields and methods to store the value/min-max range/weights and generate a random value each time we invoke the *get_rand_val* method in the *advanced_cmdline_processor*. Some of the key methods of this class are:

1. parse_opt(string) : this method takes in a string argument passed in a specific format and extracts the intended min/max value and weights and stores them for repeated use. This method is typically called only once per control knob. An example of the format is as shown below:

   +cmdline_example_1=min_val_0~max_val_0:wt_0, val_1:wt_1,...min_val_n~max_val_n:wt_n

   min_val_<n>: specifies the minimum value for the range
   max_val_<n>: specifies the maximum value for the range
   val_<n>: specifies a single value. Internally it is stored as min_<n>=max_<n>=val_<n>
   wt_<n>: specifies the weight given to the range or value. Note, within a range each value has equal weight. If not specified, the weight is defaulted to 1

   uvm_split_string() and *uvm_is_match()* standard functions are used to compute the different buckets and sort the values and weights from each other

2. parse_val(string) : function used to distinguish hexadecimal, binary or integer format specified as a string and convert them to long int. Examples provided below indicate how an user can specify hexadecimal and binary values:

   +cmdline_hex_example=h1:10,hdead,h100~h1000:89

   +cmdline_bin_example=b100:10,b111,b011~b110:89

   This support is useful especially while controlling addresses and/or data fields from the command line.

3. set_weight_q(): utilized to push weights into a weight queue which is later utilized in the actual randomization. We push value <n> (the bucket number) as many times into the queue as the wt_<n> value. This is utilized later on by the *get_rand_int()* method

4. get_rand_int(): this method is what returns a random value each time it is invoked. If an user has specified only 1 bucket with a single value, the code is optimized to just return the value without calls to *std::randomize()*. In all other cases, first a winner index is calculated by picking a random value from the weight queue. This decides which bucket (<n> value) is picked. Next step is to generate a random value in the range provided in the bucket and pass it on. In the case of a single value, the value is returned without calling the randomize function

One of the scenarios discussed earlier is the use of enumerated types. An *acmdline_opts_enum#(type T=uvm_active_passive_enum)* class is declared. This extends from the above *acmdline_opts* class. The enumerated

type is passed by the user as a parameter to this class. Most of the functionality, fields and methods from the base class are retained. The function which is overridden is:

1. parse_val(string): this function now interprets the string as an enumerated type value and uses the *uvm_enum_wrapper#(T)::from_name(string, enum_var)* function to map the string to an enumerated value and eventually store it as a long int. All the other processing remains the same as regular control knobs. Here is an example:

   +*cmdline_enum_example=ZERO:3,ONE:2,2:1* : (typedef enum {ZERO, ONE, TWO} count_t;)

This definitely helps in a more readable and intuitive way to provide command line arguments for enumerated types with especially a long list of values.

The *advanced_cmdline_processor* class maintains a database of objects of above described classes. The user is expected to call methods in this class and not directly utilize the data structure classes above. Let us explore some of the methods user will end up invoking in their code to interface

1. get_inst(): this returns a singleton class handle to the user. The idea is to have only one single instance of the advanced_cmdline_processor, so that all the control knobs are parsed, stored and invoked via a single data-base.

2. parse_cmdline(string, string, string, string, acmdline_opts): used to parse the command line by using the base class *get_arg_values()* method. In this implementation, the last specified value of a control knob is treated as the final argument. It then creates an entry in the data-base and subsequent calls to this function simply returns 1'b1. Invoking this method directly by the user is discouraged but should not have any functional consequences.

3. is_valid(string, string): this method checks and returns a true if the name + full_path(optional) control knob is provided on the command line. This is a useful check to determine if we want the code to proceed with default constraints or want overridden values provided by the cmdline_processor.

4. get_rand_val(string, string, string, string, acmdline_opts): ideally, this one single method should be called from the code and it will perform all the necessary operations and return a valid random value. If any of the user provided values are incorrect it will result in an error.

