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, Si-Vision, Cairo, Egypt (*omar.younis@si-vision.com*)

Peter Gad, Si-Vision, Cairo, Egypt (*peter.gad@si-vision.com*)

***Abstract*—*As* UVM-based verification environments grow in complexity and flexibility, the connection between the testbench and the DUT remains a critical concern—particularly in projects where the DUT may support multiple protocols or protocol versions. In most verification projects we can find an implementation of a generic or common agent. However, these classes will still be required to be extended to add protocol/interface specific functionality.**

This paper introduces GAP (Generic Agent Pattern). GAP provides a reference base implementation of a generic agent that requires no extension. GAP builds on previous literature on connecting the testbench to the DUT that avoids using virtual interfaces. Instead of requiring each agent to be tailored to specific protocols or interface variants, GAP enables a single, reusable agent to support multiple use cases through object-oriented encapsulation and dynamic behavior injection.

***Keywords*—generic agent, BFM, UVM**

I. Introduction

The traditional approach to connecting UVM testbenches to DUTs involves virtual interfaces, which allow class-based drivers and monitors to control signal-level interfaces. While effective, this model creates strong structural dependencies and requires careful management of interface instances and parameterization. These issues are particularly pronounced in designs where the DUT may be configured to operate with different protocols (e.g., *APB* vs. *AHB*) or protocol versions (e.g., *AXI3* vs. *AXI5*).

In the paper "Abstract BFMs Outshine Virtual Interfaces for Advanced SystemVerilog Testbenches" [1], an alternative approach is proposed that uses abstract classes to define protocol APIs. This approach decouples the testbench from the DUT interface allowing for cleaner layering. However, the driver and monitor classes still contain significant driving and monitoring logic, which involves calling multiple APIs implemented in the abstract BFM.

Another complementary solution is presented in [2], which addressed the integration of parameterized interfaces to reusable UVM environments. Their approach employs virtual accessor classes that abstract away parameterization, allowing class-based access to DUT signals without modifying the UVM components.

Building on the same goals of abstraction and modularity, the GAP methodology takes this further by embedding protocol behavior directly into one or more classes defined within the interface. These classes are then passed to the agent, allowing its sub-components to handle all driving and monitoring activities. As a result, the UVM driver and monitor are effectively reduced to minimal wrappers.

This design enables dynamic behavior injection, supports full runtime flexibility, and offers a unified testbench architecture suitable for multi-protocol and highly configurable designs.

II. Methodology

GAP utilizes abstract BFMs, these will be responsible for implementing all protocol logic, including *`drive()`* and *`monitor()`* tasks, as well as optional pre and post hooks and helper methods.

The UVM driver and monitor classes simply call *`bfm_h.drive()`* and *`bfm_h.monitor()`* respectively in their run phase, delegating all transaction logic to the interface. During the build and configuration phases, the testbench creates an instance of the appropriate BFM class based on the target protocol or DUT configuration and passes it to the interface.

A. Abstract Bus Functional Model Declaration

These will form the foundation of the new agent model, where protocol behavior is implemented externally and injected into a generic agent structure.

```
interface class generic_driver_bfm;
  // This task is responsible for driving the signals based on the request
  pure virtual task drive_transaction (uvm_sequence_item req);
  // Declare any common helper methods needed for the common base BFM
  // ...
endclass

interface class generic_monitor_bfm;
  // This task is responsible for monitoring the signals
  pure virtual task monitor_transaction (output uvm_sequence_item req);
  // Declare any common helper methods needed for the common base BFM
  //...
endclass
```

Figure 1 An interface class definition of the abstract BFM classes.

SystemVerilog's `interface class` construct introduces the concept of pure abstraction at the behavioral level. The interface class defines a behavioral contract through abstract methods. This enables polymorphism and decouples implementation from specification [3].

