Shotgun *E*

An Eight-Step Approach to Experience Random Verification

Peet James, Principal Engineer
peetj@qualis.com

Chris Macionski, Senior Engineer
chrism@qualis.com

Qualis Design Corporation
www.qualis.com

Abstract.

The power of the *e* language is that it brings a new dynamic to the verification space. Most groups using *e* for the first time fail to tap fully into this power. Typically, the reason is because the verification team does not know how to move from the familiar directed testcase driven mentality, to a new random generation with constrained testcases approach.

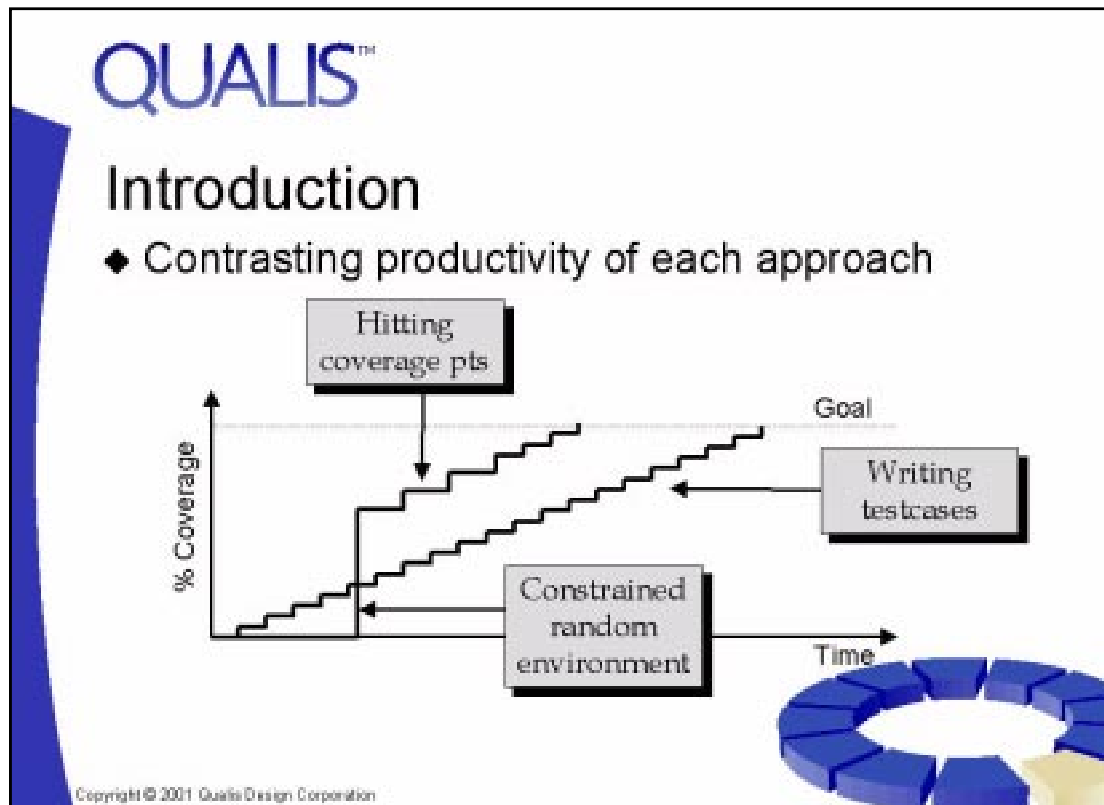
This paper outlines an eight-step methodology that lets a verification team experience the power of the random constrained approach and its ability to close the verification intent feedback loop.

1.0 Introduction

Most groups using *e* for the first time fail to fully utilize the power of the language. Their code tangles like spaghetti and they get backed into corners. They just keep using the standard Layer 1 bus functional model (BFM) approach with standard directed testcases. This is of only a small benefit over using an HDL Verification Infrastructure (VI). To really extract the value add of *e,* you have to use the random/constraint approach combined with functional coverage that gives tangible feedback data. The ability to use a directed random approach lets you cover a large amount of functionality with a handful of testcases. The functional coverage feedback lets a verification team close the verification intent feedback loop.

