Use the Sequence, Luke

Guidelines to Reach the Full Potential of UVM Sequences

Jeff Vance, Jeff Montesano, Mark Litterick

Verilab, Inc

Austin, TX, USA

www.verilab.com

ABSTRACT

Universal Verification Methodology (UVM) sequences are the standard way of controlling scenarios for design verification. Presently, no methodology exists to define an optimal collection of sequences for a UVC (UVM verification component) and/or a UVM testbench. A sub-optimal sequence library typically leads to overly complex testbenches that are hard to control, debug, and maintain. These problems pose risks to both project schedule and tapeout quality. This paper presents guidelines that solve these problems. We show how to isolate constraint solver steps through sequence application programming interface (API) layers that lead to rapid debug, improved sequence reuse, and easier control for test writers. Encapsulation guidelines demonstrate how to enhance sequences using configuration objects and package utility methods. Many of these guidelines come from extensive project experience dealing with complicated problems that occur on a large scale.

Table of Contents

Table of Figures

Table of Tables

1. Introduction

The primary problems in designing a UVM testbench are managing complexity, providing sufficient control of stimulus, and allowing reuse of implementations. Any ad-hoc collection of sequences used in a testbench is very unlikely to address all these problems. Such sequences are often poorly planned, incorrectly scoped, and inadequately constructed to meet verification requirements. Furthermore, inconsistency and inappropriate groupings of sequences make them fragile and ineffective, resulting in poor stimulus. This in turn negatively impacts project schedules.

1.1 Problem of Complexity

Management of testbench complexity becomes a major problem as many UVCs (UVM verification components) are combined into a single testbench environment. Although complexity is inevitable in most verification projects, it is imperative that we manage this complexity to keep projects progressing efficiently. Too much complexity makes it difficult to debug failures and to maintain the testbench to accommodate design changes. It is also typical that those responsible for maintaining tests and running regressions are not familiar with all the details of the design and every testbench UVC. Without properly encapsulating and managing these details, knowledge transfer and project communication become major bottlenecks in productivity and efficiency.

1.2 Problem of Insufficient Control

While we strive to reduce the complexity of a testbench, we must also ensure we have sufficient control of the scenarios generated. UVM sequences help encapsulate and manage all the stimulus options for interface UVCs. They also provide the means to control parallel and sequential stimulus patterns between multiple agents. While this helps manage complexity for a set of common well-defined scenarios, it risks crippling the capabilities of the testbench. Test writers may discover that they don't have adequate control over lower-level UVC options. It is not unusual to discover that it is impossible to randomize multiple UVCs in a single environment without generating unwanted illegal configurations[1]. UVC constraints are often managed independently and cannot be resolved together without writing directed tests. This impacts the time needed to write tests, makes it difficult to close coverage, and risks missing bugs in the design. While a simple block-level environment may get away with a default random stimulus for the drivers, this will not give sufficient control for sub-system or chip verification environments.

1.3 Problem of Insufficient Reuse

The inability to reuse sequences is a common problem in UVM testbenches. Sequence libraries[2] typically become single-project solutions that are coupled to a specific design architecture or configuration, and therefore must be rewritten for future projects. Lack of sequence reuse is also a problem within a single sequence library. It is not uncommon to find sequences cloned many times with minor adaptations. This not only increases the complexity and size of the sequence library, it becomes a major maintenance burden as duplicate code must be updated in several places.

---

[1] Each UVC can have a legal configuration according to their own constraints, but may still produce an illegal combination of configurations in terms of the overall system. Without constraints capturing the dependencies between UVCs, randomization can produce invalid test configurations without constraint solver failures.

[2] Throughout this paper we use the term "sequence library" to refer to a collection of related sequences maintained in a single file. This is not to be confused with the ***uvm_sequence_library*** class which is a specialized way of grouping and managing sets of sequences to execute.

1.4 Sequence API Solution

The problems of complexity, control, and reuse are all solved simultaneously by applying the sequence application programming interface (API) methodology described in this paper. By applying a consistent strategy to all sequences in a testbench, we show how the degree of complexity decreases while still meeting the same requirements. Section 2 describes the general strategy and defines requirements that an ideal sequence library will meet. Section 3 describes how to meet these requirements with implementation details, guidelines, and examples. Many of these guidelines come from extensive project experience dealing with complicated problems that occur on a larger scale. We show how combining these guidelines results in a more powerful testbench that is easier to use and eliminates the kinds of missteps that put projects at risk.

2. Sequence API Strategy

The overall strategy for a sequence library must ensure we simultaneously address the problems of complexity, control, and reuse. This section defines goals for how to solve these problems. These goals can be treated as requirements for any sequence library implementation. A typical UVM testbench consists of multiple tests, a single top environment, and multiple sub-environments for each UVC. This section defines the goals of a sequence library for each of these layers in a testbench hierarchy.

2.1 Test Sequences

We use the term *test sequence* to refer to the top sequence that is started by a selected `uvm_test` component. In UVM, the preferred approach is to encapsulate all constrained random stimulus in sequences. Therefore, the top sequence is responsible for defining the test stimulus while the `uvm_test` class is just a top container that instantiates the `uvm_env` class and its configuration object. This strategy keeps all stimulus generation inside sequences and out of the test component. Table 1 summarizes our primary goals for test sequences in a UVM testbench. Test sequences are the top entry point of the testbench hierarchy and are most frequently managed by testbench users[3]. Therefore, these should be the simplest to work with, by keeping all the complex details encapsulated in the sequence library. Controllability is ensured through a rich API of options provided to tests to control sequences. Since the complex details of the DUT and testbench architecture are not visible to the test, this allows them to be reused on derivative projects with different design configurations.

Table 1. Goals for Test Sequences.

Problem	Requirements
Complexity	Tests should be simple. They can be written and managed by people not familiar with testbench architecture details.
Control	Test writers are given an API that allows them to select system and interface-specific scenarios[4] to satisfy all verification goals.
Reuse	Test definitions are independent of project-specific configurations and testbench architecture details, allowing reuse between derivative projects.

[3] Throughout this paper we refer to ***users*** as those using the sequence library to achieve some goal. The user may sometimes be the same person who wrote the sequence library. However, it is best to think of an API in terms of third-party users.

[4] System scenarios allow users to control patterns of behavior between multiple interfaces. Interface-specific scenarios allow users to control low-level behavior on a specific interface.

