Automating Scenario-Level Tests with Portable Stimulus

Staffan Berg
Verification Solutions
Applications Engineer

September 2016
Subsystem and SoC-level Verification

- Requires higher-level tests
  - Scenarios vs transactions
  - High test-creation productivity

- Requires reuse of test intent
  - Develop test intent at block level
  - Reuse block-level intent at subsystem
  - Reuse subsystem-level intent at SoC

- Requires portability
  - UVM tests required at block, subsystem
  - Software-driven tests required at SoC
  - Target simulation, emulation, post-silicon
Scenarios are Everywhere

- Often described with a flowchart

- Verification scenarios
  - Sequences of transactions
  - Design-mode specific activity
  - Coordinated multi-interface activity

- Critical to verify complex designs
  - Complex protocol IP
  - Queue reordering logic
  - Processor pipelines
  - Interconnects
Scenario-Creation Challenges

- Difficult to create – even at block level

- Choices must be made iteratively
  - Must ensure we don’t “dead-end”

- Scenario state space is typically huge

- Need visibility and control over scenarios
  - Ensure we ‘cover’ the scenario space
  - Select an appropriate subset of the scenario space

- Need to have the option to migrate these scenarios forward
  - SoC level
  - System level

Automating Scenario-Level Tests with Portable Stimulus
PCIe Scenario Example

- Send PCIe TLP
- Inject a link event (optional)
- Send another PCIe TLP

- Cover all combinations of: TLP types, link events
Directed-Random Test
PCIe Scenario

- Difficult to write
  — Heavily dependent on test writer skill

- Difficult to review
  — No high-level view – just code

- Difficult to control
  — Hard to constrain from outside
  — Hard to tell what actually happens

- Difficult to migrate/reuse
  — Specific implementation
  — Specific language
  — Specific execution environment

```cpp
define enum { EV1, EV2, EV3, EV_NONE } event_t;

class pcie_scenario_seq extends uvm_sequence;
    task body();
        pcie_sequence_item tlp1 = pcie_sequence_item::type_id::create();
        pcie_sequence_item tlp2 = pcie_sequence_item::type_id::create();
        event_t ev;
        forever begin
            start_item(tlp1); tlp1.randomize(); finish_item(tlp1);
            std::randomize(ev);
            case (ev)
                EV1: run_ev1_seq();
                EV2: run_ev2_seq();
                EV3: run_ev3_seq();
                EV_NONE: // Do nothing
            endcase
            start_item(tlp2); tlp2.randomize(); finish_item(tlp2);
        end
    endtask
endclass
```
Graph-Based Portable Stimulus Test
PCIe Scenario Graph

- Easy to write
  - High-level specification
  - Efficient to create

- Easy to review
  - Hierarchical
  - Expand to see elements of interest

- Easy to control
  - Specialize from outside
  - Goal-driven generation targets goals

- Easy to migrate/reuse
  - Independent declarative language
  - Independent of specific execution env
**Questa inFact**
Graph-Based Portable Stimulus

- Efficient description of stimulus scenarios
  - 10-100x more productive than directed tests

- Efficient execution of complex stimulus
  - 10-100x more efficiently than random generation

- Broad environment support
  - SystemVerilog, UVM, OVM
  - C/C++/SystemC
  - Embedded software

- Verify more with the same resources!
Graph-Based Stimulus Description
Data, control flow, relationships

- Captures data and control flow aspects of test scenario
  - Describes legal stimulus scenario space
  - Specify textually, visualize graphically

- Data and data relationships
  - Scalar, composite data types
  - Algebraic relationships
  - Inside, if/else, foreach, etc

- Control flow
  - Sequences of operations
  - Choices
  - Loops

```c
struct my_struct1 {
    meta_action     A[unsigned 3:0];
    meta_action     B[unsigned 3:0];
}

struct my_struct2 extends my_struct1 {
    meta_action     C[unsigned 3:0];
    constraint c {
        C <= A;
    }
}
```

Automating Scenario-Level Tests with Portable Stimulus
Bringing it all Together
PCIe TLP Rules/Graph

- Define scenario with
  - Two TLP items
  - Inject event between

- Run scenario in a loop
Bringing it all Together
PCIe TLP Rules/Graph

- Define scenario with
  - Two TLP items
  - Inject event between
- Run scenario in a loop
- Built on top of VIP sequences
  - Import from library
- Concise description
- Reusable

```c
rule_graph pcie_scenario_seq {
  import "pcie_sequence_item.rseq";
  action init;
  interface do_item(pcie_sequence_item);

  set event_t[enum EV1, EV2, EV3, EV_NONE];

  struct pcie_scenario {
    pcie_sequence_item tlp1, tlp2;
    meta_action ev[event_t];
    action run_ev1_seq, run_ev2_seq, run_ev3_seq;

    symbol inject_ev = ev
      if {ev == EV1} run_ev1_seq | if {ev == EV2} run_ev2_seq | if {ev == EV3} run_ev3_seq | if {ev == EV_NONE} eta;
  }

  pcie_scenario =
    do_item(tlp1)
    inject_ev
    do_item(tlp2);
}

pcie_scenario s1;
pcie_scenario_seq = init repeat {
  s1
};
```
Test Selection and Prioritization

Coverage Strategy

- Captures test scenario goals
  - Key stimulus values, value ranges
  - Key stimulus combinations
  - Key stimulus sequences

