2025 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA

A Hybrid Functional Verification Approach of complex designs using Python based Models

Nirmal Kumar (nirmal.k@samsung.com), Joshi Pujaben Dishit(puja.joshi@samsung.com),
Vinay KH(vinay.kh@samsung.com), Kuntal Pandya(k.pandya@samsung.com),
Anil Deshpande(anil.pande@samsung.com)

Abstract: **The escalating complexity of modern IP and SoC designs has given rise to increasingly intricate reference models. These models are critical for replicating design behaviour and guaranteeing correctness, but their development can be a time-consuming and laborious process. Fortunately, Python's simplicity and flexibility make it an ideal choice for crafting these models with greater speed and concision. In this paper, we delve into the various methods for bridging the gap between Python and System Verilog, exploring their respective advantages and disadvantages. By examining the trade-offs between these approaches, we aim to provide a comprehensive understanding of the best practices for efficient verification flows.**

I. INTRODUCTION

The world of integrated circuits (ICs) is on the cusp of a revolution. The insatiable demand for artificial intelligence (AI) and high-performance computing (HPC) has sent chip design soaring to new heights of complexity, incorporating an array of functionalities that push the boundaries of miniaturization. As we strive to unlock even greater efficiency and innovation, one bold approach is emerging i.e., merging the mighty Python programming language with the traditional System Verilog hardware design environment. Python has its own advantages with rich availabilities of libraries like **NumPy, Pandas, PyTorch** to perform various tasks. It has simpler syntax and more readable code.

The reference models created from Python can be leveraged in multiple ways. From an RTL (Register-Transfer Level) design perspective, a complex module can first be prototyped in Python, serving as a reference point for implementing the equivalent Verilog code. Alternatively, from a DV (Design Verification) standpoint, we can create a golden reference model to validate specific functionalities. This allows us to apply test stimuli simultaneously to both the reference Python model and the actual Verilog design, facilitating effective comparison and validation. The output from both the models can be given to scoreboard to check the data integrity. In this paper, we delve into the exciting possibilities of this integration, showcasing a novel method for seamlessly weaving Python capabilities into the heart of System Verilog and complete with real-world examples that demonstrate the transformative potential of this hybrid approach.

II. INTEGRATING PYTHON WITH SYSTEM VERILOG TESTBENCH

In modern hardware verification, integrating a Python model with a System Verilog testbench is essential. This paper proposes an efficient hybrid methodology for interacting between System Verilog and Python, and compares it with traditional methods.

A)Command line approach:

In SV testbench, the command line method is used to pass parameters directly to the Python model during simulation. There are two approaches to achieve this. Below is an overview of how these methods function:

1) *Method 1 (Array Based Argument):* In this approach, arrays are directly passed as arguments to the simulation within the System Verilog testbench. The **"$system"** system task is utilized to invoke the Python model from within the simulation environment. The Python model can then retrieve the provided

data using the **"sys.argv"** method, enabling it to access command-line arguments that have been passed by the System Verilog testbench.

```
//populating array
initial begin
for (int i=0;i<6500;i++) begin
  exp_array[i] = i;
end
#5ns;

fd =  $fopen("Python_model.py" , "r");
//converting array to string
line = convert2string(exp_array);
//Invoke python model with array
//command line argument
$system($sformatf("python3 Python_model.py %s" , line));
$fclose(fd);
```

Figure 1. Passing array via command line method

Fig. 1 shows how to pass an array or parameters from command line argument to the System Verilog testbench using **"$system"** task and call a python model.

Disadvantages:

- Command line length limitations make it difficult to manage large arrays.

