2025 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

MUNICH, GERMANY
OCTOBER 14-15, 2025

GAP
A Generic Agent Pattern for Reusable Testbenches

Omar Younis, Peter Gad, Ahmed Adel

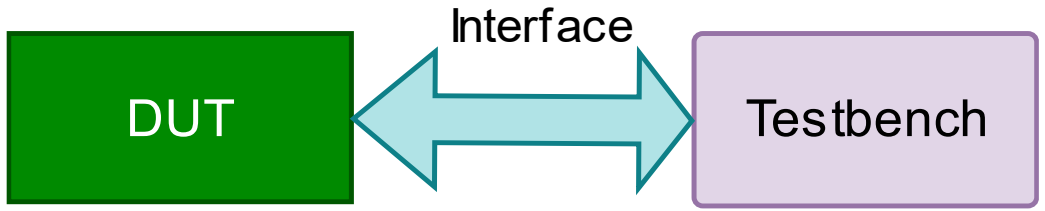
Si-VISION INNOVATIVE IC SOLUTIONS

Si-Vision LLC

accellera SYSTEMS INITIATIVE

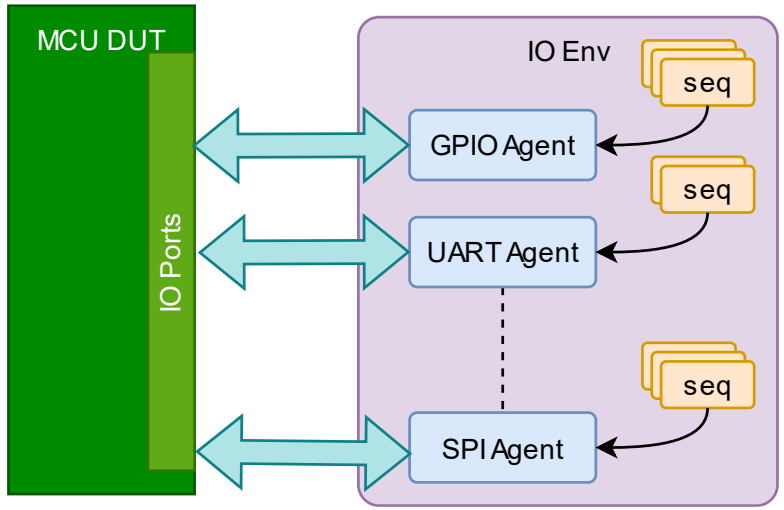
Introduction

- The traditional approach TB to DUT connection involves virtual interfaces, which allow class-based components to control signal-level interfaces.
- Creates structural dependencies.
- Requires careful management of interface instances and parameterization.
- Virtual interfaces are not classes



Introduction

- How to deal with a multi-protocol interface?
  - Build an agent per protocol. -> Complicates the testbench structure.
  - Create a super agent that supports
    all protocols through conditional code. -> Increases the maintenance overhead.



Abstract Bus Functional Models (BFMs)

```
interface class generic_driver_bfm;
  pure virtual task drive_transaction (uvm_sequence_item req);
  // TODO: Add additional helper methods as needed
endclass
interface class generic_monitor_bfm;
  pure virtual task monitor_transaction (output uvm_sequence_item req);
  // TODO: Add additional helper methods as needed
endclass
```

Abstract Bus Functional Models (BFMs)

- **Form the foundation of the new agent model**
- **Protocol behavior is implemented externally in concrete BFMs and injected into a generic agent**

```
interface class generic_driver_bfm;
  pure virtual task drive_transaction (uvm_sequence_item req);
  // TODO: Add additional helper methods as needed
endclass
interface class generic_monitor_bfm;
  pure virtual task monitor_transaction (output uvm_sequence_item req);
  // TODO: Add additional helper methods as needed
endclass
```

Abstract Bus Functional Models (BFMs)

- **Defines a behavioral contract through abstract methods**
- **Enables polymorphism and decouples implementation from specification.**
- **Cleaner implementation of software design patterns.**

```
interface class generic_driver_bfm;
  pure virtual task drive_transactio
  // TODO: Add additional helper methods as needed
endclass
interface class generic_monitor_bfm;
  pure virtual task monitor_transaction (output uvm_sequence_item req);
  // TODO: Add additional helper methods as needed
endclass
```