   Let us look at some of the function parameters accepted:

   a. name : this is the string which is used to match the command line entry. It also becomes a key to search the database
   b. full_path : an optional parameter, if provided this is concatenated with the name above to provide an unique entry to match and subsequently be a key in the database. This needs to be provided if the user wants to use the feature of per instance control. If this is non-empty the final argument is retrieved using the *uvm_config_db#(string)::get(null, full_path, name, final_arg*
   c. default_val: the user can specify a default value to be applied if no command line argument is provided. Again, this is optional but note that if an user does not provide a default value and does not provide any on the command line , it will result in an error
   d. description: an optional string parameter, which can be used to provide a quick one line on the usage of the control knob. This will stored in the database and later on used to print it in post-processing.
   e. acmdline_opts: this is solely passed when processing an enumerated type and when non-empty it adds the object directly into the data-base without creating a new entry

Similar to *acmdline_opts_enum* there is an *advanced_opt_proc#(type T=uvm_active_passive_enum)* class which is a wrapper to handle enumerated type values. Instead of *get_rand_val()* method user will call *get_rand_enum()* method which will then return a random enumerated value

IV. CODING EXAMPLES

The key strength of the advanced_cmdline_processor lies in the hands of the developer and how they decide to leverage some or all of the features described in the previous section. To get a better idea, this section deals with some of the coding examples.

Example 1

```
typedef enum {ADD,SUB,MUL} opcode_t;

opt_proc = advanced_cmdline_processor::get_inst();

class packet extends uvm_sequence_item;

  rand opcode_t   opcode;
  rand bit [2:0]  return_order;
  rand bit [31:0] operand_a;
  rand bit [31:0] operand_b;

  constraint c_return_order {
    (opcode == ADD) -> (return_order == 3);
    (opcode == SUB) -> (return_order inside {[1:4]};
  }

  constraint c_operand_a {
    (opcode == MUL) -> (operand_a[2:0] == 3'b00);
  }

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

  function void pre_randomize();

    if(advanced_opt_proc_enum#(opcode_t)::is_valid("opcode")) begin
      opcode = advanced_opt_proc_enum#(opcode_t)::get_rand_enum("opcode");
      opcode.rand_mode(0);
    end

    if(opt_proc.is_valid("operand_a")) begin
      operand_a = opt_proc.get_rand_val("operand_a");
      operand_a.rand_mode(0);
    end
  endfunction
endclass

class packet_sequence extends uvm_sequence;

 task body();
   packet pkt;

   for(int i=0;i<10;i++) begin
     pkt = packet::type_id::create("pkt");
     pkt.randomize();

     start_item(pkt);
     finish_item(pkt);
```

```
   end
  end
endclass
```

In this case, if no command line arguments are provided then default randomization kicks in and all the constraints are observed. Now let us take look at some command-line options:

a. *+opcode=ADD:80,SUB:20* : the user is trying to weight the packet in an 80:20 ratio of ADD:SUB. This code will roughly generate 8 packets with opcode = ADD and 2 packets with **opcode** = SUB and never generate a pkt with opcode = MUL. In the pre_randomize() step, the rand_mode for opcode is turned off. This means the value which is supplied by the **get_rand_val()** will be the final value. But by doing so, we do not lose the constraints. So the **c_return_order** constraint still holds true and the return_order randomization will be in accordance with the opcode value
b. *+operand=32'hfffffff0* : the user is specifying a single hexadecimal value for **operand_a** to be used in this test. Notice, that bits [2:0] = = 3'b00 and thus will generate all types of packet without an issue.
c. *+operand=32'h00000000~32'h0000000f* : user is trying to provide a range of values. There is a potential issue here though. During randomization, if opcode = = MUL is picked it will cause a constraint solver failure. User needs to make sure they use the control knob from a. above in conjunction with c. to get the desired results.

Example 2

```
class packet_sequence extends uvm_sequence;

 task body();
   packet pkt;

   for(int i=0;i<10;i++) begin
     pkt = packet::type_id::create("pkt");
     pkt.randomize();

     start_item(pkt);
     finish_item(pkt);

     p_sequencer.delay(opt_proc.get_rand_val("pkt_delay",get_full_name(),"0","Control
the delay between pkts in the sequence");  // assumption : parent sequencer has a task
                                           // delay(int num_cycles);
   end
  end
endclass

class packet_virtual_sequence extends uvm_sequence;
  packet_sequence  pkt_seq_1, pkt_seq_2;
  packet_sequencer sqr;

  function new(string name = "pkt_vir_seq");
    super.new(name);
    pkt_seq_1 = packet_sequence::type_id::create("pkt_seq_1");
    pkt_seq_2 = packet_sequence::type_id::create("pkt_seq_2");
    sqr       = packet_sequencer::type_id::create("sqr", this);
  endfunction

  task body();
    pkt_seq_1.start(sqr);
    pkt_seq_2.start(sqr);
  endtask
```

