2026
DESIGN AND VERIFICATION™
DVCON
CONFERENCE AND EXHIBITION
UNITED STATES
SANTA CLARA, CA, USA
MARCH 2 - 5, 2026

Harnessing Volatility: Innovative Strategies for Register Synchronization in UVM RAL

Bhaskar Vedula, Stephen P. Haake, Chandrakanth N. Betageri

Agenda

- Why volatile fields break RAL mirroring in real life.
- Why cookbook prediction isn’t enough for HW-autonomous updates.
- Event-Driven RAL Synchronization: Proposed Architecture.
- Key Implementation Components and Usage.
- Results, Deployment Experience, and Limitations
- Q&A

Problem Statement

Core Technical Issue:

- Volatile register fields change asynchronously due to hardware events, such as interrupts, power state transitions, or real-time monitoring updates.

Specific Problem Areas:

- Interrupt status registers that self-clear or auto-update
- Power state machine registers reflecting hardware transitions
- Configuration registers modified by the hardware microcontroller
- Thermal and performance monitoring registers with real-time updates

Bus Visibility Challenge:

- These hardware-driven changes occur independently of bus transactions and are not visible on the bus interface. As a result, standard UVM RAL prediction mechanisms—which rely on bus activity—cannot detect or update the register model to reflect these changes.

Verification Impact:

- If verification is performed based on UVM RAL, when the register model's mirrored values become stale due to undetected hardware updates, test sequences may make incorrect decisions, leading to false verification results and missed hardware events.

Current Industry Gap: No standardized UVM methodology exists for handling hardware-autonomous register field changes.

UVM RAL Architecture: Foundations and Limitations

Dual Value Model

- **Desired value:** Stores the value of what we *want* to set to the DUT.
- **Mirrored Value:** Stores the value of what we think in our DUT

Prediction Mechanisms

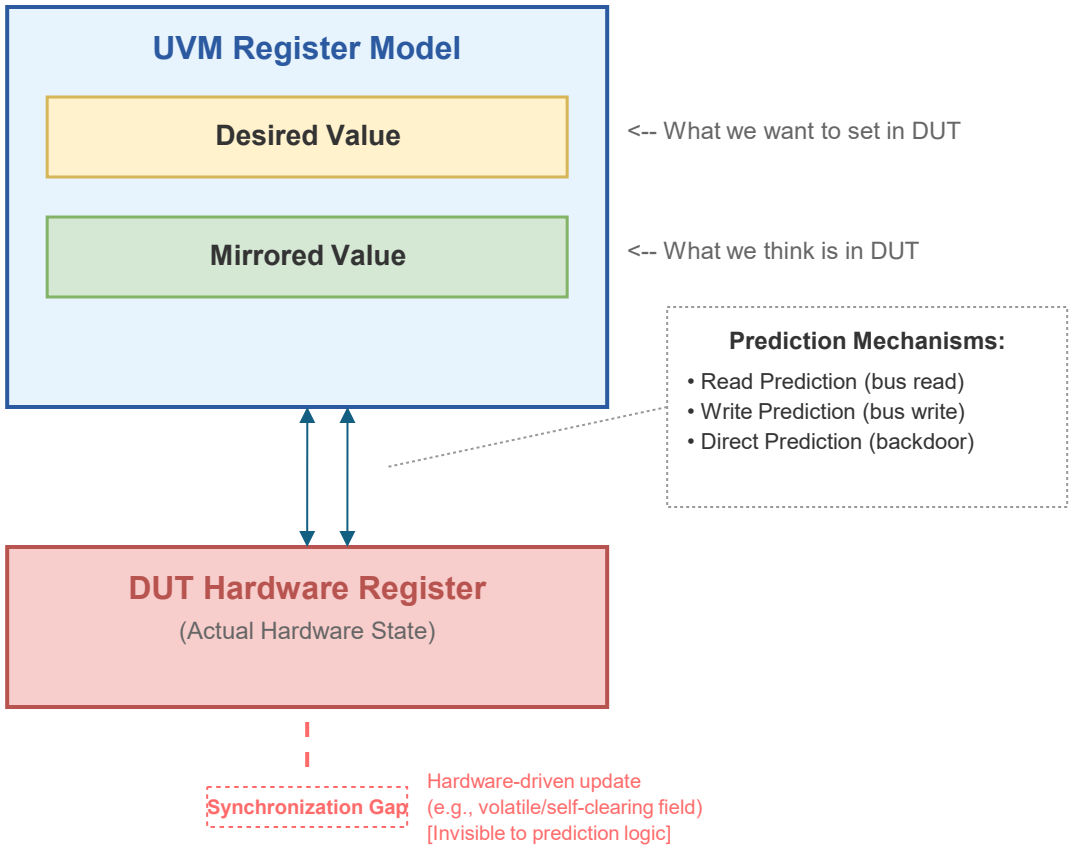
- **Read Prediction:** Updates the mirrored value using data returned from bus read transactions.
- **Write Prediction:** Updates the mirrored value after a bus write.
- **Direct Prediction:** Allows the mirrored value to be updated through backdoor access, without bus activity.

