Adaptive Protocol Checks
Configuration Aware Assertions

Mark Litterick
Introduction

• Motivation
  – inconsistent SVA setup across clients & projects
  – many protocols where adaptive SVA required

• SVA encapsulation in UVM
  – why, what & where?

• Techniques for making SVA configuration-aware
  – why & how?

• Techniques for making SVA phase-aware
  – automatically self-configuring SVA checks
  – initial build and dynamic run-time configuration
Distributed UVC Checks

- Types of UVC checks:
  - *signal protocol* and timing
  - *transaction content* and functional correctness
  - *transaction comparison* and relationships
- Each is handled by different component

All checks *belong* to UVC

Concurrent *assertions* are *not allowed* in SystemVerilog *classes*
Check Configuration & Control

• All checks can be affected by:
  – control knobs (e.g. checks_enable)
  – config fields (e.g. cfg.mode)

• Configuration object fields can be:
  – constant after build-phase
  – dynamic during run-phase

```plaintext
class my_monitor ...
if (condition)
  // update config
cfg.speed_mode = FAST;

class my_test ...
  uvm_config_db#(my_config)::set
    (this,"*","cfg",cfg);
  uvm_config_db#(bit)::set
    (this,"*","checks_enable",0);

class my_monitor ...
  if (checks_enable)
    trans.check_crc();

class my_monitor ...
  trans.check_parity(
    cfg.odd_even
  );
```
sequence s_fast_transfer;
  REQ ##1 !REQ[*1:4] ##0 ACK;
endsequence

sequence s_slow_transfer;
  REQ ##1 !REQ[*3:10] ##0 ACK;
endsequence

property p_transfer;
  @(posedge CLK)
  disable iff (!checks_enable)
  REQ |-
  if (cfg_speed_mode == FAST)
    s_fast_transfer;
  else
    s_slow_transfer;
endproperty

a_transfer:
  assert property (p_transfer)
  else $error("illegal transfer");
SVA Interface

this interface is required for UVC-DUT signals

... but what about SVA?
interface my_interface;

// local signal definitions
// modport declarations
// clocking blocks
// bus-functional methods

// support code for SVA
// property definitions
// assertion statements
endinterface

SVA code is verbose, complex & not related to signal communication

=> isolate and encapsulate SVA
interface my_interface;
  // local signals
  logic       CLK;
  logic       REQ;
  logic       ACK;
  logic [7:0] DATA;
  logic       OTHER;
  ...
  // modports, etc.
  ...
  // protocol checker
  my_sva_checker
    sva_checker(.*);
endinterface

interface my_sva_checker(
  // signal ports
  input logic       CLK,
  input logic       REQ,
  input logic       ACK,
  input logic [7:0] DATA
);
  // support code
  // properties
  // assertions
endinterface

**module not allowed** inside interface => use **interface**

required signal subset **well encapsulated**

**implicit instantiation**

ports have same name as interface signals (but can be a subset)
**SVA Configuration**

```verilog
interface my_sva_checker
  // control knobs
  bit checks_enable = 1;
  // config object
  my_config cfg;
  // local variables for SVA
  my_speed_enum cfg_speed_mode;
  int unsigned cfg_max_value;
  bit cfg_data_en;
  // properties and assertions...
  // update local vars from cfg...
endinterface
```

- **CONTROL KNOBS**
- **CONFIG FIELDS**

**Warning:**
- No class variables inside concurrent assertions
- Class variables are allowed in support code

**Interface is not a phased component**

When is `config` class built?
Method API

push CFG from class environment to SVA interface

manual operation and only partially encapsulated by interfaces
interface my_sva_checker(...);

function void set_config (my_config cfg);
  cfg_speed_mode = cfg.speed_mode;
  cfg_max_value = cfg.max_value;
  cfg_data_en = cfg.data_en;
endfunction

function void set_checks_enable (bit en);
  checks_enable = en;
endfunction

endinterface

interface my_interface;

function void set_config (my_config cfg);
  sva_checker.set_config (cfg);
endfunction

function void set_checks_enable (bit en);
  sva_checker.set_checks_enable (en);
endfunction