In this example, a zero-delay is added between each packet in both the instances of the sequence. Let us assume a scenario where we want zero or a cycle delay in the first sequence while larger delays on the second sequence. Here is how we can achieve it:

*+uvm_set_config_string=*pkt_seq_1,pkt_delay,"0:50,1:50"*
*+uvm_set_config_string=*pkt_seq_2,pkt_delay,"10~20:50,21~100:40,101~500:10"*
Slight modification in the command line and in the code-base helps us change how the arguments are picked and processed. Instead of providing a plus-arg , we need to provide a +uvm_set_config_string. Also notice, in the sequence class definition we provide **get_full_name()** as the path after the name. This enables the advanced_cmdline_processor to process it differently and add the values to the data-base. Remember, in this situation, 2 entries are made in the data-base since each instance of the sequence will have an unique {**name,path}** combination.

Example 3

```
module tb();

dut dut(.A(opt_proc.get_rand_val("dut_A","","0")),
        .B(opt_proc.get_rand_val("dut_B","","1"))
       );

endmodule
```

This is a simple yet powerful application. There are ports in the dut which are driven to a specific value while integrating at the chip level. At the unit level, they need to be held constant but can be randomized at the beginning of each test. By specifying as above we can change the default values each of these ports. This is useful in configuring a DUT as a different instance or driving clock gating signals, overrides etc

V. BENEFITS

The previous sections have discussed the intent and implementation of the advanced commandline processor. This section is solely dedicated to summarize and highlight some of the benefits.

1. Avoids redundancy of code, definitely helps save a lot of lines of code and effort. The code by itself is not too complex to re-create. But developing it in every TB just does not make sense from a re-usability stand-point.

2. Provides a central framework for code optimizations, the most efficient compiler dependent functions can be implemented to handle the randomization. Any future corrections or enhancements can be implemented and passed on to all consumers easily.

3. Scalability to all test benches and ease of use in existing frameworks. Switching over to use the advanced_cmdline_processor in existing benches is not at all complicated. It extends from uvm_cmdline_processor and none of the existing infrastructure built on top of uvm_cmdline_processor will get affected in the process.

4. Extension ensures use of the base class methods in conjunction with methods defined in the extended class. A user can choose to parse the string in a specific format and utilize the features of the component for special cases. There is no compulsion to use all methods of the advanced_cmdline_processor.

5. The processor allows users to change the randomization for a particular test without having to rework the code and compile. A quick run with new buckets is sufficient to change the configuration. Lot of power in quickly creating a specific scenario without having to change the code.

6. Controllability on a per instance granularity is pretty powerful especially in re-use both vertically to higher level benches and horizontally across blocks.

7. This can be added to an existing RAL setup to control and randomize register settings. Register configuration via a list of command line options is simpler and efficient.

VI. Drawbacks

No code is perfect and this is reality. Every system is built to tackle a specific set of issues but almost always falls behind in some areas. The advanced_cmdline_processor is just another victim of this age old conundrum. First and foremost, even though folks might be tempted to, this will not be an effective substitute for constraints and traditional randomization. It was intended to complement the constrained random approach by providing an orthogonal way to control and randomize. As discussed in the coding examples, uni-directional implications can fail if related fields are not controlled. The other drawback is string based search and match works really well for a reasonably sized data base but will start to hamper performance if the data base grows large in size. But definitely the impact outweighs the drawbacks and if utilized judiciously it is a powerful tool for any UVM based verification effort.

VII. Future Enhancements

One of the future improvements identified is to collect all the metrics for each user specified control knob – number of calls, final random values etc. The subsequent step will be to post-process and visualize it using python. User will be able to analyze the stimulus shaping of the TB and fix issues without having to enable, dump and analyze coverage. This is expected to improve development cycles especially for constrained random scenarios. This is not ready as of now, but the code will definitely makes it way into the appendix section of this paper.

