Automating SoC-Level Tests with Portable Stimulus

Staffan Berg
European Applications Engineer
Digital Functional Verification

April, 2015
Test Creation Challenges
Block to System

- Large state spaces
- Require scenarios, not just transactions

- Multiple verification platforms
  - Simulation
  - Emulation
  - Prototype

- Multiple languages
  - SystemC
  - SystemVerilog
  - Embedded software

- Must have automation
- Must have reuse
Stimulus 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

![Diagram](image-url)
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 goal-driven tests
  - Flexible focus, automation driven
  - Captures data and control flow
  - High-productivity, goal driven
Graph-Based Stimulus Description

- Stimulus scenario described using **Rules**
  - Captures data and control flow aspects of test scenario
  - Describes legal stimulus scenario space
  - Efficient description mechanism

- Rules are compiled into **Graphs**
  - Visual representation of the stimulus model
  - Easy to review

```
rule_graph trans_eng {
    action init, infact_checkcov;

    struct trans {
        meta_action data [unsigned 7:0];
        meta_action addr [unsigned 32:0];
        constraint on_addr {addr < 0x8000}
    }

    trans trans_inst;
    interface fill_trans(trans);

    trans_eng = init repeat {
        fill_trans(trans_inst)
        infact_checkcov
    }
}
```
Graph-Based Stimulus Description
Captures data and data relationships

- Scalar types
  - Signed and unsigned integer types
  - Enumerated types

- Composite data structures
  - ‘struct’, supports type extension

- Aggregate data types
  - Single and multi-dimensional fixed-size arrays

- Variables can be input or output
  - Output variables (default) send values to the environment
  - Input variables bring values in from the environment

- Constraints
  - Algebraic expressions, inside, if/else, foreach, etc

```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;
    }
};
```
Graph-Based Stimulus Description
Captures test scenario control flow

- Captures process of stimulus generation
  - Sequences of operations
  - Choices
  - Loops

- Branch-specific constraints
  - Conditional execution
  - Partitions scenario structurally

```c
constraint a_eq_b dynamic {
  inst3.A == inst3.B
}
my_graph = init repeat {
  inst1
  if (inst1.A == 5) (inst2)
  if (inst1.A >= 5) (inst3)
  if (inst1.A == 6) (a_eq_b inst3)
  infact_checkcov
};
```
Graph-Based Stimulus Description

Interfaces

- An interface specifies a port between graph and environment

- Interfaces are declared in the rules
  - Accept one or more parameters
  - Parameters may be input or output

- Interfaces are implemented in an environment-specific way
  - UVM: call start_item/finish_item
  - Embedded SW: call function

- Interface implementation is user-customizable

```c
interface do_ethmac_tx_seq_item(ethmac_tx_seq_item);
infact_ethmac_tx_seq = init repeat {
    do_ethmac_tx_seq_item(ethmac_tx_seq_item_inst)
    infact_checkcov
};
```
Test Selection and Prioritization

Coverage Strategy

- Coverage strategy expresses test scenario goals
  - Key stimulus values, combinations, sequences

- Flexible
  - Prioritize certain goals
  - Combine random/systematic generation

- Reactive
  - Adapts to changes in the DUT or the verification environment state

- Efficient
  - Automatically suppresses redundant stimulus
  - Reaches goals 10-100x faster than pure-random generation
Test Selection and Prioritization
Target value specification

- Variable domains can be divided using ‘bins’
  - Split a value range into N bins
  - Split a value range into bins of size N

- Coverage constraints select target space with expressions
  - Only active when inFact is targeting the coverage goal
  - Prioritizes specific combinations
  - Full legal space reachable otherwise

- Example: A x B
  - Full legal space is 256 (16 * 16)
  - Constraint A < B selects 120 combinations

```
my_struct1 inst;

bin_scheme small_vals {
    inst.A [0..3]: 1 [4..15]/4;
    inst.B [0..1]: 1 [2..15]/8;
}
```

```
my_struct1 inst;

constraint a_less_b coverage {
    inst.A < inst.B;
}
```
Traffic Generation Example

