Proper probing: Flexibility on the TLM level

Gergő Vékony
AImotive, Hungary
gergo.vekony@aimotive.com

***Abstract* – Complex modules sometimes call for white box test approaches, as there may be some internal variables and signals that cannot be reached from the module's outermost surface. One such test method is probing where a testbench related component is bound into the Device Under Test's (DUT) structure to provide information about its internal activities. There are several probe methodologies employing the Universal Verification Methodology (UVM) that provide different kinds of flexibility and configuration options.**

This paper focuses on emphasizing each methodology's practical strengths and weaknesses and presents a suggested solution, that was successfully deployed in an automotive field related verification project where error injection, error propagation observation and probing played a crucial role. As the probed values are propagated via Transaction-level Modelling (TLM) transactions to other UVM components, code maintainability, reusability and probe–monitor decoupling were strongly considered factors in choosing the finally implemented features.

Keywords – SystemVerilog; UVM; testbench architecture; bind; monitor; probes

I. METHODOLOGIES

There are two essential reusable techniques for implementing probes using UVM: the *interface binding* based methodology and the *polymorphic class* based methodology. In this section a short introduction is given for both. While direct hierarchical access is a valid and possible methodology, its reusability is questionable, thus it is omitted from the further discussions.

A. Interface binding

SystemVerilog provides a bind directive that can be used to specify an instantiation of a module, interface, program or checker without modifying the code of the target. This allows encapsulating further functions, instrumentation code and assertions to the target in a non-intrusive manner. Bound structures become part of the target object and can be normally referenced using the standard hierarchical naming conventions. [1, 23.11]

From a verification perspective, the possible candidates of a bind directive are the module and the interface, as not every EDA vendor supports the checker and the program construct seems to be obsolete in an UVM environment. Modules and interfaces both can contain concurrent and immediate assertions, class instances, tasks and functions; but with the virtual interface mechanism, the interface construct provides more flexibility toward the testbench components connection.

```
typedef logic[63:0] addr_t;

interface probe_if(addr_t contact);
endinterface

module dut(...);
  addr_t addr;
endmodule

module top;
  dut dut_inst(...);
  bind dut_inst probe_if probe_if_inst (.contact(addr));
  initial uvm_config_db#(virtual probe_if)::set(..., dut_inst.probe_if_inst);
endmodule
```

Figure 1 – The instruments of an interface bind

As shown in Figure 1, an interface bind requires three mandatory components:

- A bound interface (*probe_if*) with an appropriate port list, where the ports will serve as connection points toward the host;
- The target structure (*dut*) that contains the probed signals or ports;
- A compilation-unit scope (*top* module in this case) that hosts the binding.

It is always a good idea to include the UVM configuration database write of the interface (if needed) at the same spot as the bind is declared, since all hierarchical information is present there.

B. The polymorphic class

The key concepts of the polymorphic class methodology are as follows. SystemVerilog provides subtype polymorphism for its class constructs, thus allowing inheritance, dynamic method lookup and dynamic casting of objects. If a class is defined in a hierarchy construct, then it will access each port, parameter and variable of that structure. Furthermore, it becomes part of the namespace of the enclosing hierarchy construct; while it can be instantiated outside of that structure, it can *only* serve as a future superclass for descendants that are declared within a nested design element. [1, 3.13] [1, 23.4]

Figure 2 illustrates this case; if the class *holy_grail* could be used as a superclass for other namespaces, it would solve most technical related problems of binding permanently.

```
interface holy_grail_container(...);
  virtual class holy_grail extends uvm_object;
  endclass
endinterface

class tlm_probe extends holy_grail;
endclass
```

Figure 2 – An apparently perfect solution that does not work, due to not available superclass

To overcome this structural limitation of SystemVerilog, the following common technique is used: a superclass is specified in a widely available unit or package, a subclass is defined in the bound interface or module, and the UVM environment accesses an object of the subclass via a handle of the superclass. There are different well documented methods of accessing the subclass objects with SV and UVM constructs discussed in [2] [3] and [4]; thus while this paper will refer these methods, they are not explained to detail here, although an example is provided in Figure 3.