Through the `interface class`, we can support multiple inheritance and pure behavior-based interfaces, allowing SystemVerilog to model software design patterns more cleanly. If working with older simulators or with test benches where supporting the interface class is not possible, we can use a `virtual class`.

B. Abstracting the Driver and Monitor

We implement the generic `driver` and `monitor` classes to rely solely on these abstract interface classes. The driver now delegates all low-level transaction behavior to an injected `generic_driver_bfm` implementation, and likewise for the monitor.

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

  virtual task run_phase (uvm_phase phase);
    forever begin
      seq_item_port.get_next_item(req);
      driver_bfm.drive_transaction(req);
      seq_item_port.item_done();
    end
  endtask
endclass
```

Figure 2 An example of the generic driver implementation.

```
class generic_monitor extends uvm_monitor #(uvm_sequence_item);
  generic_monitor_bfm monitor_bfm;
  uvm_analysis_port #(uvm_sequence_item) monitor_ap;

  virtual task run_phase(uvm_phase phase);
    forever begin
      monitor_bfm.monitor_transaction(my_item);
      monitor_ap.write(my_item);
    end
  endtask
endclass
```

Figure 3 An example of the generic monitor implementation.

It is important to note that while the example implementations provided in this paper extend directly from the base UVM classes, most verification environments already include their own customized base classes derived from UVM. The reader is encouraged to and can easily adapt GAP to use these existing base classes instead.

C. Creating the Concrete BFMs

We implement concrete classes (adapters) that conform to the interface class definitions. These classes encapsulate all protocol-specific signal activity, binding to virtual interface instances and performing the required transactions.

```
interface apb_if(input clk);
  // signal definitions ...

  class apb_driver_adapter implements generic_driver_bfm;
    // Implement the drive method
    virtual task drive_transaction(uvm_sequence_item item);
      // Driving logic goes here
    endtask
  endclass

  class apb_monitor_adapter implements generic_monitor_bfm;
    // Implement the monitor adapter
    //...
  endclass

  apb_driver_adapter  driver_bfm  = new();
  apb_monitor_adapter monitor_bfm = new();
endinterface
```

Figure 4 A concrete implementation of a protocol BFM.

This approach that embeds the protocol adapter directly within the signal interface, proposed in [1], allows for the encapsulation of both the protocol signals and their driving semantics in one location.

An important advantage of this encapsulation style is its native support for parameterized interfaces. In traditional UVM environments, the most straightforward approach typically requires parameterizing all classes that interact with the interface. Passing a parameterized virtual interface across multiple components often results in cumbersome, error-prone, and fragile code structures.

By embedding the adapter class directly within the interface, parameter handling becomes fully localized to the interface scope. This eliminates the need to propagate parameters manually across the testbench and ensures that the adapter automatically inherits any parameters declared in the enclosing interface. As a result, verification components become more robust, easier to maintain, and highly reusable, especially when dealing with variations such as data width, address size, or timing configurations.

It is worth noting that in a previous study [2], the authors reported issues when accessing interface signals from a class defined inside an interface, particularly when multiple interface instances were used. The authors of that study observed that signals were not driven properly at runtime despite successful compilation. However, based on our experiments with modern simulators, this approach works reliably. We believe the earlier observations were likely due to tool limitations or bugs in older simulator versions, which have since been addressed.

D. Supporting Generic Subscribers

To make the agent extensible for subscribers such as; scoreboards and functional coverage components, we introduce a generic subscriber mechanism. This follows the same abstraction principles, using an interface class, as shown in Figure 5 .

```
interface class generic_subscriber_bfm;
  // This will be responsible for the subscriber behavior.
  // checkers, coverage collection, etc.
  pure virtual function void write_transaction(uvm_sequence_item req);
  // Declare any common helper methods needed for the common base BFM
endclass

interface apb_if(input clk); // ...
  class apb_checker_adapter implements generic_subscriber_bfm;
    // Implement the checker method
    virtual function void write_transaction(uvm_sequence_item item);
      // Checking logic goes here
    endfunction
  endclass

  apb_checker_adapter  checker_bfm   = new();
endinterface

class generic_subscriber extends uvm_subscriber#(uvm_sequence_item);
  apb_checker_bfm bfm; //Bus functional model

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

Figure 5 A concrete implementation of the generic subscriber.