- Simple Multi-master multi-memory system
- Goal: Exercise datapath combinations
  - C0 transfers DDR0 => IRAM0
  - C1 transfers DDR0 => DDR1
  - ...
- Constraints:
  - DMA0 cannot access IRAM
  - DMA transfer sizes are word-aligned
Traffic Generation Example
Scenario Description

- **mem_transfer** captures transfer attributes
  - source, destination region
  - size, etc

- Top-level scenario captures
  - Interfaces to the environment
  - System constraints
  - Target values

```c
// Memory regions
set region_e[enum DDR0, DDR1, IRAM0, IRAM1];

struct mem_transfer {
  meta_action src[region_e];
  meta_action dst[region_e];
  meta_action src_offset[unsigned 0..15];
  meta_action dst_offset[unsigned 0..15];
  meta_action size[unsigned 31:0];
  // Limit size between 1 and IM
  constraint region_sz_c {
    size inside [1.1024*1024*1024];
  }
}

// Interfaces to launch traffic
interface start_cpu0(mem_transfer t);
interface start_cpu1(mem_transfer t);
interface start_dma0(mem_transfer t);
interface start_dma1(mem_transfer t);

mem_transfer cpu0_t, cpu1_t, dma0_t, dma1_t;

// Engine-specific constraints
constraint engine_c {
  (dma0_t.size & 0x7) == 0;
  (dma1_t.size & 0x7) == 0;
  // DMA0 cannot access IRAMs
  dma0_t.dst outside [IRAM0, IRAM1];
  dma0_t.src outside [IRAM0, IRAM1];
}

// Target various transfer sizes
bins mem_transfer.size [1..7]:1 [8..65535]/8;

mem_traffic_scenario = init repeat {
  start_cpu0(cpu0_t) start_cpu1(cpu1_t)
  start_dma0(dma0_t) start_dma1(dmal_t)
  init_fcheckov
};
```
Traffic Generation Example
Coverage-Target Selection

- Legal scenario space is huge
  - >10 billion combinations

- Target all legal src/dst combinations
  - C0 src=DDR0, dst=DDR1

- Target combinations of transfer size
  - C0 size=25 DMA0 size=32

- Total:
  - 16384 legal src/dst combinations
  - 20736 legal size combinations
Reuse and Portability

- Leverage existing descriptions
  - Existing constraints
  - Existing register descriptions

- Share knowledge across domains
  - Block-level verification
  - SoC-level verification
  - Domain experts

- Automation derives specific implementations
  - Simulation, emulation, prototype
  - HVL, embedded software
Graphs Enable Reuse and Portability

- Graphs are language and environment-independent
  - Self-contained
  - Describe data and sequence scenario

- Graphs are mapped to specific environments
  - Communicate data to/from environment
  - Synchronized to environment execution

- Can reuse graphs across environments
  - Create with UVM, reuse with embedded sw
  - Create with C model, reuse in UVM
Graph Reuse
Mapping Traffic Generation to UVM

- Graph implemented in a virtual sequence
- Interfaces launch sequences on agents
  - C0, C1 agents
  - DMA-programming agents
- Sequences could be
  - Graph-based
  - Existing sequences
Graph Reuse
Mapping Traffic Generation to Embedded SW

- Graph implemented as a C program
- Interfaces call C functions
  - Perform programmatic transfers via processors
  - Program DMA to carry out transfers

Virtual Sequence

Test

```c
main() {
    ...
}
```

DMA Driver

Memory Library

DDR0

DDR1

C0

IRAM0

C1

IRAM1

DMA0

DMA1
Graph Reuse Across Languages
C Model Verification

- Verify C model for high-level synthesis
  - SystemC simulation environment

- Re-run same tests on RTL result of synthesis
  - SystemVerilog simulation environment
  - UVM testbench

- Graph Benefits
  - Same graph used in both environments
  - Highly-productive test creation SystemC
  - Systematic verification
Graph-Based Portable Stimulus

- **High-Productivity Input Specification**
  - Familiar data constructs and constraints
  - Formally captures control flow

- **Efficient and flexible execution**
  - 10-100x more efficient than random execution
  - Scalable to a simulation farm

- **Portable**
  - Existing HVLs, embedded software
  - Simulation, Emulation, Prototype

- **Highly Automatable**
  - Import/export
  - Analysis