Key Limitation

Hardware-driven updates—such as changes to volatile or self-clearing fields—occur without bus transactions and are therefore invisible to standard prediction logic.

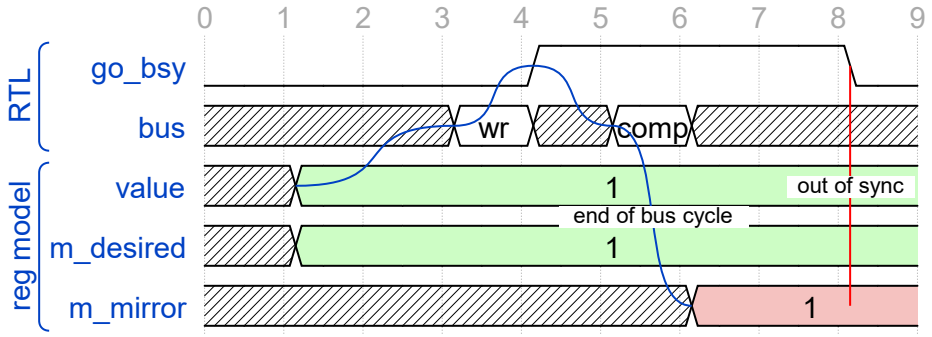
Synchronization Gap

UVM RAL provides no built-in mechanism to automatically detect and synchronize autonomous hardware updates, leading to potential mismatches between the register model and actual hardware state.



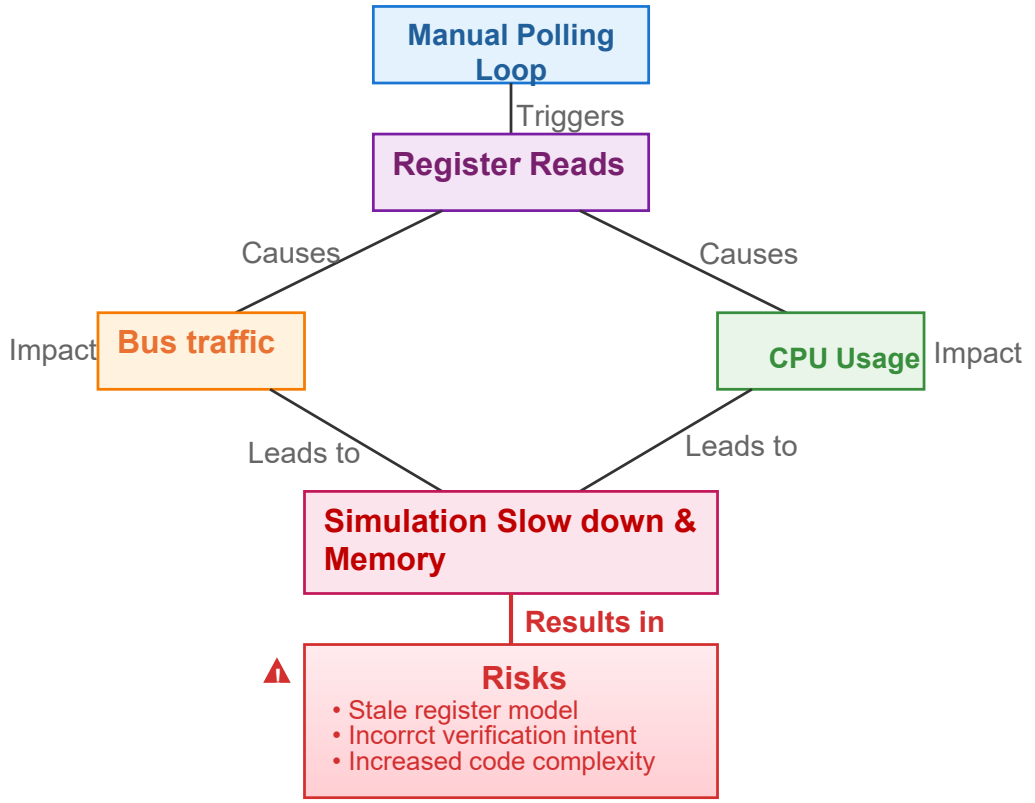
Volatile Register Challenges

- Volatile register fields can be updated by hardware at any time, independent of testbench actions.
- These hardware-driven changes are not visible on the bus, so standard UVM RAL prediction mechanisms cannot detect them.
- Example: After a bus write sets a field (for example, **go_bsy**), hardware may clear it autonomously, but the register model still shows the old value, causing "**out of sync.**"
- The register model's **m_mirror** value may become stale, causing it to mismatch the actual hardware state.
- Using **set() and update()** methods in sequences may not trigger a bus transaction, further masking hardware changes.
- Lack of synchronization can result in incorrect stimulus generation.



