A Novel Approach to Standardize Verification Configurations using YAML

Nikhil Tambekar, Technical Specialist, Nokia Solutions and Networks Oy, Tampere, Finland, (*nikhil.tambekar@nokia.com*)

***Abstract*— With the increasing complexity of System-On-Chip designs, hardware verification has evolved into a multidimensional challenge. Balancing first pass silicon and a fast time to market is delicate but essential consideration for the SOC organizations. Verification has been always in critical path in the SOC development and any improvements in verification execution efficiency has exponential effect on the quality of the SOC. Effective configuration management of different areas in the verification environment combined with automation scripting helps to achieve appreciable boost in the execution efficiency. In this paper, a single YAML format is proposed to handle all the configurations in the verification environment. Using a standardized configuration format along with scripting saves verification time by an order of magnitude and makes the environment more reusable and maintainable across the projects.**

Keywords*— *System-On-Chip, Verification, Configuration, YAML

I. INTRODUCTION

The scope of verification environment encompasses a comprehensive set of tasks including testbench development, various tool integration, automation scripting, regression setup and Continuous Integration (CI).

A verification environment consists of three key components which includes firstly, a testbench to simulate testcases for desired Design Under Test (DUT) configurations, which involves configuring testbench using simulation runtime options, controlling randomization constraints, functional coverage monitors etc. Secondly, the environment includes configuring RTL, establishing testbench build system, creating models for generating reference data, automating testcase generation and setting up regressions. Thirdly, scripts are used for processing simulation results, parsing log files to aid debugging and generating coverage reports. Additionally, verification engineers play a crucial role in setting up CI workflows, creating verification plan, and status reporting. It involves learning different tools, inhouse methodology, and learning scripting languages. The modern-day IPs and SOCs are configurable and developing a user-configurable verification environment is a necessity for maintainability and reusability across multiple projects.

II. CHALLENGES ASSOCIATED WITH CONFIGURATION

As stated earlier, configurability plays a pivotal role in the verification environment. However, configurations are unique for each component of the environment and currently there is no standard way of defining the verification configurations. Owing to the non-standard formats, automation scripting becomes more complicated and takes more effort for the development because engineers need to define custom configuration format. Consequently, engineers may opt for manual methods which, unfortunately can lead to decreased execution efficiency and an environment vulnerable to mistakes. Moreover, custom configurations can detrimentally impact the reusability and maintainability of scripts and tools. These user-specific configurations also contribute to a steeper learning curve for new team members in familiarizing the setup and carrying out enhancements. To summarize, standardization of a configuration format emerges as a crucial requirement in SOC organizations to enhance execution efficiency.

III. PROPOSED SOLUTION FOR HANDLING VERIFICATION CONFIGURATIONS

In this paper, a YAML (YAML ain't markup language) format is proposed for representing all the configuration requirements in the verification environment which is depicted in Figure 1.