Acknowledgements

Colleagues in the past and present have had a significant influence on shaping the concept and this paper itself. Want to take a moment to thank these folks. John Dickol, who assisted in solving some of the technical challenges of the code and provided a lot of feedback and resources during the paper writing process. Multiple brainstorming sessions with Vamsi Chavali on various scenarios was immensely productive in refining the codebase. Profusely thank my first mentor in the industry, Brent Vestal, whose general guidance on verification strategies and coding have always been a source of inspiration.

References

[1] System Verilog LRM IEEE Std 1800™-2017 (Revision of IEEE Std 1800-2012)
[2] Universal Verification Methodology (UVM) 1.2 Class Reference

APPENDIX I

```
`ifndef __ACMDLINE_OPTS_SV__
`define __ACMDLINE_OPTS_SV__

///
/// Protected data class used to create an entry in the database maintained by
/// advanced_cmdline_processor
/// Not intended to be used directly, handles created by advanced_cmdline_processor
///
class acmdline_opts;
  string name;        ///< stores the name of the option
  string description; ///< holds the user provided description
  string default_val; ///< holds the default value provided by the user

  longint min[];
  longint max[];
  int     wt[];
  int     weight_q[$];

  extern         function          new(string name, string default_val="",  string
                                       description="");
  extern virtual function void     set_len(int len);
  extern virtual function void     set_weight_q();
  extern virtual function bit      parse_opt(string parse_string);
  extern virtual function longint  get_rand_int();
  extern virtual function int      parse_val(string val_string, ref longint val);
  extern virtual function string   val2string(longint val);
  extern virtual function string   convert2string();
endclass

function    acmdline_opts::new(string    name,    string    default_val="",    string
   description="");
  this.name        = name;
  this.description = description;
  this.default_val = default_val;
endfunction

/// function to set the dynamic array size based on number of arguments provided
function void acmdline_opts::set_len(int len);
  min = new[len];
  max = new[len];
  wt  = new[len];
endfunction

/// sets up the weight queue which is used by the rand_get_int method to generate a
/// random value
function void acmdline_opts::set_weight_q();
  for(int i=0;i<wt.size();i++) begin
    for(int j=0;j<wt[i];j++) begin
      weight_q.push_back(i);
    end
  end
endfunction

/// parse a single value string.  May be re-implemented in typed derived class to
/// handle enum values.
function int acmdline_opts::parse_val(string val_string, ref longint val);
  if($sscanf(val_string, "%d", val)) begin
    `uvm_info("parse_val", $sformatf("Processing %s as a DEC", val_string), UVM_DEBUG)
    return 1;
  end
  else if($sscanf(val_string, "h%x", val)) begin
    `uvm_info("parse_val", $sformatf("Processing %s as a HEX", val_string), UVM_DEBUG)
```

```
    return 1;
  end
  else if($sscanf(val_string, "b%b", val)) begin
    `uvm_info("parse_val", $sformatf("Processing %s as a BIN", val_string), UVM_DEBUG)
    return 1;
  end
  else begin
    return 0;
  end
endfunction: parse_val

/// return a string representation of a longint value.  May be re-implemented in
/// derived class to handle enum values
function string acmdline_opts::val2string(longint val);
  return $sformatf("%0d", val);
endfunction: val2string

/// parses the string and sets the min/max + wt
function bit acmdline_opts::parse_opt(string parse_string);
  string splits[$];
  string subsplits[$];

  if( (parse_string == "") && (default_val == "") ) begin
    return 1'b0;
  end
  else if( (parse_string == "") && (default_val != "") ) begin
    parse_string = default_val;
  end

  // splits into multiple sets of min/max or val + wt combo
  uvm_split_string(parse_string, ",", splits);

  // sets lengths based on how many values the user has provided
  this.set_len(splits.size());

  foreach(splits[i]) begin
    uvm_split_string(splits[i], ":", subsplits);

    `uvm_info("parse_opt",   $sformatf("subplit_size   =   %d   subsplit[0]   =   %s",
   subsplits.size(), subsplits[0]), UVM_DEBUG)

    if(uvm_is_match("*~*", subsplits[0])) begin  // indicates a min~max range
      string min_max_split[$];

      uvm_split_string(subsplits[0],"~",min_max_split);

      if(!parse_val(min_max_split[0],this.min[i])                                    ||
   !parse_val(min_max_split[1],this.max[i])) begin
        `uvm_error("parse_opt",  $sformatf("Invalid  args  %s  provided  for  %s  option",
   parse_string, this.name));
      end
    end
    else begin                                  // user just indicated a single value
      if(!parse_val(subsplits[0],this.min[i])  ||  !parse_val(subsplits[0],this.max[i]))
   begin
        `uvm_error("parse_opt",  $sformatf("Invalid  args  %s  provided  for  %s  option",
   parse_string, this.name));
      end

if(subsplits.size()>1) begin
      if(!$sscanf(subsplits[1],"%d",this.wt[i])) begin
        `uvm_error("parse_opt",  $sformatf("Invalid  args  %s  provided  for  %s  option",
   parse_string, this.name));
      end
    end
    else begin
      this.wt[i] = 1; //if  second  subsplit  does  not  exist  -  indicates  user  did  not
   specify a wt and a default wt of 1 is assigned
    end
  end

  set_weight_q();
```

```
  return 1'b1;

endfunction

/// returns a random value based on distribution provided by the user in terms of
/// weight. Returns without randomizing if there is only 1 value specified by the user
function longint acmdline_opts::get_rand_int();
  int     rand_index;
  int     winner_index;
  longint rand_int;

  if( (wt.size() == 1) && (min[0] == max[0]) ) return min[0];

  if (!std::randomize(rand_index) with {rand_index inside {[0:weight_q.size()-1]};} )
   begin
     `uvm_error("parse_opt",  $sformatf("Randomization  failed  for  rand_index  inside
   range:[0:%d]",weight_q.size()-1));
  end

  winner_index = weight_q[rand_index];

  if(min[winner_index] == max[winner_index]) return min[winner_index];

  if(!std::randomize(rand_int)    with    {rand_int    inside    {[min[winner_index]    :
   max[winner_index]]};}) begin
     `uvm_error("parse_opt",  $sformatf("Randomization  failed  for  rand_int  inside
   range:[%d:%d]", min[winner_index],max[winner_index]));
  end

  return rand_int;
endfunction


function string acmdline_opts::convert2string();
  string print_msg;

  print_msg = $sformatf("option_name : %s description : %s default_value : %s", name,
   description, default_val);
  foreach(this.wt[i]) begin
    print_msg     =     $sformatf("\n%s     min[%0d]=%s     max[%0d]=%s     wt[%0d]=%0d",
   print_msg,i,val2string(this.min[i]),i,val2string(this.max[i]),i,this.wt[i]);
  end

  return print_msg;
endfunction

`endif
```

