Testbench Linting – open-source way

Using slang/pyslang/PySlint

Srinivasan Venkataramanan, Solutions Architect, AsFigo Technologies, London, UK (svenkat@asfigo.com)

Deepa Palaniappan, DV Consultant, AsFigo Technologies, Zurich, Switzerland (deepa@asfigo.com)

Satinder Paul Singh, Europe CTO, Cogknit EU, Munich, Germany (satinder.singh@cogknit.com)

***Abstract*— Linting or rule-checking is a proven technique in RTL design and software domain to maintain high quality of code across distributed teams. SystemVerilog, the leading design verification language for chip designs will immensely benefit from such technology/process. However, lack of focus in EDA, wide variety of coding styles, and commercial factors have slowed down development of such tools in the past. We present our experience in building and deploying such lint tools based on a popular open-source platform slang/pyslang. As rules are developed in Python, it is very easy to customize it at finer granularity with reasonable cost. We also share how such rule checking can be integrated into a Continuous Integration/development (CI/CD) flow such as Git Actions.**

Keywords—SystemVerilog; Linting; Coding style; UVM; Python

I. Introduction

Linting is a well-known, popular technique in large code development projects. Software engineers have been using Lint based rule checking to maintain high quality code adhering to a set of predefined coding guidelines. In hardware design field, Lint tools have been very popular in RTL design phase and designers use lint to ensure the design is synthesis friendly, DFT compliant etc. Several popular coding guidelines have evolved over the last few decades such as Reuse Methodology Manual (RMM [1]), Japan's STARC standard [2] etc. Often tools group a set of rules and create policies to ensure design code is RMM compliant etc. Verification on the other hand does not enjoy the same level of lint tools. ensure that SystemVerilog is well established as the design verification language in semiconductor design flow. While rule checking (a la Linting) is well established for synthesisable code well over a decade, the verification counterpart lacked such technology in the past. With software giants such as Google, Meta investing in this domain, many fully featured parsers are now available, many as open source as well. In this paper, authors share their experiences in developing a highly customizable linter for SystemVerilog testbench code based on open source pyslang API [3]. Verification methodologies such as UVM [4] have evolved over last decade and more and have provided much more than just guidelines including base class libraries, applications, examples etc. While there are no read-made/standard rule policies for coding guidelines available for verification (unlike in design), many customers tend to develop their own rules/policies primarily based on UVM and picking important ones as they deem fit for their projects. However, in the past there was no easy way of adhering to those rules and flagging any potential violations – thereby affecting quality, reusability aspects of codebase. We intend to change that with our work.

II. Lint and Static Code Analysis

In the realm of chip design, "Lint" and "Code Analysis" refer to critical aspects of static analysis—a method that involves scrutinizing the source code RTL (Register Transfer Level) description of a digital design and TB (TestBench) code without actually executing it. This meticulous review is performed by specialized tools to identify potential issues and deviations from coding standards, ensuring a higher quality end product. Linting is a static analysis technique used to spot coding issues, irregularities, and potential bugs in RTL & TB code. The term originated from the UNIX utility "lint," which identified syntax and style errors in C programming. In chip design, lint tools perform a similar function but are tailored to hardware description languages like SystemVerilog and VHDL. These tools analyze the code for syntax errors, naming conventions, type mismatches, uninitialized variables, and other common coding mistakes. Linting tools help catch problems early in the design process, preventing downstream issues and reducing the chances of encountering subtle bugs during later stages of development. Modern Lint tools go beyond simple syntax checking and delves into more complex structural and design-related aspects of the code or RTL. It involves a deeper examination of the code's logic, organization, and potential interactions. Code analysis tools identify potential functional problems, design flaws, and performance bottlenecks. This process aids in maintaining consistency, enhancing readability, and ensuring that the design adheres to best practices and established design guidelines.

III. STATIC CODE ANALYSIS OPPORTUNITIES IN VERIFICATION PROCESS

RTL Lint has been very popular and well adopted over decades. However, on the verification side, lint usage has been limited in adoption. This does not necessarily mean the lack of need for it, rather there are ample opportunities for using linting in verification. Some of them are enumerated below.

A. Assertions

SystemVerilog Assertions (SVA) are very powerful feature used by both designers and verification engineers. There are plenty of opportunities to perform static analysis of SVA code including coding style, functionality, performance etc.

B. Coverage models

SystemVerilog includes powerful constructs to capture functional coverage that can be used in various contexts such as interface/module/class. Poorly coded coverage models can heavily mislead the project – either to give a false positive or false negative picture of verification status. This is another good avenue for static analysis tools to provide quantitative picture of the verification code-base.

