

# Halstead, McCabe, and Lint in Action

## Quality Metrics

### for

## SystemVerilog Testbenches

Hemamalini Sundaram  
Ajeetha Kumari Venkatesan  
Sivabharati Chinnaswamy  
Manoj Rajendran

# Agenda

- Halstead metrics Introduction
- SystemVerilog & Halstead examples
- Applying Halstead to a constraint block
- Impact and carry forward

# UVMLint – Analytics apps

- Datamine existing code for key metrics
  - RTL
    - Code coverage
  - SVA
  - SV TB
  - UVM
  - DPI

# UVMLint – Analytics apps

- Cyclomatic complexity
- Halstead
  - Operator and Operand
  - Volume, Complexity
  - BugRate

# Halstead metrics - Introduction

- Developed by Maurice Halstead (1977)
- A set of software metrics to measure
  - Complexity
  - Effort
  - Maintainability
- Based on operators and operands in source code

# Halstead metrics

## Basic metrics

- $n_1$  = number of distinct operators
- $n_2$  = number of distinct operands
- $N_1$  = total occurrences of operators
- $N_2$  = total occurrences of operands

## Derived metrics

- Program Length ( $N$ ) =  $N_1 + N_2$
- Program Vocabulary ( $n$ ) =  $n_1 + n_2$
- Volume ( $V$ ) =  $N \times \log_2(n)$
- Difficulty ( $D$ ) =  $(n_1 / 2) \times (N_2 / n_2)$
- Effort ( $E$ ) =  $D \times V$

# SystemVerilog & Halstead example

```
class Packet;  
  rand bit [7:0] length;  
  rand bit [15:0] addr;  
  constraint valid_range {  
    length > 5;  
    addr inside  
    {[16'h1000:16'h1FFF]};  
  }  
endclass
```



<b>Metric</b>	<b><u>Description</u></b>	<b><u>Value</u></b>
<b><math>n_1</math></b>	Number of unique operators	
	class, rand, bit, constraint, >, inside, {}, :	8
<b><math>n_2</math></b>	Number of unique operands	
	Packet, length, addr, 5, 16'h1000, 16'h1FFF	6

# SystemVerilog & Halstead example

```
constraint valid_range {  
    (length > 5 && length < 128) &&  
    (addr inside {[16'h1000:16'h1FFF]} || addr == 16'h2000)  
    && (length != 0 && addr != 16'h0000);  
}
```

# SystemVerilog & Halstead example

<u>Metric</u>	<u>Description</u>	<u>Value</u>
$n_1$	Unique operators	10
	$> , < , != , == , \&\& ,    , \text{inside}, \{ \} , : , \text{constraint}$	
$n_2$	Unique operands	8
	length, 5, 128, addr, 16'h1000, 16'h1FFF, 16'h2000, 16'h0000	
$N_1$	Total operators	13
	$> , < , !=x2 , == , \&\& x3 ,    , \text{inside}, \{ \} , : , \text{constraint}$	
$N_2$	Total operands	13
	length $\times$ 3, 5, 0,128, addr $\times$ 3, 16'h1000, 16'h1FFF, 16'h2000,16'h0000	

# SystemVerilog & Halstead example

- Program vocabulary:  $\eta = \eta_1 + \eta_2$
- Program length:  $N = N_1 + N_2$
- Calculated estimated program length:  $\hat{N} = \eta_1 \log_2 \eta$
- Volume:  $V = N \times \log_2 \eta$
- Difficulty :  $D = \frac{\eta_1}{2} \times \frac{N_2}{\eta_2}$
- Effort:  $E = D \times V$

Metric	Value
Program Length (N)	26
Program Vocabulary (n)	18
Volume (V)	108
Difficulty (D)	8
Effort (E)	864

# Interpreting Halstead metrics

- Volume (V): Size of implementation
- Effort (E): Cognitive effort to develop or understand
- Difficulty (D): Relative difficulty of implementation
- Thumb rule: High values → May indicate need for refactoring

# Constraint complexity & solver performance

- Complexity Correlates with Solver Performance
- High Halstead difficulty and effort scores usually mean the constraint
  - Has many logical operators (&&, ||, !, ->)
  - Uses many variables and nested conditions
  - Contains intertwined dependencies that are hard to resolve quickly
- Why it matters?
  - Constraint solvers use backtracking and heuristics.
  - More complex logic increases search space, making solving slower or causing solver timeouts.