Certain applications further extend this methodology with the introduction of an abstract superclass or with interface classes, to enforce the implementation of certain subroutines. However, these extensions do not fundamentally change the application in all but one case, which will be discussed in II.B later.

```
virtual class probe_abstract;
  pure virtual function addr_t get_data();
endclass

interface poly_if(addr_t contact);
  class probe_specialized extends probe_abstract;
    function addr_t get_data();
      get_data=contact;
    endfunction
  endclass

  probe_specialized probe_inst=new();
  initial uvm_config_db#(probe_abstract)::set(..., probe_inst);
endinterface

bind dut_inst poly_if poly_probe_inst (.contact(addr));
```

Figure 3 – The instruments of a polymorphic class access

Figure 3 illustrates an example of the polymorphic class access. The bound interface (*poly_if*) encapsulates a specialized descendant (*probe_specialized*) of the global abstract superclass (*probe_abstract*). This subclass implements all the parent's required methods, therefore providing access to the host interface's ports and signals. As a final step, it is instantiated and set to the configuration database, to make it accessible for the rest of the testbench.

It is worth noting that the bound interface type, the instance, and the specialized class type are unknown to the testbench, as they only access the class instance via a superclass typed handle, as shown in Figure 5 later. This detached nature provides a huge advantage for this methodology, albeit some restrictions apply as will be discussed in II.B.

II. APPLICATION

There are several considerations for instrumenting a probing. In this section, the usual challenges and solutions are discussed of both methodologies.

A. The binding host

No matter what methodology one may take, a bind directive will be used to inject a structure into the DUT. As the bind directive can be specified in a module, in an interface or in a compilation-unit scope, there are at least two straightforward places to consider as possible hosts.

Placing the bind into the verification top module seems quite straightforward, but as the bind command may contain hierarchical paths from the DUT, a modification will also be required if there is a structural change in the DUT. Furthermore, this cripples the flexibility of the testbench to accept different kind of DUTs if their inner structure isn't exactly similar regarding the probing targets.

```
module dut (...);
  addr_t addr;
endmodule

module dut_wrapper (interface tb_if);
  dut dut_inst(...);
  bind dut_inst probe_if probe_if_inst (.contact(addr));
  initial uvm_config_db#(virtual probe_if)::set(..., dut_inst.probe_if_inst);
endmodule

module top;
  dut_wrapper dut_wrapper_inst(...);
endmodule
```

Figure 4 – A detached wrapper structure example

Encapsulating the bind into a wrapper structure as shown in Figure 4 seems to be a better choice, as for example, a DUT wrapper could follow the changes in the DUT while being detached from the rest of the testbench. This kind of separation could be used without modification until the wrapper structure's ports are changed and possibly even after that, if the wrapper employs generic interfaces as ports.

This proves to be especially useful if the DUT and the testbench use different repositories or version handling systems and can be modified individually. With such a structure, it is also possible to use the very same testbench for different versions of the DUT, as the hierarchical path definitions follow the appropriate module instantiation.

B. The probe API

As soon as the probes are in place, processing components are needed. There could be a theoretical debate about if they can be named monitors in the UVM's terminology, as they may or may not be part of an agent. However, since they translate bus level signals to the TLM world, this paper will address those components as monitors.

The mandatory requirement of the monitor is to know the type of the virtual interface or polymorphic class it is expecting from the DUT. As the interface construct doesn't support polymorphism, the exact type shall be declared, greatly reducing the component's vertical reuse capability. The class-based access is much more simple and flexible, as the superclass is widely advertised, and the actual object will be compatible with it.

```
class probe_monitor extends uvm_monitor;
  /*  exact interface type required  */
  virtual probe_if probe_vif;
  /*  only the superclass type required  */
  probe_abstract probe_inst;

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    uvm_config_db#(virtual probe_if)::get(..., probe_vif);
    uvm_config_db#(probe_abstract)::get(..., probe_inst);
  endfunction
endclass
```

Figure 5 – Accessing the probes in UVM with both methodologies

Now that the monitor has access to the probe, it is time to compare the API options of the methodologies.

The interface-based approach gives the most freedom to the processing components, as the native port signals and net values are accessible. While usually it can be considered as an advantage, it also makes the monitors error-prone to both human mistakes and interface connection or port list changes. However, as there are no access restrictions enforced, any kind of data transformation or lookup change is possible without modifying the bound component.