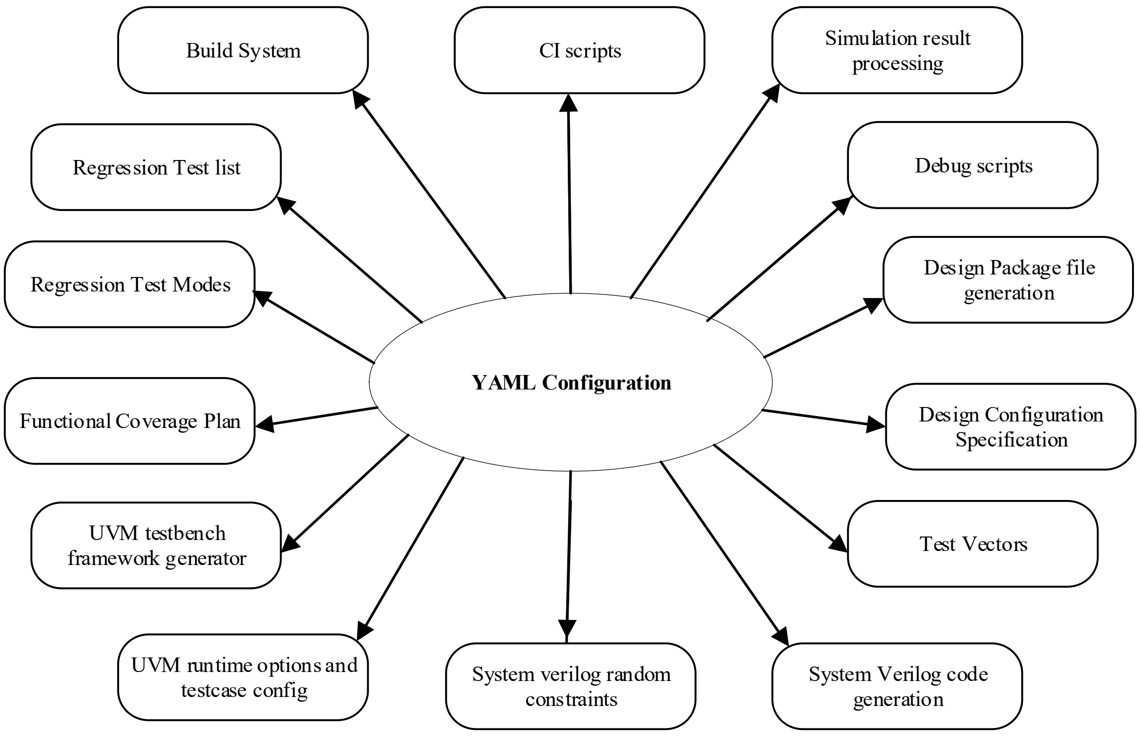

YAML Configuration
Build System
CI scripts
Simulation result processing
Regression Test list
Debug scripts
Regression Test Modes
Design Package file generation
Functional Coverage Plan
Design Configuration Specification
UVM testbench framework generator
Test Vectors
UVM runtime options and testcase config
System verilog random constraints
System Verilog code generation

Figure 1 YAML based configuration in verification environment

This paper comprises of brief introduction to YAML format and templating language, example implementation of how a single format is applied to different verification environment setups, future enhancements, and conclusion.

IV. YAML FORMAT FOR CONFIGURATIOONS

YAML (YAML Ain't Markup Language) is a human-readable data serialization format often used for configuration files and data exchange between languages with different data structures. It is often used in software development, particularly for specifying configurations, settings, or data structures that need to be human-readable but also machine-parseable. Key features of YAML are simple syntax, supports various data types which are suitable for cross language data exchange and programming language agnostic. Some examples of YAML configuration formats are shown in Figure 2 below:

```
#Interface definition in YAML
input_data_if:
      Interface_type: axi4
      Data_width: 128
      Min_addr: 0x1000
      Max_addr: 0xffff

Output_data_if:
      Interface_type: axi_stream
      Data_width: 128
      tkeep_support: Yes
```

```
#YAML test configuration
files:
      input_file: in.bin
      output_file: in.hex

variables:
      test_name: smoke.test
      test_params: cfg_mode0
      random: True
```

Figure 2 YAML file configuration format

XML, JSON and YAML are commonly used formats for representing structured data. XML is a markup language, whereas JSON and YAML are data serialization formats. XML uses tags to define the elements and stores data in hierarchical structures, whereas data in JSON is stored like a map with key/value pairs. YAML, on the other hand, allows representation of data both in list or sequence format and in the form of a map with key/value pairs.

XML is used for data interchange (that is, when a user wants to exchange data between two applications). JSON is better as a serialization format and is used for serving data to application programming interfaces (APIs). YAML is best suited for configuration.

V. Proposed Implementation of Confgurations

Figure 3 below illustrates how a YAML file parsing library and a templating language like Jinja2, can be used to process the configurations in YAML for verification environment which would help in the automation process.

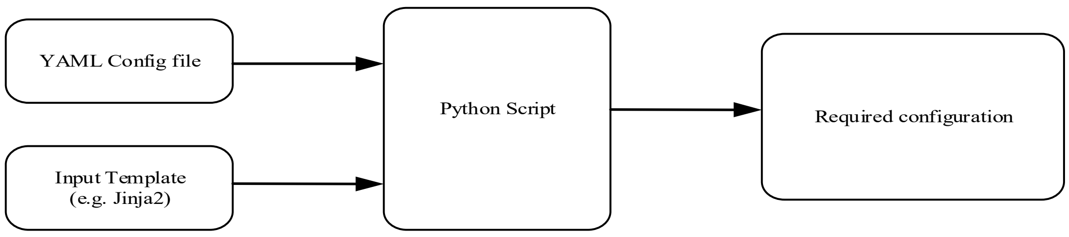


Figure 3 Scripting Setup using YAML and template

A templating engine, such as Jinja2, is a library that enables the separation of code and content. It allows developers to define templates with place holders for dynamic data, which can be populated at run-time. Special placeholders in the template allow writing code like Python syntax as shown below in Figure 4.

```
Module {{module_name}}_top #Variable
example
{% if loop.index is divisibleby 3 %}
#condition Code
{% endif %}

{% for item in seq %} #Looping
    <li>{{ item }}</li>
{% endfor %}
```