An analogy to this approach is to fire a shotgun at a target -- a large percent of the verification space is covered with few strategic shots (or testbases). After looking at the target (or coverage reports), you would target specific (uncovered) areas with a high powered sniper rifle (directed testcases). The old-school, directed verification strategy using HDL is like using a pea shooter to individually target the gnats (bugs) within the verification target area. The new-school way, if done correctly, will need only a few sniper shots fired near the end of the verification project.

Figure 1. Constrained random approach improves coverage productivity.



So how does a verification team switch weapons? They have these shinny new shotguns that management bought for them, but they have a white-knuckling grip on their trusty old pea shooter. Engineers need to overcome their fears. Another new language (gun) to learn? Putting random data into their design? The verification team needs a big leap-of-faith to let themselves experience the new approach. The power of the shotgun must be experienced first-hand for the team to fully “Get it” and let go of their pea shooters.

2.0 *E* Verification Infrastructure

The building of an *e* verification infrastructure (*e*VI) is like building an automatic control panel with knobs and gauges that automagically creates and runs testcases for you. The control panel has built-in mechanisms (shotguns) that run testcases over and over again while tweaking various knobs in interesting random ways. The knobs are set (shotguns are aimed) via the reading of gauges that convey real, tangible feedback information. The goal of this verification control panel is to be able to run by itself for the most part (like a robotic sentry fires at will). The verification team members just need to monitor the control panel from time to time, aiming it into corner cases that have yet to be hit.

How does one set up this control panel? What are the gauges? What are the knobs? What is the feedback mechanism? What does the code look like? A high-level, practical eight-step methodology is presented that evolved out of several *e* projects and the generation of both beginner and advanced *e* class material. Each step will be explained with practical how-to tips.

In addition, a simple, detailed example with code snippets is presented in the Appendix. Each one of these steps could warrant a detailed paper. This paper gives a 30,000-foot, big-picture overview of the eight-step approach. The eight steps are:

- Feature extraction
- Testbase extraction
- Per-interface random generation
- Scenarios
- Error injection
- Scoreboards
- Coverage feedback
- Constraining testcases

The first two steps create a global list of features to mark off when done, and a list of testbases that intelligently group and aim the verification work into strategic enviornments.

The next six steps are performed on a per-testbase basis. In most cases, the eight steps do not flow one right after another, but proceed after some code writing, code running and code debugging at each step along the way. Also, the order of these six steps may be different. For instance, injecting errors may occur after scoreboarding.

3.0 Step 1: Feature Extraction

Description

Feature extraction is a basic part of any modern conformance verification plan (CVP). To obtain buy-in from the team, gather all knowledgeable parties (ASIC architect engineers, RTL engineers, verification engineers). Brainstorm up front on the verification approach and on a list of practical features for verifying functionally.

Plan on verifying functionality -- don't get stuck thinking about verifying synthesis. To facilitate extraction, the feature lists are typically subdivided into three or four feature groups. Each feature group is extracted separately.

Key additional resource

For information on the process of generating a CVP and doing feature extraction, refer to a paper that was given at the 2000 Boston Synopsys Users Group (SNUG) entitled, *The Five-Day Verification Plan* by Peet James. For convenience, this paper has been added to the ClubV proceedings.

The resulting feature list is the same regardless of doing an HDL-based VI or an *e*-based VI. The feature extraction and the subsequent list that is created is an extremely useful tool to guide and gauge the overall verification work.

- First, the feature list serves as a launching point to describe the necessary testbases (next step) as they are grouped into meaningful subcategories.
- Second, from this list of features, a shopping list of necessary verification components can be extracted. A list of BFMs, monitors, generators, scoreboards, and coverage metrics can be drawn up.

  Extracting the features makes the amount of work and schedule more clear. How many people will be needed for the verification team. (On several occasions, this is what sold a company on having a separate verification team.) With *x* number of people, it will take about *y* long. What about *x*+1? Priorities, assignments, and dependencies follow naturally. What are the must-haves? What are the nice-to-haves? Which tests do we need to check first?
- Finally, the feature list acts as a checklist. As testbases are run repeatedly and driven into corner cases, the features that are covered can be crossed off the list until a tangible threshold of conformance is achieved.