APPENDIX II

```
  `ifndef __ACMDLINE_OPTS_ENUM_SV__
  `define __ACMDLINE_OPTS_ENUM_SV__

  ///
  /// Protected data class used to create an enum option entry in the database
  /// maintained by advanced_cmdline_processor
  /// Not intended to be used directly, handles created by advanced_cmdline_processor
  ///
  class acmdline_opts_enum#(type T=uvm_active_passive_enum)extends acmdline_opts_enum;

    extern         function        new(string name, string description="",
                                       string default_val="");
    extern virtual function int    parse_val(string val_string, ref longint val);
    extern virtual function string val2string(longint val);
  endclass

  function   acmdline_opts_enum::new(string   name,   string   description="",   string
default_val="");
    super.new(name, description, default_val);
  endfunction

  function int acmdline_opts_enum::parse_val(string val_string, ref longint val);
     T   enum_var;
     int rc;

     `uvm_info("parse_val", $sformatf("calling parse_val of enum class"), UVM_HIGH)

     rc = super.parse_val(val_string, val);
     if( rc != 0 ) begin
        // If we get here, the val_string is a numerical value.
        enum_var = T'(val);
        if(enum_var.name() != "") begin
          `uvm_info("parse_val", $sformatf("return rc!=0"), UVM_HIGH)
          return rc;
        end
        `uvm_error("parse_val",$sformatf("numerical value = %d is not a valid value
for the enum: %s\n", val, $typename(T)))
        return 0;
     end else if( uvm_enum_wrapper#(T)::from_name(val_string, enum_var)) begin
         val = longint'(enum_var);
         return 1;
     end else begin
        return 0;
     end
  endfunction: parse_val

  function string acmdline_opts_enum::val2string(longint val);
     T      enum_val = T'(val);
     string val_str  = enum_val.name();

     if(val_str == "") begin
        return super.val2string(val);
     end else begin
        return $sformatf("%s(%s)", val_str, super.val2string(val));
     end
  endfunction: val2string

  `endif // __ACMDLINE_OPTS_ENUM_SV__
```

