Verification Patterns – Taking Reuse to the Next Level

Harry Foster
Mentor Graphics Corporation
Texas, USA
Harry_Foster@mentor.com

Michael Horn
Mentor Graphics Corporation
Colorado, USA
Mike_Horn@mentor.com

Bob Oden
Mentor Graphics Corporation
North Carolina, USA
Bob_Oden@mentor.com

Pradeep Salla
Mentor Graphics Corporation
Bangalore, India
Pradeep_Salla@mentor.com

Hans van der Schoot
Mentor Graphics Corporation
Ottawa, Canada
Hans_vanderSchoot@mentor.com

Abstract-Design patterns provide an optimized, reusable solution to many of today's engineering problems. Experience has shown that they are an effective tool for sharing best practices and building skills within a project team. However, one problem that has prevented the widespread adoption of design patterns within the microelectronic verification community is a lack of an easily searchable library of patterns. In this paper, we demonstrate a systematic set of guidelines for creating and organizing an extendable library of verification patterns that are applicable across multiple technologies and engines (or platforms) in the verification space—ranging from property specification to UVM testbench development—and formal verification, simulation, and emulation.

1. INTRODUCTION

What is a pattern? In the process of designing something (e.g., a building, software program, or an airplane) the designer often makes numerous decisions about how to solve specific problems. If the designer can identify common factors contributing to the derived solution and abstracts the solution in such a way that it can be applied to other similar recurring problems, then the resulting generalized problem-solution pair is known as a pattern. Documenting patterns provides a method of describing good design practices within a field of expertise and enables designers to improve the quality in their own designs by reusing a proven solution on a recurring problem. In the context of this paper, we define a pattern library as a collection of pattern entries—where each documented pattern entry provides a solution to a single problem.

Design patterns originated as a contemporary architectural concept from Christopher Alexander in 1977, and they have been applied to the design of buildings and urban planning. [2] In 1987, Kent Beck and Ward Cunningham proposed the idea of applying patterns to programming. [11] However, it was Gamma et al., also known as the Gang of Four (GoF) who popularized the concept of patterns in computer science after publishing their book *Design Patterns: Elements of Reusable Object-Oriented Software* in 1994. [1]

Applying patterns to verification is not a new idea. For example, Dwyer et al. [3] in 1998 initially proposed the idea of codifying and reusing property specifications for finite-state verification using patterns. Foster et al. [4] in 2004 extended the idea of applying patterns to the general domain of assertion-based verification. More recently, the literature for many of today's testbench verification methodologies (such as UVM) often reference various software or object-oriented related patterns in their discussions, which can provide valuable insight into how to achieve better vertical and horizontal reuse for testbenches and stimulus and even platform portability with the advent of hardware-assisted acceleration. [5][6][7][8][9][10]

Although patterns have been referenced in the context of design verification in numerous conference papers, it is generally difficult to search, reference, and leverage the published solutions these patterns provide since they are distributed across multiple heterogeneous platforms and databases and documented using multiple varied formats. Furthermore, prior work in patterns applied to hardware verification problems has been limited in scope by focusing predominantly on the coding aspect of simulation testbenches. In this paper, we address these concerns by extending the application of patterns across the entire domain of verification (i.e., from specification to methodology to implementation) and introduce a systematic set of steps for organizing and documenting an easily referenced verification patterns library that is applicable across verification engines (or platforms).

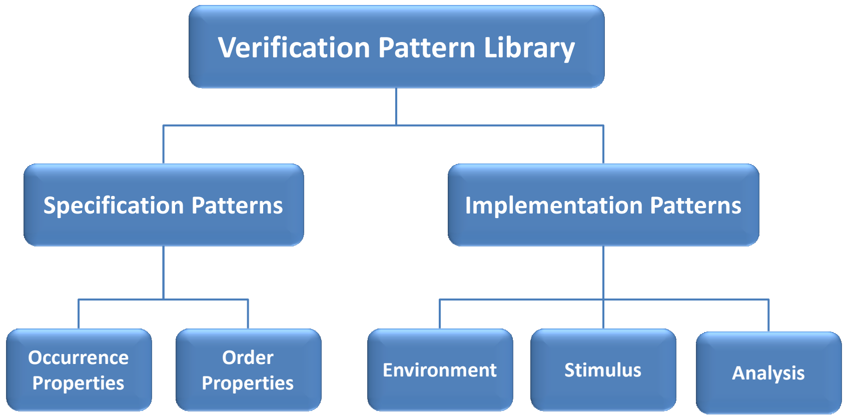
***Paper Organization*—**The remainder of this paper is organized as follows. In Section 2, we provide a recommendation for categorizing and organizing patterns into a library. In Section 3, we turn our focus to a recommended style for documenting patterns and requirements for creating an easily referenced verification pattern library. In Section 4, we present verification pattern examples related to various disciplines in the verification space (such as property specification, UVM testbench development, and hardware acceleration). In Section 5, we discuss other considerations when creating a patterns library—and introduce our freely available, online verification patterns library, which is based on the systematic patterns creation approach and guidelines we discuss in the paper. In Section 6, we summarize our work and then discuss future work, such as extending patterns into newer and emerging areas of verification (for example, portable stimulus).

2. CATEGORIZING AND ORGANIZING PATTERNS

To facilitate learning, ease of use, and quick access when searching for verification pattern content, careful thought should go into organizing the library into searchable categories whose patterns solutions are related and exhibit similar characteristics. Although the verification patterns contained in our library extend beyond traditional patterns used within the software community, we felt that it was important to first understand and then build on the prior work that went into pattern categorizations from this domain.

