Faster TB Development with Automated Code Checks

Pushkar Kumar
Application Engineer, Synopsys

October 2022

Introduction

- Efficiency is crucial when working with SystemVerilog UVM Testbenches. Syntax mistakes, coding violations, and the resulting problems increase the daily stress of engineers.
- Every time an engineer submits a compile after coding a part of TB, it is extremely likely for the code to fail at compile or run time due to syntactical or structural errors – Needs iterative compile and simulation to fix.
- That time is measurable
- Engineers would rather have the problems:
  - in their editor
  - annotated with exactly the right error message to fix
  - be given quick feedback if their fix worked

What Do We Need During Code Development?

- Testbench linting
- Can you even enforce tabs-versus-spaces today?
- Naming convention enforcement
- Templates
- Shared settings
- Version control frontend
- Common environment

Automated Code Checks

On-the-fly Rule Checking

- **Avoid Testbench errors**
  - Testbench linting with elaboration and advanced analysis
  - Find bugs in testbench and uncover RTL errors hidden by erroneous testbench code

On-the-fly SVTB/UVM checking

- **Prevent late-stage Design-under-Test (DUT) bugs and assure code quality**
  - Design rule checking as you type
  - Assure high quality code by finding bugs early before it becomes risky to modify code

On-the-fly DUT checking

- **Improve productivity**
  - Integrated Development Environment
  - Coding acceleration, code viewing and navigation, integration with version control, bug tracking, task management
- **Easily adopt coding styles and project methodology**
  - Verify legal syntax/semantics, naming conventions, unified project indentation and code formatting
  - Custom rules for methodology enforcement

Integrated Development Environment (IDE)

Testbench Checks

- Procedural code analysis
  - Dead code detection
  - Time-0 problems detections, such as out-of-bounds accesses, 'unique' case violations, etc.
  - 'null' class / virtual interface access
  - Dynamic casting violations
  - Infinite loops & recursions
- UVM specific rules
  - Non-compliant or incorrect code
  - Migration to UVM 1.2 or IEEE standard
- Method overrides & implementation inconsistencies or irregularities
- SVA checks
- Class constraint checks
- Data type mismatches
  - In method argument, including DPI
  - In assignments
  - In expressions in any context
- Simulation performance rules
- Custom methodology rules

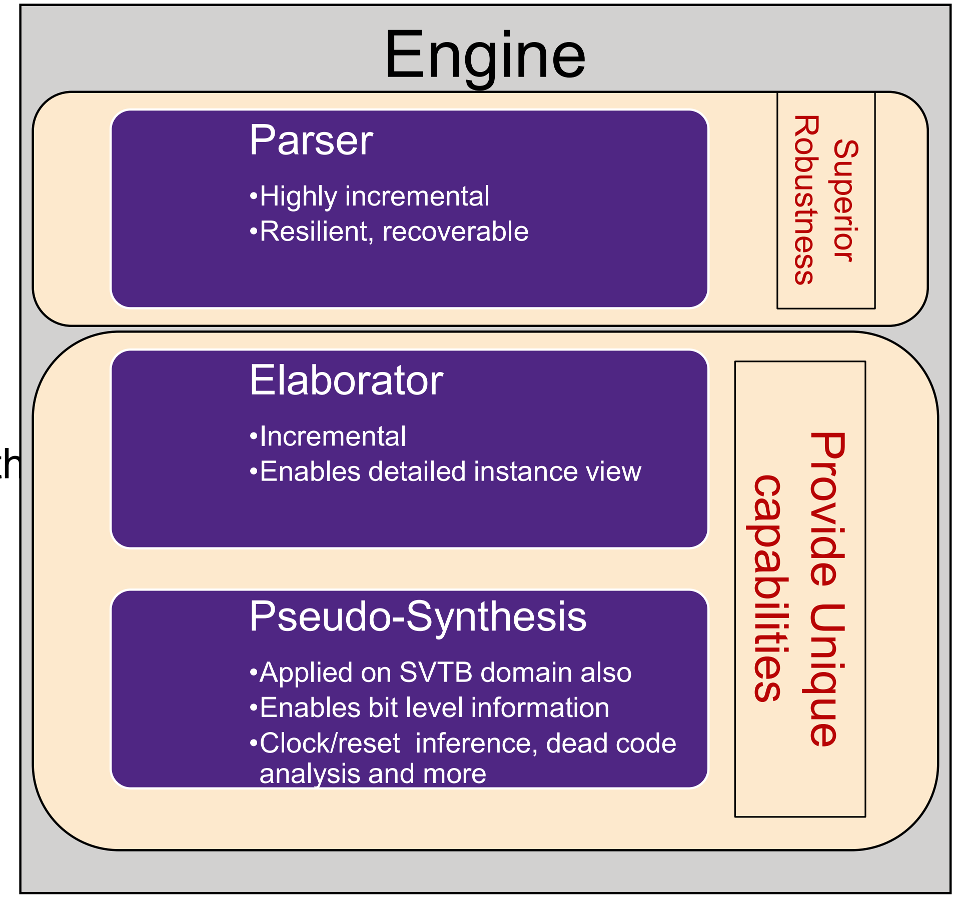
Design Checks

- Synthesis results
  - Combinational loops, latches, ungated flops, etc.
  - Clock & reset related rules
- Drives / loads violations
  - Per Index, leveraging time-0 information
  - Undriven, unread
  - Multiple drives, multiple driving procedures
  - Interface & modport violations
- Procedural code analysis
  - Dead code detection
  - Time-0 problems detections, such as out-of-bounds accesses, ‘unique’ case violations, etc.
- Width mismatches and other data type mismatches
  - Parameterized information
- Simulation performance rules

Euclide

Technology Highlights

- Powerful EDA Engine
- Engine Differentiation:
  - High performance
  - Resilient and recoverable
    - Robust feedback even on incomplete code or code with errors
  - Incremental and interactive
    - Enable live feedback and support for IDE features
- For more :
  - Getting Started with Synopsys Euclide



Engine
Parser
•Highly incremental
•Resilient, recoverable
Superior Robustness
Elaborator
•Incremental
•Enables detailed instance view
Pseudo-Synthesis
•Applied on SVTB domain also
•Enables bit level information
•Clock/reset inference, dead code analysis and more
Provide Unique capabilities

Thank You