C. Class based SystemVerilog testbench

One of the key reasons why SV is popular is due to the class-based testbench capabilities. Given that many hardware engineers end up writing/maintaining/debugging such testbench code, it is crucial to enforce coding guidelines via a lint tool.

D. UVM testbench code

UVM has been very well adopted by teams across the globe and comes with a multitude of choices and features. Many teams develop internal coding guidelines while using UVM and these rules are ideal candidates for linting. Some examples are:

- Proper use of base classes
- Consistent macro usage
- Avoiding potentially dangerous macros
- Reuse aspects in UVM agents
- Performance friendly coding styles with UVM Config-DB etc.

E. VIP selection

One of the benefits of standards is the abundant availability of products using such standards. UVM based VIPs are excellent examples wherein several new innovative verification IP providers can compete for same protocol such as USB, PCIe etc. As end users, teams often need to select appropriate VIP for their chips and often find clouded by marketing slide decks that tend to hide technical quality and qualitative aspects of VIPs. With static analysis tools, one can objectively evaluate multiple VIPs and make informed choices.

F. Formal Verification, Accelleration etc.

Often design verification goes beyond simulation to leverage on adjacent technologies such as Formal Verification, Hardware Acceleration, Emulation etc. Though SystemVerilog as a language yield to all these technologies, there are sub-sets within SV that are more suitable to Formal, acceleration etc. A testbench lint tool can help users' code to adhere to such sub-sets on a need basis.

IV. RTL LINT USING OPENSOURCE TOOLS

As a quick recap – RTL lint using opensource tools has been in use for few years in the industry. Though at times limited in capabilities compared to a commercial lint tool, opensource tools help customers do quick check, catch low-hanging fruits in an efficient manner without incurring expensive license costs. This is especially effective when combined with continuous integration (CI) and continuous development (CD) flows. A subset of RTL lint checks can be effectively enforced on every check-in of RTL code by integrating opensource lint tools to source code management systems such as GitHub, SVN, Perforce etc. Readers are encouraged to look at options such as:

- Verilator
- Svlint
- Verible
- PySlint

V. TESTBENCH LINTING - MYTHS

For a long time, end users have been seeking lint tools for testbenches, especially after seeing the clear benefits of RTL linters. However, there are several myths around testbench code not so yielding to lint rules, some of them being:

- Developing linting tools for SystemVerilog testbenches presents significant challenges due to the dynamic nature of testbench code.
- Unlike traditional RTL (Register Transfer Level) code, testbenches components and objects are not "born" till run-time (Though this is applicable for "simulation", static analysis of code is still feasible).
- Lack of common rules
- Developing new rules takes lot of manpower and skilled workforce

Fast forward to 2023, good news is most of these myths are busted and the industry is seeing new wave of open-source parsers enabling teams to develop testbench linters.

VI. SYSTEMVERILOG PARSERS

A SystemVerilog parser is a program that converts SystemVerilog source code into a data structure that can be analyzed by other tools. The parser takes the source code as input and produces a syntax tree as output. The syntax tree is a hierarchical representation of the source code, which can be used to find errors, perform code analysis, and generate other output formats. SystemVerilog is a complex language, and its parsers can be challenging to develop and maintain. The language has a lot of features, and it is constantly evolving. Additionally, SystemVerilog parsers must be able to handle a wide variety of input, including well-formed code, poorly formed code, and code that contains errors.

One of the major bottlenecks in implementing custom rules for testbenches has been the lack of easy-to-access SystemVerilog parsers in the past. While there were few good commercial ones, it was beyond reach for many end-users. In recent years, many open-source parsers have become available thanks to larger community getting involved in this domain and increased interest from Artificial Intelligence and Machine Learning (ML) teams trying to extract key factors form complex testbench code.

A. Slang

slang is a is a recursive descent parser for SystemVerilog, which means that it breaks the input code down into smaller and smaller pieces until it reaches the smallest possible pieces, which are called tokens. The tokens are then assembled into a syntax tree, which is a hierarchical representation of the input code.

The slang SystemVerilog parser is designed to be robust and efficient. It can handle a wide variety of input code, including well-formed code, poorly formed code, and code that contains errors. The parser is also designed to be efficient, so that it can be used to parse large amounts of code quickly.