The first attempt at design pattern categorization is attributed to the GoF, who proposed the following pattern organization in the context of creating reusable object-oriented software: *Creational Patterns* (which deal with object creation), *Structural Patterns* (which identify a simple way to realize relationships between entities), and *Behavior Patterns* (which identify common communication patterns between objects). [1] Buschmann et al., in their *Pattern Oriented Software Architecture* book series built on the work of the GoF and proposed three broader category levels for patterns: *Architectural Patterns* (e.g., Layers, MVC, P2P), *Design Patterns* (e.g., GoF proposal), and *Idioms* (e.g., language-specific patterns like Pimpl, RAII in C++). [12] However, Martin Fowler, in his book *Pattern of Enterprise Application Architecture* (POEA) points out that patterns are not just for documenting solutions to code specific problems—they are also useful for describing solutions to how data and system components are arranged and interconnected. [13]

Since our goal in creating verification patterns is to broaden the application of patterns beyond the software domain, we decided that our categories should align from a high level with the digital design and verification process. Hence, we have identified two main verification pattern categories, which should be familiar to any design and verification engineer working in this domain. That is, *Specification Patterns* and *Implementation Patterns*, as illustrated in the following figure.



2.1 SPECIFICATION PATTERNS

Specification Patterns provide solutions to notational problems when specifying design intent. Various forms of specification notation are being explored in the digital design and verification industry today, such as UML, graphs, and property specifications. Probably the most prevalent form of formally specifying design intent in the digital verification domain is through the use of properties, which can be implemented as either assertions or cover properties. However, learning how to specify properties has historically been a challenge. Hence, for the initial release of our verification pattern library, we decided to focus on the challenge of creating property patterns—with the goal of facilitating reuse of common solutions to various property specification problems.

As shown in the previous figure, our property specification patterns are furthered organized into two subcategories, which is based on the work of M. Dwyer et al. [3]. The subcategories are: *Occurrence Properties* and *Order Properties*. Occurrence Property patterns require that either some state or event[1] must occur (e.g., a Universality Property Pattern or an Existence Property Pattern [14]) or not occur (e.g., an Absence Property Pattern [14]). Alternatively, Order Property patterns constrain the order of states and events (e.g., a Precedence Property Pattern or a Response Property Pattern [14]). More complex, compound properties can then be built up from combinations of more basic occurrence and order patterns contained in our library. In section 5.1, we demonstrate an Occurrence Pattern from our library.

2.2 IMPLEMENTATION PATTERNS

Implementation Patterns provide solutions to the construction problem for various verification infrastructures. Since the construction of contemporary testbenches are essentially large software projects, which utilize object-oriented features found in SystemVerilog and UVM, a lot of the prior work in software patterns is applicable to verification Implementation Patterns. In fact, many of the patterns referenced in prior verification publications originated from the set of reusable software object-oriented patterns proposed by the GoF. For example:

- Factory – Factory Pattern (Abstract Factory) & Singleton
- Policy Knobs (Objects) – The Policy Pattern [Strategy Pattern]
- Analysis Ports & Analysis Components – Observer Pattern
- Phases – Template Method Pattern
- TLM – Command Pattern
- UVM Object and Component registration – Proxy Pattern
- UVM_TOP – Singleton
- Object Wrappers – Adopter or Wrapper Pattern

However, verification is more than just creating testbenches. Hence, our category of Implementation Patterns is intended to cover a broader set of solutions to commonly occurring problems involved in verification—such as stimulus and analysis.

We have further organized our Implementation Patterns into the following three subcategories, as illustrated in the previous figure: *Environment*, *Stimulus*, and *Analysis*. The Environment patterns are those that are used in testbench architecture, construction, configuration, and communication/synchronization (e.g., Façade Pattern or Component Configuration Pattern [14]). These patterns are the ones that capture the structural and behavior aspects of the core verification environment model. Stimulus patterns capture the behavior, strategy, and types of stimulus (e.g., Layering Sequence Pattern [14]). Similarly, Analysis patterns are used to capture the behavior, strategy, and types of response checking and coverage (e.g., Walking Pattern [14]). For reuse, analysis and stimulus have no interaction or interdependency. The environment (testbench) includes infrastructure, interconnect, resource sharing, synchronization, and so on. As such, it touches both analysis and stimulus. It is the structure in which both analysis and stimulus reside and operate.

3. A TEMPLATE FOR DOCUMENTING VERIFICATION PATTERNS

When creating a pattern library, it is important that each documented pattern follow a consistent format and style. This consistency simplifies learning and facilitates ease of use when reviewing different patterns contained in the library. The documentation for a verification pattern should describe the context in which the pattern is used—a problem within this context that the pattern is seeking to address—and a suggested solution.

[1] An event could be specified as a Boolean equation that references state elements or variables from the RTL.

Historically, no single standard format for documenting patterns exists. Rather, a variety of different formats have been used by different pattern authors. One example of a commonly used documentation format is the one used by the GoF in their book *Design Patterns*.[1] However, previous pattern documentation proposals focused on software patterns. We found that there are other necessary pattern documentation requirements that are specifically related to verification patterns and that are not addressed by previous proposals—such as documenting Specification Patterns. Hence, our verification pattern template extends previous software pattern documentation proposals to address a broader set of requirements related to verification.

3.1 VERIFICATION PATTERN TEMPLATE

Our verification pattern template consists of the following sections:

- **Pattern Name:** A unique (descriptive) name that helps in identifying and referencing the pattern.
- **Intent:** A very brief description of the goal behind the pattern and the reason for using it.
- **Motivation:** A description of a specific scenario consisting of a problem and a specific context in which this pattern can be applied. Think of this as a problem statement that describes a concrete example. (The solution to the problem will be discussed in the subsequent Implementation and Example sections.)
- **Applicability:** Situations in which this pattern is usable; the general context for the pattern.
- **Structure:** *(Required for Implementation Patterns and optional for Specification Patterns)* Abstract graphical representation of the pattern (e.g., UML class diagrams, interaction diagrams, etc.).
- **Implementation:** A description of an implementation of the pattern; the solution part of the pattern.
- **Example:** A code (or pseudo-code) example of how the pattern can be used. It is suggested that the example addresses the original problem scenario presented in the motivation section.
- **Scope:** *(Recommended for Specification Patterns and optional for Implementation Patterns)* A scope defines the extent of the verification execution over which the pattern must hold. More property scope details will be discussed in section 3.2, and an example is provided in section 4.1.
- **Consequences:** *(Optional)* A description of the results, side effects, and tradeoffs caused by using this pattern.
- **Related Patterns:** *(Optional)* Other patterns that have some relationship with the pattern with a discussion of the differences and similarities between the related patterns.
- **Contribution:** Identification of person and/or references for this pattern contribution to the library.