2.2 Top Environment Sequences

The top environment in a UVM testbench (class extending `uvm_env`) typically contains one or more UVC environment instances[5]. The top environment sequences act as an interface between tests and the UVC sequences (test sequences *call* top environment sequences). Our goals for top environment sequences are summarized in Table 2. Test writers should not be required to directly call UVC sequences. Top environment sequences should resolve all test options in order to guarantee legal stimulus in terms of the overall system. The top environment sequences must also resolve relationships between multiple UVCs to achieve control of certain scenarios (i.e., top environment sequences are not hard-coded to a specific design architecture). By keeping the project-specific details outside the sequences, these sequences can be reused on other derivative projects. However, note that these sequences need resources to control things properly (e.g., configuration objects and register models). The top environment sequences should encapsulate all register model operations since hard coding these into test sequences would seriously impact portability and reuse, and it is inappropriate for lower interface UVCs to have a handle to a device-specific register model.

Table 2. Goals for top environment sequences.

Problem	Requirements
Complexity	• Guarantees legal stimulus in terms of the system, even if no sequence variables are constrained by users. • Option to solve constraints in isolated steps for easier debug. • Hides details of managing UVCs from tests.
Control	• Derives UVC control options from test options. • Provides tests a sufficient API to control required verification scenarios. • Manages relationships between UVCs to ensure proper control of stimulus. • Manage system resources such as register models which can't be managed by lower interface UVCs.
Reuse	• Changing UVCs has minimal impact on top environment and no impact on tests. • Sequences can be used on projects with different architectures and configuration options. • Eliminates constraint duplication by reusing sequences within the library.

2.3 Interface UVC Sequences

Table 3 summarizes the goals for interface UVC sequences. They must provide a comprehensive sequence API that covers all reasonable legal operations and anticipated protocol-specific error modes. Users of a UVC should never have to be concerned with low-level encoding schemes or standard data/frame packing and unpacking. A UVC's sequence library should handle all of these operations and keep the details hidden from the top environment sequences that call it. This keeps the top environment sequences simpler while often allowing them to be reusable with different UVCs. Any configuration-specific concerns for an interface UVC (e.g., channel number, port ID) should be

[5] Not all UVM testbenches require multiple environments encapsulated inside a top environment. Some may have all agents inside a single top environment. This paper focuses on the more complex problem of integrating multiple UVC environments into a single testbench.

managed through that UVC's configuration object accessor methods[6][2]. This allows sequences to automatically tune their behavior by constraining operation based on the current configuration settings.

Table 3. Goals for Interface UVC sequences.

Problem	Requirement
Complexity	• Guarantees legal stimulus in terms of the protocol, even if no sequence variables are constrained by users. • Allows the enabling of necessary error injection • Manages routine data handling (packing and unpacking), CRC calculations, and other configuration details specific to the interface.
Control	• Provides a sufficient API to top environment sequences to control all legal aspects of the protocol and appropriate error injection supported by UVC.
Reuse	• Implements only UVC-specific protocol (which may be generic between projects). • Shall not contain anything specific to a design. No architecture details, memory maps, or register models exist inside these.

3. Sequence Library Solutions

This section gives guidelines for implementing a complete UVM sequence library that meets all the goals defined in Section 2. We first give general encapsulation[1] guidelines that apply to all sequences throughout the testbench. We then highlight guidelines that apply more specifically to UVC and top environment sequences.

Figure 1 shows a typical UVM testbench architecture, where a single top environment contains multiple UVCs. Each UVC has one or more agents, a sequence library for interacting with those agents, and a set of utility functions for common calculations and configuration access. Additionally, a top testbench environment must itself provide a sequence library. The top environment can have its own configuration and utility functions. This environment contains sequences for managing the configuration of all the lower UVCs and ensuring they work together. However, most importantly, this environment provides an API to test writers that encapsulate all the complex details of managing the agents. Applying the guidelines in this section helps ensure that both UVC and environment sequences can integrate into different testbench environments and that users have sufficient control.

[6] Common practice in UVM is to provide configuration object handles to a sequence via the *p_sequencer* handle. We recommend providing accessor functions (e.g. get_[varname]()) that return fields needed for sequence operation.