2) *Method 2 (File I/O Based Argument):* To address the limitations of the previous method (*Array Based Argument*), we employ an alternative strategy that utilizes file input/output operations to pass arguments. In this approach to call a Python model from a System Verilog testbench, data exchange takes place through intermediary files. The "$system" task is then used to invoke the Python model, passing the file path as an argument. The Python script reads the input file, processes the data, and writes the results to another file which read back by system Verilog for further operations. Application for the same is explained below:

- *FEC and CRC algorithm verification:* In MIPI Unipro-3.0 IP, the link layer makes use of Forward Error Correction (FEC) and Cycle Redundancy Check (CRC) algorithm using Reed Solomon encoder to detect and correct errors in Datapath. We utilize a Python-based reference model to validate the correctness of the FEC and CRC implementations within our design.

```
fd = $fopen("fec_input.txt" , "w");
`uvm_info(get_type_name() , $sformatf("File created" ) , UVM_HIGH)

for(int i=0;i<=24;i++)
begin //{
    $fwrite(fd , "%d\n" , unipro_intf.ip_fec_data[0][i][7:0]);
end //}

$fclose(fd);

`uvm_info(get_type_name() , $sformatf("Fec input file closed" ) , UVM_HIGH)

`uvm_info(get_type_name() , $sformatf("Calling python script" ) , UVM_HIGH)

$system($sformatf("python3 $TB_HOME/tb/top/rs_fec_crc.py %s" , "fec_input.txt"));

`uvm_info(get_type_name() , $sformatf("Python script ended" ) , UVM_HIGH)
```

Figure 2. Calling the python model using $system task

```
fec_file = sys.argv[0];
data_stream = []
with open(fec_file , 'r') as rd_file:
  lines = rd_file.readlines()
  for line in lines:
    data_stream.append(line.strip())

s0_temp = 0xff
s1_temp = 0xff
s2_temp = 0xff

for i in range(0 , 241):
  s2_temp = add(s0 , data_stream[i])
  s1_temp = add(s2 , mpy(235 , s7_temp))
  s0_temp = add(s1 , mpy(164 , s7_temp))
```

Figure 3. Actual python model to implement CRC 64 algorithm

To verify our System Verilog implementation of a CRC-64 algorithm, we created a Python model that was called from within the System Verilog code using the "**$system**" task (Fig. 2). The input/output data was exchanged through text files (Fig. 3). This hybrid approach enabled us to quickly identify and fix RTL bugs by leveraging the strengths of both languages. While this approach may have several advantages, it also has some significant drawbacks:

- Limitations prevent direct access to specific Python functions within the process.
- I/O operations (reading and writing to files) introduce unnecessary overhead, which can slow down performance.
- Storing data in files leads to increased memory usage, potentially causing resource constraints.

The generic way of command line approach (Method 1 and Method 2) is depicted in the form of flow chart in below Fig. 4.

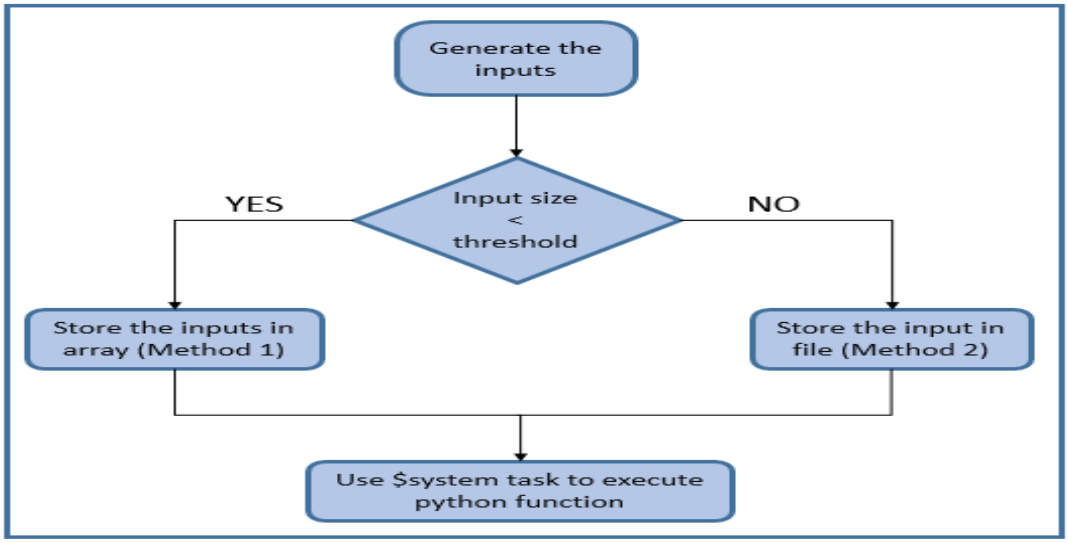

Figure 4. Command Line Method

B) DPI (Direct programming interface):