For the novice pattern creator, there is often confusion related to the differences between the sections *Intent*, *Motivation*, and *Applicability*. The *Intent* section is a generic, high-level description of the problem being solved, and it should be limited to one or two sentences. However, the *Motivation* section is a description of a "specific" example of a problem where a solution is needed (without describing the solution), and it does *not* need to be brief. Note that the *Motivation* section is sometimes labeled as the *Problem Statement* in other proposed pattern templates. The *Applicability* section will then generalize the "specific" problem example described in the *Motivation* section so that the reader understands how the proposed pattern can be applied to other similar problems. Finally, we recommend that the *Example* section demonstrate the solution on the same problem previously described in the *Motivation* section.

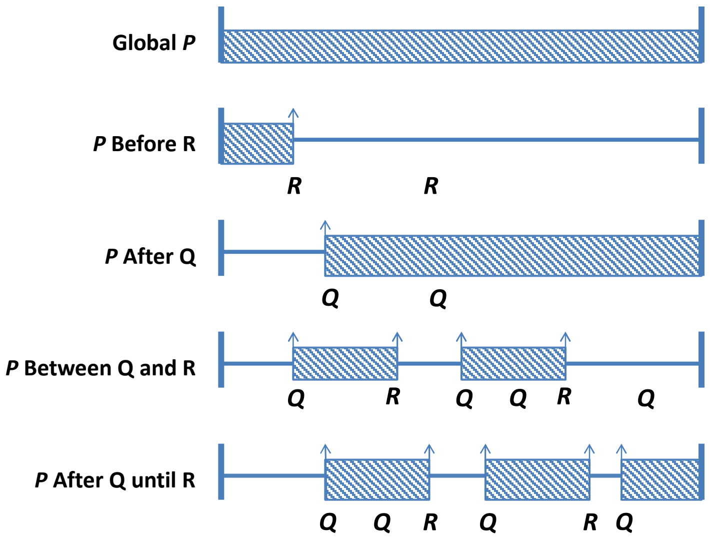
3.2 SPECIFICATION PATTERNS AND PROPERTY SCOPES

Foster et al. documented sixteen patterns in their book *Assertion-Based Design*. [4] However, a recent and deeper analysis of these patterns revealed that the base property for a number of the pattern examples had a similar implementation—and the main difference between these patterns were often due to the boundary enabling and fulfilling conditions that delimited the base property. This resulted in a slightly larger set of documented patterns than necessary. Furthermore, in some cases, it was difficult to see how the example for a specific property specification could then be generalized to address new specification problems.

M. Dwyer et al. proposed an elegant solution to the specification patterns generalization problem by defining property scopes—where a scope defines the extent of the verification execution (i.e., the design model execution in simulation, emulation, or formal verification) over which the base pattern must hold. [3] The advantage of defining scopes is that it greatly simplifies the process of documenting property specification patterns since it generalizes the application of a base pattern problem-solution pair to many other similar, yet slightly different problems—and thus greatly reduces the amount of required documentation. Hence, we have adopted the concept of property scopes when documenting property specification patterns released in our library.

We use the figure below to help illustrate the concept of various scopes for property P and its potential boundary enabling and fulfilling conditions Q and R, respectively.

Experience has shown that there are five common scopes (as shown in the following figure) that cover a many properties related to digital design [3], which are: *global* (the entire verification execution that is under consideration), *before* (if R occurs, then the verification execution up to but not necessarily including the fulfilling condition R), *after* (the verification execution including and after the enabling condition Q), *between* (If R occurs, then the verification execution including and after the enabling condition Q and up to, but not necessarily including, the fulfilling condition R), *after-until* (the verification execution including and after the enabling condition Q and up to, but not necessarily including, the fulfilling condition R). The difference between the *between* and *after-until* scopes is that the *between* scope of P only applies if the fulfilling condition R occurs after the enabling condition Q. However, the *after-until* scope of P applies after the enabling condition Q, even if the fulfilling condition R never occurs.



Scope: Extent of the model's execution over which the property *P* holds.

Note, instead of defining a much larger set of scopes for what we found to be less common instances where property P does not overlap with the enabling condition Q, or the instance where property P is required to overlap with the fulfilling condition R, it is easy to extend our smaller set of scopes to include these conditions by coding the property appropriately (e.g., qualify the enabling condition Q to prevent overlap with P, or qualify the fulfilling condition R to ensure overlap with P).

Pattern scopes are not necessarily applied to the overall property. That is, often a scope is applied to a sub-property that is used to form a more complex overall property (e.g., `sequence_expr |-> property_expr`, where the pattern scope might be applied to the sub-property `property_expr` versus the overall property).

4. Pattern Examples

Due to space limitations, we have chosen to present three examples from our library in this paper. Each example is representative of patterns from the main categories contained in our library: Specification Patterns and Implementation Patterns.

4.1 Specification Pattern Example

The following example demonstrates our documentation approach for Specification Patterns and how property scopes can be used as a powerful tool to generalize the implementation of a specific specification pattern solution to an alternative form, which is then used to address a different specification problem.

Pattern Name: *Forbidden Sequence Property Pattern.*

Intent: The Forbidden Sequence Property Pattern is used to specify portions of a design model's verification execution that forbids a specific sequence of designated states or events.

Motivation: In the normal verification execution of an RTL model, there are often specific sequences of states or events that must never occur. The classic example of applying the Forbidden Sequence Property Patterns relates to checking fairness in an arbiter. For example, if a specific client A issues a request to the arbiter, and the arbiter issues a sequence of multiple grants to client B before client A is issued a grant, then the arbiter is not fair.

Applicability: Any sequence of states or events that describes undesirable behavior in a design, can be formulated into a forbidden sequence property.

Implementation: The Forbidden Sequence Property Pattern can be expressed using any of the industry standard specification languages (such as SVA or PSL).