B. Pyslang

PySlang is a Python library for parsing and analyzing SystemVerilog code. It is based on the slang SystemVerilog parser, and it provides a number of features for linting, code analysis, and simulation. PySlang is a free and open-source library, and it is available on GitHub. It is actively maintained and developed by a team of engineers and researchers. PySlang can be used to improve the quality of SystemVerilog code by finding errors, performing code analysis, and generating other output formats. It can also be used to automate a variety of tasks related to SystemVerilog code, such as linting, code analysis, and simulation.

VII. PySlint – a collaborative approach to Testbench Linting

A. PySlint: Empowering Innovative Hardware Design Verification with Python-Based Linting

PySlint builds on top of PySlang that melds the power of Python programming with the intricacies of hardware description. PySlint stands on the robust foundation of the Python programming language, an industry favorite known for its readability and adaptability.

B. From Minimalism to Mastery: Unveiling Case Studies

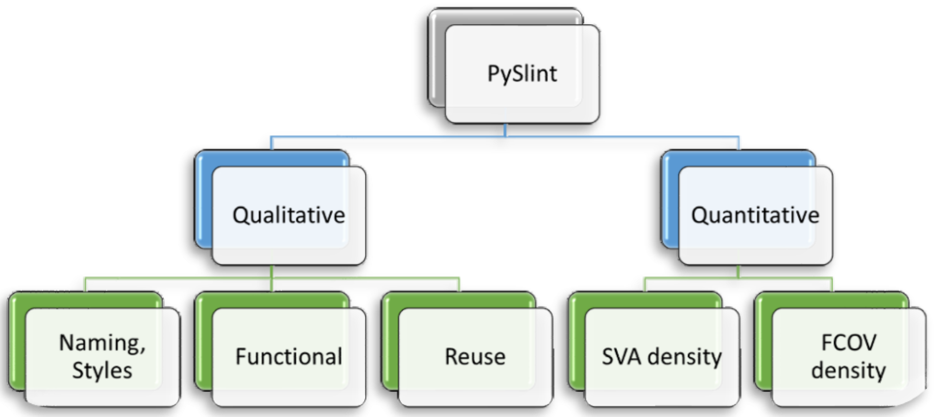
At its core, PySlint isn't a tool; it's a platform. It provides minimalist rules meticulously crafted as case studies. These case studies serve as beacons of best practices, guiding users through effective linting strategies. Yet, beyond mere guidelines, PySlint invites users to harness these foundational principles and evolve them into custom-tailored solutions.

C. Customer-Centric Customization

PySlint doesn't merely present predefined rules; it empowers users to mold their own. Designers have the liberty to curate rules that align seamlessly with their unique design requirements. This opens a realm of possibilities—users can customize existing rules, forge entirely new analysis tools, and even maintain proprietary rule sets, ushering in a new era of linting adaptability. Unlike conventional EDA products, PySlint defies the norm. It steps beyond the boundaries of traditional solutions and embraces innovation as its guiding principle. By providing a canvas for user-driven rules and novel analysis tools.

VIII. Static analysis of SystemVerilog Testbenches using PySlint

PySlint provides both qualitative and quantitative analysis of SystemVerilog testbench code. Qualitative analysis includes classical lint style checks.



IX. SAMPLE PYSLINT RULES

We start with simple naming rules to demonstrate how linter can be built on top of pyslang.

- All classes shall have a suffix _c
- All interfaces must be named with a suffix _if
- All covergroups must have cg_ as prefix.

Each of this rule is customizable and can be enabled/disabled on need basis.

We also show rules such as (indicative list):

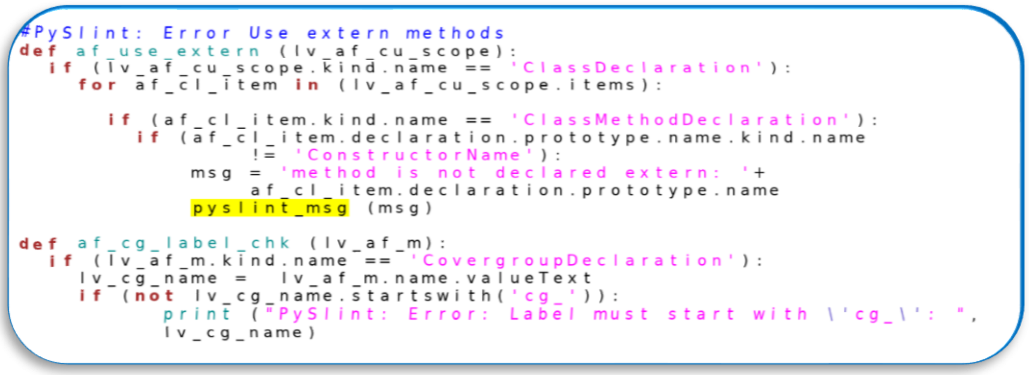
- All class methods shall be declared as extern (except new()).
- All extern methods shall have implementation.