This implementation relies on the TLM analysis ports to connect the subscribers to the monitors. This connection is done in the agent. Another implementation, presented in [4], relies mostly on the interface class implementation. We will explore this implementation next and it is up to the reader to decide which approach to choose.

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

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

  virtual task run_phase(uvm_phase phase);
    //...
    forever begin
      monitor_bfm.monitor_transaction(req);
      foreach(subscribers_h[i])
        subscribers_h[i].write_transaction(req);
      end
  endtask
endclass

class apb_checker_adapter extends uvm_component implements
generic_subscriber_bfm;
  virtual function void write_transaction(uvm_sequence_item item);
    //...
  endfunction
endclass
```

Figure 6 An alternative concrete implementation of the generic subscriber using interface class.

E. *Putting all together (the Generic Agent)*

The resulting system is modular, maintainable, and protocol-agnostic. Adding a new protocol requires only the definition of a signal interface, and adapter classes.

The following figure shows a UML class diagram of the resulting generic agent (right), alongside the traditional protocol-specific implementation of the agent (left).

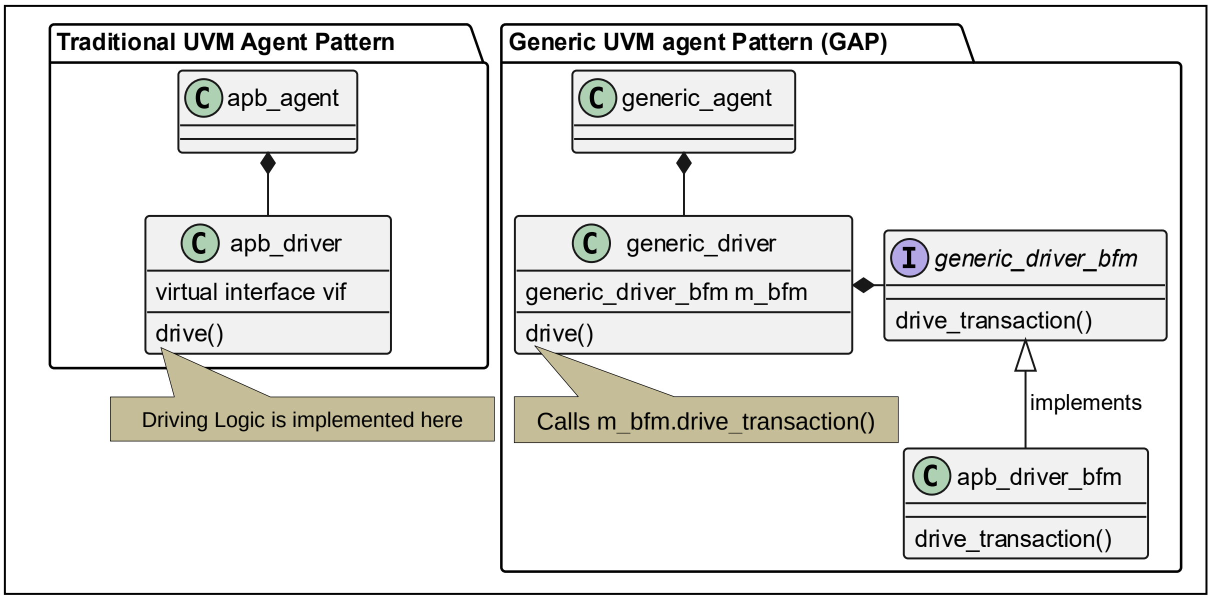

Figure 7 UML diagram of the generic agent pattern and the traditional agent.

It is important to note that since the BFMs here are now class based, they can be dynamically assigned to the UVM components. The BFMs can be assigned to the components at any point in the simulation. In this example, we chose to use the connect phase just because this is how it is traditionally done.