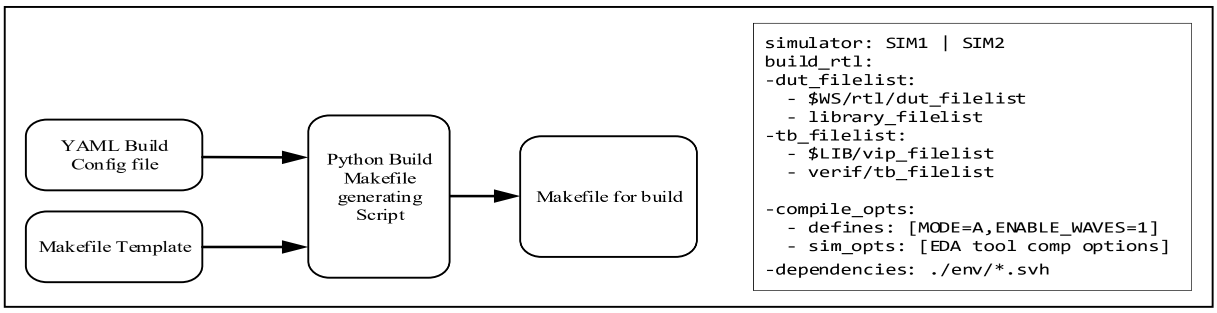
```
from jinja2 import Environment
env = Environment()
template=env.get_template('regr_list.tpl')
config=yaml.safe_load(open('./
regr_cfg.yaml','r'))
test_list = template.render(config)
```

Figure 4 Jinja2 Template and Python code

VI. YAML Configuration Examples In Verification Environment

A. Build System

Build system is used to transform the source code written by engineers into executable binaries. GNU makefile is commonly used in verification environment to build simulator executables. The build complexity increases at subsystem and SOC level, where multiple IPs and libraries are reused and writing a Makefile becomes a tedious job. To overcome this problem, the build specification can be represented using YAML format and the makefile syntax in a template format. Using a YAML and template processing script, the Makefile can be generated. This approach can speed up the build setup and improves maintainability and ease of defining a build. Figure 5 below is a sample representation of a build specification.



```
simulator: SIM1 | SIM2
build_rtl:
-dut_filelist:
   - $WS/rtl/dut_filelist
   - library_filelist
-tb_filelist:
   - $LIB/vip_filelist
   - verif/tb_filelist

-compile_opts:
   - defines: [MODE=A,ENABLE_WAVES=1]
   - sim_opts: [EDA tool comp options]
-dependencies: ./env/*.svh
```