Traditional Approaches & Limitations

- Manual polling causes excessive CPU usage and simulation slowdown.
- Arbitrary timing delays can lead to race conditions and unpredictable results.
- Unnecessary bus traffic and memory overhead degrade performance.
- Debugging and maintaining polling logic is difficult, especially for large designs.
- No standardized UVM solution; each team implements custom logic.
- Risks:
  - Stale register model.
  - Incorrect verification intent.
  - Increased code complexity and maintenance burden.

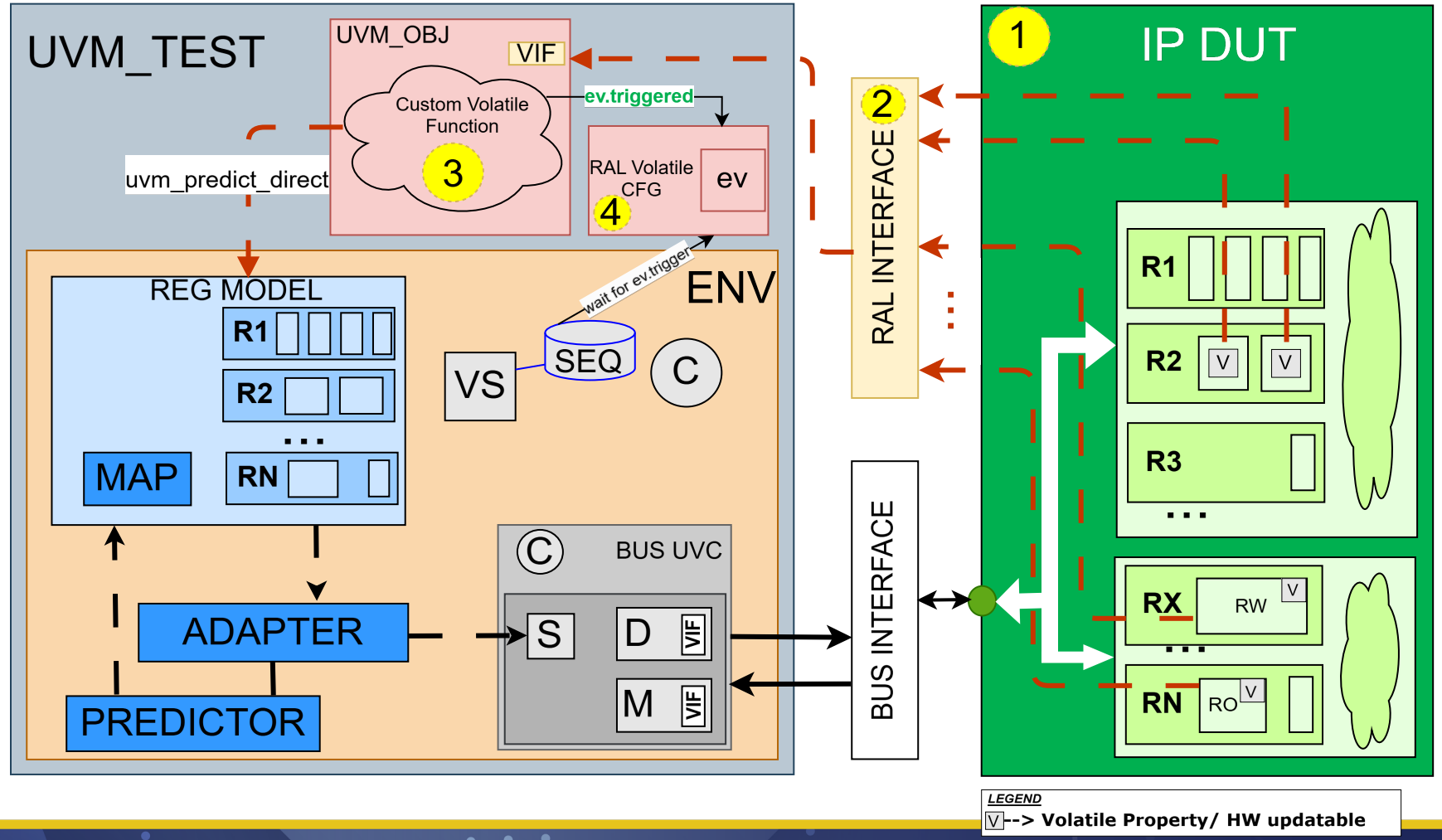



```
// 1) Synchronize reg model with actual HW state
register.mirror();    // updates mirrored value (m_mirror) from DUT

// 2) Program the new intent
register.set();       // updates desired value (m_desired) in the model

// 3) Drive the bus only if model sees a delta
register.update();    // bus access occurs only if (m_mirror != m_desired)
```

```
// 1) Program the new intent
register.set();

// 2) Force a bus write regardless of mirrored state
register.write();     // always issues a bus transaction
```