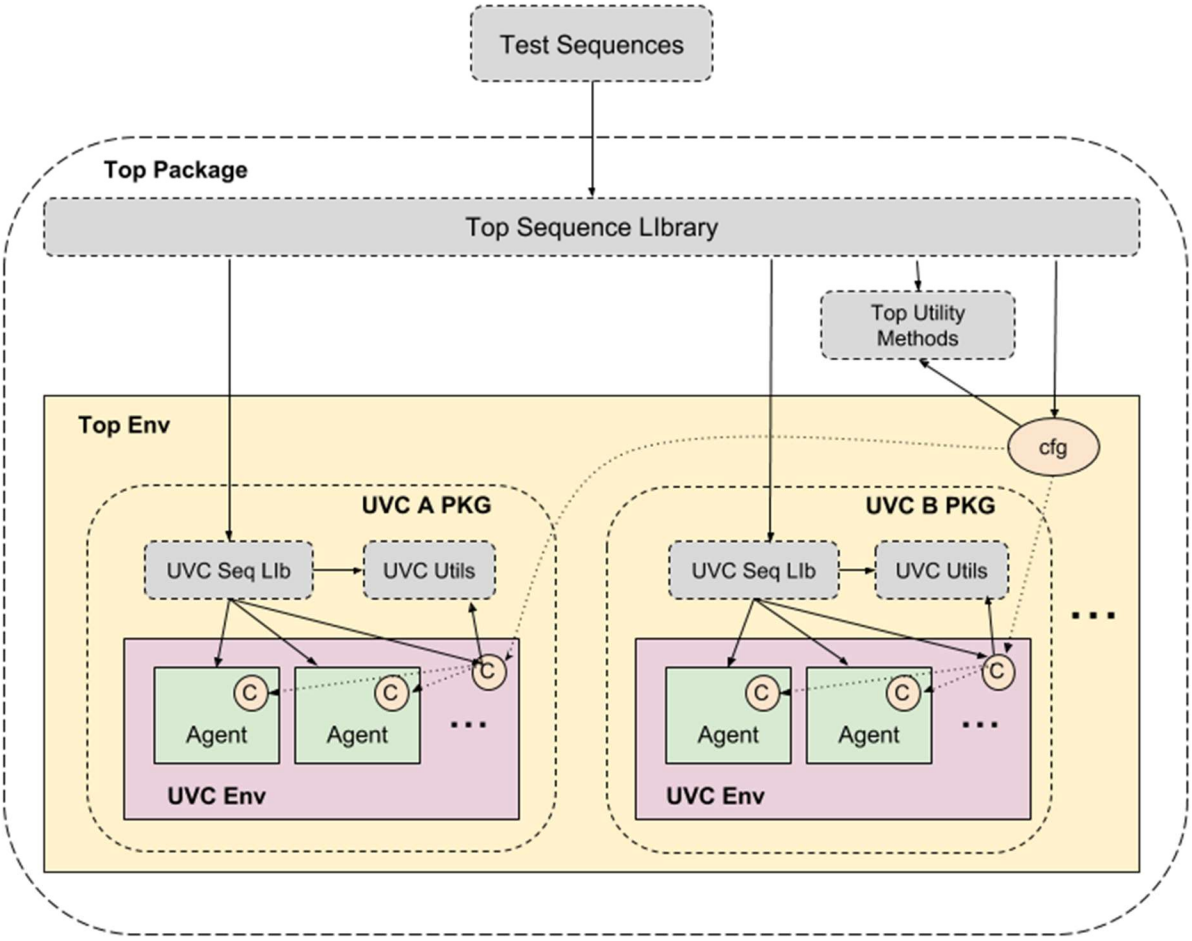

Figure 1. Typical UVM Environment Hierarchy

3.1 Encapsulation Overview

The guidelines presented in this paper aim to provide effective encapsulation for sequence users. This is achieved at the expense of more effort from sequence developers, but we believe this is easily justified. Each sequence provides a specific, targeted functionality for users, relieving them of the complex details, while giving them options for controlling the behavior.

3.1.1 Control Knobs and Derived Fields

Sequence classes can have a dedicated set of constrained random fields which the user can optionally constrain at execution using inline constraints. We typically refer to these random fields as *control knobs*[7] in UVM. By exposing control knobs to users, the sequence behavior can be tuned to reach more specific scenarios. We typically use control knobs in the following ways:

- Users specify no control knob values and rely on built-in constraints for random operation.
- Users specify all control knobs using inline constraints to provide a purely directed use-case.
- Users specify a subset of one or more control knobs using inline constraints to tune some random behavior of the sequence.

[7] Control knobs were referred to as "sequence variables" in section 2.

In addition to user control knobs, there are typically other class variables that are derived from control knobs. These derived fields may be non-random variables whose values are calculated from the random control knobs, or they may also be random variables where the choice of values is influenced by the control knobs. Derived fields are discussed further in Section 3.1.3.

3.1.2 Types of Encapsulation

Sequences encapsulate any of the following:

- The control knob fields and corresponding constraints (which may propagate to lower level sequences or be used to select appropriate alternate sequences)
- The lower sequence (or multiple sequences) chosen for execution
- The sequencer (or multiple sequencers) chosen to execute the lower sequence(s)

In all these cases, the sequences are encapsulating functionality (including system-wide scenario control of overlaps, parallelism, and data flows between interfaces). To ensure the purpose of each sequence is clearly understood, we recommend the following tip:

> ***Tip:*** *Use sequence names that are derived from the things they are encapsulating.*

The following are example sequence names for each type of encapsulation:

- **Control Knob Encapsulation:** `burst_write_seq` - indicates it is constraining transaction control knobs to perform a burst write[8]. Users only have control of fields needed to specify the kind of write transaction to be performed, which typically includes address and data, burst type, and size.
- **Sequence Encapsulation:** `initialize_dut_seq` - indicates it will perform initialization actions. However, it may call different initialization sequences based on the current DUT configuration. For example, this could call various combinations of power-up, wake-up, reset, configuration, and boot-up sequences. Some scenarios may require additional sequences that wait for the system to stabilize before taking certain actions.
- **Sequencer Encapsulation:** `drive_active_ch_seq` - indicates it is driving stimulus to only the currently active channels. Each channel may be controlled by a different agent. Users don't have to specify which sequencers are performing the actions if they don't want to. Without specifying a sequencer, a randomly selected active channel is selected by default[9].

3.1.3 Control Knob and Derived Field Encapsulation

When required, a sequence sets the control knobs of another sequence using inline constraints, typically with the `` `uvm_do_with() `` macros (without macros it is typically done using the `randomize() with` construct). Similar to minimizing the number of arguments in a function[10],

---

[8] Assuming a lower sequence has a READ/WRITE control knob that is randomized by default.

[9] The active channels may come from the DUT configuration and may change during simulation.

[10] Best coding practices advocated throughout programming disciplines stress the importance of reducing the number of arguments to functions in order to ensure loose coupling, promote reuse, and simplify functions[3].

we should minimize the number of control knobs available for a sequence. As described in Section 3.1.1, there are often other variables for fields derived from control knobs. In order help with this distinction, we recommend the following tips to hide the derived fields from the user:

> ***Tip:*** *If the absolute value of a derived field can be calculated from control knobs with no variance, declare this field inside the body() method instead of declaring it random and constraining it.*

This tip reduces the amount of work required by the constraint solver when randomizing the sequence and makes debug easier. Derived non-random fields do not require a constraint solver and can be easily calculated in methods.

> ***Tip:*** *If the resolved value of a derived field is influenced by control knobs, but can itself be chosen from a random set of values, declare it as a* ***protected*** *field and provide appropriate constraints.*

Declaring random derived fields as **protected** variables clearly captures the intent that these are derived random fields not to be directly constrained by users. It will also help enforce the encapsulation since VCS generates an error if an inline constraint is applied to these fields.