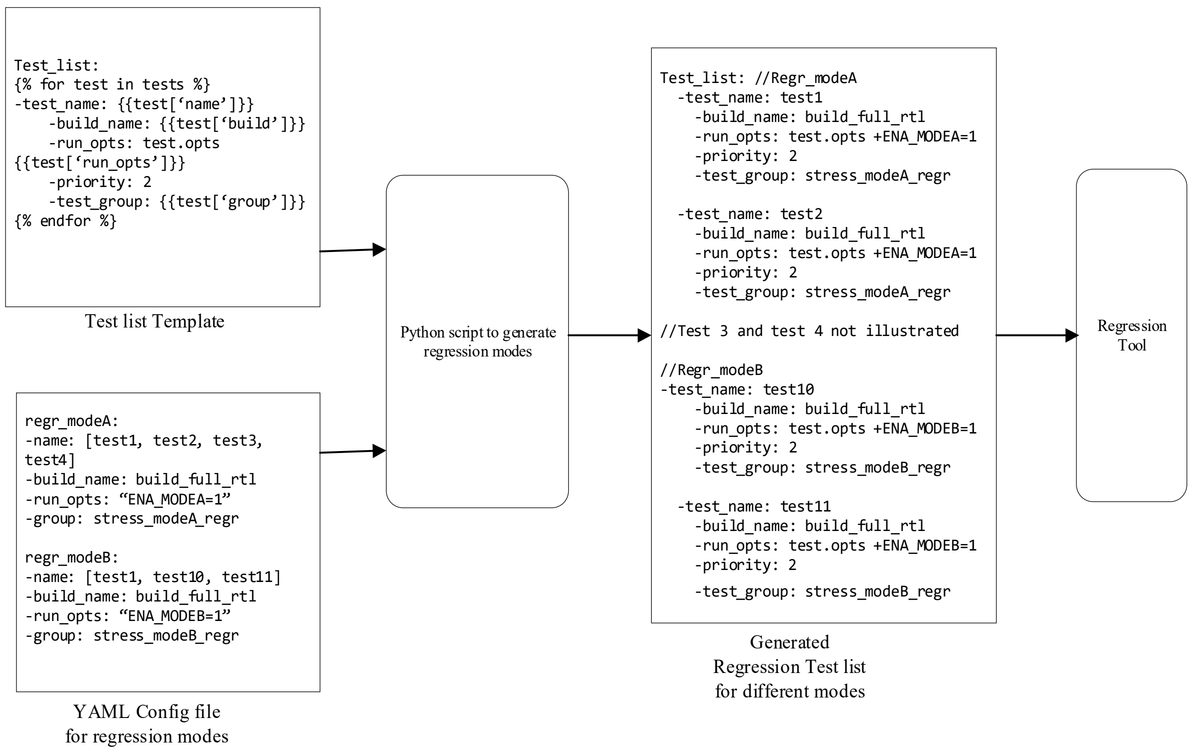
Figure 5 YAML Config and Template Based Build flow

B. Regression Test list and Regression mode specification

Regression is a key element of verification process which involves a collection of testcases and simulating the tests regularly on the DUT with different randomization seeds and DUT configurations to achieve coverage. A typical regression test list would contain approximately thousands of tests with a combination of simulation runtime options. Developing a test list and maintaining many tests for regression is a time-consuming effort and prone to mistakes.

To achieve coverage goals and ensure that DUT is verified in all the configurations, testcases need to be run in multiple regression modes. It means testcases in a base test list are simulated for different DUT features using another set of runtime arguments. When the base test list grows, creating multiple variations from it would be challenging. Also, if any changes are needed in the tests, they would be done at multiple places in the regression lists which causes hardship to engineers, and it can lead to missing certain crosses of the configurations.

A proposed method to overcome this limitation is to specify a base test list and different DUT configurations in the YAML format which is illustrated in Figure 6 below. It leaves engineers to maintain only one base test list and regression modes making the regression maintenance manageable. Further efficiency improvement could be achieved using a templating language to represent the base test list. Leading EDA vendors have been using YAML format in their regression tool offerings.


Test list Template
YAML Config file for regression modes
Python script to generate regression modes
Generated Regression Test list for different modes
Regression Tool

```
Test_list:
{% for test in tests %}
-test_name: {{test['name']}}
    -build_name: {{test['build']}}
    -run_opts: test.opts
{{test['run_opts']}}
    -priority: 2
    -test_group: {{test['group']}}
{% endfor %}
```

```
regr_modeA:
-name: [test1, test2, test3,
test4]
-build_name: build_full_rtl
-run_opts: "ENA_MODEA=1"
-group: stress_modeA_regr

regr_modeB:
-name: [test1, test10, test11]
-build_name: build_full_rtl
-run_opts: "ENA_MODEB=1"
-group: stress_modeB_regr
```

```
Test_list: //Regr_modeA
  -test_name: test1
    -build_name: build_full_rtl
    -run_opts: test.opts +ENA_MODEA=1
    -priority: 2
    -test_group: stress_modeA_regr

  -test_name: test2
    -build_name: build_full_rtl
    -run_opts: test.opts +ENA_MODEA=1
    -priority: 2
    -test_group: stress_modeA_regr

//Test 3 and test 4 not illustrated

//Regr_modeB
-test_name: test10
    -build_name: build_full_rtl
    -run_opts: test.opts +ENA_MODEB=1
    -priority: 2
    -test_group: stress_modeB_regr

  -test_name: test11
    -build_name: build_full_rtl
    -run_opts: test.opts +ENA_MODEB=1
    -priority: 2
    -test_group: stress_modeB_regr
```