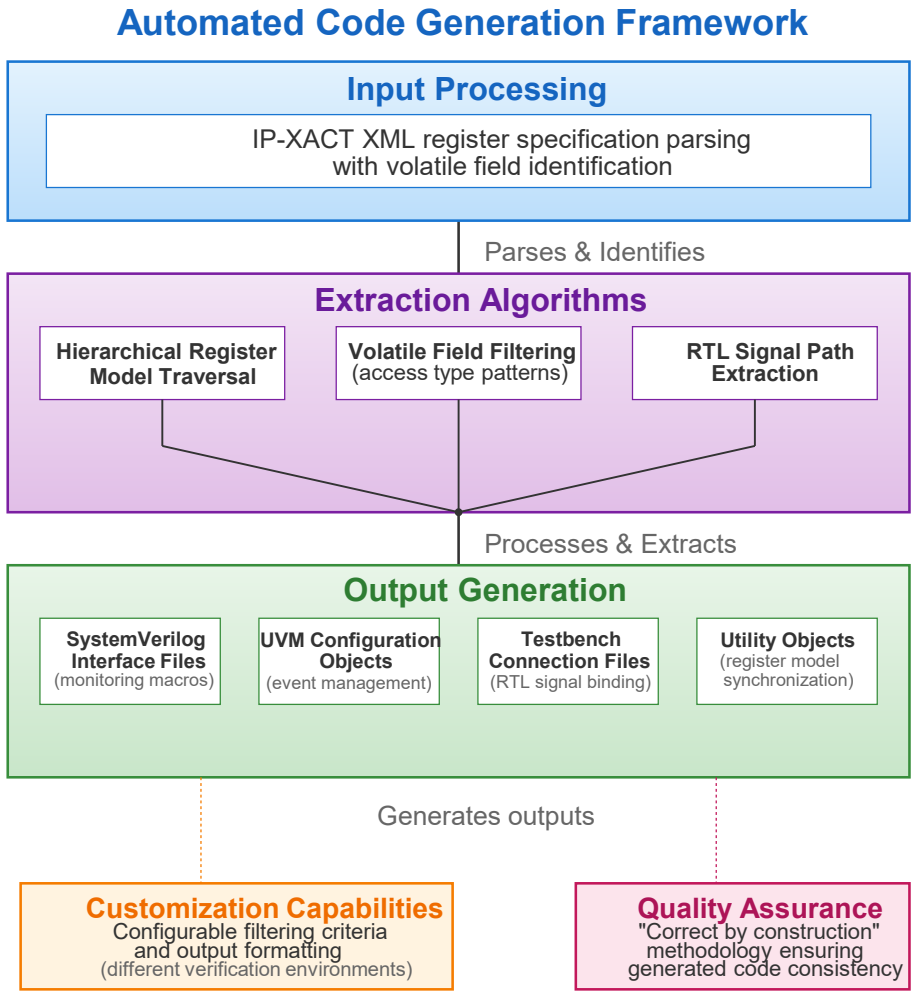
Solution Overview – Proposed Technical Architecture

- Shifts from reactive polling to proactive, automated hardware monitoring for volatile register fields.
- Integrates core components:
- Python based RAL script auto-generates infrastructure from IP-XACT XML.
  - RAL Monitor Interface detects hardware signal changes in real time. (Label 2)
  - RAL Volatile Utility Object bridges hardware and UVM class domains. (Label 3)
  - RAL Volatile Configuration Object manages UVM events and data containers. (Label 4)
- **Key innovation:** Automatic register model synchronization via ***UVM_PREDICT_DIRECT,*** triggered by hardware signal changes.
- Eliminates manual polling in verification sequences, reducing code complexity and improving efficiency.