For example, in Figure 2 control knobs are provided for users to constrain inline if they like, but derived random variables are also present. These derived variables will benefit from the randomization engine, without being available to the user to constrain directly. The non-random derived fields in the body() method are calculated based on current randomized values for the control knobs, but the result is absolute.

```
class ramp_signal_seq extends base_seq;

  rand ramp_mode_t  ramp_mode;    // * control knob
  rand real         ramp_target;  // * control knob

  protected rand real ramp_step;  // derived random variable
  protected rand time ramp_delay; // derived random variable

  constraint ramp_target_c {
    ramp_target inside {[0.0:1.5]};
  }
  constraint ramp_step_c {
    ramp_step <= ramp_target;
  }
  constraint ramp_delay_c {
     ramp_delay inside {[calc_min_delay(ramp_mode):calc_max_delay(ramp_mode)]};
  }

  task body();
    time ramp_time;  // derived non-random variable
    time clk_period; // local variable
    clk_period = p_sequencer.cfg.get_sys_clk_period();
    ramp_time = calc_ramp_time(clk_period, ramp_step, ramp_target);
    `uvm_do_with(ramp_seq, {ramp_mode, ramp_target, ramp_step, ramp_delay, ...})
```

Figure 2. Derived field examples

Another option for hiding fixed control knobs is to pass these as explicit values in the inline constraints instead of using random variables with fixed constraints. Figure 3 shows a random write sequence allows that users to optionally control the address and data without having to know the current status fields that should be applied.

When applying inline constraints, be careful of scope rules of the variables referenced in the constraint. By default, variables in an inline constraint resolve to the scope of the sequence being randomized (e.g. *bus_item* in Figure 3), not the current sequence (e.g. *rand_write_seq* in Figure 3). For this reason, we recommend the following tip:

> ***Tip:*** *Always use the* ***local::*** *scope resolution keyword when passing variables in inline constraints.*

In many cases, the local scope resolution may not be necessary. However, we've found it helps avoid any chance for confusion if there happen to be common variable names in both sequences, as shown in the following example (e.g. *addr* and *data*).

```
class rand_write_seq extends uvm_sequence;
  //control knobs exposed to users
  rand bit[23:0] addr;
  rand bit[31:0] data;

  task body();
    x_bus_item  bus_item;
    bit[6:0] status_fields = p_sequencer.cfg.get_status_fields();

    `uvm_do_with(bus_item, {
                  addr==local::addr;  // local:: scope resolution
                  data==local::data;  // local:: scope resolution
                  rw==BUS_WRITE;      // fixed explicit value
                  status==local::status_fields; // tuned by config
                 })
  endtask
endclass
```

Figure 3. Use of local scope resolution

3.1.4 Sequences Layering Strategy

While UVM gives us the standard hierarchy of test, environment, and UVC sub-environment, this is not enough to achieve the amount of encapsulation we need to meet the sequence library goals defined in Section 2. Our strategy is to provide multiple layers of sequences, both within lower UVCs and within the top environment. Table 4 describes this layering approach. Each layer serves a particular purpose and focuses constraints on randomizing a particular aspect of the scenarios needed for tests. The following sections provide guidelines for implementing sequences according to these layers.

Table 4: Sequence Layering Strategy

Layer		Constraint Focus	Purpose
TEST		• Test scenario or flow	• Top entry point • Keep test count down by providing a few scenario options
TOP	User	• System use cases • System scenarios • Convenience routines	• Provides few or no knobs • Encapsulates knobs, lower-level sequences, and sequencers
	Lower	• System requirements • Architecture restrictions • Resolve dependencies between UVCs	• Provides minimal knobs • Provides base configurations of UVCs • Intermediate constraints to resolve knobs for UVC user sequences
UVC	User	• Bus / feature use cases • Bus / feature scenarios • Convenience routines	• Derives & propagates knobs • Provide UVC config object & util function access to top env • May encapsulate UVC sequencer
	Middle	• High-level protocol rules	• Provides many knobs • General purpose sequences
	Low	• Packing conventions	• Resolves final knobs for item • May pack fields for layered protocols
	Item	• Low-level protocol rules	• Provides knobs for almost everything • Calculates checksums and metadata

3.2 Hierarchy Guidelines

Building a hierarchy of sequences results in multiple levels of encapsulation, with each level focused on a particular aspect of randomization and control. The following guidelines ensure sequences are useful in any context as they are called throughout a sequence library.

3.2.1 Keep Stimulus Legal

Guideline 1: *Produce legal stimulus by default.*

This guideline ensures a sequence is always useful, even if the caller provides zero, one, or more inline constraints, as shown in the example code of Figure 4. Users require random legal values for fields they don't specify. This means that the sequence's built-in constraints must ensure legal operation for all randomization occurrences. These constraints also enforce legal rules on users. If a

higher layer constrains a knob to something illegal, the constraint solver will fail. This is the kind of failure we want to occur since it protects users from running invalid scenarios[11].

```
class ahb_fabric_master_write_seq extends ahb_fabric_base_vseq;
  …
  task body();

    `uvm_do(ahb_burst_seq) // no inline constraints, any legal burst
    `uvm_do_with(ahb_burst_seq, {direction == WRITE;}) // any legal write burst
    `uvm_do_with(ahb_burst_seq, { // specific write starting at 4000 (random data)
      direction == WRITE;
      start_addr == `h4000;
      length == 256;
    }
```

Figure 4. Sequence called with no inline constraints; legal stimulus assured by Guideline 1

Following this guideline also makes sequences much more valuable than tasks or functions in a constrained random testbench. Similar to method calls, sequences allow us to control functionality by passing arguments (passed as inline constraints). However, unlike tasks or functions, arguments not specified are randomized by default and guaranteed to be legal (when applying Guideline 1). This is a powerful way to manage complexity while retaining controllability. However, note that sequences with constraints are equally valuable for directed tests. Even if there is no desire to randomize anything, the constraints capture the legal rules that must be enforced and protect users from writing illegal directed test cases.

3.2.2 Layer Sequences by Composition or Inheritance

We can layer sequences either by composition or inheritance. The choice between these can have major consequences on the testbench usability and complexity.

> ***Tip:*** *Layer sequences by inheritance when you can add simple constraints without modifying the original body() task.*

Layering by inheritance works by extending an existing sequence class and declaring additional constraints to be solved concurrently with the base constraints (recall that a constraint block is a *member* of the class). A typical example is to extend a random sequence and apply more directed constraints to reach a specific scenario. We've found this works best in situations where you don't need to modify the body() and don't redefine the base constraints. However, as a sequence library grows, sequence inheritance can be harder to debug due to solving many constraints concurrently.

Many other situations call for layering sequences by composition. Rather than extending a sequence, we declare instances of lower sequences. This is ideal when we need to provide a variety of scenarios and data-flows by grouping several lower sequences. It is also ideal when a constrained random value needs to be distributed among several sequences.