The introduction of a new signal to the monitor and to the interface is a simple task, as only the interface's port list and bind statement needs to be updated. Since different monitors may use different signals, adding another one to the port list will not change the operation of the existing components, since they will simply ignore the new item. Such an extension example is shown on Figure 6, where a new data field is added to the existing probe infrastructure and a bind statement that ignores this port.

```
typedef logic[127:0] data_t;

interface probe_if(addr_t address, data_t data);
endinterface

module top;
  ...
  bind dut_inst probe_if probe_if_inst (.address(addr), .data('z));
endmodule
```

Figure 6 – Extending a probe interface with a new field

On the other hand, as the polymorphic solution uses getter functions for data retrieval, the monitor's access is restricted to the implemented methods. While this makes this probing methodology more robust, adding another API lookup function might become a far from trivial task to accomplish, especially if a common superclass is used in multiple probe variants. If an abstract superclass is used as the base of the specialized probes, there is a chance that the API functions are declared as *pure virtual* functions. In this case, adding a new one also requires supplying an implementation in each non-abstract descendant class, thus modifying *all* specialized subclasses.

```
virtual class probe_abstract;
  pure virtual function addr_t get_address();
  /*  data is always [7:3]  */
  virtual function data_t get_data();
    get_data = -1;
  endfunction
endclass
```

Figure 7 – Mixing virtual and pure virtual methods

While there is the option to use a virtual method instead of a pure virtual one and skip this requirement, it introduces uncertainty to the probing infrastructure, as there is no easy way to tell if the value returned came from the function of the superclass or the child class. Of course, if there is an identifiable valid range of values that the function can return in a legal use-case and a default invalid return value is specified in the super class, using a simple virtual method can be justified, just as shown in Figure 7.

C. The parametrized complexity

Until this point, the presented examples haven't contained any parameters for the sake of simplicity. However, one may wonder what happens if parameters are introduced to enhance the reusability of probe infrastructure. Both methodologies share some common points, as the bound interface needs to be supplied with parameters; in the interface bind case to set the width of the ports; in the polymorphic case to supply the specialized class with parameters. Figure 8 and Figure 9 show parametrized examples for both variations.

```
interface probe_if #(int ADDRW=32)(logic [ADDRW-1:0] contact);
endinterface

module top;
  dut dut_inst();
  bind dut_inst probe_if#(16) probe_if_inst (.contact(addr));
  initial uvm_config_db#(virtual probe_if#(16))::set(..., dut_inst.probe_if_inst);
endmodule
```

Figure 8 – Parametrized interface bind structure and configuration database access

```
virtual class probe_abstract #(int PW=32);
  pure virtual function logic[PW-1:0] return_data();
endclass

interface poly_if #(int ADDRW=16)(input logic[ADDRW-1:0] contact);
  class probe_specialized extends probe_abstract #(ADDRW);
    function logic[ADDRW-1:0] return_data();
      return_data=contact;
    endfunction
  endclass

  probe_specialized probe_inst;
  initial probe_inst=new();
  initial uvm_config_db#(probe_abstract#(ADDRW))::set(null, "*", "probe", probe_inst);
endinterface
```

Figure 9 – Parametrized polymorphic class structure and configuration database access

As the interface binding methodology makes the bound structure accessible to the testbench components via a virtual interface variable, the probe monitor needs to know the exact parameterization of the interface to have a type-compatible variable that will be used as a handle. Since such parameters (pseudo-constants) need to be supplied during elaboration time, they shall appear as class parameters to the monitor and its upstream parental chain, as seen on Figure 10.

```
class probe_env extends uvm_env;
  probe_monitor   #(32) probe_mon_32;
  probe_subscriber#(32) probe_subscr_32;
endclass

class probe_monitor #(int DATAW=64) extends uvm_monitor;
  virtual probe_if #(DATAW) probe_vif;

  function void connect_phase(uvm_phase phase);
    uvm_config_db#(virtual probe_if#(DATAW))::get(..., probe_vif);
  endfunction
endclass
```

Figure 10 – Parametrized environment and monitor accessing a bound interface

The other methodology has seemed to be more flexible so far; however, it suffers from the same problems. The abstract class typed handle needs to match in parameters with the instantiated one, so the same type of parameter propagation is also required here. Figure 11 demonstrates this scenario:

```
class probe_monitor #(int DATAW=64) extends uvm_monitor;
  probe_abstract#(DATAW) probe_inst;

  function void connect_phase(uvm_phase phase);
    uvm_config_db#(probe_abstract#(DATAW))::get(this, "", "probe", probe_inst);
  endfunction
endclass
```

Figure 11 – Parametrized monitor accessing a polymorphic class

In both cases, if the TLM analysis components are parameter-dependent, their parental chain and even the mediator transaction *may* need to be parametrized. As the parental chain may go as deep as the *uvm_test* descendant class, implementing a fully parametrized environment might be a serious effort. So it needs to be considered if it worth the invested time. A more detailed discussion and possible workarounds are presented in [5].

