Perspec™ System Verifier
SW Driven Verification

Larry Melling
April 2015
The Eras of Dynamic Verification
Looking at the past and into the future

- **Directed Testing Era** (aka “Stone Age”)
  - Simple ad-hoc testbenches
  - Tests created by hand
  - Not scalable – requires more engineers to scale
  - Difficult to achieve good quality

- **HVL Coverage Driven Verification Era**
  - Constrained-random stimulus automates test creation
  - Coverage metrics used for coverage closure
  - Ideal for exhaustive “bottom up” IP/Subsystem verification
  - Scale verification by automatically generating more tests
  - Best suited for RTL simulation on IP and subsystem

- **SW-Driven Verification Era**
  - SoC “top down” use case testing
  - Automated constrained random generation of SW tests
  - With coverage metrics
  - Reuse across simulation, emulation, FPGA prototype, and post-silicon

Timeline:
- 1980
- 1990
- 2000
- 2010
- 2020
A System-Centric Look at a Modern SoC

- **Many IPs**
  - Standard IO
    - Wifi, USB, PCIe, etc.
  - System infrastructure
    - Interconnect, interrupt control, uart, timers…
  - Differentiators
    - custom accelerators, modem…

- **Many cores**
  - Both symmetric and asymmetric
  - Both homogeneous and heterogeneous

- **Lots of software**
  - Part of core functionality
    - communication stack, DSP software, GPU microcode…
  - User application sw infrastructure
    - Android, Linux…
The Gaps

• How to develop and debug the tests
  – Bare-metal – more directed but more complicated to develop
  – Multicore coherency, concurrency, and power
  – On top of OS – more infrastructure to speed development, but more difficult to debug issues because of side effects that can be introduced by a complex OS

• How do I measure what I have actually tested?
• How do I reuse tests developed for IP at the system-level?
• How to develop tests that work on all my validation platforms?
• What tests to develop?
  – Testing done manually so what to test is driven by what you can think to test and resources
  – Need to drive tests from legal use cases
SoC level Verification & Validation Requirements

- How to communicate/share use cases between users
- How to Create and Reuse Use Cases from IP to SoC
- How to use c code to execute natively on many cores and communicate between cores
- How to run use cases across platforms and run more constrained random variants on faster platforms
The Solution: Perspec System Verifier

Scope (Integration)
- Middleware (Graphics, Audio, etc.)
- OS & Drivers
- Bare Metal SW
- System on Chip (HW + SW)
- Sub-System
- IP

Use Case Reuse

User
- Architect
- HW Developer
- SW Developer
- Verification Engineer
- SW Test Engineer
- Post-silicon Validation Engineer

Vertical Reuse

Horizontal Reuse

Abstract Model with Reusable Use Cases

Powerful Solvers

Perspec System Verifier

Multi-core Verification OS
- Firmware / HAL
- Many cores

Scheduling, Inter-processor communication, runtime randomization

Virtual Platform
- Simulation
- Emulation
- FPGA Prototype
- Silicon Board

c test

IP

Use Case Reuse
The Flow

Abstract System Level Reusable Use Cases
Powerful Solvers and Automatically Generate SW Tests

Virtual Platform | Simulation | Emulation | FPGA Prototype | Silicon Board

Scope (Integration)
- Middleware (Graphics, Audio, etc...)
- OS & Drivers
- Bare Metal SW
- System on Chip (HW + SW)
- Sub-System
- IP

User
- Define Use Case
- Solve abstract use case to create concrete solutions
- Generate testcases
- Debug UML view of testcases & review coverage
- Execute testcases
- SW

Scope (Integration)
- Power
- Communication
- App Processors
- Multi-cluster Apps Processors
- 3D GFX
- Audio
- Boot
- Many cores

Platform
- Virtual Platform
- Simulation
- Emulation
- FPGA Prototype
- Silicon Board

Vertical Reuse
- Define Use Case
- Solve abstract use case to create concrete solutions
- Generate testcases
- Debug UML view of testcases & review coverage
- Execute testcases
- SW

Horizontal Reuse
ST DVCON Presentation on Use of Perspec System Verifier

**Choice of automation tool:**

**Perspec System Verifier:**
- **Model based** approach of the tool
- **Graphical representation** of the generated test
  - UML activity diagram showing all transition from initial to final state
- Generated C test is linear and readable
  - Simplify debug
- First model was developed with local AE support
- No need for deep knowledge of the language
  - Learning curve is in days
  - Not the complexity of UVM for example

**Results**

- **Higher** coverage in **less time** than manual tests development
  - All 192 generated tests are different and cover all states
  - Covering transition we did not think off
  - Estimated manual effort to reach same coverage: 192 days