APPENDIX III

```
`ifndef __ADVANCED_CMDLINE_PROCESSOR_SV__
`define __ADVANCED_CMDLINE_PROCESSOR_SV__

///
/// Class extending uvm_cmdline_processor provide the following:
///  - all methods of the uvm_cmdline_processor
///  - ability to add an user option to the data base
///  - both plus args and uvm_config_db support to extract values from cmdline
///
/// Use model:
///  - advanced_cmd_line_processor p = advanced_cmdline_processor::get_inst() will get
///    a handle to the singleton class instance
///  - call methods using the above handle
///  - get_rand_int() method can be called from any class/function within the UVM
///    framework.
///  - provide the unique name to be used to parse the cmdline and subsequently
///    access the database
///  - full_path,if provided will invoke the config_db call. Useful to specify options
///    from the command line for a particular instance
///  - plus arg format
///      - "+<option_name>=min1~max1:wt1,min2~max2:wt2,val3:wt3 and so on
///      - if wt is not specified then a default wt of 1 is assigned to that bucket of
///        min/max or val
///
class advanced_cmdline_processor extends uvm_cmdline_processor;

  local static advanced_cmdline_processor inst = null;
  ///< database of all options used in the test run with the name serving as the
  ///  unique key
  acmdline_opts acmdline_opts_db[string];

  extern        function         new (string name = "advanced_cmdline_proc");
  extern static function         advanced_cmdline_processor get_inst();
  extern        function void    add_opt(string name, string full_path="", string
                                         default_val="", string description="",
                                         acmdline_opts eopt=null);
  extern        function bit     exists_in_db(string name, string full_path="");
  extern        function bit     is_valid(string name, string full_path="");
  extern        function bit     parse_cmdline(string name, string full_path="",
                                               string default_val="", string
                                               description="", acmdline_opts
                                               eopt=null);
  extern        function string  return_final_arg(string name, string full_path="");
  extern        function longint get_rand_val(string name, string full_path="", string
                                              default_val="", string description="",
                                              acmdline_opts eopt=null);
  extern        function string  get_key(string name, string full_path);
  extern        function string  convert2string();

endclass

function advanced_cmdline_processor::new(string name = "advanced_cmdline_proc");
  uvm_cmdline_processor p = uvm_cmdline_processor::get_inst();

  //Initialize the protected variables to match uvm_cmd_processor.
  this.m_argv      = p.m_argv;
  this.m_uvm_argv  = p.m_uvm_argv;
  this.m_plus_argv = p.m_plus_argv;
endfunction
```

```
function advanced_cmdline_processor advanced_cmdline_processor::get_inst();
  if (inst == null) begin
     inst = new("advanced_cmdline_proc");
  end
  return inst;
endfunction // get_inst



/// user can add an opt to the database. The get_rand_int will access the default
/// value provided here in case a cmdline argument is missing
/// user can add an opt to the database. The get_rand_int will add it to DB incase no
/// called explicitly
function void advanced_cmdline_processor::add_opt(string name, string full_path = "",
   string default_val="", string description="", acmdline_opts eopt=null);
  void'(parse_cmdline(name, full_path, default_val, description, eopt));
endfunction

/// method to check if an option is already registered with the db. Function is also
/// used by parse_cmdline to check if it already exists in the database
function bit advanced_cmdline_processor::exists_in_db(string name, string
                                                      full_path="");
  if(acmdline_opts_db[get_key(name,full_path)] == null) return 0;
  return 1;
endfunction

/// method to indicate if control knob is passed as part of the test/cmdline
function bit advanced_cmdline_processor::is_valid(string name, string full_path="");
  if(return_final_arg(name,full_path) == "") return 1'b0;
  else return 1'b1;
endfunction

/// helper function which ideally will only be functioning once per option
function bit advanced_cmdline_processor::parse_cmdline(string name, string
   full_path="", string default_val="", string description="", acmdline_opts
   eopt=null);
  string final_arg;

  if(!exists_in_db(name,full_path)) begin
    acmdline_opts opt;
    if(eopt != null) begin
       opt = eopt;
    end else begin
       opt = new(name, default_val, description);
    end

    final_arg = return_final_arg(name,full_path);
    if(final_arg == "") final_arg = default_val;

    if(final_arg == "") return 1'b0;

    if(!opt.parse_opt(final_arg)) begin
      return 1'b0;
    end
    // adding it to database only if default value and/or value provided in cmdline is
    // not empty
    acmdline_opts_db[get_key(name,full_path)] = opt;
  end

  return 1'b1;
endfunction
```

```
/// helper function to parse cmdline and return final arg
function   string   advanced_cmdline_processor::return_final_arg(string   name,   string
   full_path="");
  string args[$];
  string final_arg;

  if(full_path=="") begin
    void'(get_arg_values($sformatf("+%s=",name), args));
    if(args.size()!=0) final_arg = args[args.size()-1];
    else               final_arg = "";
  end
  else begin // support for uvm_config_db - uvm_opts in command line
    if(!uvm_config_db#(string)::get(null, full_path, name, final_arg))
      final_arg = "";
  end

  `uvm_info("return_final_arg", $sformatf("final_arg = %s", final_arg), UVM_DEBUG)

  return final_arg;
endfunction

/// users will call this function to get a random/same value each time based on the
/// sets and distributions provided in default or overridden using the cmdline
function longint advanced_cmdline_processor::get_rand_val(string name, string
   full_path = "", string default_val="", string description="", acmdline_opts
   eopt=null);

  if(!parse_cmdline(name, full_path, default_val, description, eopt)) begin
    `uvm_error("get_rand_val", $sformatf("No valid argument provided for option : %s
   ", get_key(name, full_path)))
    return 0;
  end

  return acmdline_opts_db[get_key(name,full_path)].get_rand_int();
endfunction

/// locally used function to concatenate the option name and path
function string advanced_cmdline_processor::get_key(string name, string full_path);
  if(full_path == "") return name;
  return {full_path,".",name};
endfunction

/// returns print information of all the options in the database
function string advanced_cmdline_processor::convert2string();
  string print_msg;

  foreach(acmdline_opts_db[i])
    print_msg = {print_msg, acmdline_opts_db[i].convert2string()};

  return print_msg;
endfunction

`endif
```