The following table has been created only to provide a better understanding of the semantics of the Forbidden Sequence Property Pattern for our five basic scopes—where a scope is the extent of the model's verification execution over which the property must hold.

The exampled described in the following table illustrates one instance of the Forbidden Sequence Property Pattern that specifies that the sequence of three low-to-high T transitions is forbidden. For this example, Q, R and T are Boolean expressions.

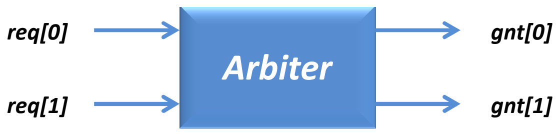
Property Scope	Definition
Global[2]	`not ($rose(T)[->3])`
Before[3] **R**	`(R[->1]) implies (not ($rose(T)[->3]) s_until R)`
After Q	`always (Q |-> not ($rose(T)[->3]))`
Between[4] **Q and R**	`always (((Q & !R) ## R[->1]) implies (not ($rose(T)[->3]) s_until R))`
After Q until R	`always (Q |-> (not ($rose(T)[->3]) until R))`

Example: Let us consider a fair, two-client arbiter as illustrated in the figure below, where signals `req[0]` and `req[1]` are input requests to the arbiter from clients 0 and 1, and `gnt[0]` and `gnt[1]` are the output grants, respectively. For our example, a request is defined as a rising edge occurrence for either `req[0]` or `req[1]` and similarly for a grant.

[2] For this particular pattern, and for performance reasons, the *global* scope is generally more applicable for defining a sub-property versus an overall property.

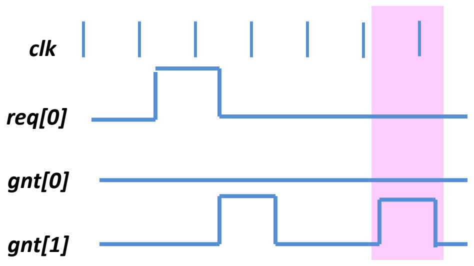
[3] What this pattern is expressing is that *if R eventually occurs, then the forbidden sequence must hold before R*. If R does not occur, then the forbidden sequence is not required to hold. The *Before* scope is generally more applicable when defining a sub-property within a more complex property. Caution, this pattern may incur performance issues, and the SVA `implies` construct is not supported by all tools with respect to *property* `implies` *property*.

[4] What this pattern is expressing is that *if R eventually occurs after Q, then the forbidden sequence must hold between Q up to but not including R*. If R does not occur after a Q, then the forbidden sequence is not required to hold. This pattern can incur performance issues, and the SVA `implies` construct is not supported by all tools.



2-client Arbiter

For our two-client arbiter to be fair, a client with a pending request should never have to wait more than two arbitration cycles to receive a grant (where an arbitration cycle is the interval between issued grants). In another example, if client 0 has a pending request, and a grant is issued twice to client 1 before a grant is issued to client 0, then the arbiter is not fair. The following waveform illustrates this failing case.



Client 0 has a pending request, yet
grant was issued twice to client 1.

We can create a sequence that represents the condition where two grants are issued to client 1, and throughout this sequence, no grant is issued to client 0, as follows:

```
( !gnt[0] throughout $rose(gnt[1])[->2] )
```

This forbidden sequence can be used to construct our fairness property. For this case, we specify that client 1 should never be granted more than one grant when we have a pending request from client 0.

```
property p_arbiter_fair_0_1;
   @(posedge clk) disable iff (!reset_n)
       $rose(req[0]) |-> not ( !gnt[0] throughout $rose(gnt[1])[->2] );
endproperty
```

Similarly, we can write a separate property to specify the fairness with respect to client 1 (e.g., p_arbiter_fair_1_0). Alternatively, we could create a parameterized property that could be used to check for any pairwise arbiter grants (i, j) as follows:

```
property p_arbiter_fair(i,j);
   @(posedge clk) disable iff (!reset_n)
      $rose(req[i]) |-> not ( !gnt[i] throughout $rose(gnt[j])[->2] );
endproperty
```

This parameterized property could then be used to fully specify any *n*-client arbiter (e.g., `p_arbiter_fair(0,1)`, `p_arbiter_fair(0,2)`…`p_arbiter_fair(0,n)`, `p_arbiter_fair(1,2)`…). A SystemVerilog `generate` statement could be used to efficiently create the full set of properties using this parameterized property.

Scope: The scope for the Forbidden Property Pattern in our previous fairness property example is *after*.

Obviously, depending on the property we wish to specify, the Forbidden Sequence Property Pattern is applicable to other scopes (e.g., *global*, *before*, *between,* or *after-until*).

Consequences: The Forbidden Sequence Property Pattern does not specifically require the occurrence of any number of instances for a given state or event (rather it forbids the number of occurrences). A separate property needs to be written if we want to specify that a specific number of occurrences are required.

Related Patterns: The Forbidden Sequence Property Pattern is classified as a Specification Pattern, with a sub-classification of an Occurrence Pattern. This pattern is a special case of the Absence Property Pattern.

4.2 IMPLEMENTATION PATTERN EXAMPLES

The following are two verification pattern examples that demonstrate how we document Implementation Patterns. The first example demonstrates a pattern targeted at both simulation and emulation, while the second example demonstrates how to share resources between objects without requiring detailed knowledge of the resource.