endinterface

user API via VIF methods

required fields well encapsulated

internal SVA details well hidden
API – Initial Configuration

could be done in **agent** or **monitor**

call set_* methods via virtual interface after build and connect

```plaintext
class my_monitor extends uvm_monitor;
...
  function void end_of_elaboration_phase(...);
...
    // set interface config after build
    vif.set_config(cfg);
    vif.set_checks_enable(checks_enable);
endfunction
...
endclass
```

must be after **VIF** assigned
API – Dynamic Configuration

class my_monitor extends uvm_monitor;
  ...
  task run_phase(...);
    forever begin
      ...
      if (cfg updated) vif.set_config(cfg);
    end
  endtask
endclass

call set_config via virtual interface when update required

must be PASSIVE component

additional work for monitor hard to debug if done wrong
Phase-Aware Interface

reference CFG from class environment inside SVA interface

automatic and fully encapsulated in interface
class is UVM phased component

locally declared class can see all local variables in interface

interface my_sva_checker(...);
  // declare local variables (cfg, checks_enabled, etc.)
...
  class checker_phaser extends uvm_component;
  // use local variables (cfg, checks_enabled, etc.)
  // use UVM phases (build, connect, run, etc.)
endclass

// construct unique instance of phaser
// (at the top-level under uvm_top)
checker_phaser m_phase = new($psprintf("%m.m_phase"));
endinterface

no component_utils if multiple instances required
(can’t register same type multiple times with factory)

unique name required for multiple instances

unique name provides better debug messages
Parallel Phases

**SVA interface class in parallel with main env under uvm_top**

- **UVM_TOP**
- **ENV CLASS**
- **SVA CLASS**

UVM phases run in parallel

Both build phases complete before either connect phase starts

... but order of build phase completion cannot be predicted!

Potential race scenario

© Verilab & Accellera
SVA Class – Initial Configuration

```verbatim
class checker_phaser extends uvm_component;
...
  function void end_of_elaboration_phase(...);
  ...
  if (!uvm_config_db#(my_config)::get(this,"","cfg",cfg))
    `uvm_fatal("CFGERR","no my_config cfg in db")
  void'(uvm_config_db#(bit)::get
    (this,"","checks_enable",checks_enable));

  cfg_speed_mode = cfg.speed_mode;
  cfg_max_value  = cfg.max_value;
  cfg_data_en    = cfg.data_en;
endfunction
endclass
```

- **Use** `end_of_elaboration_phase` to avoid race condition
- `cfg` must be provided by db
- `checks_enable` may be overloaded by db
- Copy required `cfg` fields to local variables for use in SVA
- Required `cfg` fully encapsulated
- No demands on monitor class
Class – Dynamic Configuration

```verilog
interface my_sva_checker(...);
...
  always @(cfg.speed_mode or cfg.data_en) begin
    cfg_speed_mode = cfg.speed_mode;
    cfg_data_en = cfg.data_en;
  end
endinterface

class checker_phaser ...;
...
  task run_phase(...);
...
    forever begin
      @(cfg.speed_mode or cfg.data_en) begin
        cfg_speed_mode = cfg.speed_mode;
        cfg_data_en = cfg.data_en;
      end
      `uvm_info("CFG","cfg updated",UVM_LOW)
    end
endtask
endclass
```

- `uvm_info("CFG","cfg updated",UVM_LOW)` detects change in `cfg` object.
- `always@` and `always_comb` in do not work since `cfg` is null at start of simulation.
- Use `run_phase` to check for dynamic `cfg` changes.
- Only required dynamic `cfg` fields.
- `forever @*` and `@(*)` do not work (cfg object does not change only fields inside the object).
Conclusion

• Practical suggestions for **SVA encapsulation**
  – basic encapsulation inside interface

• Demonstrated **configuration-aware SVA**
  – method API with demands on UVC class
  – **phase-aware SVA** checker with embedded class

• Successfully used in many **UVM projects**

• Validated with **multiple tools** in different clients
References and further reading

• “SVA Encapsulation in UVM – Enabling Phase and Configuration Aware Assertions”
  M.Litterick, DVCon 2013, www.verilab.com/resources
Questions