Figure 6 Regression mode automation using YAML and template

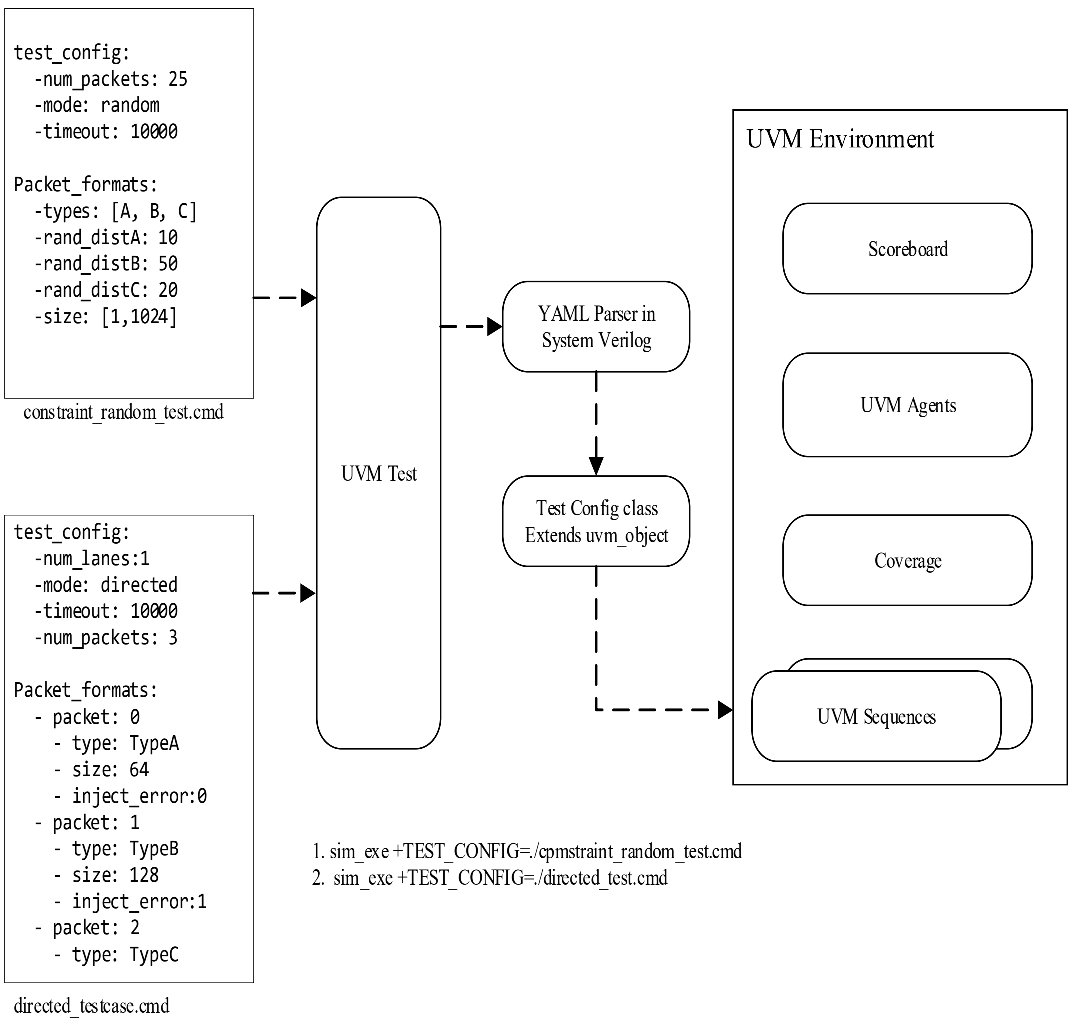
C. UVM Testbench Simulation time argument control

System Verilog UVM (Universal Verification Methodology) testbench is made configurable using runtime options. The runtime options are used for controlling testbench parameters, DUT features and randomization constraints without the need of recompilation. The literature on UVM methodology discourages writing functional code in testcases. However, a clear guidance on writing configurable testcases is missing. The UVM command line processor and plusargs provide limited capability to specify configurations and maintaining a large set of runtime options is a challenge.

As shown in Figure 7, the YAML format can be used to specify testcase configurations. A generic System Verilog config class (extended from uvm_object) can be developed to parse YAML and map the configurations to SV class variables. This SV class objects can then be used in the testbench to enable scenario-specific sequences and to provide constraint hooks for randomization. It would make the UVM testbench more reusable for multiple configurations.