Abstract Bus Functional Models (BFMs)

- **If the simulator / TB cannot support** interface class, **we can use a** virtual class.

```
interface class generic_driver_bfm;
  pure virtual task drive_transaction (uvm_sequence_item req);
  // TODO: Add additional helper methods as needed
endclass
interface class generic_monitor_bfm;
  pure virtual task monitor_transaction (output uvm_sequence_item req);
  // TODO: Add additional helper methods as needed
endclass
```

Abstract Bus Functional Models (BFMs)

```
interface class generic_driver_bfm;
  pure virtual task drive_transaction (uvm_sequence_item req);
  // TODO: Add additional helper methods as needed
endclass
interface class generic_monitor_bfm;
  pure virtual task monitor_transaction (output uvm_sequence_item req);
  // TODO: Add additional helper methods as needed
endclass
```

Drives the interface signals based on the input

Monitors the interface signals

Creating Concretions

```
interface apb_if(input clk); // signal definitions ...
  class apb_driver_adapter implements generic_driver_bfm;
    virtual task drive_transaction(uvm_sequence_item item);
    // ...
    endtask
  endclass
  class apb_monitor_adapter implements generic_monitor_bfm;
    //...
  endclass
  // Instantiate the driver and monitor adapters
  apb_driver_adapter driver_bfm = new();
  apb_monitor_adapter monitor_bfm = new();
endinterface
```

Creating Concretions

- **Encapsulate all protocol-specific signal activity in interface class implementations.**
- **Native support for parameterized interfaces**

```
interface apb_if(input clk); // signal
  class apb_driver_adapter implements generic_driver_bfm;
    virtual task drive_transaction(uvm_sequence_item item);
    // ...
    endtask
  endclass
  class apb_monitor_adapter implements generic_monitor_bfm;
    //...
  endclass
  // Instantiate the driver and monitor adapters
  apb_driver_adapter driver_bfm = new();
  apb_monitor_adapter monitor_bfm = new();
endinterface
```

Abstracting the Driver and Monitor

```
class generic_driver extends uvm_driver #(uvm_sequence_item);

  generic_driver_bfm driver_bfm;

  virtual task run_phase (uvm_phase phase);
    uvm_sequence_item req;
    forever begin
      seq_item_port.get_next_item(req);

      driver_bfm.drive_transaction(req);

      seq_item_port.item_done();
// ...
```

Abstracting the Driver and Monitor

```
class generic_driver extends uvm_driver #(uvm_sequence_item);

  generic_driver_bfm driver_bfm;

  virtual task run_phase (uvm_phase phase);
    uvm_sequence_item req;
    forever begin
      seq_item_port.get_next_item(req);

      driver_bfm.drive_transaction(req);

      seq_item_port.item_done();
// ...
```

The interface class BFM handle

Delegates all signal-level transaction behavior to BFM implementation

Supporting Generic Subscribers

```
interface class generic_subscriber_bfm;
  pure virtual function void write_transaction(uvm_sequence_item req);
  // ...
endclass
interface apb_if(input clk); // ...
  class apb_checker_adapter implements generic_subscriber_bfm;
    virtual function void write_transaction(uvm_sequence_item item);
    // ...
    endfunction
  endclass

  apb_checker_adapter  checker_bfm  = new();
endinterface
```

Supporting Generic Subscribers

```
interface class generic_subscriber_bfm;
  pure virtual function void write_transaction(uvm_sequence_item req);
  // ...
endclass
interface apb_if(input clk); // ...
  class apb_checker_adapter implements generic_subscriber_bfm;
    virtual function void write_transaction(uvm_sequence_item item);
    // ...
    endfunction
  endclass

  apb_checker_adapter  checker_bfm  = new();
endinterface
```

Implements the subscriber behavior; checkers, coverage collection, etc

Instance of the subscriber adapter BFM

Supporting Generic Subscribers

```
class generic_subscriber extends uvm_subscriber#(uvm_sequence_item);

  generic_subscriber_bfm bfm;

  virtual function void write(uvm_sequence_item t);
    bfm.write_transaction(t)
  endfunction
endclass : generic_subscriber
```