```
class generic_agent extends uvm_agent;
  generic_driver     driver_h;
  generic_monitor    monitor_h;
  generic_subscriber subscriber_h[];

  // The build_phase will construct all the agent components
  // This approach assumes that all components will be passed/constructed with
the name "parent"+"_type"
  virtual function void build_phase(uvm_phase phase);
    if(!uvm_config_db#(CFG)::get(this, "", {get_name(),"_cfg"}, cfg_h)) begin
      `uvm_fatal("NO_CFG", $sformatf("Failed to fetch %0s from CGFDB",
{get_name(),"_cfg"}))
    end

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

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

    monitor_h = generic_monitor::type_id::create({get_name(),"_mon"}, this);
    //...
  endfunction: build_phase

  // The connect phase will pass all the BFMs to the components
  virtual function void connect_phase(uvm_phase phase);
    if (cfg_h.is_active == UVM_ACTIVE) begin
      driver_h.driver_bfm = cfg_h.driver_bfm;
      //...
    end
    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];
        // Connect the subscribers to producers
        this.connect_subs(subscriber_h[i]);
        //...
      end
    end
  endfunction: connect_phase
endclass : generic_agent
```

Figure 8 Complete implementation of the generic agent.

III. Use Cases

Dynamically Configurable Protocols in the same DUT

A DUT may need to switch between different protocol versions depending on its configuration. In traditional approaches, this is often handled by a single agent containing large conditional branches to manage the variations, which can make the code complex and difficult to maintain. In contrast, GAP allows different BFM classes to be plugged into the same agent dynamically, the driving logic is simpler, readable, and maintainable.

Another common scenario is pin multiplexing, where the same physical pins are shared across multiple protocols depending on the DUT configuration. Traditionally, this requires creating separate agents for each supported protocol, increasing overhead and code duplication. With GAP, a single agent can be used, and only the BFM implementation needs to be swapped when the protocol changes. The following diagram illustrates this: on the right, the conventional approach enables or disables different agents as the DUT reconfigures its I/O registers; on the left, GAP uses a single agent and simply replaces the BFM whenever a reconfiguration occurs.

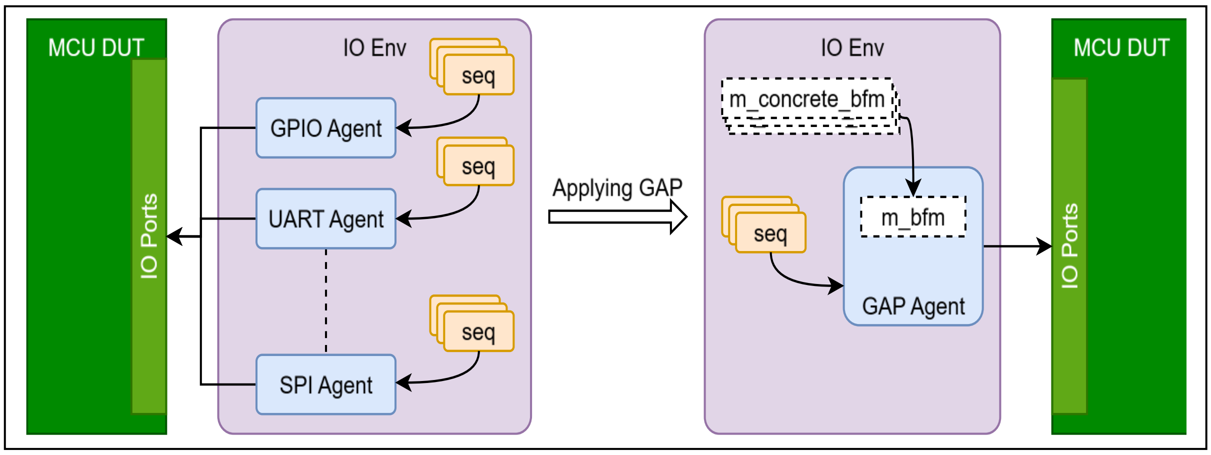

Figure 9 Simplified environment comparison — traditional agents versus GAP single-agent design for reconfigurable I/O.

IV. Result and Findings

A. *Advantages*

This approach offers a cleaner separation between structure and behavior, following principles similar to the Strategy Pattern. By breaking functionality into smaller, focused classes, the resulting architecture is easier to maintain and modify. The methodology also enables faster bring-up for small and mid-scale projects, as the modular design reduces integration complexity.

Additionally, it is easy to extend for new protocols or variants by simply adding new adapter implementations, without modifying the core agent logic. Finally, GAP supports both dynamic and static DUT configurability, allowing the same framework to be used across a wide range of designs and configuration scenarios.

B. *Disadvantages*