<table>
<thead>
<tr>
<th></th>
<th>Nb tests</th>
<th>Lines of code</th>
<th>Development</th>
<th>Maintenance (each change)</th>
<th>Nb tests /Day</th>
<th>Nb test/day in case of 5 changes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Manual</strong></td>
<td>20</td>
<td>2k (100x20)</td>
<td>20 days</td>
<td>3 to 4 days</td>
<td>1</td>
<td>0.57</td>
</tr>
<tr>
<td><strong>Perspec</strong></td>
<td>192</td>
<td>800</td>
<td>10 days</td>
<td>1 day</td>
<td>19.2</td>
<td>12.8</td>
</tr>
<tr>
<td><strong>Ratio</strong></td>
<td>0.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>22.4</td>
</tr>
</tbody>
</table>
A Closer Look at the Technology
Why are Use-Cases Important
Translating end user use-case to system-level bare-metal actions

End user use-case:
Mobile phone requirement:
view a video while uploading it
(6 words)

System-level bare-metal actions:
Take a video buffer and convert it to MPEG4 format with medium resolution using any available graphics processor. Then transmit the result through the modem via any available communications processor and in parallel decode it using any available graphics processor and display the video stream on any of the SoC displays supporting the resulting resolution.
(55 words)
Abstract Use Case >> Solver >> Concrete Solution

**Solver**
- Constrained random Data and Control Flow

**Randomize creation of video buffer**
- Randomize video stream attributes
- Randomly select a display for mpeg4 video stream

**Distribute available computing resources**

**Generate C-Code, supplementary files and platform configurations**

**UML activity diagram**
- Choose a mpeg4 convertible video format and save it
Example Use Cases for ARM Multi Core Subsystems
CPU and Memory Configuration Tables

- CPU and memory sub system models are automatically generated by reading system configuration tables (shown below)

User provides the system information:
- Processor info
- Memory info
- MAIR settings
- Preset page table

User can extend the tables to add more user specific attributes and generated code.
Coherency

• Cache
  – Modeling cacheable pages
    – Fixed gen-time or dynamic configuration
    – Read/write allocate, write-through/right-back
  – A cache group
    – Multiple processors targeting same cache region in parallel
    – Sequences of reads/writes/checks
    – Clean/invalidate/barrier instructions
  – Complex cache scenarios
    – Multiple concurrent cache groups
    – Cache groups in parallel fabric/slave stress…
    – Cache groups with power-down/up of cores/L2/CCI

• IO
  – MMU programming for IO redirection
  – Incorporating IPs with IO-coherent masters inside cache groups
    – Allocation of input/output buffers in cache
    – IO coherent IP processing as part of cache group
The MOESI Protocol

- A state machine describes the transitions on a given cache block
- State transitions are caused by
  - Processor read/write instructions
  - External probe (snoop) requests
- The instructions of one processor affect probes on the others

Notes:
- Other transitions are caused by
  - Special instructions
  - Cache management (e.g. eviction)
- Corresponds to a certain cache policy
  - write-back, read-allocate, write-allocate
- MESI and MOSI are variants in which exclusive and owned states respectively are collapsed to shared
The SLN Model

• SLN – System Level Notation
• Transitions are Actions
• States are Tokens
• Constraints are natural translation of state diagram
• Configurable number of processors

```cpp

type line_state_e: [modified, owned, exclusive, shared, invalid];

action read like instruction_base {
    constraint (prev_state.val != invalid) // cache hit
    => next_state.val == prev_state.val;

    constraint (prev_state.val == invalid) => // cache miss
        (snoop_out.hit ?
            (next_state.val == shared) :
            (next_state.val == exclusive));

    // cause snoop if miss
    constraint enabled(snoop_out) == (prev_state.val == invalid);
    constraint snoop_out.req_op == read;
}
```
Reaching Specific Situations

Scenario goal: write to a line when it’s in *shared* state

Control flow view of generated scenario

Scheduling view of generated scenario

Proc2 line transitions: Invalid -> Modified -> Owned -> Invalid
Crossing State Between Cores

Generate scenarios for all combinations

Collect cross coverage on legal combinations
Traversing All Multi-step Transitions

Scenario goal: observe any state on core 3 followed by any state on core 3

Unreachable paths – cannot get back from modified to exclusive (without implicit cache maintenance)

Activity diagram for generated scenarios – getting from exclusive to modified by invalidating the line using another core write
Connecting It Together

Coherency use case

User1

Coherency

Mixed Scenario

User3

Power Shutdown use case

User2

Power Shutdown

1. Cache transactions

2. Power down

3. Cache transactions

4. Power up

5. Cache transactions
Perspec System Verifier Solution

- **Productivity**
  10x improvement for complex SoC test creation

- **Abstraction**
  UML style use-case diagrams

- **Automation**
  System use-case test generation

- **Portability**
  Reuse across all execution platforms

- **Measurement**
  SoC-level HW/SW coverage metrics