Subscriber BFM handle

Subscriber behavior is fully contained in the BFM

Supporting Generic Subscribers

```
class generic_monitor extends uvm_component;
  local generic_subscriber subscribers_h[$];

  function void add_subscriber(generic_subscriber subscriber_h);
    subscribers_h.push_back(subscriber_h);
  endfunction

  virtual task run_phase(uvm_phase phase);
    forever begin
    // ...
      foreach(subscribers_h[i])
        subscribers_h[i].write_transaction(req);
// ...
```

Subscriber queue

Subscriber connection method

Sending transactions to subscribers

Generic Agent Build Phase

```
virtual function void generic_agent::build_phase(uvm_phase phase);
  if(!uvm_config_db#(CFG)::get(this, "", {get_name(),"_cfg"}, cfg_h))
    `uvm_fatal("NO_CFG", $sformatf("Failed to fetch %0s from CGFDB", {get_name(),"_cfg"}))

  if (cfg_h.is_active == UVM_ACTIVE)
    driver_h = generic_driver::type_id::create({get_name(),"_drv"}, this);

  if (cfg_h.sub_en != 0) begin
    string mode_s;
    subscriber_h = new[cfg_h.sub_en];
    foreach (subscriber_h[i])
      subscriber_h[i] = generic_subscriber::type_id::create({get_name(), "_sub"}, this);
  end
  monitor_h = generic_monitor::type_id::create({get_name(),"_mon"}, this);
```

Generic Agent Build Phase

```
virtual function void generic_agent::build_phase(uvm_phase phase);
  if(!uvm_config_db#(CFG)::get(this, "", {get_name(),"_cfg"}, cfg_h))
    `uvm_fatal("NO_CFG", $sformatf("Failed to fetch %0s from CGFDB", {get_name(),"_cfg"}))

  if (cfg_h.is_active == UVM_ACTIVE)
    driver_h = generic_driver::type_id::create({get_name(),"_drv"}, this);

  if (cfg_h.sub_en != 0) begin
    string mode_s;
    subscriber_h = new[cfg_h.sub_en];
    foreach (subscriber_h[i])
      subscriber_h[i] = generic_subscriber::type_id::create({get_name(), "_sub"}, this);
  end
  monitor_h = generic_monitor::type_id::create({get_name(),"_mon"}, this);
```

Build subscribers based on configuration settings

Generic Agent Build Phase

```
virtual function void generic_agent::build_phase(uvm_phase phase);
  if(!uvm_config_db#(CFG)::get(this, "", {get_name(),"_cfg"}, cfg_h))
    `uvm_fatal("NO_CFG", $sformatf("Failed to fetch %0s from CGFDB", {get_name(),"_cfg"}))

  if (cfg_h.is_active == UVM_ACTIVE)
    driver_h = generic_driver::type_id::create({get_name(),"_drv"}, this);

  if (cfg_h.sub_en != 0) begin
    string mode_s;
    subscriber_h = new[cfg_h.sub_en];
    foreach (subscriber_h[i])
      subscriber_h[i] = generic_subscriber::type_id::create({get_name(), "_sub"}, this);
  end
  monitor_h = generic_monitor::type_id::create({get_name(),"_mon"}, this);
```

Assume that all components have the name "parent"+"_type"

Generic Agent Connect Phase

```
virtual function void generic_agent::connect_phase(uvm_phase phase);
  if (cfg_h.is_active == UVM_ACTIVE)
    driver_h.driver_bfm = cfg_h.driver_bfm;

  monitor_h.monitor_bfm = cfg_h.monitor_bfm;

  if (subscriber_h.size() != 0) begin
    foreach (subscriber_h[i]) begin
      subscriber_h[i].bfm = cfg_h.bfm[i];

      this.connect_subs(subscriber_h[i]);
    end
  end
endfunction: connect_phase
```