D. Interchanging capabilities

In both methodologies, the bound interface is strongly typed and not polymorphic, it may only be changed within the compilation time, without editing the interface's instantiation. Several solutions are discussed in [6], but it seems that the easiest one is to interchange files in the compilation script of the testbench. Other methods include parameter-based compiler directives for example, but these methods introduce complex structures and/or code redundancy, thus reducing reusability. To ensure compatibility, the bound interfaces shall share their name and port list, and the specialized classes shall descend from the same superclass.

On the TLM side of the probing infrastructure, type override methods can be employed to change monitor instance types, and while the interface or the superclass compatibility is ensured, their workflow shouldn't be disturbed at all.

III. LARGE SCALE PROBING

Until now, this paper discussed the possibilities of probing from an academic perspective, like what structures can be employed and how they should be used. However, as one moves toward a real-world application, the value of certain qualities, like reusability and maintainability, increases by a substantial amount. This is especially true if the verification environment employs tens or hundreds of probes with differing requirements of port widths and port numbers, and maintaining many different components is not possible. This chapter will discuss the options of generalization and presents a solution based on the "maximal footprint" approach.

A. Generic interface

SystemVerilog provides a generic port construct which serves as a placeholder for an interface, acting as a weakly-typed variable that accepts all kinds of connected interfaces. This approach pushes the responsibility to the implementer that each referenced signal of interface will be found in the supplied interface. However, there is no such thing as generic interface declaration.

```
interface probe_if(addr_t contact);
endinterface

module top;
  probe_if probe_if_inst;
  initial uvm_config_db#(virtual interface)::set(..., probe_if_inst);
endmodule
```

Figure 12 – Generic interface to the configuration database and a syntax error

A virtual interface is a strongly-typed variable that represents an instance of an actual interface. Since of this representation, it is a strongly-typed property and there is no such thing as configuration database access with a generic interface type; although it would be one of the ultimate solutions for probing and interface access in general. Figure 12 illustrates this case, but similar to the example in Figure 2, the result is an error.

B. Generalized interface

As it was shown, it is not possible to use a fully generic interface due to language limitations. Instead, one may try to generalize the employed interface to enable wide range of horizontal and vertical reuse. A well-established method for this is the so-called "maximal footprint" approach, where the signal bundle sizes are determined to encompass the largest possible use case size and let the built-in vector truncation and extension of SystemVerilog manage the rest of the values. To enhance flexibility, instead of a propagated parametrization, a global package-based parametrization may be used, while omitting the interface parametrization entirely, as suggested by Figure 13.

```
package testbench_base_pkg;
  parameter int ADDRW =   64;
  parameter int DATAW =  512;
  parameter int PARW  = 2048;
endpackage

interface generalized_probe_if(
    input var logic             sync_rst,
    input var logic             async_rst_n,
    input var logic             event_one,
    input var logic             event_two,
    input var logic [ADDRW-1:0] address_one,
    input var logic [ADDRW-1:0] address_two,
    input var logic [DATAW-1:0] data_word_one,
    input var logic [DATAW-1:0] data_word_two,
    input var logic [PARW -1:0] parameter_one,
    input var logic [PARW -1:0] parameter_two
  );
endinterface: generalized_probe_if
```

Figure 13 – A simple generalized interface example

This solution may appear ineffective for several reasons at first sight, since the memory footprint of the virtual interface variables may grow quite large, but as this shall not account more than a few megabytes, it is negligible. Furthermore, this approach suffers from the same error proneness that was discussed in II.B, but this can be relieved by several well-defined rules and conventions. Such rules may include connection conventions, like using the address ports only for addresses and adjusting the least significant bit (LSB) of the address to the LSB of the port, etc.

As the interface serves as a listening only instrument in this application, it is possible to force the input ports to be variable types instead of ambiguous logic or strict net types. This allow the signals to fulfill ref argument roles in functions and tasks, thus allowing more agile application in the probe monitor.

C. Generalized polymorphic class

Likewise, the polymorphic methodology may as well be generalized by employing a similarly parametrized host interface for the specialized class and a corresponding parametrized API function, as illustrated on Figure 14.

However, in many cases, architecting getter functions without any kind of added functionality (e.g. data transformation, etc.) seems redundant, and the including functionality might cripple reusability. As the UVM monitor is completely decoupled from the *type* and *implementation* of the probe class instance, the separation of data manipulation could easily result in faulty results due to implementation mismatches. Therefore, to avoid such errors, if data processing is restricted to the monitor, the polymorphic accessor API doesn't offer a lot in terms of added value.