---

[11] In some cases, we might want to verify illegal scenarios by providing additional dedicated illegal and error sequences. However, this is an exceptional case that should be handled outside the default API options. This makes it easier on users to keep track of what is legal and isolates the intent of the sequence.

> ***Tip:*** *Layer sequences by composition for complex constraint models, where multiple lower level sequences are being managed, or when the body must be redefined.*

The choice between sequence composition and inheritance will ultimately depend on the application. Some projects may be entirely composition or inheritance based, while others may require a mixture of both styles.

3.2.3 Isolate Randomization Between Layers

We can specify constraints for sequences in two ways: the class constraints declared within the sequence, and the inline constraints an upper sequence provides to a lower sequence. The choice between these can have major impact on the reuse and complexity of the sequence library. In particular, debugging constraint solver failures can be considerably harder if we don't choose wisely. Guideline 2 addresses this problem:

> ***Guideline 2:*** *Constrain control knobs with class constraints and pass the results with inline constraints.*

The primary difference between choosing class and inline constraints is controlling the order in which the constraints are solved. Combining class and inline constraints gives us a 2-step randomization process:

Step 1: The top sequence randomizes its class variables using class constraints.
Step 2: The top sequence executes the `body()` method and passes these variables as control knobs to the lower sequence (using "uvm_do_with" macros or "randomize with" calls).

Figure 5 shows two versions of a sequence. The first violates the guideline with expressions passed as inline constraints. The second version is revised so control knobs are only influenced by class constraints and passed inline to the lower sequence.

```
// first version
class ahb_write_burst_seq extends ahb_base_seq;
  …
  `uvm_do_with(ahb_seq,
               ahb_seq.hwrite == HWRITE_WRITE;
               ahb_seq.hburst inside {HBURST_SINGLE, HBURST_INCR}; //BAD!

// second version
class ahb_write_burst_seq extends ahb_base_seq;
  hburst_t  hburst;

  constraint c_hburst {
    hburst inside {HBURST_SINGLE, HBURST_INCR};
  }
  `uvm_do_with(ahb_seq,
               ahb_seq.hwrite == HWRITE_WRITE;
               ahb_seq.hburst == hburst; //GOOD!
```

Figure 5. Demonstration of inline constraint usage according to Guideline 2

Guideline 2 ensures that control knobs for a sequence are solved and finalized *before* they are passed to the lower sequence for randomization. If we instead pass range expressions for control knobs, the lower sequence has influence on the control knob (it must satisfy its class constraints and inline constraints concurrently). This can make debugging randomization problems much more complicated if there are many of these constraints. This effectively mixes the concerns of two different sequences into one constraint solver step[12].

There are two major benefits from debugging constraints that are isolated between sequence layers. First, if there is a constraint solver failure resolving control knobs, the solver will stop and allow us to debug *before* considering the constraints of the lower sequence[13]. A second benefit exists if there is no constraint solver failure, but users are getting unexpected results (i.e., test case is not reaching the goal or coverage is lower than expected). In a large hierarchy of sequences, users can review each stage of control knob randomization in isolation and easily pinpoint the problem. This is much harder to do if most of the constraints are being solved concurrently.

This guideline also promotes reuse since the lower sequence is completely decoupled from any inline expressions supplied by the calling sequence. The sequence behavior is more precisely defined and can be called by a variety of sequences with minimal risk of constraint solver failures.

3.2.4 Expose Control Knobs Appropriately

In order to preserve the separation of sequence layers, we need to be careful in choosing which control knobs to expose to callers of the sequence.

> ***Guideline 3:*** *Minimize the number of control knobs.*

Guideline 3 can be applied following the tips described in Section 3.1.3. Decide what minimal set of control knobs is required. All other fields should be *protected* fields or declared in the body() method.

```
class ahb_master_write_seq extends ahb_base_seq;
  rand int          slave_num; // ** user CAN specify which slave to write
  …
  virtual task body();
    // ** user CANNOT change sequence to be a read, (hwrite explicitly set)
    // ** user CANNOT change cacheableness (hprot3) (config defines it)
    ahb_seq_item req;
    `uvm_do_with(req,{
                      req.hwrite == HWRITE_WRITE;
                      req.hprot3 == p_sequencer.cfg.get_hprot3();
                      req.haddr[31:24] == slave_num;})
```

Figure 6. Sequence writing exposes control knob for slave number, not for other fields

[12] Some complex constraint models may require concurrent solving of disparate constraints, but these are rarely needed for most situations.

[13] This benefit is especially true for sequence libraries that manage very large sets of control knobs that influence many class constraints.

This ensures callers cannot deviate from the intent of the sequence. A sequence designed to enforce a specific kind of transaction (Figure 6 shows a sequence for doing writes) or scenario cannot guarantee legal stimulus if users constrain things in unexpected ways.

> ***Tip:*** *Make sure sequences lower in the hierarchy have more control knobs compared to higher sequences.*

Consistently applying these hierarchy guidelines naturally reduces the number of control knobs that are exposed as we move up the sequence hierarchy. Each sequence encapsulates how it constrains a lower set of control knobs to reach a specific scenario. The lowest sequences and sequence items in a hierarchy should have the most control knobs, giving users control of almost anything (within the legal constraints). The highest user sequences will be encapsulating the hierarchy of options that reach a more specific use-case, therefore they will have a much smaller number of control knobs. This also promotes better reuse of sequences since lower sequences can be applied in a larger variety of situations.

3.2.5 Allow Sequences to be Extendable

Although we try our best to provide a comprehensive sequence library that anticipates all reasonable use-cases, we have to accept that some users may want to adapt our sequences to provide additional, or indeed reduced, functionality. This is a common theme in all aspects of UVM testbench architectures and sequences are no different. In addition to registering sequence classes with the factory using *uvm_object_utils*, we also need to allow the sequences to be extendable for modified functionality. With this in mind we would recommend that dedicated constraint blocks for individual control knobs are provided rather than putting all variables into the same constraint block.