Checking items off is a metric that can show progress and how much is left to do. However, it does not tell you if or when you are done. This metric tells you only that you are confident that this part of the design is checked and that there is more to do.

3.1 How To Extract Features

You need to adjust the basic CVP generation approach as described in the SNUG paper mentioned above (which is non-HVL- or *e*-specific) with these *e* slants:

- Brainstorm on overall approach to identify Layer 1 bus functional models (BFMs) to start with. Go on to other layers if time allows, but do not get bogged down.
- Brainstorm on the supporting VI. Stuff like directory structure and scripts to run *e* with your HDL simulator.

  Parking lot any sticky issues -- many times they flush themselves out as the VI is built. Also, remember to use all of your resources. Hand-off non-design work (writing scripts, directory structure, mailing lists, etc.) to network admin / MIS types.
- Identify two to four feature groups (such as basic sanity, intentional, or stress), but use a more random slant to them (such as basic random, random day-in-the-life, constrained stress corner cases).

  These feature groups are just a natural way to slice up the overall verification work. The basic ones still might be directed at first, but the random approach can be added as a way to let the verification team experiment and experience the new approach.

  When you start listing features, you need also to start thinking about coverage. For just about every feature, you can detail how to watch and see if this feature was covered. Start thinking randomization, scenarios, coverage and feedback right up front and put it in the CVP.

The actual feature extraction is done on each feature group using something we call the Yellow-Sticky method. The method is detailed in the SNUG paper, but basically you gather a group of designers and verification people (anyone who knows the architecture, control, protocols, and dataflow of the chip or system), and have them brainstorm a list of things to verify.

On a per-feature group basis (such as basic for short, start-up type features; day-in-the-life for normal things that the chip will be doing 90% of the time; stress for trying to break the design), just brainstorm, putting one idea on each yellow sticky. Each sticky can have a short blurb about the feature, followed by any thoughts or constraints on randomness, how to check it, and/or coverage ideas. Massive detail is not needed at this point. Just capture a mark on the wall that points in the right direction. Every item in the CVP should have a corresponding sticky.

Unlike in a pure HDL environment, the features are not grouped into categories and then individual directed testcases. Instead, the features are put into groups that will lead to testbases (which is the next step).

4.0 Step 2: Testbase Extraction

Description

So far the development of the CVP and corresponding VI has been much the same as with an HDL-only environment, but with some minor random approach ideas included. The testbase extraction is where we deviate from grouping features into categories that will lend themselves to further subdividing them into individual directed testcases.

Instead, we group the features so that they lead into common testbases. What is a testbase? A testbase is just a name given to an *e*VI that will handle a certain flow of data, or a certain group of similar testcases. It is a harness (also called testbench) set up with various verification components that will be needed to test a particular group of features.

E's ability to randomize data will give the harness the power to check many different areas. In HDL-only VIs you typically want to minimize the number of harnesses to one or two because they are painful to create, update and maintain. In an *e*VI, you will want to keep the number of testbases down to three or four, but it is not as crucial because the *e* language allows (via the when statement and the ability to extend) for configurable harness setups that can be easily constrained into testbases.

In some cases (such as in the Appendix example), due to the simplicity of the design or the constrainability of *e*, a single testbase is sufficient. This single testbase is configurable enough to test all the features. This situation is ideal. Typically, in practice several testbases are needed to test all the features.

Examples of testbases, might be a CPU-only mode that only folds in the CPU BFMs. If the CPU instruction is a read/write to external memory, the external memory interface BFM would be loaded into the harness.

Another way to define testbases is by which interfaces (and their associated BFMs, GEN, MON, etc.) will be included.

4.1 How To Extract Testbases

Testbase extraction flows out of the Yellow-Sticky method done in the feature extraction step.

- Have all the engineers put their yellow stickies up on a whiteboard or a flip chart.
- Have everyone start grouping the features together. Natural categories will form.