The potential of such a platform is way beyond such simple rules – though these rules do bring in discipline and maintain code integrity. For instance, some of the applications of such a versatile rule checker are:

- Check (and maintain via CI/CD) code compatibility among simulators – often SV/UVM code has vendor specific enhancements/relaxations making it harder for customers to switch.
- Evaluating competing VIPs from vendors while making choices for a new SoC design project.
- Build custom policy for acceleration-friendly VIPs etc.

A. Adding rules via Python API

One of the challenges in using pyslang today is the lack of documentation. Our team analyzes each rule as specification/requirement and develops a sample algorithm to implement on top of slang. We use existing C++ code and documentation and extrapolate it to the Python API and with a few trail-and-error we build rules in pyslint [6]. Figure-1 below shows sample implementation of one such rule:



```
#PySlint: Error Use extern methods
def af_use_extern (lv_af_cu_scope):
  if (lv_af_cu_scope.kind.name == 'ClassDeclaration'):
    for af_cl_item in (lv_af_cu_scope.items):

      if (af_cl_item.kind.name == 'ClassMethodDeclaration'):
        if (af_cl_item.declaration.prototype.name.kind.name
                != 'ConstructorName'):
            msg = 'method is not declared extern: '+
                af_cl_item.declaration.prototype.name
            pyslint_msg (msg)

def af_cg_label_chk (lv_af_m):
  if (lv_af_m.kind.name == 'CovergroupDeclaration'):
    lv_cg_name =  lv_af_m.name.valueText
    if (not lv_cg_name.startswith('cg_')):
          print ("PySlint: Error: Label must start with \'cg_\': ",
          lv_cg_name)
```

Figure 1. Sample rule implementation in pyslint

Below is a table of sample rules available in PySlint for SystemVerilog:

PySlint Rule ID	Description
NAME_CL_PREFIX NAME_CG_PREFIX NAME_ASM_PREFIX NAME_COV_PREFIX	Naming/style checks
SVA_MISSING_LABEL SVA_MISSING_ENDLABEL SVA_NO_PASS_AB SVA_MISSING_FAIL_AB	Assertion specific. Performance, Debug categories
CL_METHOD_NOT_EXTERN CL_MISSING_ENDLABEL FUNC_CNST_MISSING_CAST	Class specific – style, functionality

UVM rules are developed on customer specific requirements. Sample rules on PySlint are below:

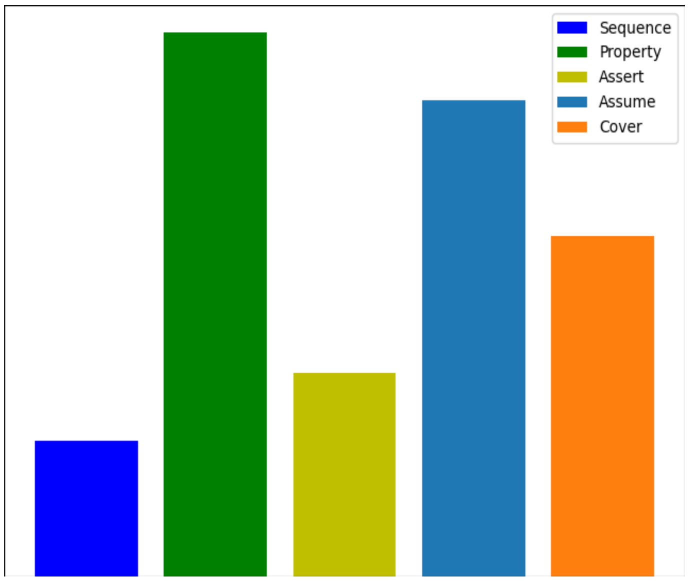
UVMLint Rule ID	Description
UVM_DRV_MISSING_PARAMS	Driver should have REQ parameter overridden
UVM_AGENT_IS_ACT_REUSE	Agent should check is_active before constructing drivers, sequencer for reuse
UVM_AGENT_IS_ACT_HIDE	Do not re-declare is_active field in uvm_agent
UVM_MON_MISSING_VIF	Monitor should have a handle to virtual interface