Python RAL Script Technical Implementation

- **Automated Code Generation Framework**
- **Input Processing:** IP-XACT XML register specification parsing with volatile field identification.
- **Extraction Algorithms:**
  - Hierarchical register model traversal.
  - Volatile field filtering based on access type patterns.
  - RTL signal path extraction for hardware monitoring.
- Shifts from reactive polling to proactive, automated hardware monitoring for volatile register fields.
- **Output Generation:**
  - SystemVerilog interface files with monitoring macros.
  - UVM configuration objects with event management.
  - Testbench connection files for RTL signal binding.
  - Utility objects for register model synchronization.
- **Customization Capabilities:** Configurable filtering criteria and output formatting for different verification environments.
- **Quality Assurance:** "Correct by construction" methodology ensuring generated code consistency.



Hardware Signal Monitoring Mechanism

- Macro instantiated for each volatile field in the interface.
- Detects hardware signal changes (compares current and previous values).
- On change, calls **notify_ral()** in the utility object.
- Clean separation between the hardware monitoring domain and the UVM class-based register model domain.
- Enables robust, automated synchronization.

```
`define RAL_DETECT_AND_NOTIFY_CHANGE_MACRO(clk, resetn, SIGNAL_WIDTH, input_signal, cfg,
    ral_path) \
    logic [SIGNAL_WIDTH-1:0] ``input_signal``_ff; \
    logic event_detected;                         \
    always @(posedge clk) begin \
        if (~resetn) begin \
            ``input_signal``_ff <= {SIGNAL_WIDTH{1'b0}}; \
        end else begin \
            proxy_pointer.notify_reset_ev(cfg.``input_signal``_change); \
            event_detected = (input_signal != ``input_signal``_ff) ? 1'b1 : 1'b0; \
            ``input_signal``_ff <= input_signal; \
            if (event_detected) begin \
                proxy_pointer.notify_ral(input_signal, ral_path, cfg.``input_signal``_change)
                    ; \
            end \
        end \
    end \
```

Macro to detect and notify

Change detection

Notification to the UVM side

RAL Monitor Interface Architecture

- SystemVerilog interface auto-generated to declare and monitor all volatile register fields.
- Wire declarations for each volatile field, with correct bit widths and signal names.
- Automated instantiation of monitoring macros for each field, enabling change detection and event generation.
- Centralized management of handles to RAL configuration objects, utility objects, and register block references.
- Integrated clock and reset signals ensure proper synchronization with system timing.
- Designed for scalability, supporting thousands of volatile fields without performance degradation.

```
//Start: Python script generated code
interface ral_monitor_if  (clk, resetn);
    input clk, reset;
    // Handle declarations
    ral_config_pkg::ral_config                  ralCfg;
    uvm_mon_pkg::ral_monitor_util               proxy_pointer;
    reg_block_pkg::ip_reg_block                 ip_rb;

    wire [0:0] ctrl_reg_go_bsy;
    `RAL_DETECT_AND_NOTIFY_CHANGE_MACRO(clk, resetn, 1, ctrl_reg_go_bsy, ralCfg, ip_rb.
       ctrl_reg.go_bsy)
    //...
    //Other registers
endinterface // ral_monitor_if
//End: Python script generated code
```

RAL Macro Utilization

Pointer to RAL Monitor Utility Object

IP Reg Model Instance

RAL Volatile Utility Object

Key Features:

- Acts as a communication bridge between the hardware monitoring interface and the UVM class domain.
- Maintains handles to the RAL virtual interface, IP register block, and RAL configuration object for seamless integration.
- **set_bfm()** method links the utility object to the virtual interface and relevant register components in the testbench.
- Implements core methods for volatile field management:
  - **notify_ral()**: Updates the register model using **UVM_PREDICT_DIRECT (backdoor)**, then triggers the corresponding UVM event with updated signal data.
  - **notify_reset_ev():** Manages event lifecycle by resetting active events as needed.
- **Note:** Updates register model based on observed changes; correctness of predicted values should be verified separately.

```
class ral_monitor_util extends uvm_object;
   `uvm_object_utils(ral_monitor_util)
    virtual ral_monitor_if              mon_ral_intf; // Virtual Interface
    ip_reg_block                        ip_rb; // IP register block
    ral_config                          ralCfg; //Script generated RAL Configuration
    function void set_bfm(virtual ral_monitor_if bfm, ip_reg_block ip_rb, ral_config ralCfg);
        mon_ral_intf                = bfm;
        mon_ral_intf.proxy_pointer = this;
        mon_ral_intf.ralCfg         = ralCfg;
        mon_ral_intf.ip_rb          = ip_rb;
        this.ip_rb                  = ip_rb;
        this.ralCfg                 = ralCfg;
    endfunction: set_bfm

    // Proxy Methods: Add here other useful methods
    function void notify_ral(longint input_signal, uvm_reg_field ral_field, uvm_event
        signal_change_event);
        ral_obj              ralObj; // Instantiate ral_obj and create object
        ralObj             = new("ralObj");
        ralObj.data        = input_signal;

        ral_field.predict(.value(input_signal), .be(-1), .kind(UVM_PREDICT_DIRECT), .path(
            UVM_BACKDOOR), .map(null), .fname(""), .lineno(0));
        signal_change_event.trigger(ralObj);
    endfunction
    function void notify_reset_ev(uvm_event event_to_reset);
        if (event_to_reset.is_on())
            event_to_reset.reset();
    endfunction
    // Add other RAL required methods here
endclass
```