To address the limitations of memory utilization and accessibility of internal python functions described in above method (Command line approach – *Section 2.1*), we propose the DPI(Direct Programming Interface) method to interact between System Verilog and Python. DPI is a set of functions or procedures that allow functions coded in one language to be called from another language. In the context of SV DPI, "import" and "export" keywords are used for function exchange between SV and foreign languages. This approach leverages the strengths of both SV for hardware description and Python for computational tasks.

While the Language Reference Manual (LRM) for SV DPI (SystemVerilog Direct Programming Interface) restricts foreign languages to only C or C++, users seeking to interact with SystemVerilog from Python encounter a limitation, as no direct solution is offered. To circumvent this constraint, developers can employ an intermediate layer written in C, which serves as a bridge between the SystemVerilog code and the Python environment.

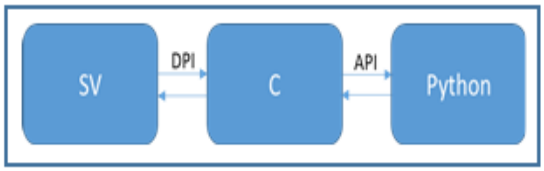

Figure 5. SV to Python integration

As shown in Fig. 5, SystemVerilog code employs the DPI (Direct Programming Interface) method to pass required parameters to a C function. The C function leverages the API (Application Programming Interface) provided by the Python model to establish communication between the two environments. Upon receiving input data from the SystemVerilog testbench, the Python model processes this information and returns a result in the form of a "**PyObject**" object. This result is then received and converted back into a format compatible with SystemVerilog by the C wrapper function. Finally, the C function transfers control back to the SystemVerilog testbench along with the processed output data from the Python model.

1) Pseudo Code for DPI Implementation:

a) Python Code Function: In Fig. 6, the "py_func" Python function is shown to be called by a corresponding C-code function.

Py Function

```
def py_func(nbitsdump , pList1 , cdr_off , num1):
    for i in range(0 , len(num1) , 10):
    temp_data = 0
    for j in range(10):
      temp_data = temp_data << 8 | x[i+j]
   Outsig_signed1 , Outsig_unsigned1  = Analog_Model.AnalogModel_Wrapper(pList1)
   Outsig_signed2 , Outsig_unsigned2  = Analog_Model.AnalogModel_Wrapper(pList1)
   Outsig_dp1_list = Outsig_dp1.tolist()
   Outsig_dp2_list = Outsig_dp2.tolist()
   return Outsig_dp1_list  , Outsig_dp2_list
```

Figure 6. Example python function which is called by C code

b) C-Code Function: **"sv_to_python"** function shown in Fig. 7 illustrates the master C code function which includes calling python funtions and reading the returned values in C-code.

```
void sv_to_python (char *file1 , int nbitsdump , int num1 , int arr_h[4] ,
                   int cdr_off , int hex_arr[1000] , svOpenArrayHandle res_arr00  , int *arr0_size)
{
   PyObject *name , *loadmodule , *func , *callfunc , *args ;
   PyInitialize();
   calling_python_func(file1 , nbitsdump  , num1 , arr_h , cdr_off  , hex_arr);
   reading_python_values(file1 , res_arr00 , &arr0_size);
   PyFinalize();
}
```

Figure 7. Master C Code Function

- The "**Py_Initialize**" and "**Py_Finalize**" functions create a channel between C and Python.
- The definition of calling_python_func and reading _python_values functions are explained further.