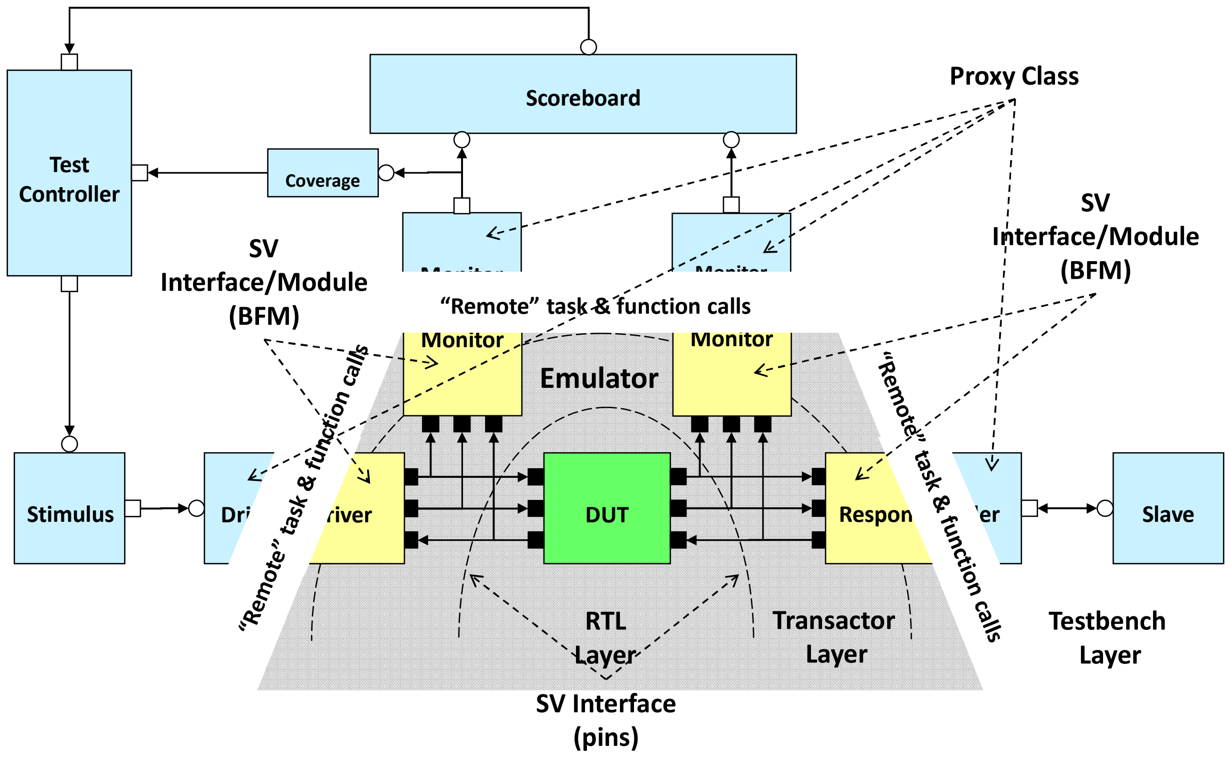
Pattern Name: *The BFM-Proxy Pair Pattern.*

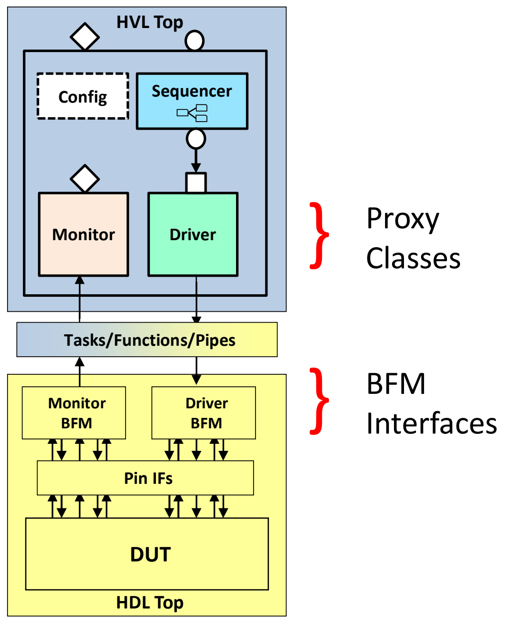
Intent: The BFM-Proxy Pair Pattern is categorized as an Environment Pattern and facilitates the design of transactors like drivers and monitors for dual domain partitioned testbenches that can be used for both simulation and emulation, and across verification engines (or platforms) in general.

Motivation: In order to enable and promote a verification process that is abstracted from underlying verification engines, particularly a software simulator and a hardware emulator, modern testbenches should exhibit (from conception) a dual domain architecture with partitioned HVL and HDL module hierarchies targeted for the simulator and emulator, respectively, and linked together to run in unison. Fundamental to this architecture is the employment of BFM-proxy pairs to devise so-called split transactors, where components in the HVL domain typically implemented as classes act as proxies to BFMs implemented as interfaces or modules in the (synthesizable) HDL domain. An HVL proxy provides a surrogate or placeholder for the associated cross-domain HDL BFM to control access to it via a transaction-based HVL-HDL communication model using remote function and task calls. Effectively, the proxy embodies the transactor API to upper testbench layers, abstracting the cross-domain communication and the implementation details of the BFM's bus cycle state machines.

Applicability: The BFM-Proxy Pair Pattern is applicable in any situation demanding a common dual domain partitioned testbench architecture (i.e., separated HVL and HDL module hierarchies) for both simulation and emulation, and across verification engines in general.