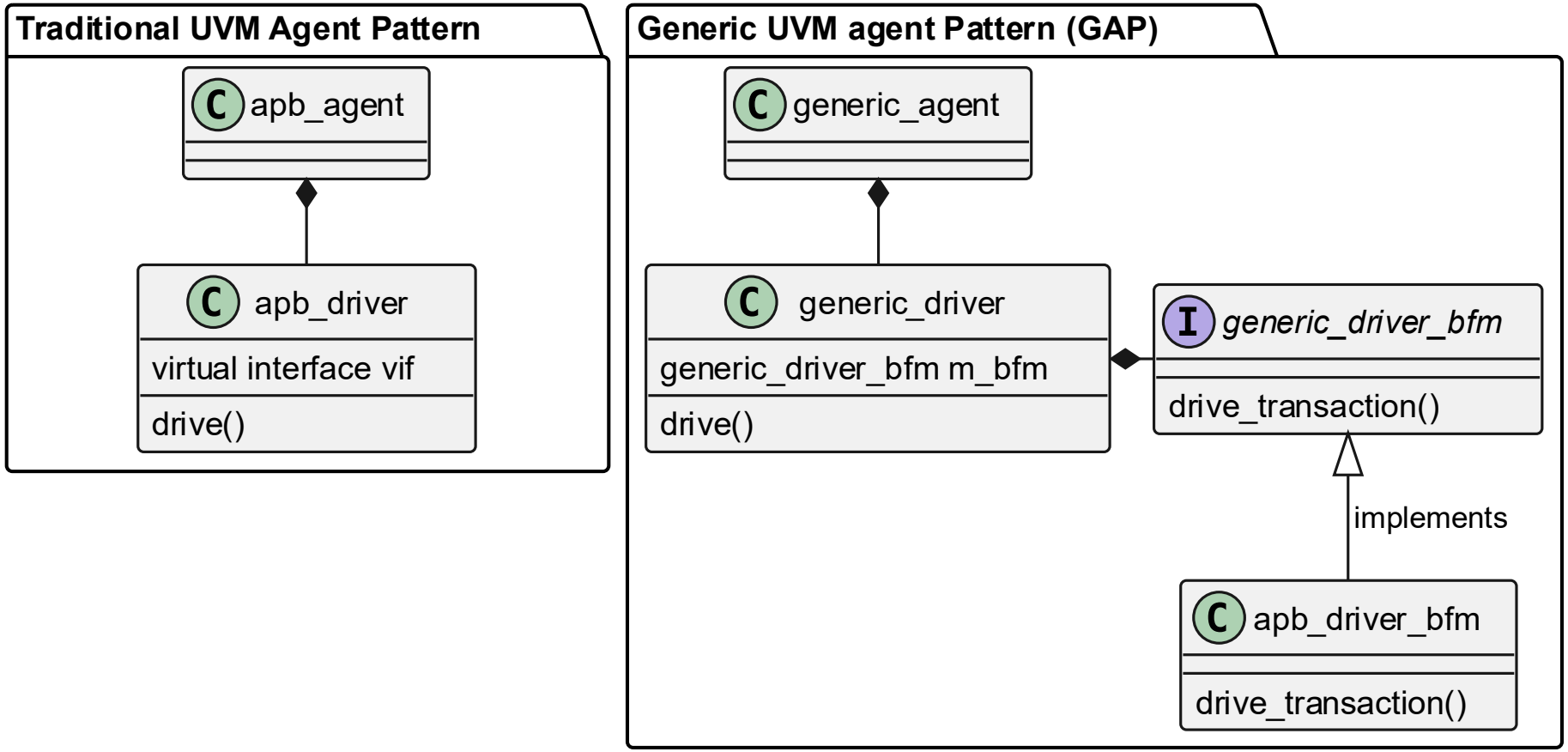
Generic Agent Connect Phase

```
virtual function void generic_agent::connect_phase(uvm_phase phase);
  if (cfg_h.is_active == UVM_ACTIVE)
    driver_h.driver_bfm = cfg_h.driver_bfm;

  monitor_h.monitor_bfm = cfg_h.monitor_bfm;

  if (subscriber_h.size() != 0) begin
    foreach (subscriber_h[i]) begin
      subscriber_h[i].bfm = cfg_h.bfm[i];

      this.connect_subs(subscriber_h[i]);
    end
  end
endfunction: connect_phase
```