c) Calling Python Function : The C code function "**calling_python_func**" which is used to call the actual Python function "**py_func**" is shown in Fig. 8 .

```
void calling_python_func(char *file1 , int nbitsdump , int num1 , int arr_h[4] ,
                         int cdr_off , int hex_arr[1000])
{
  int hex_length = 1000;
  PyObject* pList1 = PyList_New(hex_length);
  for (int i = 0; i < hex_length; i++) {
  PyObject* pItem1 = PyLong_FromLong(hex_arr1[i]);
  PyList_SetItem(pList1, i, pItem1);
    }
  name = PyUnicode_FromString("conv_8_to_80b");
  loadmodule = PyImport_Import(name);
  func = PyObject_GetAttrString(loadmodule , "py_func");
  args = PyTuple_Pack(4 , nbitsdump , pList1 , cdr_off , num1);
  callfunc = PyObject_CallObject(func , args);
}
```

Calling Python function

Figure 8. Calling python function in C Code

- The list is created for a definite length using "**PyList_New**" function. The integer variable which is received as input from SV is converted to "**PyObject**" using "**PyLong_FromLong**" function.
- The list and individual variables are stored in terms of "**PyObject**".
- The **"PyImport_Import"** function is used to import the required Python module.
- The input arguments are passed from C using the "**PyTuple_Pack**" function.
- The "**PyObject_CallObject**" function is used to call the required Python function and execute it.

d) Return values from Python: The function to return the values from Python model through C-Code is done by calling various C-API(Application Programming Interface) functions as explained below:

- The "**svOpenArrayHandle**" data type is used to store the returned array values. The pointer variable *arr0_size can be used to store the size of the array.
- Once the python function is executed, the returned output variables are stored in PyObject type variable "**callfunc**".
- The returned values are stored in each PyObject variable using "**PyTuple_GetItem**" function. In Fig. 8, the second argument of "**PyTuple_GetItem**" is 0, which indicates the first returned output value from Python.
- If there are multiple returned values, the second argument will be used in incremental order accordingly
- The output array is in "**PyObject**" Type. So "**PyList_AsLongArray**" function is used to convert it to normal array which can understandable by SV.

The implementation of C code to read the returned values from python is shown in Fig.9

```
void reading_python_values(char *file1 , svOpenArrayHandle res_arr00 , int *arr0_size)
{
   int* res_arr0;
   int size0;

   res_arr0 = (int *)svGetArrayPtr(res_arr00);

   callfunc = PyObject_CallObject(func , args); //call the python function using CallObject method

   PyObject* pArray0 = PyTuple_GetItem(callfunc , 0); // to read the data returned from python

   int *arr0 = PyList_AsLongArray(pArray0 , &size0);
      *arr0_size = size0;

      for(int l=0;l<*arr0_size;l++)
      {
        *(res_arr0+l) = arr0[l];
      }
}
```

Figure 9. C code function to read the returned values from Python

e) System Verilog Code: As shown in Fig. 10, the SV code consists the "**import"** keyword for the C code function to be called. In the below example **"sv_to_python"** is the name of C code function. The direction of the variables is by default "input". In the example two arguments are having direction as "output" whose values will be assigned in the C function based on the values returned from Python code.

```
import "DPI-C" context function void sv_to_python(string file1 ,int nbitsdump, int num1 ,
int arr_h[4] , int arr_l[4] ,  int hex_arr[1000] , int hex_arr1[1000] , output int size);

class sv_dpi_c extends uvm_monitor;

  `uvm_component_utils_begin(sv_dpi_c)
   `uvm_component_utils_end

   string file_name ;
   int num , nbitsdump , size;

   int hex_arr1[2][1000];
   int arr_h[2][4];

   virtual task run_phase(uvm_phase phase);

     sv_to_python("File.txt" , nbitsdump , num , arr_h[0] , arr_h[1] ,
                  hex_arr1[0] , hex_arr1[1] , size );

   endtask
endclass
```

Figure 10. SV code calling the C code function using DPI method

Hence, Fig. 6, 7, 8, 9 and 10 shows the seamless integration of the python code with System Verilog, where C code act as an intermediary bridge.

