2023 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE 10 YEAR ANNIVERSARY

A Novel Approach to Standardize Verification Configurations using YAML

Nikhil Tambekar

NOKIA

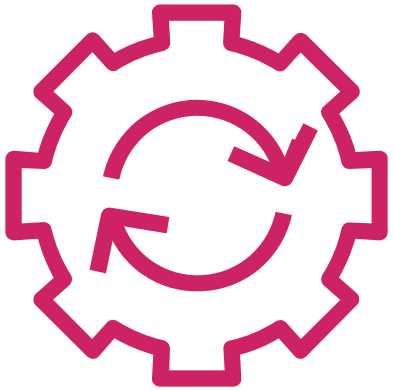
accellera SYSTEMS INITIATIVE

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

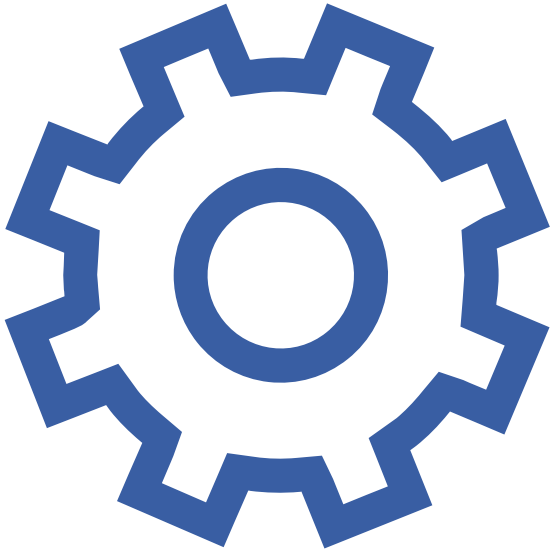
- Design specification
- Design code generator
- Package files
- Design Parameters
- Config registers

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

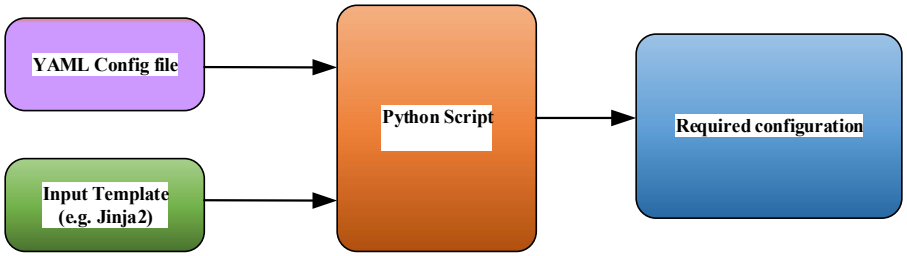
Proposed Format - YAML

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

YAML is best suited for configurations

Proposed Implementation

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




```
#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 Config

```
module {{module_name}}_top #Variable
example

{% if loop.index is divisibleby 3 %}
#condition Code
{% endif %}


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

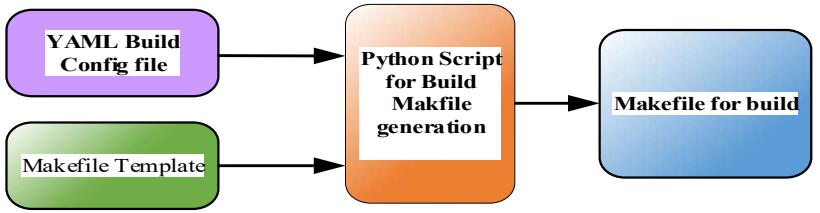
Input Template

```
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)
```