The advantages of using YAML based testcase definition are as follows:

- It allows writing both directed and random test scenarios.
- Testcases can be written and modified without recompiling the testbench code.
- Vertical reuse (IP to Subsystem) would be easier because the testcase defines only test scenarios.
- Language independent testcase specification
- Testcase generation can be easily automated for configuration variations.



```
test_config:
  -num_packets: 25
  -mode: random
  -timeout: 10000

Packet_formats:
  -types: [A, B, C]
  -rand_distA: 10
  -rand_distB: 50
  -rand_distC: 20
  -size: [1,1024]
```

constraint_random_test.cmd

```
test_config:
  -num_lanes:1
  -mode: directed
  -timeout: 10000
  -num_packets: 3

Packet_formats:
  - packet: 0
    - type: TypeA
    - size: 64
    - inject_error:0
  - packet: 1
    - type: TypeB
    - size: 128
    - inject_error:1
  - packet: 2
    - type: TypeC
```

directed_testcase.cmd

```
1. sim_exe +TEST_CONFIG=./cpmstraint_random_test.cmd
2. sim_exe +TEST_CONFIG=./directed_test.cmd
```

UVM Test
YAML Parser in System Verilog
Test Config class Extends uvm_object
UVM Environment
Scoreboard
UVM Agents
Coverage
UVM Sequences

Figure 7 YAML testcase used in SV UVM Testbench

D. *YAML based design specification for automation in the verification environment*

As the SOC designs have become more complex, the process of interpreting architecture into design parameters and converting specification into a design has become cumbersome. In case of specification updates, the changes need to be made in all the environment components which further delays the execution, and the manual changes may introduce bugs in the system.

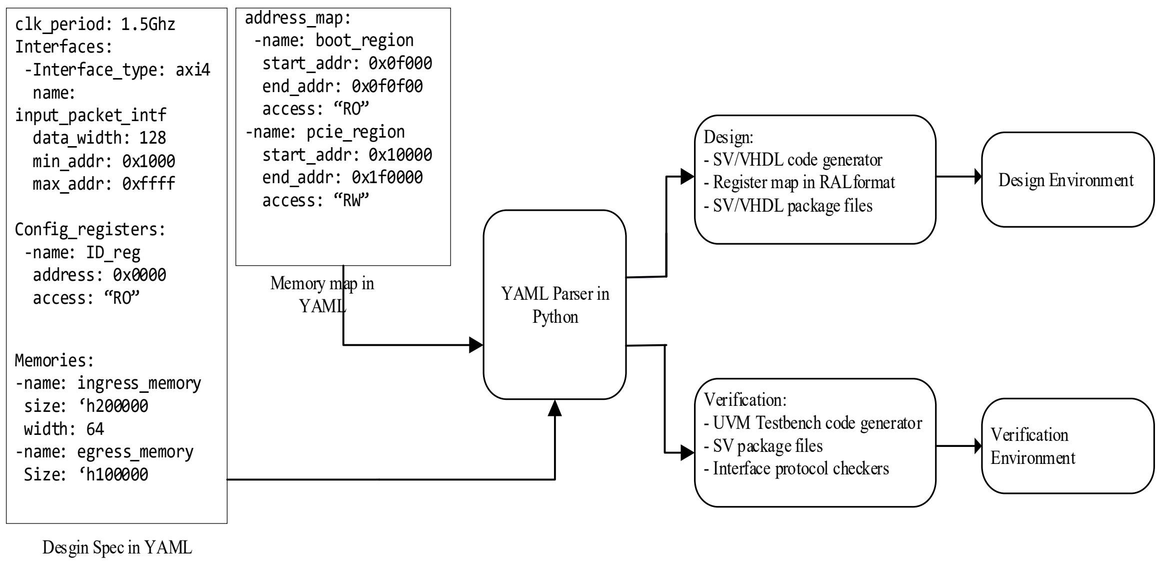
Machine readable specifications can facilitate design and verification automation more effectively. The proposed method is to represent the design micro-architecture specifications in the YAML format which includes design interfaces, memory configurations, config registers, instruction formats, packet formats, data structures, and project specific address map. The configuration file can be maintained in the project repository as a single point reference for the specification. As the specification is available in the machine-readable format, automation scripts can consume the YAML configuration file to generate the design and verification environment parameters as shown in Figure 8. This approach makes the environment more scalable and easier to maintain while handling large set of parameters which further improves execution efficiency and quality of deliverables. The YAML based specification file can be used for various utilities in the project such as:

1) Design Environment:

- Creating package files in VHDL or System Verilog.
- Code generator for Configurable RTL modules/entities.
- Generating module instance parameters for design integration.
- Representing IO cells to create SOC top level entity.
- Code generation for Software configuration register blocks in RTL.