> ***Guideline 4:*** *Constrain each control knob in a dedicated constraint block.*

This means user's that need to modify the behavior of a sequence can extend the class, redefine a single constraint for the required control knob only (instead of having to recode all the other constraints as well), and then use the new sequence directly or provide a factory override to ensure all occurrences of the original sequence are replaced by the new code. An example of this is shown in Figure 7.

```
// first version - user has to repeat all constraint lines to modify one
class ahb_burst_seq extends ahb_base_seq;
  // one constraint for many fields
  constraint all_burst_c {
    saddr inside ([MIN_ADDR_C:MAX_ADDR_C]);
    length <= MAX_LENGTH_C;
    hburst inside {HBURST_SINGLE, HBURST_INCR, ...};
    ... // etc.
  }

class my_burst_seq extends ahb_burst_seq;
  // redefine all constraint lines, just to knock out HBURST_INCR
  constraint all_burst_c {
    saddr inside ([MIN_ADDR_C:MAX_ADDR_C]);
    length <= MAX_LENGTH_C;
    hburst inside {HBURST_SINGLE, ...}; // changed
    ... // etc.
  }

// second version - user can redefine a single constraint
class ahb_burst_seq extends ahb_base_seq;
  // dedicated constraints for each field
  constraint saddr_c {saddr inside ([MIN_ADDR_C:MAX_ADDR_C]);}
  constraint length_c {length <= MAX_LENGTH_C;}
  constraint hburst_c {hburst inside {HBURST_SINGLE, HBURST_INCR, ...};}
  ... // etc.

class my_burst_seq extends ahb_burst_seq;
  // redefine only one constraint, to knock out HBURST_INCR
  constraint hburst_c {hburst inside {HBURST_SINGLE, ...};}
```

Figure 7. Example of providing dedicated constraint blocks per control knob

3.2.6 Be Careful with Soft Constraints

Soft constraints must be treated with care since they can produce unexpected results for users. The primary problem is how soft constraints can silently restrict the random solution space for users. Soft constraints are only applied if there is no conflict with other constraints. If a user provides inline constraints for control knobs that also have a soft constraint, one of two things will happen: If there is an overlapping solution between the constraints, both constraints will apply. If there is no overlapping solution between them, the soft constraint will be disabled. In either case, there is no indication to the caller what is happening. Furthermore, if there are several soft constraints that can't be all satisfied, there is no deterministic rule for which soft constraints are ignored.

Soft constraints that supply a default value (or smaller range of values) will potentially continue to be applied and ignore the inline constraints supplied by the caller (see Figure 8). If there are many of these constraints, callers may be unaware that the control knob constraints they supply are being silently ignored.

```
class ahb_master_write_seq extends ahb_base_seq; // ** UVC-level sequence
  rand int          slave_num;
  constraint c_slave_num {soft slave_num == 0;}
  ...

class __ahb_fabric_master_write_seq extends ...; //** Test level sequence
  task body();
`uvm_do_on_with(ahb_master_write_seqs[b],
                  p_sequencer.p_env.mst_ahb_agents[b].sequencer,
                  {ahb_master_write_seqs[b].slave_num inside {[0:3]};})
```

Figure 8. Lower-level soft constraint silently undermines test writer's intention

There are a couple of common scenarios where soft constraints are helpful. The first example is when we want to provide a Boolean (bit) flag to control a mode of operation where the default is clear, and we never really randomize the value to a range, but just select the mode or not - for example an error enable flag. By default, we might want only normal scenarios, but we want to give callers the option to enable errors with a control knob. A soft constraint to disable errors is helpful here. Users get error-free behavior by default, but can easily enable them with an inline constraint without having to disable the constraint.

Another common example is where we have a random configuration sequence which contains no soft constraints but a full set of legalization constraints. In order to isolate verification concerns we often extend this sequence to generate a fixed configuration sequence (discussed also in Section 3.3). In this case the derived sequence can set all the fixed values using soft constraints since they will be a subset of the legal constraints in the random base configuration sequence. This allows users to override a single field when they call the fixed sequence and thereby provides a whole suite of tunable but nearly fixed configuration behavior if required (otherwise call the fixed sequence with no inline constraints to have no variance).

Guideline 5: *Use soft constraints carefully and sparingly.*

In conclusion, we recommend not using soft constraints if they aren't truly required. For the cases they are required, use them with caution.

3.2.7 Use Enumerated Types

Guideline 6: *Use enumerated types for control knobs.*

The control knobs of a sequence should be easy to use. By using enumerated types for control knobs, users of the sequence are:

- guaranteed to only use legal values for the knobs
- less likely to make encoding-related errors (e.g. specify HSIZE_8 when the intention of their test was to use HSIZE_16)
- more likely to write tests that are easily readable by others

In Figure 9 we see a package that defines enumerated types for the AHB protocol, and later see a test which calls that sequence.

```
package ahb_names_pkg;
  typedef enum bit [2:0] { HSIZE_8    = 3'b000,
                           HSIZE_16   = 3'b001,
                           HSIZE_32   = 3'b010,
                           HSIZE_64   = 3'b011,
                           HSIZE_128  = 3'b100,
                           HSIZE_256  = 3'b101,
                           HSIZE_512  = 3'b110,
                           HSIZE_1024 = 3'b111
                         } hsize_t;

class ahb_master_seq extends ahb_base_seq;
  hsize_t hsize; // ** control knob is an enumerated type!
  ...
  task body();
    ahb_seq_item req;
    `uvm_do_with(req, {
                      req.hsize == local::hsize;

class my_test_seq extends ahb_fabric_base_vseq;
  ...
   task body();
     ahb_master_seq master_seq;
     `uvm_do_with(master_seq, {
                  master_seq.hsize == HSIZE_256) //** Test is more readable!
```

Figure 9. Control knob for AHB hsize using an enumerated type

3.3 Top Environment Sequence Guidelines

There are several guidelines that specifically apply to top environment sequences. These guidelines help achieve the goals defined in Section 2.2.

3.3.1 Keep Tests Generic

Section 2.2 describes the goal for top environment sequences. The primary goal is to create a sequence layer between test sequences and the UVC sequences. The sequences at this level guarantee legal stimulus in terms of the overall system. However, the other benefit is this allows us to make test sequences more generic and independent of project-specific testbench architectures. Therefore Guideline 7 helps achieve both reuse and complexity goals for environment sequences.