Python Script

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

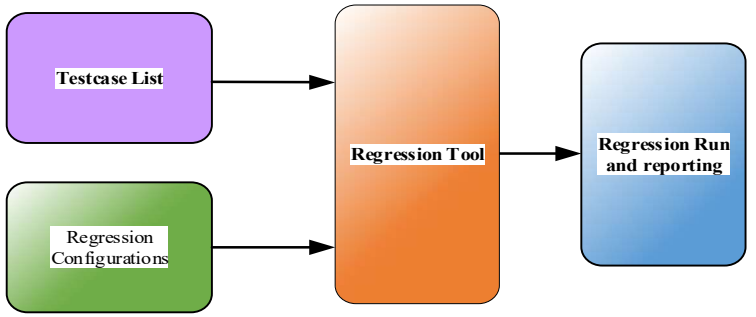



```
simulator: SIM1 | SIM2
build_rtl:
 dut_filelist:
  - '$WS/rtl/dut_filelist'
  - 'library_filelist'
 tb_filelist:
  - '$LIB/vip_filelist'
  - 'verif/tb_filelist'

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

Regression Setup

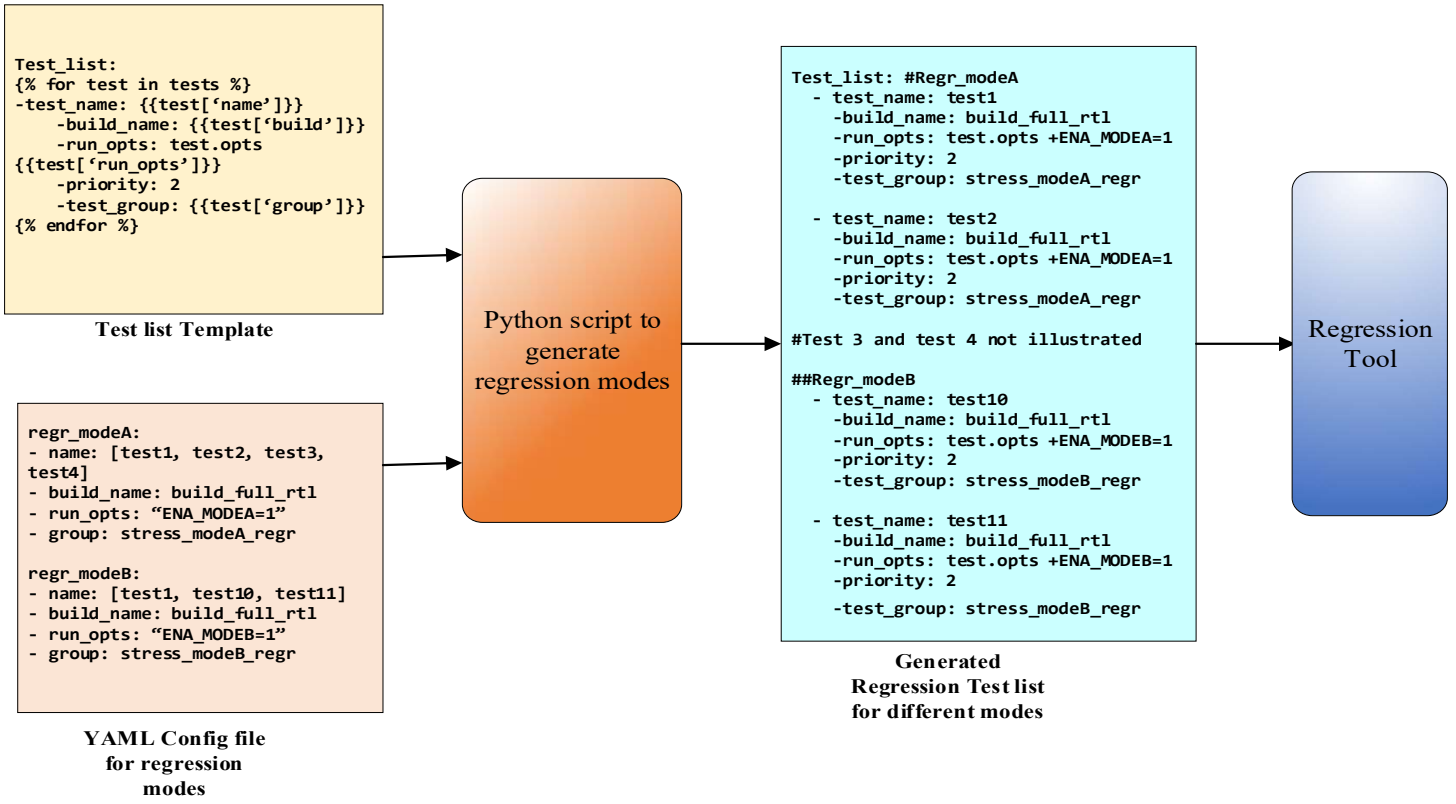
- Regression requirements (IP level)
  - 1000+ testcases
  - Run with multiple random seeds
  - Modes for DUT configurations
  - Regression tool Configurations
- Challenge to maintain large testcase lists and runtime options
- Synopsys Execution Manager supports YAML test list format