X. Quantitative Analysis of Testbenches

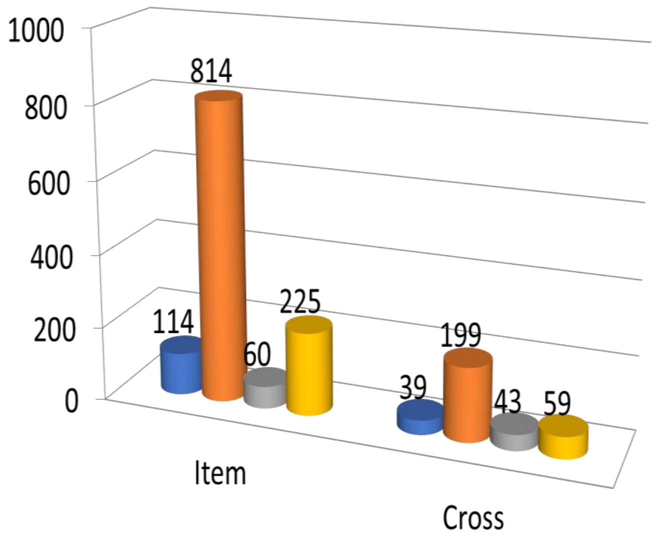
Quantitative analysis provides a measurable information of users' testbench code by extracting key information such as:

- SVA Sequences, Properties, Assertions, Assumptions, Covers
- Ability to infer metrics such as "Assertion Density", number of assertions vs. number of signals in a module etc.
- Constraint analysis
- Functional Coverage code – extract number of coverpoints, bins, crosses etc.

SVA statistics:



Functional Coverage statistics:



Evaluating the effectiveness of hardware verification involves a multifaceted analysis. Metrics play a crucial role in this endeavor, offering insights that go beyond conventional measurements. PySlint offers quantitative analysis that can help as follows.

- Consistent use of Active Line Count, Comments: The percentage of active lines in a block of code is the number of lines that are not comments or blank spaces. Strong correlation exists between complexity of a block/protocol and the VIP active line count in the code that implements the same. A consistent percentage of comments indicates that the code is well-written and readable. It also suggests that the code is easy to maintain.
- SystemVerilog constraints: A statistical analysis of SV constraints enables users to choose correct solver for complex problems such as PCIe lane management, a credit buffer model, cache coherency protocol etc. By annotating tool specific constraint solving metrics, teams can predict the resources needed to solve such complex constraints.
- Functional Coverage code analysis: A block of code with a lot more coverage points is a block of code that has been tested more thoroughly. At times this can also point to wasted efforts as there maybe inadvertently be a large state-space being coded that does not provide a good ROI.
- There should be a strong correlation between the number of coverage points and the execution metrics of a block of code. This means that the blocks with more coverage points should also have more execution metrics, such as execution time, tests, runs etc.
- Maximum number of "checks": The maximum number of checks in a block of code is the number of ways that the code can be verified. These checks can include unit tests, integration tests, and system tests. Blocks with a higher number of checks are more likely to be bug-free and reliable.
- Blocks under development: Blocks under development are blocks of code that are still being worked on. These blocks tend to have a lower percentage of active lines, as they are not yet complete. Additionally, blocks under development tend to have fewer coverage points, as they have not yet been tested as thoroughly. Subsequent phases introduce more, enhancing coverage and the overall verification process.

In essence, these metrics collectively contribute to a comprehensive verification strategy, ensuring that hardware designs are not only functional but also thoroughly examined for robustness.

XI. Conclusion

We present pyslint – an open-source Python based rule checker/linter for SystemVerilog testbenches built on top of pyslang/slang. It should be noted that rules presented in this work are easy to port to any other platforms such as C++/TCL/Perl/Rust etc. on need basis provided an underlying API is available. Also, this work as available on opensource is still in its initial stages.

Acknowledgments

Authors acknowledge the authors of slang and pyslang, especially that of Mike Popoloski for his timely replies on our queries on the Python API. We also would like to acknowledge the timely reviews, rule suggestions etc. done by various early adopters of this exciting technology.

REFERENCES

[1] Michael Keating , Pierre Bricaud, Reuse Methodology Manual https://link.springer.com/book/10.1007/978-1-4615-5037-2

[2] Japan group publishes HDL design style guide https://www.eetimes.com/japan-group-publishes-hdl-design-style-guide/

[3] Pyslang - https://github.com/MikePopoloski/pyslang

[4] IEEE 1800.2 UVM - https://ieeexplore.ieee.org/document/9195920

[5] slang - https://sv-lang.com/

[6] PySlint – https://www.github.com/AsFigo/PySlint