Guideline 7: *Make tests independent of testbench architecture.*

Designs are often reused from one project to the next, whether it be to add/remove functionality, or to instantiate the DUT within another design. In these situations, reusing test cases can be an important time saver. As described in section 3.1.2, we achieve this by encapsulating the specific sequencer selected for running the lower sequence. Test sequences only have to call a mid-level wrapper sequence without any knowledge of which UVC sequencers will be used. The mid-level sequence then determines which UVC sequencer should run the stimulus. In Figure 10 the mid-level sequence is called ahb_fabric_master_write_seq. The test sequence is ahb_fabric_master_write_seq.

```
class ahb_fabric_master_write_seq extends ...; // Mid-level sequence
  ...
  task body();
    ahb_master_write_seq ahb_master_write_seqs[string]; //UVC-level sequence
    ...
      `uvm_do_on_with(ahb_master_write_seqs[b],
                      p_sequencer.p_env.mst_ahb_agents[b].sequencer,
    ...
  endtask: body

class ahb_fabric_master_write_seq extends ...; // Test API-level sequence

  task body();
    ahb_fabric_master_write_seq master_write_seq;
    `uvm_do(master_write_seq)
  endtask: body
```

Figure 10. Sequence layering achieves tests that are independent of testbench architecture

With this approach, if the testbench architecture changes, all tests remain unchanged. The only thing that will change is the mid-level sequences. In situations where a DUT has hundreds or thousands of tests, the benefits of this approach are significant.

3.3.2 Provide Random and Directed Sequences

While the big goal of a set of constrained random sequences is to uncover bugs in unexpected places, a less lofty but still important goal is to verify that the DUT operates correctly in any given mode. To achieve this, randomization needs to be restricted to those aspects which affect that particular mode. This can be achieved by having both random and directed versions of a given sequence, in which the directed version targets a specific mode of operation.

Tip: *Provide users with both random and directed versions of a common sequence[14]. The directed version further constrains the random version with fixed default values.*

3.3.3 Use Descriptor Objects

Sometimes sequence constraints can get extremely complicated which results in code that is hard to read, prone to errors and difficult for the constraint solver to resolve. For example, when we have arrays of complex features with many fields to configure and the results of randomization must be distributed to many sequences and registers. In cases like this we have found it useful to create an additional container class for the sole purpose of encapsulating these complex constraint relationships and enforcing an extra layer into the constraint resolution process. These descriptor classes contain all the necessary fields and constraints to describe one complex feature independently of how we will use the fields.

Guideline 8: *Use descriptor objects to encapsulate complex constraint sets.*

[14] A directed sequence can either extend the random sequence with additional constraints, or it can call the random sequence with fixed inline constraints.

The sequence can easily contain arrays of these descriptors and after randomization can use the descriptor content to affect other sequence behavior. These descriptor objects do not typically represent an existing transaction class in the testbench architecture and are not an executable sequence either, rather they are an artificial grouping of related fields with the purpose of improving stimulus effectiveness. One example of such a descriptor and its usage is given in Figure 11.

```
class shape_descriptor extends uvm_object;
  rand shape_kind_t shape; // TRIANGLE, SAWTOOTH
  rand shape_polarity_t polarity; // NORMAL, INVERTED
  rand int ramp_step_up, ramp_samples_up, .... // 20 other fields
  ...
  constraint {...} // comprehensive set of constraints for all fields
  ...

class config_shape_Seq extends base_seq;
  rand int num_shapes;            // number of shapes (1-8)
  rand shape_descriptor shapes[]; // array of shapes

  constraint size_c { // simple (only) constraint
    num_shapes inside {[1:8]};
    shapes.size() == num_shapes;
  }

  task body();
    foreach (shapes[i]) begin
      // use the descriptor contents in sequences, register model, etc.
      regmodel.CFG0.RMPSTEP3.set(shapes[3].ramp_step_up);
      regmodel.CFG0.update(status);
      `uvm_do_with(start_pll_seq, {
        polarity == shapes[cfg.get_active_shape()].polarity; ... })
```

Figure 11. Descriptor example

3.3.4 Enumerate Scenario Control Knobs

A final tip to consider for top environment sequences is to apply Guideline 6 to define enumerated types for system scenarios.

> ***Tip:*** *Use enumerated types for control knobs to encapsulate system scenarios.*

For example, data-flow patterns between different interfaces can often be characterized as sequential (interfaces not active at the same time) or parallel (interfaces active at the same time). By encapsulating these scenarios in enumerated types, sequences can declare the data-flow pattern as a control knob. These control knobs can then influence the control knobs fed to each interface UVC. The data-flow control knobs can be randomized and users can optionally constrain them as needed for each test case.

3.4 Configuration and Reuse Guidelines

In UVM testbenches constrained-random stimulus is controlled by both sequences and configuration object fields. It is important that the configuration objects are also well encapsulated and provide a comprehensive set of fields, constraints and associated accessor methods (to get current values) as well as utility functions for calculating derived values.

Guideline 9: *Use configuration objects and accessor methods to adapt to project-specific configurations.*

These configuration objects are part of the hierarchy and support each layer or UVC independently. They contain static project-specific configuration as well as dynamic UVC configuration that is related to the current DUT configuration register content and testbench specific modes of operation. This information should not be hard-coded inside sequences since this limits reuse.

In order to make the configuration objects available to the sequences, we would normally have a handle to the corresponding configuration object in the associated sequencer. All sequences should automatically tune their operation to current configuration by accessing helper methods in the configuration object that can be accessed via the p_sequencer handle (e.g. *p_sequencer.cfg.field* or *p_sequencer.cfg.get_field* as shown in several places throughout the paper).

Guideline 10: *Use utility methods to support self-tuning sequences.*

A sequence library often requires the same calculation to be done multiple times and in different places. This functionality is best placed inside a SystemVerilog function, and perform common activities such as the following:

- calculate the capabilities of a UVC based on current configuration(i.e., max throughput, rate, max cycles for a transaction)
- calculate a data offset based on an address (see Figure 12)
- calculate a derived value using a formula
- convert random integers to real or time values based on configured precision
- calculate delays for transaction stimulus
- calculate timeouts for waiting and checking for outputs

A good practice is to define these functions outside of a class and import them into the UVC package[15]. This allows the package-scope methods to be called from any sequence, as well as other UVC components (e.g. drivers, monitors, etc.).