Connect BFMs to the driver and monitor

Connect subscribers to producers

Generic Agent UML



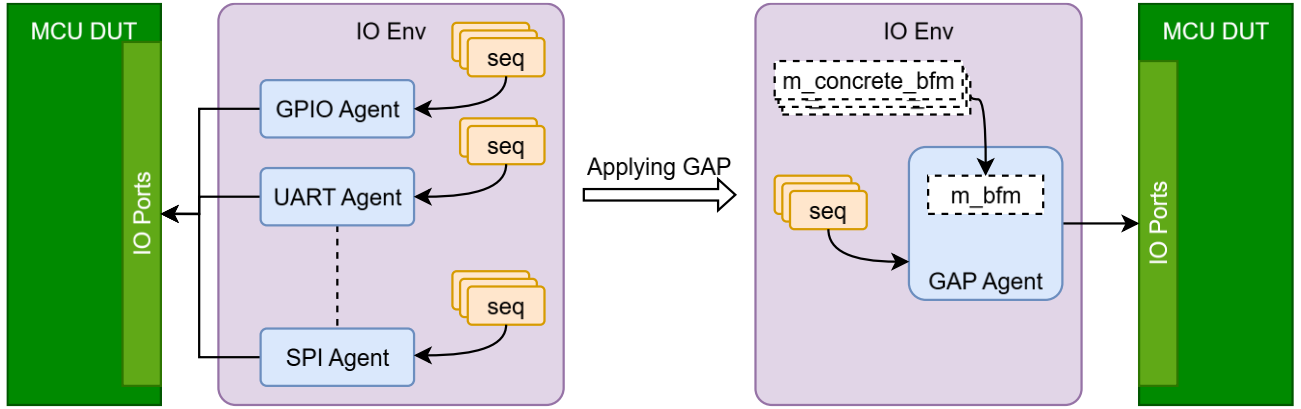
Use cases (1)

- A DUT may need to switch between different protocol versions / driving logic depending on its configuration.
- In traditional approaches:
  - A single agent containing large conditional branches to manage the variations.
  - Complex code and difficult to maintain.

- GAP:
  - Allows different BFM classes to be plugged into the same agent dynamically
  - The driving logic is simpler, readable, and maintainable.

Use cases (2)

- Pin multiplexing, where the same physical pins are shared across multiple protocols
- In traditional approaches:
  - Create separate agents for each supported protocol
  - Increases overhead and code duplication.



Traditional Agents VS GAP (1)

Feature	Traditional Agents	GAP
Dynamic behavior injection	Limited	Fully dynamic vis BFM classes
Code reuse across protocols	One agent per protocol	One agent, multiple BFMs
TB-DUT decoupling	Partial (via virtual interfaces)	Full (class-in-interface binding)

Traditional Agents VS GAP (2)

Feature	Traditional Agents	GAP
Follows OOP Principles	Virtual interfaces can limit traditional agents	Fully compliant with OOP principles
Use in Single-protocol and/or low complexity environments	Traditional agents offer faster bring-up time in cases where GAP needs to be developed from scratch	Truly shines in environments where multiple related protocols are in use. For single protocol environments, the case for using GAP is not strong
Statically configurable DUTs	Factory overrides can be superior to GAP.	Can be used but does not offer any advantage as the DUT itself is statically configurable

Conclusion (1)

- GAP provides a clean, modular, and flexible method for connecting UVM testbenches to DUTs without relying on virtual interfaces
- GAP enables object-oriented, strategy-driven verification components that are easy to build, extend, and maintain.
- GAP simplifies agent development, enhances reusability, and aligns closely with modern software design best practices while remaining fully UVM-compliant.

Conclusion (2)

- GAP is not intended as a replacement for the traditional UVM agent approach.
- In many cases, traditional agents remain simpler and perfectly suitable, and the choice between approaches should be guided by the project's specific requirements and complexity.

Conclusion (3)

- Disadvantages:
  - The indirection introduced through interface classes can make debugging more difficult without proper tooling or comprehensive logging support.
  - For simple or one-off protocols, the extra abstraction might be unnecessary and even counterproductive.
  - Using a single, highly generic agent to support many diverse use cases—such as pipelining, reactivity, or incorporating specialized behaviors needed only in a few blocks—can become counterproductive and lead to significant maintenance challenges.
- Define separate generic agents for each major use case or protocol family.

Thank You!