C) Comparison of DPI with FILE I/O Method:

For any complex IP design, the Clock and Data Recovery (CDR) block is of utmost importance in achieving accurate clock recovery at the receiver end. This component is responsible for extracting the clock signal from the incoming data stream, which is essential for proper data synchronization and error-free communication. The amount of data generated by the UVC can be quite substantial, while performing stress checks on CDR module.

If the FI LE I/O method is employed, all the data from the UVC would need to be dumped into a text file, which consume a significant amount of memory, especially when dealing with large volumes of data.

By adopting the proposed DPI method, the need for additional text files to store the input data is eliminated. Instead, an array can be utilized to pass the required data directly to the Python function, significantly reducing memory consumption and improving efficiency.

Advantages of DPI over FILE I/O method:

- Flexibility: Allows calling foreign language functions from SV.
- Memory Efficiency: Saves memory by not requiring files for input and output.
- Function Call ability: Enables calling internal functions from Python code.

III. Application And Result

The application of this method is to enable the seamless integration of Python models into System Verilog test benches for hardware verification by enabling the use of Python's extensive libraries and capabilities in hardware design and verification. Some of the realtime applications are:

1) *Serdes DSP Verification:* In high-speed SERDES (Serializer/Deserializer) IP, the transmitter (TX) and receiver (RX) DSP (Digital Signal Processing) models are implemented in both Python and Verilog. The purpose is to verify whether the outputs of the Verilog and Python DSP models are exactly the same for the same set of inputs. This helps to ensure the accuracy and reliability of the Python model, which can then be used as a powerful tool for testing and verifying complex algorithms before they are ported to the Verilog model. By using Python as an intermediate layer, designers can save time and effort by identifying and fixing bugs early in the development process, ultimately leading to more efficient and reliable hardware designs.

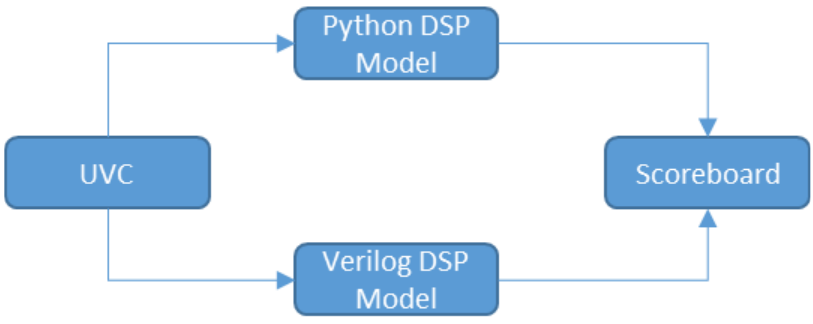

Figure 11. TX/RX DSP Python and Verilog model with UVC and scoreboard

Result: A regression test of 1000 tests was conducted using both the file I/O method and the DPI method. The results are as follows:

a) The result shown in Table 1, demonstrates the advantage of the DPI method in terms of memory efficiency, making it a more suitable choice for large-scale hardware verification projects

Table 1
Memory usage with both methods

Method	Single Test (in MB)	Regression (in MB)
File I/O	13	13000
DPI	0	0

b) Table 2, shows significant difference in runtime highlights he Python model's advantage in terms of speed

Table 2
Simulation time comparison

Reference model	Simulation time (per test) (hours)
Third party model	7:23
Python model	4:11

IV. CONCLUSION

In conclusion, this paper presented the efficient methods for integrating Python into the System Verilog environment. By seamlessly bridging the gap between Python and System Verilog, we've unlocked a game-changing approach to chip design. Our innovative integration method harnesses the best of both worlds, empowering designers to tackle even the most complex projects with unprecedented speed and precision.

V. REFERENCES

[1] System Verilog, Language Reference Manual – IEEE 1800-2003
[2] https://docs.Python.org/3/library, The Python Standard Library
[3] mipi_Unipro_v_3_0_Specifcation