[15] Unlike class methods, package-scope methods are *static* by default. You may want to declare all of these methods as *automatic* to avoid unexpected behavior.

```
(file ahb_common.sv)
function automatic int calc_data_offset_from_address(ADDR_t addr);
  return(addr / DATA_WORD_SIZE) % DATA_WORDS_PER_ADDR);
endfunction: calc_data_offset_from_address...
(end file ahb_common.sv)

package ahb_pkg; // ** ahb_agent UVC package

  `include "ahb_common.sv"
  `include "ahb_vip_cfg.sv"
  `include "ahb_sequencer.sv"
  `include "ahb_driver.sv"
  `include "ahb_monitor.sv"
  `include "ahb_agent.sv"

  `include "ahb_seq_item.sv"
  `include "ahb_base_seq.sv"
  `include "ahb_master_rand_seq.sv"
  `include "ahb_master_write_seq.sv"
  `include "ahb_slave_mem_seq.sv"
  `include "ahb_slave_fifo_seq.sv"
endpackage

`endif
```

Figure 12. Utility methods defined outside a class, imported by a sequence package

A similar type of utility function exists for querying configuration objects. These functions should live inside the configuration object class. Some examples of these functions are to:

- get the current configuration of the DUT or UVC (see Figure 13)
- get relationships between current configuration settings of the DUT or UVC
- get derived values based on the current configuration of the DUT or UVC

```
class ahb_cfg extends uvm_object;
  rand int slv_fifo_depth;
  ...
  constraint {
    slv_fifo_depth inside {[1:15]};
  };

  function int get_fifo_depth();
    return(this.slv_fifo_depth);
  endfunction
  ...
```

Figure 13. Configuration object utility functions for getting values related to UVC configuration

Configuration object accessor methods also allow powerful options for test sequences to control scenarios across multiple UVCs. Very often we want constrained-random scenarios between multiple

interfaces. However, default random configurations on each UVC may give undesired results[16]. The configuration accessor methods help solve this problem. Top environment sequences can coordinate randomization of control knobs between UVC's. The typical process is to first randomize one UVC configuration, extract some of its configuration fields using a `cfg.get()` call, then use these values to randomize derived control knobs to pass to other UVC sequences.

3.5 Sequence Library Management

A sequence library typically exists as multiple sequence classes within a single file. When the number of sequences gets large, it can be difficult for users of the library to navigate the file. To address this situation, the authors recommend listing all sequences as SystemVerilog typedef statements at the top of the sequence library file (see Figure 14). Beside each typedef statement is a short description of the sequence.

```
(file ahb_fabric_seq_lib.sv)

// list of all sequences in this file
typedef class ahb_fabric_clk_rst_seq;    // performs reset and starts clocks
typedef class ahb_fabric_slave_fifo_seq;//slave sequence for storing in FIFO
typedef class ahb_fabric_master_write_seq; // master write sequence
...
class ahb_fabric_clk_rst_seq extends ahb_fabric_base_vseq;
...
class ahb_fabric_slave_fifo_seq extends ahb_fabric_base_vseq;
...
class ahb_fabric_master_write_seq extends ahb_fabric_base_vseq;
```

Figure 14. Using typedef statements for readability and functionality

This approach offers the additional benefit that by listing the sequences as typedefs, they can then be implemented and used in any order inside the file, and compilation will pass regardless of any dependencies between them.

Tip: *Use* `typedef` *header at the top of sequence library files.*

When debugging a complex environment with many sequences running in parallel on many agents, it can be very difficult to debug testbench issues arising from things like synchronization, ordering, and prioritization. A good technique to help with this is to place `` `uvm_info `` messages announcing the beginning and ending of every sequence, and to have them at the `UVM_HIGH` (or `UVM_FULL`) messaging levels. These can be added as the very first, and very last statements of the base sequence pre_start and post_start methods respectively. In addition, the entire sequence object should be printed out at UVM_HIGH verbosity to enable inspection and debug of the control knobs and deriver fields.

Tip: *Use messaging to announce the beginning, content and ending of a sequence.*

[16] Default randomization may not accurately target the scenarios needed by a test. It might also produce invalid combinations of UVC configurations in terms of the system.

Note that the *pre/post_do* methods should not be used since these are only called when a sequence is executed using the *uvm_do* macros, and the *pre/post_body* methods should also not be used since these are only called when the sequence is explicitly started on a sequencer (i.e. they do not work when the *uvm_do* macros are used). An example of such sequence messaging is shown in Figure 15.

```
class ahb_base_seq extends uvm_sequence_item;
  ...
  task pre_start();
    `uvm_info(get_type_name(), "Sequence starting...", UVM_LOW)
    `uvm_info(get_type_name(), $sformatf("Sequence content:\n%s",
      this.sprint()), UVM_HIGH)
  endtask

  task post_start();
    `uvm_info(get_type_name(), "Sequence completed", UVM_HIGH)
  endtask
```

Figure 15. Example of messaging added to base seq and inherited by all derived sequences

4. Conclusions

UVM sequences, while appearing routine on the surface, are a topic worthy of much thought and effort. Sequences permeate every layer of a verification environment, and have significant influence on how powerful, reusable, and readable the testbench will be for its users. In this paper, we have explored several important aspects of sequences that verification engineers need to keep in mind when dealing with them, and have formed that knowledge into a list of guidelines (Table 5).

Table 5. Summary of guidelines

Guideline	
1	*Produce legal stimulus by default.*
2	*Constrain control knobs with class constraints and pass the results with inline constraints.*
3	*Minimize the number of control knobs.*
4	*Constrain each control knob in a dedicated constraint block.*
5	*Use soft constraints carefully and sparingly.*
6	*Use enumerated types for control knobs.*
7	*Make tests independent of testbench architecture.*
8	*Use descriptor objects to encapsulate complex constraint sets.*
9	*Use configuration objects and accessor methods to adapt to project-specific configurations.*
10	*Use utility methods to support self-tuning sequences.*

The content of this paper comes from lessons learned across many UVM projects. Presently, the need to use UVM sequences in an optimal fashion is often underappreciated amongst verification engineers, who must face the consequences of complex testbenches under tight schedules. We hope that through this paper, the situation will begin to change for the better.

5. References

[1] https://en.wikipedia.org/wiki/Encapsulation_(computer_programming)
[2] https://en.wikipedia.org/wiki/Mutator_method
[3] Martin, Robert C. "Clean Code: A Handbook of Software Craftsmanship", Prentice Hall, 2008.
[4] Accelera, "Standard UVM Class Reference, v1.2" http://www.accellera.org/downloads/standards/uvm