  Typically, the categories will be interface- or block-driven CPU, mem, etc. or mode driven (normal, multicast, etc.). The mode approach is likely where we want to go to get testbases, because we are verifying an entire system, entire chip or at least multiple modules. The interface categories are fine if you are verifying a single module or block, but will not work for larger combinations. The appropriate subdivision of categories is crucial here. You need to guide the knowledgeable parties toward it.

  While guiding toward mode-driven grouping, it is appropriate at this point to give a short talk on where you are going with the testbase. Something like, “A testbase is one configuration of the harness that will be set up to facilitate the verification of a category (or categories) of like features. Instead of the old-school way of grouping features into directed testcases, a testbase is created to handle verifying similar categories of features.”

  This is where we start the leap of faith from the traditional HDL approach -- sometimes engineers don't want to leap -- they need to be pushed. Take away their pea shooter, distribute the shotguns.
- Next, have the team group these categories into a testbases. Come up with some possible testbase configurations together by drawing on the mode-style categories just created and then run with them. It would be great to come up with all the testbases at this point, but typically only one or two emerge. Once the engineers see how the process works, they can easily add few more later.
- Another practical approach to adding testbases, especially if the team is resistant in this step, is to let the team group the features into any categories that emerge naturally.

  Do this activity together on a big whiteboard. Circle the categories and give them names. Now, ask this question, “Given the VI that we identified in Step 1, what configurations of the harness and what verification components are needed to handle each of these category groups?”

  You might circle five of the seven groups and point out that they can be handled by one configuration. The other two groups can be handled with another configuration. You now have two testbases.

5.0 Step 3: Random Generation

Description

In the past, random generation, if used at all, was done at the testcase level. Each testcase used the $random to apply random data to a BFM. The goal of an *e*VI is to develop an actual random generation layer that can be intelligently and easily controlled via extension and constraints.

It is much more than just a $random approach. The random generation layer must be architected up front. Control via external constraints requires visibility over all relevant parameters. Strategically define knobs and levers on the control panel to facilitate easy control of the random generation. This layer controls the spread of your shotgun output, and where it is aimed initially.

The trick here is to set up lots of knobs, but in such a way that most will never have to be set manually. Only a few select knobs will need to be changed by the average testcase constrainer. The actual knob implementation and most knobs themselves will be hidden. The bottom line is to make the randomization both easy to extend and constrain.

5.1 How To Set Up Random Generation

Pick a testbase. The first one should be one of the easier ones. You will perform the final five steps one at a time on each testbase.

- Outline what interfaces are included.
- Pick an interface, identify the information and abstracted basic operations that will be generated for each interface.

  Think of what type of information (fields) are required to describe each transaction, read/write, address, data, etc.. Think basic -- additional values/types for a specific field can be extended later.

  If the *e* has not been written for a BFM for a particular interface, you can start to outline the unit and its fields. Now, start thinking about what the knobs can control in that transaction. For instance, number of wait states, number of burst cycles, locking, etc.. Note which fields are to be randomized and what the ranges and default constraints (keeps) should be. Don't be afraid to use an existing, legacy BFM written in HDL. But if a BFM does not exist, it may be quicker to create it in *e*.
- Now, go over the monitors for that interface. Define what you looking for.

  Decide if you want to look at internal signals (white box testing), or just monitor the I/O (black box testing). White box testing is useful for coverage, however it is implementation dependant. Black box testing is preferred, but the visibility into the design is limited. What indicates good/bad? Think first in terms of the protocol, then in terms of data. Reference the timing diagrams. Which events will you need for your temporal expressions (TEs)? What asynchronous signals, drivers or checkers should be used? Which events do you want to use for coverage?
- Prepare your code to be extended. Add hooks into your code at the beginning, at the end, or at any other interesting point. These hooks might just be empty methods at this time, but they will allow strategic extension later. These hooks will give you more control and will overcome the limitations of the is first, is only, and is also extension constructs.
- Next, go into defensive programing mode. Ask how to write these to make them easily extendable with contraints. What if there are multiple instantiations of the interface? Can I constrain each sequence, or does a constraint apply to all sequences? Do I need to set up semaphores? Need to use computed HDL names? Code it. Check it for limitations. Re-code it.
- Repeat the last three steps for each interface. Notice any interaction between various interfaces.