Proxy Container Class for RAL Utilities

All Core Methods are Defined Here

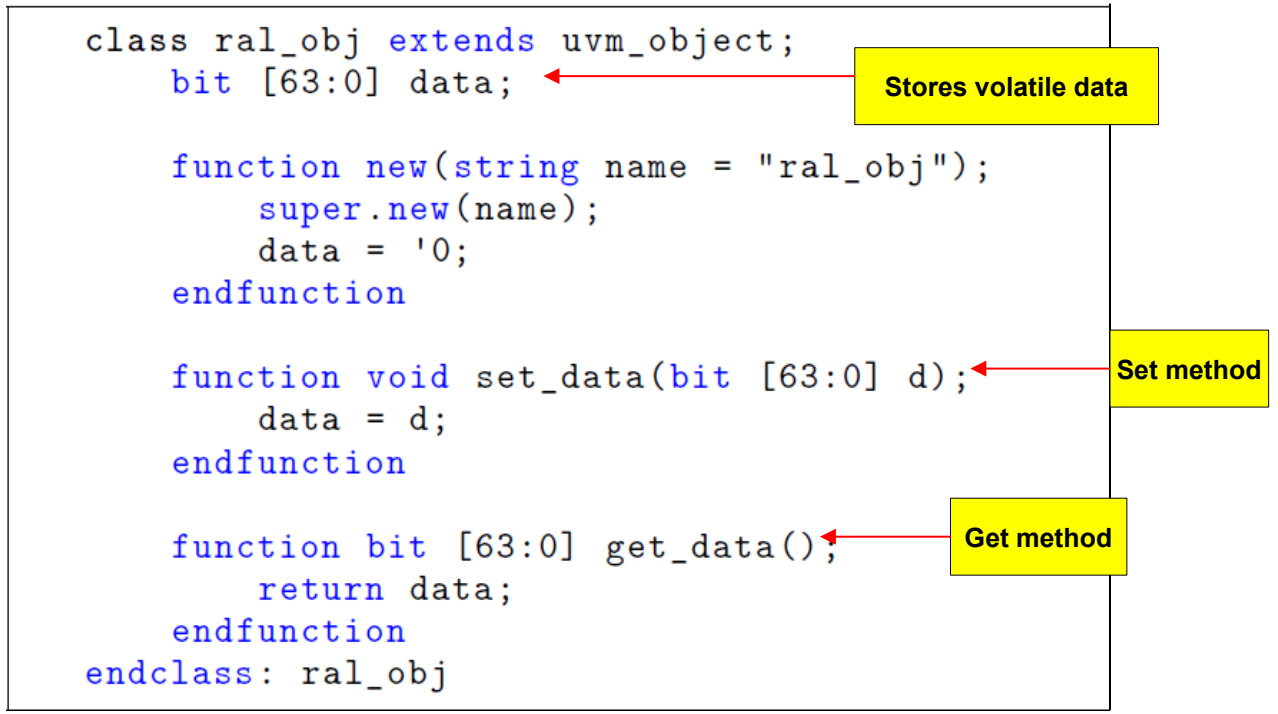
RAL Volatile Configuration Object

- Auto-generated to manage UVM events for all volatile fields.
- Ensures every volatile field has a corresponding event for synchronization.
- Uses the **ral_obj** class to encapsulate and pass updated volatile field data to verification components.

```
//Start: Python Script Generated
class ral_config extends uvm_object;
    uvm_event ctrl_reg_go_bsy_change;
    ...
    function new(string name = "ral_config");
        ctrl_reg_go_bsy_change = new("ctrl_reg_go_bsy_change");
        ...
    endfunction: new
endclass: ral_config
//End: Python Script Generated
```

RAL Data object

- **ral_obj** is a dedicated data container class used in the infrastructure.
- When a volatile register field changes, a new ral_obj instance is created and populated with the updated value.
- This object is passed as trigger data when notifying the corresponding **uvm_event**.
- Enables verification components and sequences to:
  - Detect when a volatile field change occurs.
  - Immediately access the updated value without requiring additional register reads.