APPENDIX IV

```
`ifndef __ADVANCED_ENUM_OPT_PROC_CLASS_SV__
`define __ADVANCED_ENUM_OPT_PROC_CLASS_SV__

/// Wrapper class for managing enum command line options.
class advanced_enum_opt_proc#(type T=uvm_active_passive_enum);

  extern static function T    get_rand_enum(string name, string full_path="", string
                                            default_val="", string description="");
  extern static function bit  is_valid(string name, string full_path="");
endclass

/// Get a random enum value from the command line.
function advanced_enum_opt_proc::T  advanced_enum_opt_proc::get_rand_enum (string
   name, string full_path="", string default_val="", string description="");
  advanced_cmdline_processor   opt_proc = advanced_cmdline_processor::get_inst();
  acmdline_opts_enum#(T)       eopt;

  if(!opt_proc.exists_in_db(name, full_path)) begin
    eopt = new(name,default_val,description);
  end

  return T'(opt_proc.get_rand_val(name, full_path, default_val, description, eopt));
endfunction

/// method to check if arg is passed as part of the test/cmdline
function bit advanced_enum_opt_proc::is_valid(string name, string full_path="");
  advanced_cmdline_processor   opt_proc = advanced_cmdline_processor::get_inst();

  return opt_proc.is_valid(name,full_path);
endfunction

`endif //__ADVANCED_ENUM_OPT_PROC_CLASS_SV__
```