2) Verification Environment

- SV/UVM testbench framework generator code including VIP configuration.
- Creating package files, data structures for testbench.
- Interface specification for protocol checkers, formal verification setup.
- Extracting parameters for functional coverage implementation



```
clk_period: 1.5Ghz
Interfaces:
 -Interface_type: axi4
  name:
input_packet_intf
  data_width: 128
  min_addr: 0x1000
  max_addr: 0xffff

Config_registers:
 -name: ID_reg
  address: 0x0000
  access: “RO”


Memories:
-name: ingress_memory
 size: ‘h200000
 width: 64
-name: egress_memory
 Size: ‘h100000
```

```
address_map:
 -name: boot_region
  start_addr: 0x0f000
  end_addr: 0x0f0f00
  access: “RO”
-name: pcie_region
  start_addr: 0x10000
  end_addr: 0x1f0000
  access: “RW”
```

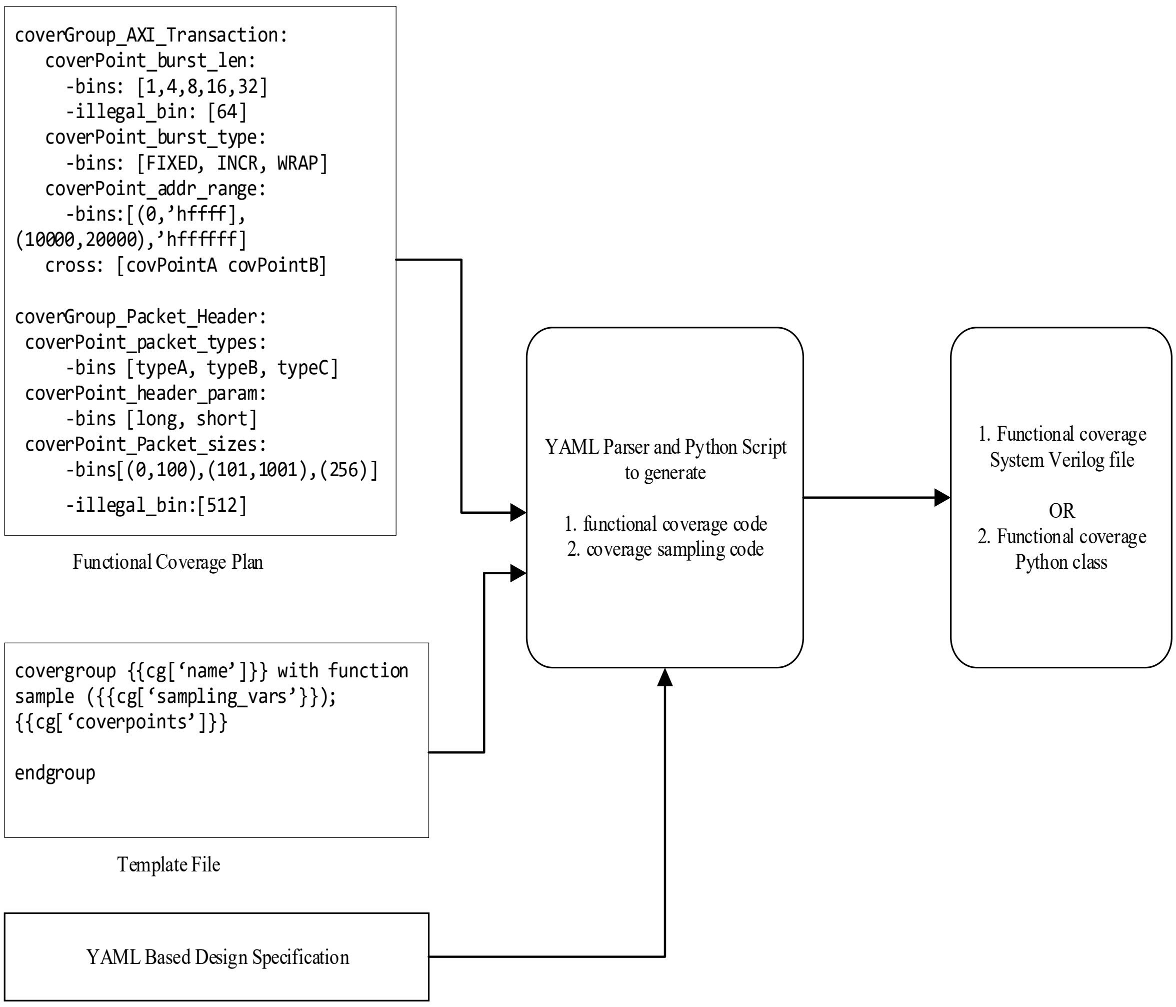
Figure 8 Automation with Design Specification in YAML

