A Novel Approach to Standardize Verification Configurations using YAML

Nikhil Tambekar

Nokia
Agenda

• Introduction
• Verification Configurations
• Challenges with Configurations
• Proposed Solution
• Example Implementations
• Conclusion
• Q/A
Introduction

• System-On-Chip Complexity
• Long SOC Verification Cycle
• Configurable Systems
• First Pass Silicon
• Automation is key

Standardizing automation improves reusability
Configurations in Verification Environment

**Testbench**
- UVM framework
- Generator
- Simulation arguments
- Functional Coverage Plan
- Test Vectors

**Design**
- Design specification
- Design code generator
- Package files
- Design Parameters
- Config registers

**Tools**
- Build System
- CI pipeline scripts
- Regression Test list
- Regression Test Modes
Configuration Challenges

- Unique requirement for each component
- No standard format
- Custom formats
- Language dependent formats (PERL hashes)
- Leads to automation not reusable
- Increased Learning curve

Standardization of Configuration Format is necessary
Proposed Format - YAML

<table>
<thead>
<tr>
<th>Property</th>
<th>XML</th>
<th>JSON</th>
<th>YAML</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Markup language</td>
<td>Data format</td>
<td>Data format</td>
</tr>
<tr>
<td>Structure</td>
<td>Tags and tree structure</td>
<td>Map with key/value pairs</td>
<td>list/sequence and key/value pairs</td>
</tr>
<tr>
<td>Comments</td>
<td>Allowed</td>
<td>Not allowed</td>
<td>Allowed</td>
</tr>
<tr>
<td>Interpretation</td>
<td>Difficult</td>
<td>Easily readable</td>
<td>Easily readable</td>
</tr>
<tr>
<td>Parsers</td>
<td>Available</td>
<td>Available</td>
<td>Available</td>
</tr>
<tr>
<td>Use</td>
<td>Data interchange between 2 APPs</td>
<td>Serving Data to APIs</td>
<td>Suited for Configurations</td>
</tr>
</tbody>
</table>

YAML is best suited for configurations
Proposed Implementation

- YAML config file
- Templates using Jinja2
- Script for Config and template parsing

```python
import yaml
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)
```
Build System Configuration using YAML

- Source Code to Executable binary
- GNU Make
- Build Requirements
  - Build targets and dependencies
  - Source code files (design, testbench, reference model)
  - Compile options (`define, debug level, Coverage)
  - Reusable compiler target for SS/Top
  - Simulator choices and simulator modes (X-prop)
  - Runtime Option Control

```
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
```

Diagram:
- YAML Build Config file
- Python Script for Build Makfile generation
- Makefile for build
Regression Setup

- Regression requirements (IP level)
  - 1000+ testcases
  - Run with multiple random seeds
  - Modes for DUT configurations
  - Regression tool Configurations

- Challenge to maintain large testcase lists and runtime options

- Synopsys Execution Manager supports YAML test list format

```yaml
test_list:
  - test_name: testname_1
    - build_name: RTL
    - run_opts: test.opts
    - num_seeds: 10
  - test_name: testname_2
    - build_name: PART_RTL
    - run_opts: test2.opts
    - num_seeds: 5
  - test_name: register_test
    - build_name: RTL
    - run_opts: +BIT_BASH
```
Regression Modes in YAML Format

- Test list in Jinja2 Template format
- Regression Modes in YAML

```yaml
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
```

Python script to generate regression modes

YAML Config file for regression modes

Test list Template

Generated Regression Test list for different modes

Regression Tool
Simulation runtime arguments in YAML

- UVM Runtime Options
- YAML for testcase configuration
- Directed and random tests
- Modification without re-compile
- Reusable
- Testbench Language agnostic
- Needs YAML config parser in System Verilog

```yaml
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]

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

1. sim_exe +TEST_CONFIG=./cpmstraint_random_test.cmd
2. sim_exe +TEST_CONFIG=./directed_test.cmd
Design Specification in YAML

- Machine Readable Specification
- Design Environment
  - Package files in VHDL/System Verilog
  - Code generator for RTL modules
  - Module instance parameters
  - Software Configuration Registers
- Verification Environment
  - UVM Testbench framework generator
  - Creating uvm_object classes
  - Formal verification setup
  - Functional Coverage classes

YAML based design Spec

YAML Parser in Python

Code Generators for design and verification

clk_period: 1.5Ghz
Interfaces:
- Interface_type: axi4
  name: input_packet_intf
data_width: 128
min_addr: 0x0000
max_addr: 0xffffffff

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: 0x0000
  end_addr: 0x0000
  access: "RO"
- name: pcie_region
  start_addr: 0x100000
  end_addr: 0x100000
  access: "RW"
**Functional Coverage Plan in YAML**

- Measures functionality covered
- SV or Python implementation
- YAML based coverage plan
- Automate cover groups and SV sampling class generation
- Easy to review and maintain
- Improves efficiency

```yaml
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, hfffff], (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]
```

**YAML Parser and Python Script to generate**

- functional coverage code
- coverage sampling code

**YAML Based Design Specification**

**Functional Coverage Plan**

**Template File for cover group**

**Functional coverage System Verilog files**

**Cover group and classes for sampling**

---

**accellera**

**SYSTEMS INITIATIVE**

**DVCON 2023**

**DESIGN AND VERIFICATION CONFERENCE AND EXHIBITION**

**EUROPE 10 YEAR ANNIVERSARY**
Conclusion

• Necessity for common configuration format
• YAML format for all configurations
• Standardization improves efficiency and reusability
• Automation improves quality
• Improves time to market
Questions