While GAP offers significant flexibility, it also introduces certain trade-offs. The indirection introduced through interface classes can make debugging more difficult without proper tooling or comprehensive logging support. To mitigate this, teams should invest in strong transaction-level logging, standardized debug hooks, and consistent simulation checks. These practices help improve traceability, enhance clarity, and make the overall system's behavior easier to understand during debugging and analysis.

For simple or one-off protocols, the extra abstraction might be unnecessary and even counterproductive, introducing indirection where a direct implementation would suffice. In these cases, using traditional UVM agents without additional layering is more practical and recommended to keep things simple.

Moreover, 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. A more sustainable solution is to define separate generic agents for each major use case or protocol family. This approach balances generalization with clarity, prevents overcomplicating shared agents, and enables easier maintenance and scalability as the design evolves.

C. A comparison between traditional agents and GAP

Table I. A comparison between Traditional Agents and GAP

Feature	***Traditional Agents***	***GAP***
Supports dynamic behavior injection	Limited	Fully dynamic via BFM class
Code reuse across protocols	One agent per protocol	One agent, multiple BFMs
Testbench-DUT decoupling	Partial (via virtual interfaces)	Full (class-in-in-interface binding)
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	GAP 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.	While GAP can be used here, it does not offer any advantage as the DUT itself is statically configurable.

V. CONCLUSION & FUTURE WORK

GAP provides a clean, modular, and flexible method for connecting UVM testbenches to DUTs without relying on virtual interfaces. By embedding class-based behavioral models inside interfaces, GAP enables object-oriented, strategy-driven verification components that are easy to build, extend, and maintain. This approach is especially well-suited for designs that support multiple interface protocols or runtime configurations. GAP simplifies agent development, enhances reusability, and aligns closely with modern software design best practices while remaining fully UVM-compliant.

It is important to emphasize that GAP is not intended as a replacement for the traditional UVM agent approach. Instead, GAP offers an alternative methodology that can provide significant advantages in scenarios where greater flexibility, reusability, or configurability is required. 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.

Future work may explore the use of automation tools for generating BFM templates and extending GAP to cover more expansive agent use case scenarios.

REFERENCES

[1] D. Rich and J. Bromley, "Abstract BFMs Outshine Virtual Interfaces for Advanced SystemVerilog Testbenches," in *Design & Verification Conference*, San Jose, CA, 2008.

[2] W. Yun and S. Zhang, "Deploying Parameterized Interface with UVM," in *Design & Verification Conference*, San Jose, CA, 2013.

[3] IEEE, "IEEE Standard for SystemVerilog - Unified Hardware Design, Specification, and Verification Language," IEEE Std 1800-2023, 2023.

[4] S. Sokorac, "SystemVerilog Interface Classes – More Useful Than You Thought," in *Design & Verification Conference*, San Jose, CA, 2016.