```
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
```

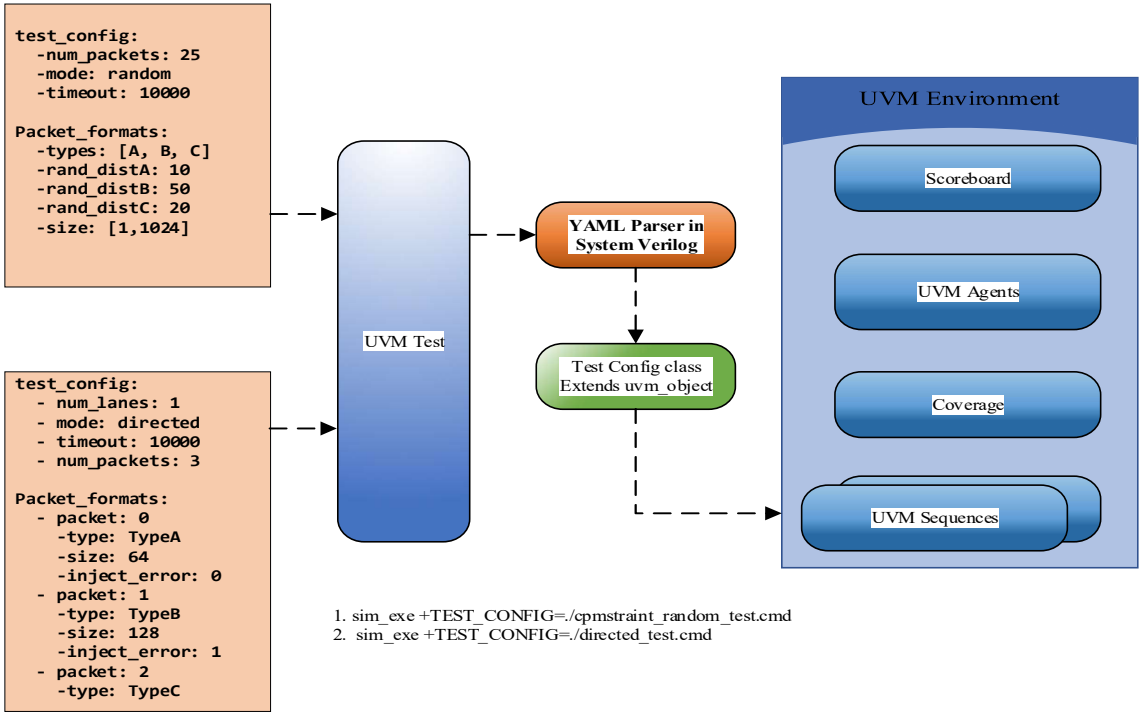
Regression Modes in YAML Format

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



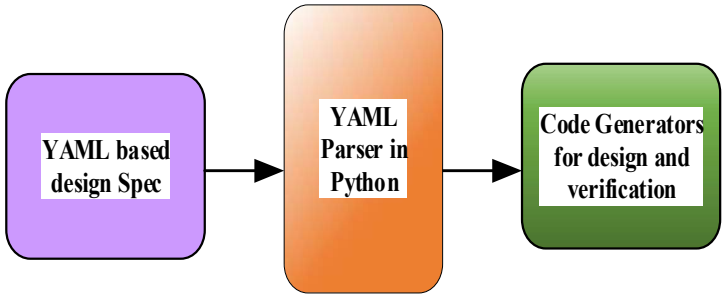
Simulation runtime arguments in YAML

- 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



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




```
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
```

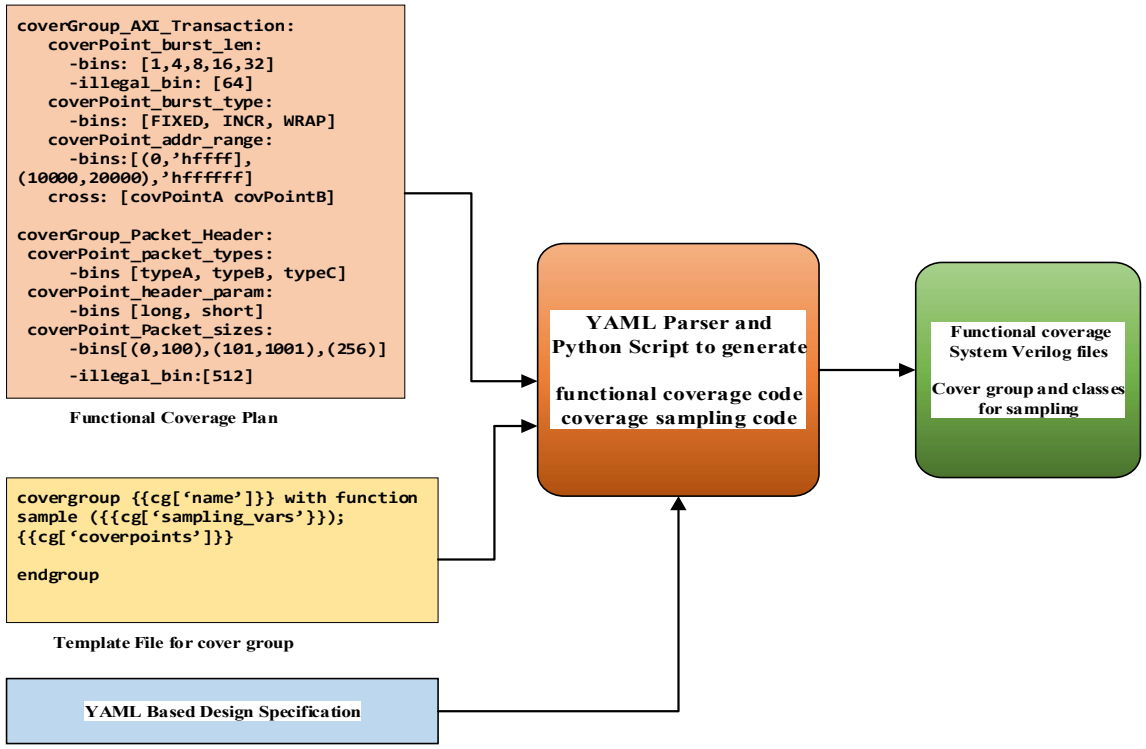
Interface Specification in YAML

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

Memory map in YAML

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



Conclusion

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

Questions