Modern methodologies in a TCL test environment

Matteo De Luigi, Alessandro Ogheri
Ogheri Consulting GmbH

A. Ogheri consulting
http://www.alessandroogheri.com
Overview

• Introduce DUT, existing environment, and challenges
• Analysis of critical quantities
• Evaluation of different approaches
• Architecture of chosen solution
• Results
Device Under Test (DUT)

• Microprocessor-driven controller for analog module
• Firmware, digital hardware, and analog hardware are co-developed in-house by three different teams
• Task: check flow of signals between digital and analog parts
Challenges

• Protocol too complex for simple assertions
• HW/SW co-development: none of the three components was/is in a stable state. Must track three specifications at once
• Test run times may be very long
• Limited amount of project time to develop and test checkers
More about the environment

• Limited input parameters per test
  – Enumeration was not only possible, but required
• Conceptually part of a directed firmware test infrastructure.
Existing status

• Simple VHDL-based directed test environment
• Protocol was too complex for the simple VHDL-based environment.
• Customer decided to reuse the existing hardware test environment, built with a Constrained-Random-Verification (CRV) tool, for firmware verification.
• A CRV-based module, developed for a previous project, performed the needed checks
Analysis: critical quantity

• Run-time for tests: not critical
• Stimuli and coverage: test is directed, enumeration
• Development and turnaround time: critical
• Reaction time to changes in specification: critical
CRV-based tool: pros

• Code reuse (a module for our task had already been developed)
• Language integrates well with simulator
• Native support for verification and modern methodologies
CRV-based tool: cons

• Slow development turnaround
• Need to maintain two separate code environments (CRV and VHDL based) to perform directed tests.
  – Specialized CRV skills needed just to write stimuli
• Proliferation of stimulus code
• Heavy reliance on complex metaprogramming
CRV tool: the bottom line

• CRV based tools are powerful, but our project did not exploit that power

• The existing environment was paying the price of using a CRV tool without reaping the benefits
TCL: benefits

• Fast development turnaround (if used outside the simulator)
  – Startup time is negligible
• Can be plugged into existing VHDL environment
• Can re-use existing stimulus files
• Native fit for complex text processing (no separate metaprogramming step needed)
TCL: desiderata

Parallel, “offline” flow, based on processing logs:

• Edit logs and/or checkers
• Develop prototype
• Once done, plug in test as-is
Challenges

• Existing CRV-based module does not fit TCL's programming model
• Plain TCL provides no constructs for verification
• Must plug into existing environment (e.g., testbench) and adhere to specific conventions for logging, reporting errors, etc.
• Interacting with design is non-standard across simulators, and not supported by vanilla TCL
Architecture

High-level platform independent code: Methodology-Oriented Infrastructure (MOI)

User code
(Application-Specific Environment: ASE)

Methodology support
(Object Infrastructure and Observer: OIO)

Simulator Abstraction Layer (SAL)

Simulator or Log replayer

Platform Extras Adapter (PEA)

Testbench

Platform specific code

Platform abstractor: thin wrapper layers
(Simple Unified Translation Interface: SUTI)
Simulator Abstraction Layer (SAL)

• Abstracts simulator or log replayer (backend)
• Provides functionality normally provided by a simulator, nothing more.
SAL architecture

SAL-common

driver

pathmapper

Simulator or Log replayer
SAL: components

• Driver: minimal set of primitives to interact with backend. Very small.
• SAL-common: Plain TCL code which implements the “virtual simulator” model. Main part of code.
• Pathmapper: associates design- and simulator-specific path names to their logical/virtual equivalents.
Platform Extras Adapter (PEA)

- Handles logging conventions
- Relays DUT errors to main environment, and tallies them
- Provides a configuration database
- Can be expanded to provide more functionality as needed
PEA: components

• Driver: minimal set of primitives to interact with backend. Very small.

• PEA-common: Plain TCL code which abstracts everything else which was not abstracted by SAL. Main part of code.

• SAL and PEA together **abstract away the details of the platform**. Anything built solely on them can be reused on multiple backends without change.
Object Infrastructure and Observer

• Provides a simplified object system
• Provides *handlers*, similar in concept to UVM components.
• Provides *porthandlers*, special primitive handlers which monitor hardware ports.
• Supports *phases*
Application-Specific Environment

• Application layer.
• Agents go here
A second target: NoSim

•~800 lines of code
•Wrote SAL and PEA drivers + pathmapper
•Logs from a simulation run can be replayed **without** restarting the simulator.
NoSim benefits

Parallel, “offline” flow, based on processing logs:

- Edit logs and/or checkers
- Develop prototype
- Once done, plug in test as-is
- **Tests run much faster and**
- By concentrating on critical sections of logs, tests which normally require hours to run can be debugged in minutes.
Results

![Graph showing results of tests with different simulation modes: Simulator quiet, NoSim verbose, NoSim quiet. Tests include test_1 to test_11.]
Questions

Finalize slide set with questions slide