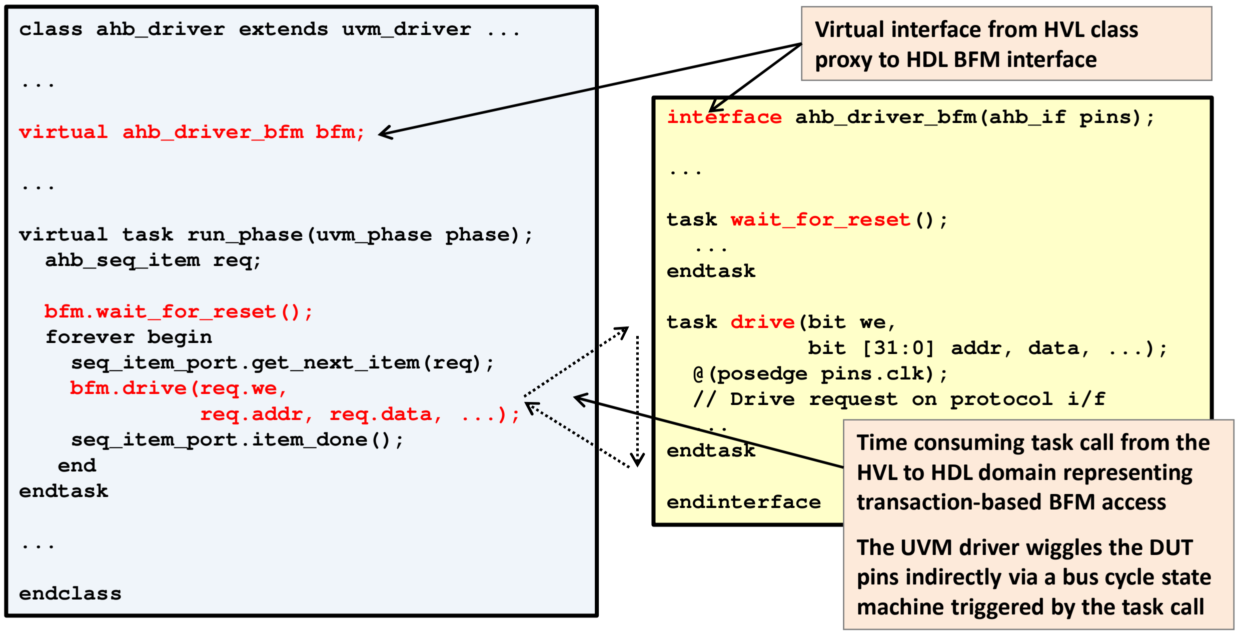
Structure: The diagrams below illustrate the dual domain testbench architecture and the according UVM agent structure, respectively, with the transactors depicted as BFM-proxy pairs.





Implementation: A transactor following the prescribed BFM-Proxy Pair Pattern implements a BFM as a SystemVerilog interface (or module) with dedicated functions and tasks to be called from a class proxy through a virtual (or DPI-C) interface to execute bus cycles, set parameters, or get status information. Additionally, a BFM interface (or module) may call functions defined in the class proxy via a proxy object back-pointer mechanism to provide notifications of transactions and other interesting events and conditions for control and analysis. Transaction-based cross-domain communication is thus enabled in both directions with either the HVL proxy or the HDL BFM as initiator. Each proxy-BFM pair is regarded as a joint pair representing a single transactor.

Example: BFM-Proxy Pair Pattern source code examples for a UVM driver and monitor are provided below:



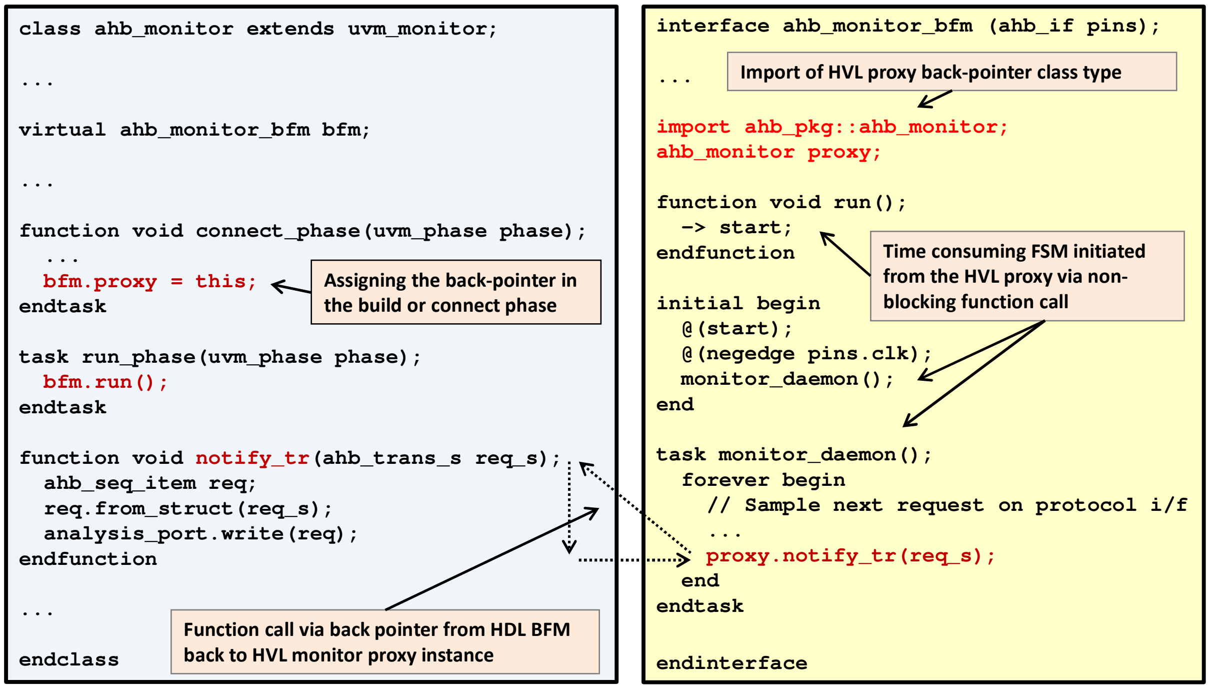
```
class ahb_driver extends uvm_driver ...

...

virtual ahb_driver_bfm bfm;

...

virtual task run_phase(uvm_phase phase);
  ahb_seq_item req;

  bfm.wait_for_reset();
  forever begin
    seq_item_port.get_next_item(req);
    bfm.drive(req.we,
              req.addr, req.data, ...);
    seq_item_port.item_done();
   end
endtask

...

endclass
```

```
interface ahb_driver_bfm(ahb_if pins);

...

task wait_for_reset();
  ...
endtask

task drive(bit we,
           bit [31:0] addr, data, ...);
  @(posedge pins.clk);
  // Drive request on protocol i/f
  ...
endtask

endinterface
```

Virtual interface from HVL class proxy to HDL BFM interface

Time consuming task call from the HVL to HDL domain representing transaction-based BFM access

The UVM driver wiggles the DUT pins indirectly via a bus cycle state machine triggered by the task call