Testbench Integration

- Instantiate the RAL monitor interface in the testbench.
- Create and assign a virtual interface handle.
- Set the virtual interface into the UVM configuration database.
- Assign volatile DUT RTL signals to the RAL monitoring interface for real-time observation.

```
module tb_top();

    ral_monitor_if ral_intf_inst(.clk(system_clk), .resetn(system_rst_b));
    virtual ral_monitor_if ral_mon_vif = ral_intf_inst;

    // DUT instantiation
     ip_core ip_dut (
        .clk(system_clk), .rst_n(system_rst_b),
    //  .... other dut inputs/outputs
    //  ....
      );

    initial begin
        uvm_config_db#(virtual interface ral_monitor_if)::set(null, "*", "vif", ral_mon_vif);
    end

    // Start : Python script generated : Assign Volatile field signals to monitor interface
    // All connections are generated before compile time and included here.
    // `include "ral_tb_connection.sv"
    // For illustration purposes showing directly in the testbench.
    assign ral_intf_inst.ctrl_reg_go_bsy = ip_dut.dut_rtl.registers.ctrl_reg.go_bsy;
    //...
    //...
    // End : Python script generated: Assign Volatile field signals to monitor interface
endmodule
```

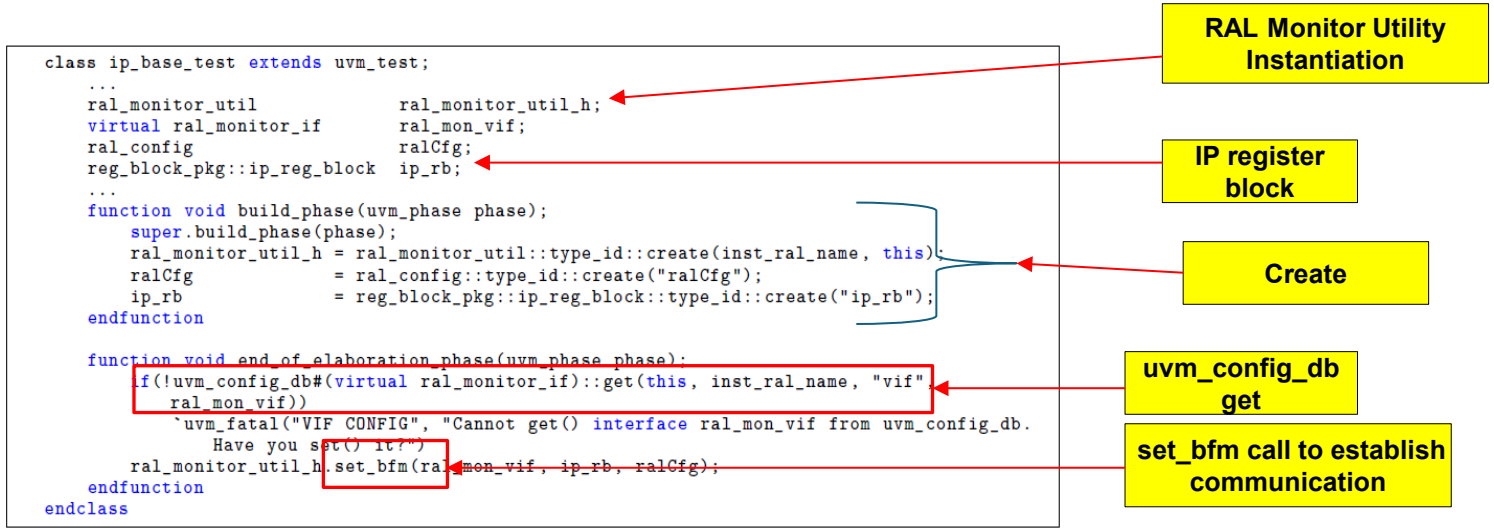
RAL Virtual Intf

Set intf in uvm_cfg_db

Cross-module reference

UVM Test Setup

- Instantiates and connects all RAL monitoring infrastructure components:
  - RAL utility object
  - RAL configuration object
  - Register block
- Retrieves the virtual interface handle from the UVM configuration database.
- Establishes communication between the utility object and the virtual interface using **set_bfm()**.
- Configures the utility object with handles for the configuration object and register block.



Sequence Usage

- Event-driven synchronization: sequences wait for UVM event notifications.
- Immediate access to updated values as trigger data.
- No need for manual polling loops.
- Simplifies sequence logic and improves responsiveness to hardware changes.

```
task body();
    cfg.wait_for_uvm_event_with_expected_data(
        .ev(cfg.ralCfg.ctrl_reg_go_bsy_change),
        .exp_data('h1),
        .max_count(7000),
        .data(event_data)
    );
    if (event_data != null)
        `uvm_info(get_name(), $sformatf("event_data: %s", event_data.convert2string()),
            UVM_LOW)
    else
        `uvm_info(get_name(), "event_data is null (timeout/error)", UVM_LOW)
endtask
```

Sequence using RAL event

Benefits of the Proposed Infrastructure

- Enhances verification accuracy—**regmodel.field.get()** always returns the current hardware value.
- Correctness - **regmodel.register.update()** or **regmodel.field.update()** does not skip the bus access if we neglect to call register.mirror().
- Reduces debugging time and risk of verification errors due to stale register model.
- Eliminates manual polling, reducing code complexity and verification effort.
- Centralized, automated infrastructure simplifies testbench integration and maintenance.