Another disadvantage of the API functions is that they cannot be used natively as a value reference for other functions. They need some kind of trigger event to initialize reevaluation, while a *var* declared port of the interface will fit the reference's role perfectly. This seems to be a small nuisance, but it is often desirable to deploy several fire-and-forget checks both for synchronous and asynchronous triggers, without creating a specific task with specific signal names to work with. In this case, supplying the checker tasks with references of the probe values seem to be the straightforward option.

```
virtual class generalized_probe_abstract;
  pure virtual function logic[ADDRW-1:0] get_address();
endclass

interface generalized_poly_if (
    input var logic [ADDRW-1:0] address_one
  );
  class generalized_probe_specialized extends probe_abstract;
    function logic[ADDRW-1:0] get_address_one();
      get_address_one=address_one;
    endfunction
  endclass
endinterface
```

Figure 14 – A simple generalized polymorphic class example

To illustrate this with an example, take the following situation: the probe needs to check the value of a data field when another data field changes value and do some further processing, like writing a transaction to an analysis port. While clock gating and timing implementation are *possible* in the specialized probe class (as illustrated in [2, Fig. 4.] for example), it *requires* a workaround to track the value changes of an asynchronous signal *purely* from the TLM side. Opposed to this, the interface described in Figure 13 will solve this problem with a monitor shown below in Figure 15 with ease. The lack of this nimble ability is a huge disadvantage for the polymorphic methodology.

```
class probe_example extends probe_base;
  generalized_probe_if probe_vif;
  uvm_analysis_port#(uvm_transaction) blue_ap;
  uvm_analysis_port#(uvm_transaction) red_ap;

  task run_phase(uvm_phase phase);
    fork
      check_data_field(address_one, data_one, blue_ap);
      check_data_field(address_two, data_two, red_ap);
    join_none
  endtask

  task check_data_field(
      ref logic[FIELDW-1:0] _trigger,
      ref logic[FIELDW-1:0] _value,
      ref uvm_analysis_port#(uvm_transaction) _ap
    );
    forever @(_trigger) begin
      if (_trigger==3'h3) begin
        /*  write transaction to analysis port etc.  */
      end
    end
  endtask
endclass
```

Figure 15 – Asynchronous deployment example in a probe monitor

D. The suggested methodology

Taking all the above into consideration, the suggested probe methodology that ensures maximum flexibility on the TLM level consists of the following components:

- A generalized, maximal footprint interface, similar to the one shown at Figure 13, to avoid parametrization altogether

- Bind statement wrapper structures around the DUT, similar to the one described in Figure 4, to guarantee DUT version tracking ability and decoupling option
- A specialized UVM monitor component that is a descendant of a probe monitor base class or *uvm_monitor*, to ensure factory override and interchange capability
- Furthermore, the aforementioned bound interface deployment conventions need to be created

Such a probing environment takes the most advantages from each of its components, while it preserves simplicity and offers easy extensibility.

IV. Conclusion

This paper has presented the current options of probing found in SystemVerilog, describing both the interface binding and the polymorphic class access methodologies. Practical considerations have been made regarding the complexity of deployment, the amount of required overhead and the pitfalls of parametrization. A solution has been suggested that employs the "maximal footprint" approach to avoid the difficulties of parameter handling, while providing flexibility and reusability on the TLM level of the testbench. Examples have been used to illustrate the key point of the design and to point out advantages and disadvantages in different levels of the methodologies to help verification engineers solve common probe-related problems.

This generalized interface technique was successfully applied to a highly complex neural network related automotive design. The ability to have an array of probes deployed with different configurations, with minimal modification of the verification environment saved considerable amount of time and effort. The author firmly believes this approach can be used effectively in other projects and ported to different use cases.

References

[1] IEEE, "Standard for SystemVerilog — Unified Hardware Design, Specification, and Verification Language," in *IEEE Std 1800-2012*.

[2] D. Rich and J. Bromley, "Abstract BFMs Outshine Virtual Interfaces for Advanced SystemVerilog Testbenches," in *Proceedings of DVCon 2008*, San Jose, CA.

[3] D. Rich, "The Missing Link: The Testbench to DUT Connection," in *Proceedings of DVCon 2012*, San Jose, CA.

[4] S. Bhutada, "Polymorphic Interfaces: An Alternative for SystemVerilog Interfaces," *Verification Horizons*, vol. 7, no. 3, pp. 19-24.

[5] J. Bromley, "Slicing Through the UVM's Red Tape," in *Proceedings of DVCon EU 2016*, Munich.

[6] G. Blake and S. Chappel, "One Compile to Rule Them All: An Elegant Solution for OVM/UVM Testbench Topologies," in *Proceedings of DVCon 2013*, San Jose, CA.

[7] Verification Academy, UVM Cookbook, Mentor Graphics.