Assigning the back-pointer in the build or connect phase
Function call via back pointer from HDL BFM back to HVL monitor proxy instance
Import of HVL proxy back-pointer class type
Time consuming FSM initiated from the HVL proxy via non-blocking function call

```
class ahb_monitor extends uvm_monitor;

...

virtual ahb_monitor_bfm bfm;

...

function void connect_phase(uvm_phase phase);
  ...
  bfm.proxy = this;
endtask

task run_phase(uvm_phase phase);
  bfm.run();
endtask

function void notify_tr(ahb_trans_s req_s);
  ahb_seq_item req;
  req.from_struct(req_s);
  analysis_port.write(req);
endfunction

...

endclass
```

```
interface ahb_monitor_bfm (ahb_if pins);

...

import ahb_pkg::ahb_monitor;
ahb_monitor proxy;

function void run();
  -> start;
endfunction

initial begin
  @(start);
  @(negedge pins.clk);
  monitor_daemon();
end

task monitor_daemon();
  forever begin
    // Sample next request on protocol i/f
    ...
    proxy.notify_tr(req_s);
  end
endtask

endinterface
```

Consequences: The dual domain partitioned testbench architecture enabled by this BFM-Proxy Pair Pattern offers maximum leverage of established simulation-based verification practices into emulation, including the benefits of using SystemVerilog and UVM for creating modular, reusable verification components and environments.

Related Patterns: A precursor to this BFM-Proxy Pair Pattern is the Dual Domain Hierarchy Pattern, which advocates the HVL and HDL domain partitioning as a sound and necessary separation of concerns fundamental to emulation and other hardware-assisted verification platforms. Additionally, the BFM-Proxy Pair Pattern resembles the proxy pattern as one of the structural patterns of the GoF's OOP design patterns (though applying instead between a dynamic proxy object and a static interface or module).

Pattern Name: *Resource Sharing Pattern.*

Intent: The Resource Sharing Pattern is categorized as an Environment Pattern is used to share resources between objects without requiring detailed knowledge of the resource. Related resources share common access attributes thereby creating simple associations.

Motivation: A hierarchical simulation environment contains resources required by other components as well as test writers. A consistent mechanism for sharing resources promotes horizontal and vertical reuse. A simple mechanism that requires no knowledge of the simulation environment hierarchy eases the task of test and stimulus creation.

Applicability: The Resource Sharing Pattern can be used when reusing verification components. It can also be used to reduce the overhead of adding test writers to a project.

Structure:

Resource Sharing Attributes	
Attribute	**Value**
`cntxt`	null
`inst_name`	String identifying resource group
`field_name`	String identifying specific resource

Implementation: The Resource Sharing Pattern can be implemented using either the `uvm_config_db` or `uvm_resource_db` within UVM. For the example below the `uvm_config_db` was selected because of its simplicity of use and use model available. The `uvm_config_db` has two methods for resource sharing: set and get. It also supports the generic scope use model shown in this example as well as a hierarchical scope use model required to share specific resources with specific objects. Resources shared can include but not be limited to configuration objects, virtual interface handles and sequencer handles.

Example: Let us consider a DUT with various protocol interface ports. Each interface on the DUT is given a unique string identifier. For this example let us give one of the interface ports the unique string identifier "INGRESS_DATA_PORT". This unique string identifier is used for the `field_name` attribute listed in the structure table. All resources for that interface including the virtual interface handle, agent configuration handle and sequencer handle are identified using the unique string identifier. The table below shows the values used to provide and access the various resources associated with this interface.

The required constructs are available to environment developers to share resources within the environment. The test writer only needs to know the information in the table to access resources associated with the ingress data port on the design. No detailed knowledge of environment hierarchy is required to write test scenarios.

Resource	cntxt	`inst_name`	`field_name`
Configuration handle	null	"CONFIGURATIONS"	"INGRESS_DATA_PORT"
Virtual interface handle	null	"INTERFACES"	"INGRESS_DATA_PORT"
Sequencer handle	null	"SEQUENCERS"	"INGRESS_DATA_PORT"

5. Considerations for Creating our Pattern Library

It is important to establish usability goals early on when developing any type of online library. And since achieving usability goals often places constraints on the required infrastructure, this is one area requiring thoughtful consideration. Consequently, we chose to broadly identify the usability goals for our verification pattern library as a knowledge base that is easily discoverable, referenceable, and relatable.

In addition to usability goals, we felt it important to set goals on the pattern creation process and how to effectively populate the library. On a related note, you might have wondered why there is such a large set of authors listed on this paper (which we refer to as the Gang of Five). The reality is that verification is a diverse field, and it often requires expertise in varied areas, such as methodologies, technologies, tools, and languages. No single person is a master in every aspect of verification. Thus, to create patterns across the broad field of verification, we built a team made up from experts in assertion-based verification, formal verification, constrained-random and coverage-driven verification, UVM, hardware-assisted verification, and emulation. However, even with this diverse team of experts we recognize that there is still additional verification expertise required for solving verification problems in specific application domains. Hence, for our verification patterns library, we set a goal that the pattern creation process should harness the power of online social communities made up from a diverse set of verification experts that work in multiple application domains. In turn, this community of experts would foster collective problem solving for the creation of novel patterns and provide alternative, optimized solutions for existing pattern content. To achieve these goals, we developed a web-based infrastructure that allows new content to be contributed in a consistent format, which follows the pattern template guidelines that we discussed in Section 3.

The final verification patterns library goal we set was to grow the user base as quickly as possible. For this goal, we decided to leverage the Verification Academy since it consists of an existing online social community with over 35,000 design and verification engineers. Furthermore, the Verification Academy provided us an existing online infrastructure (reducing our development costs), which enabled the creation of a patterns knowledge base that is easily discoverable, referenceable, and relatable. Our freely available, online verification patterns library can be accessed at www.verificationacademy.com. [14]

6. Summary and Future Work

As we stated in the introduction, pattern examples from previous publications are generally difficult to search, reference, and leverage since they are distributed across multiple heterogeneous platforms and databases and documented using multiple varied formats. In addition, prior work in verification patterns has been limited in scope by focusing predominantly on the coding aspect of simulation testbenches. In this paper, we addressed these concerns by extending the application of patterns across the entire domain of verification (i.e., from specification to implementation) and then introduce a systematic set of steps for organizing and documenting an easily referenced verification patterns library. Finally, we demonstrated our documentation format on three examples, which are representative of the class of patterns found in our library.