Results, Performance, and Deployment

Performance

- 20% more simulation cycles per CPU second.
- 60-70% reduction in verification code lines for volatile field handling.
- 15% reduction in memory usage.

Deployment

- Used in Intel Power Management IP verification.
- Managed 2500+ volatile fields across power state machines, interrupt controllers, configuration registers.
- 30% reduction in sequence development time, less debugging.

Qualitative Benefits

- Enhanced synchronization, maintainability, and scalability.
- Correct-by-construction: changes in XML propagate automatically.

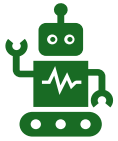
Limitations, Future Work, and Conclusion

Current Limitations:

Optimized for IP-level verification; SoC-level requires additional optimization.

Custom Python tooling dependency for infrastructure generation.

Limited to volatile fields with accessible RTL signals.



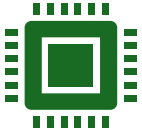
Future Enhancements:

SoC-Level Optimization: Resource management for large-scale deployment.

ML Integration: Predictive modeling for volatile field behavior and anomaly detection.

Standardization: Industry-wide UVM package development.

Advanced Verification: Explore use of UVM_CHECK in mirror() calls for volatile registers.



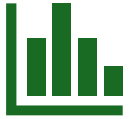
Technical Contributions:

Automated infrastructure for volatile register synchronization via hardware signal monitoring

Event-driven notification system eliminating manual polling in verification sequences

Python-based "correct by construction" code generation framework

Production-validated scalability in complex IP verification environments



Industry Impact:

Paradigm shift from reactive polling to proactive event-driven synchronization in UVM RAL methodologies with demonstrated production effectiveness.

Thank You

Backup Slides

Appendix: Sequence Implementation Examples

- Provides practical examples of sequences using RAL volatile events for register monitoring and synchronization.
- Illustrates how to wait for and respond to hardware-driven volatile field changes within UVM sequences.
- Designed as templates—users should adapt and extend these utilities to fit their specific verification environment and requirements.

```
task automatic env_config::wait_for_uvm_event_with_timeout(uvm_event ev, int max_count = 100,    output ral_obj data);
    int wait_counter = 0;    bit triggered = 0;    uvm_object tmp_data;
    int effective_max_count = (ral_max_timeout_count != 0) ?
                              ral_max_timeout_count : max_count;
    ral_obj local_data = new();
    fork
        begin
            ev.wait_ptrigger_data(tmp_data);
            if (!$cast(local_data, tmp_data)) begin
                `uvm_error(get_full_name(), "Failed to cast tmp_data to ral_obj")
                local_data = null;
            end
            triggered = 1;
        end
        begin
            while (!triggered && wait_counter < effective_max_count) begin
                @(posedge clk_agt_h.vif.clk);
                wait_counter++;
            end
        end
    join_any
    disable fork;

    if (!triggered) begin
        `uvm_error(get_full_name(),
                   $sformatf("Timeout: uvm_event '%s' not triggered after %0d cycles",
                             ev.get_name(), wait_counter))
        local_data = null;
    end
    data = local_data;
endtask : wait_for_uvm_event_with_timeout
```

Continuation…

```
task automatic env_config::wait_for_uvm_event_with_expected_data(
    uvm_event ev,
    bit [63:0] exp_data,
    int max_count = 100,
    output ral_obj data
);
    ral_obj expected = new();
    ral_obj local_data = new();
    expected.set_data(exp_data);

    wait_for_uvm_event_with_timeout(ev, max_count, local_data);

    if (local_data == null) begin
        `uvm_error(get_full_name(),
                   $sformatf("Event '%s' did not trigger or timed out",
                             ev.get_name()))
        data = null;
        return;
    end

    if (!local_data.compare(expected)) begin
        `uvm_error(get_full_name(),
                   $sformatf("Event '%s' data mismatch", ev.get_name()))
    end

    data = local_data;
endtask : wait_for_uvm_event_with_expected_data
```

2026

DESIGN AND VERIFICATION™

DVCON

CONFERENCE AND EXHIBITION

UNITED STATES

SANTA CLARA, CA, USA

MARCH 2 - 5, 2026