- Flexible
  - Prioritize certain goals
  - How random, how systematic

- Goal-Driven Systematic Generation
  - Automatically suppresses redundant stimulus
  - Reaches goals 10-100x faster than pure-random generation
Bringing it all Together
PCIe Scenario Coverage Goals

- Total scenario space: >6 quadrillion
  — TLP1 fields X events X TLP2 fields

- Target combination of
  — TLP1 type
  — Event type
  — TLP2 type

- Total targeted scenarios: 3600

- Efficient scenario generation
  — No redundancy, predictable completion
Graphs and Reuse
Compose Hierarchical Scenarios

- Easily compose larger scenarios
  - Graphs are object oriented
  - Support composition, inheritance

- Over-constrain to shape

- Two back-to-back scenarios
  - Customized TLP types
  - Customized events

- 28,800 scenarios
**Bringing it all Together**

**UVM Integration**

- **Scenario graph inside a UVM virtual sequence**
  - Generate TLPs via PCIe agent
  - Launch event-injection sequences

- **Just Another UVM Sequence**
  - Integrates with the UVM factory
  - Use without understanding the internals

### Automating Scenario-Level Tests with Portable Stimulus

---

© 2014 Mentor Graphics Corp. Company Confidential

[www.mentor.com](http://www.mentor.com)
**Bringing it all Together**

Embedded SW Integration

- Scenario graph realized as embedded software
  - Calls utility functions to trigger PCIe traffic
  - Calls utility functions to trigger link events

- Just another C-Test
  - Call just like any other test routine
Accellera Portable Stimulus Standard
- Mentor & Cadence submitted joint language proposal
- Selected as baseline for standard in August 2016
- Standard will also have semantically-equivalent C++ input format

Declarative graph-based scenario specification
- Raises abstraction level
- Efficient scenario capture and review
- Single specification portable across multiple targets

Questa inFact: Portable goal-driven stimulus
- 10-100x more efficient than random
- Faster coverage closure at block level
- Block to system-level reuse
Integration Example
Embedded software

- Graphs call existing ‘C’ methods
  - Select parameter values
  - Sequence method calls

- Graphs can run
  - Independently
  - Cooperatively

- Supports
  - Simulation
  - Emulation
  - Silicon

Automating Scenario-Level Tests with Portable Stimulus
Graphs and Reuse
Build on existing SV Classes

- Imports
  - SV classes
    - Fields, constraints
  - Covergroups
    - Coverpoints, crosses

- Creates
  - Graph rules
  - Coverage directives

- Leverages existing SV

- Raises abstraction level
  - Import transaction-level details
  - Build complex scenarios on top

Automating Scenario-Level Tests with Portable Stimulus
Microsoft U2U Presentation

Productivity benefits beyond efficient stimulus

- inFact is primarily positioned as a stimulus-generation tool

- Also significant additional value
  - Constraint analysis and debug
  - Functional coverage creation

- Microsoft cited these at U2U
  - In addition to efficient stimulus

---

Productivity Benefits

- Expand from a few hundred Functional Coverage bin per bench to 10’s of thousands bin
- Cut Random stimulus development time from 3, 4 weeks to 1 week
- Cut back on Functional Coverage development time to almost nothing
- Spend double amount of time on bug hunting (from 2 weeks to 3, 4 weeks)
Stimulus Creation Techniques

- **Directed tests**
  - Focused, user-created
  - Captures data and control flow
  - Low-productivity

- **Constrained-random tests**
  - Open loop, automation driven
  - Captures data
  - High-productivity

- **Graph-based portable stimulus**
  - Flexible focus, automation driven
  - Captures data and control flow
  - High-productivity, goal driven
Graph-Based Scenario Generation Results

<table>
<thead>
<tr>
<th>Industry</th>
<th>Server on Chip</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design</td>
<td>Cache Controller</td>
</tr>
<tr>
<td>Existing</td>
<td>Questa SystemVerilog CRT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>New</th>
<th>Questa Ultra iTBA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value 1</td>
<td>+90% coverage</td>
</tr>
<tr>
<td>Value 2</td>
<td>No need to write directed tests</td>
</tr>
<tr>
<td>Integration</td>
<td>½ Day</td>
</tr>
</tbody>
</table>

Complex coverage efficiently hit. No need to write directed tests

Simulation: 1 hour simulation
Coverage: 10%

Weekend regression: 100%
Graph Integration Example
Virtual Sequence UVM

- Graph runs in a UVM virtual sequence

  - Graph execution
    - Selects sequence parameters
    - Starts sequences

  - Sub-sequences can be
    - Graph-based UVM sequences
    - Random or directed sequences
Test Specification Fundamentals

- **What is legal**
  - Universe of what could happen
  - Captures both data and scenario
  - Enables creation of ‘unexpected’ cases

- **What to target**
  - Cases of specific interest
  - What to verify today, during this test
Questa inFact Stimulus
10.4x-10.5 Update Overview

- New modeling constructs
  - Direct support for more SystemVerilog constructs
  - Coverage goal specification enhancements

- Distributed Simulation Enhancements
  - Save coverage progress across SDM runs
  - Performance improvements

- SystemVerilog Import Enhancements
  - Coverage variable mapping
  - Multi-library support

- Graph Trace for Debug
  - Improved visibility into graph execution
Effective Verification Process

- Starts with a plan
  - Functionality to exercise
  - How functionality will be exercised

- Tests implement the plan items

- Test-creation challenges
  - Large numbers of configurations to test
  - Need to generate scenarios, not just transactions
  - Functionality must be exercised at block and SoC level