E. Functional Coverage Plan in YAML

Functional coverage is a measure of how much functionality of a design has been exercised by the testbench. It involves extracting IP features, design configurations, various data structures and packet formats. The completeness of functionality also depends upon how exhaustive the coverage plan is and the correctness of its implementation. Although functional coverage is important, it's implementation is considered as a secondary work in the execution.

This work can be automated if the coverage plan is defined in a machine-readable format like YAML, which will help to enable the coverage reporting early phase of the project. Using a templating language like Jinja2, the syntax of cover groups can be specified and if the design specification is also available in machine-readable format, the scripts can be developed for the coverage sampling parsers. This proposed setup would significantly improve quality of the coverage generation.

Nowadays, Python based libraries are available for implementing functional coverage. It gives added benefit of running functional coverage on reference models and representing abstract architectural features in coverage. Figure 9 below illustrates YAML based functional coverage flow.



```
coverGroup_AXI_Transaction:
   coverPoint_burst_len:
     -bins: [1,4,8,16,32]
     -illegal_bin: [64]
   coverPoint_burst_type:
     -bins: [FIXED, INCR, WRAP]
   coverPoint_addr_range:
     -bins:[(0,'hffff],
(10000,20000),'hffffff]
   cross: [covPointA covPointB]

coverGroup_Packet_Header:
 coverPoint_packet_types:
     -bins [typeA, typeB, typeC]
 coverPoint_header_param:
     -bins [long, short]
 coverPoint_Packet_sizes:
     -bins[(0,100),(101,1001),(256)]
     -illegal_bin:[512]
```

```
covergroup {{cg['name']}} with function
sample ({{cg['sampling_vars'}});
{{cg['coverpoints']}}

endgroup
```

Functional Coverage Plan
Template File
YAML Based Design Specification
YAML Parser and Python Script to generate
1. functional coverage code
2. coverage sampling code
1. Functional coverage System Verilog file
OR
2. Functional coverage Python class

Figure 9 YAML based functional coverage flow

VII. Conclusion and Future work

In this paper, a single YAML format is proposed for handling verification environment configurations. An effective configuration management along with automation scripts would expedite design verification by maximizing productivity, optimizing resource efficiency, and accelerating time to market. The saved bandwidth of engineers using the automation could be utilized for the domain specific use-case verification which would further improve quality of the SOCs. Investing in automation has significant benefits and it is a foundational pillar in the shift-left strategy of SOC design cycle.

Future work includes, developing Python libraries for handling various configurations and deploying in multiple SOC projects. Additionally, developing a System Verilog UVM based YAML parser which then could be integrated in any testbench environment.

Acknowledgements

I am thankful to my current managers Sakari Patrikainen and Axel Jahnke for giving me the opportunity to work on cutting edge technology and providing continuous support in exploring new verification methodologies. I would like to thank my ex-colleague and friend Maghawan Punde, who demonstrated from his scripting skills, the importance of automation and is a true inspiration for me. I am profusely thankful to my ex-manager and mentor Milind Patil who guided me on verification strategies, automation techniques and creative way of working through his innovative leadership skills.

References

[1] Universtal Verification Methodology(UVM) 1.2 User Guide
[2] YAML Documentation (https://yaml.org/)
[3] Jinja Templating engine (https://jinja.palletsprojects.com/en/3.1.x/)
[4] Python YAML library (https://pyyaml.org/)