Our current plans are to continue populating the library with new patterns, many of which will be drawn from previous publications. In addition, we are opening our pattern library up to an online social community to contribute one's own unique patterns to the library. Our future plans are to continue exploring new applications of patterns in emerging verification domains, such as portable stimulus and system-level analysis. Finally, we plan to explore a few additional scopes that leverage the expressiveness of regular expressions, which were not considered by the work of Dwyer et al. [3]

REFERENCES

[1] E. Gamma et al. (1994) Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley.

[2] C. Alexander. (1979) The Timeless Way of Building, New York: Oxford University Press.

[3] M. Dwyer et al. (1998) 2nd Workshop on Formal Methods in Software Practice.

[4] H. Foster et al. (2004) Assertion-Based Design, 2nd-Edition, Kluwer Academic Publishers.

[5] M. Glasser (2009) Open Verification Methodology Cookbook, Springer.

[6] J. Sprott (2008) Improve Your SystemVerilog OOP Skills by Learning Principles and Patterns, SystemVerilog Users Group

[7] H. van der Schoot et al. (2011) "Off to the races with your accelerated SystemVerilog testbench," DVCon 2011

[8] UVM Cookbook—Emulation, Verification Academy, https://verificationacdemy.com/cookbook/emulation

[9] Accellera Portable Stimulus Specification Working Group, http://www.accellera.org/activities/working-groups/portable-stimulus

[10] H. van der Schoot, A. Yehia (2015) "UVM & Emulation: How to get your ultimate testbench acceleration speed-up," DVCon Europe 2015

[11] Beck, Kent; Cunningham, Ward (1987). "Using Pattern Languages for Object-Oriented Program," OOPSLA '87 workshop on Specification and Design for Object-Oriented Programming.

[12] Buschmann, Frank (2000). *Pattern-Oriented Software Architecture*, John Wiley & Sons.

[13] Fowler, Martin (2003). *Patterns of Enterprise Application Architecture*. Addison-Wesley.

[14] Verification Academy Patterns Library, www.verificationacademy.com

APPENDIX: VERIFICATION ACADEMY PATTERN LIBRARY

The following table represents the Verification Academy Pattern Library released content as of the time of this publication. For an up-to-date listing of released patterns, visit www.verificationacademy.com.

Pattern Name	Category	Subcategory	Description
Absence Property Pattern	Specification	Occurrence	Used to specify states or events1 in a system that must never occur during execution. Also known as Never.
Existence Property Pattern	Specification	Occurrence	Used to specify portions of a system's execution that contains an instance of a certain states or events. Also known as Eventually or Future
Forbidden Sequence Property Pattern	Specification	Occurrence	Used to specify portions of a system's execution that forbids a sequence of states or events.
Bounded Existence Property Pattern	Specification	Occurrence	Used to specify portions of a system's execution that contains at most a specified number of instances of a designated state transition or event.
Precedence Chain Property Pattern	Specification	Order	Used to specify relationships between chains (i.e., sequence of states or events), where an occurrence of the cause chain must be have been preceded by an occurrence of the effect chain. We say that an occurrence of the effect chain is enabled by an occurrence of the cause chain.

Precedence Property Pattern	Specification	Order	Used to describe relationships between a pair of states or events where the occurrence of the first is a necessary pre-condition for an occurrence of the second. We say that an occurrence of the second is enabled by an occurrence of the first.
Response Chain Property Pattern	Specification	Order	Used to specifies relationships between chains (i.e., sequence of states or events), where an occurrence of the cause chain must be followed by an occurrence of the effect chain. Also known as Follows and Leads-to.
Response Property Pattern	Specification	Order	Used to describe cause-effect relationships between a pair of states or events. An occurrence of the first, the cause, must be followed by an occurrence of the second, the effect. Also known as Follows and Leads-to
Universality Property Pattern	Specification	Occurrence	Used to specify states or events1 or states in a system that must always hold (that is, have a desired property) during execution. Also known as Henceforth and Always.
Adapter Pattern	Implementation	Environment	Used to convert the interface of a class/module into another expected interface. Adapter Pattern lets classes/modules work together easily that couldn't otherwise be easy because of incompatible interfaces. Also called the Wrapper.
BFM-Proxy Pair Pattern	Implementation	Environment	Used to facilitate the design of transactors like drivers and monitors for dual domain partitioned testbenches that can be used for both simulation and emulation, and across verification engines (or platforms) in general
BFM Notification Pattern	Implementation	Analysis	Used to provide effective and efficient notifications of protocol transaction occurrences, and any other interesting protocol and design events and conditions, for control and analysis in a dual domain partitioned testbench, from HDL to HVL domain
Component Configuration Pattern	Implementation	Environment	Used to create a coherent configuration structure for the component hierarchy from top to bottom. It promotes self-containment and data-hiding techniques in the configuration and creation of component hierarchy.
Dual Domain Hierarchy Pattern	Implementation	Environment	Used to facilitate the design of testbenches that can be used for both simulation and emulation, and across verification engines (or platforms) in general.
Environment Layering Pattern	Implementation	Environment	Used to provide consistent configuration and structure for vertical reuse of environments
Façade Pattern	Implementation	Environment	Provides a simple interface to a complex system, making it easier for the client or external world to use
Layering Sequence Pattern	Implementation	Stimulus	To be able to select and execute one of several sequences at will
Resource Sharing Pattern	Implementation	Environment	Used to share resources between objects without requiring detailed knowledge of the resource. Related resources share common access attributes thereby creating simple associations.
Strategy Pattern	Implementation	Stimulus	Used to define a set of behaviours/algorithms than can be interchanged seamlessly.
Utility Pattern	Implementation	Environment	Encapsulate small, useful functionality in a portable, easy-to-use object
Walking Pattern	Implementation	Analysis	Used to ensure toggling of signals to ensure connectivity and ensure higher toggle coverage is achieved.