# UVMLint.Analytics – value proposition

- Halstead metrics provide a quantitative measure of complexity before running simulations.
- A constraint flagged with a high score can be reviewed before it causes runtime solver bottlenecks.
- This proactive step prevents hours wasted/saved on simulation stalls or cryptic solver errors.

# Refactoring Guidance - UVMLint

When Halstead flags high complexity, refactoring can:

- Split large constraints into smaller, focused ones
- Remove redundant conditions
- Use implication instead of nested ifs
- Avoid mixing multiple conditions in single expressions

# Code Review Best Practices with Halstead Metrics

- Use consistent metrics across the codebase
- Set thresholds for acceptable complexity
- Refactor code with high complexity scores
- Treat metrics as guidance, not strict rules

# AXI - constraint example

```
constraint complex_access_c {
    // Apply only for secure or privileged accesses
    if ((is_secure && is_privileged) || (!is_secure && debug_mode)) {
        // Constrain address range based on access mode
        if (access_mode == BURST && burst_len > 4) {
            addr inside {[32'h8000_0000:32'h8FFF_FFFF]};
        } else if (access_mode == SINGLE || burst_len == 1) {
            addr inside {[32'h0000_1000:32'h0000_1FFF]};
        }
        // Apply protection based on multiple flags
        if ((user_mode && !is_privileged) || (secure_mode &&
        override_prot)) {
            protection == 3'b010;
        } else if (!user_mode && !override_prot) {
            protection inside {3'b001, 3'b100};
        }
    }
}
```

- Operators: &&, ||, ==, !, >, inside, if, else, brackets — 10+ unique operators
- Operands: many unique operands
- High difficulty and effort
- Halstead metrics will suggest this block -complex
- Potentially needs refactoring.

# Is this bad code?

- This level of complexity might be justified
- Expresses real conditional behaviors of an AXI transaction model or secure bus protocol Not necessarily.
- Encodes legitimate hardware rules for secure vs. non-secure transactions
- Needs to match spec edge cases tightly

# Refactored constraint model

- In this case, a high Halstead score is acceptable, but a code reviewer should check:
- Can it be split for readability?
- Are all conditions needed?
- Would implication constraints simplify it?

```
((access_mode == SINGLE) || (burst_len == 1)) ->  
  addr inside {[32'h0000_1000:32'h0000_1FFF]};
```

```
((user_mode && !is_privileged) || (secure_mode &&  
override_prot)) ->  protection == 3'b010;
```

```
((!user_mode) && (!override_prot)) ->  
  protection inside {3'b001, 3'b100};
```

## Summary:

- Use halstead as guidance
- Very handy to compare code across files/implementations
- Develop coding guidelines and UVMLint them same!

# UVM Quantitative Analysis –IBEX (RV)

```
rv_ibex_sv_files/vendor/google_riscv-dv/src/riscv_load_store_instr_lib.sv  
Processing single file: rv_ibex_sv_files/vendor/google_riscv-dv/src/riscv_
```

Metric	Value
Unique Operators	25
Unique Operands	59
Total Operators	251
Total Operands	184
Vocabulary	84
Length	435
Estim. Length	463.2
Volume	2780.7
Difficulty	39.0
Effort	108398.5
Time	6022.1
Bugs	0.757811202809796

Halstead  
for CNST  
blocks

## Summary:

- Use halstead on targeted expressions such as constraints, covergroups, assertions
- Very handy to compare code across files/implementations
- Develop coding guidelines and UVMLint them same!

# Questions