6.0 Step 4: Scenarios

Description

If you have used the random features of *e* at all, you know that the combinations can quickly snowball into an almost infinite (and memory-hogging) number of combinations. The goal of scenarios is to quickly and easily narrow the amount of random combinations down to ones that are strategic and make sense.

If you have done a couple rounds (code, check for limits, re-code) on the random generation, you have narrowed down the potential combinations somewhat already. This is the first level of abstraction on the random generation front.

Scenarios are the second level. What you want to do is create a scenario descriptor object and then use the when statement to define the strategic constraints for each scenario. You already have the knobs and gauges described for the random generation. Now, draw out a list of interesting scenarios, and make it easy to add new scenarios as they develop.

Scenario-specific methods can be used to iterate over the available abstracted BFM methods (read and write of a CPU interface for instance). New methods can be added to use the hooks built into the existing methods created previously. Also, scenario-specific generation of random sequences can also be included. You can even modify when generation occurs and when data is applied on a per-scenario basis. The bottom line is to use scenarios to quickly and consistently narrow the randomness down into interesting setups. Aim the shotgun strategically. Some simple strategies here can cut simulation time way down.

6.1 How To Create Scenarios

- Review each interface and its random hooks. Think in terms of memory uses and number of potential combinations.
- Next, constrain only the most interesting and common scenarios. Make a scenario descriptor (normal, hammer, multi, etc.).
- Now, use the when construct to define the constraints for each scenario. How about modifying/extending certain methods or a special method to iterate over existing methods? Modify generation times. Make certain random generation dependant on other previous generated data and so forth.
- You can start with some typical scenarios and then add more or build on existing ones later.
- Scenarios are a way to achieve or at least approach the one-testbase ideal. If you code correctly, you can have the one testbase and then use the scenario layer to handle configuring and constraining any particular feature group setup.

7.0 Step 5: Error Injection

Description

Many interfaces are designed to detect and recover from violations. A good *e*VI should be able to apply errors randomly to test the *e* monitors and DUT for compliance. We need to architect an *e*VI that can control the injection of all the different kinds of errors and error combinations, and check for proper recovery from these errors.

We also need to architect the *e*VI to detect the errors, and verify that we injected it or flag that we did *not* inject it. With our existing setup of random generation and scenarios, we have options on how to proceed. We may be able to add error injection mechanisms to existing code via the hooks, or we may want to develop an independent unit that handles all the error injection for us. Typically, it is hard to extend the existing setup with the is also or is first constructs, and the is only construct makes for high maintenance. The typical use of for loops also makes it hard to add error injection with is also or is first.

Instead, an error descriptor can be made to define and inject errors. The default should be no errors, but via constraints, errors can be injected. These errors can be injected easily if pre- or post-hook methods are used, or as direct calls in the BFM code itself. Experience shows that using a separate error mechanism, with its random generation and scenarios works best.

7.1 How To Inject Errors

With error injection you need to move over to the dark side of verification and start thinking about breaking the design. Some teams never get this far and only have time to test normal operations. You might have to trick them into thinking outside the box, especially this early in the project.

- First, look at each interface again and outline all the possible errors.

  Where and how are they injected? What kinds? Parity, protocol corruption, cache missing, missing or delayed signals, and so on. What is likely? Most interfaces have a timing diagram. Draw it on a whiteboard and it should lead to a list of errors.
- Next, define an error descriptor for each interface. In some cases this is a natural extension of the existing scenarios.
- Now, go back and review your random generation code. Add mechanisms for each error. Decide whether to do it as a pre- or post-hook method, or as a call in the BFM code.
- Revisit scenarios with the new error hooks. Use constraints to force the injector always on/off, or use a weighted keep soft. You might add scenarios all together that are error specific.

8.0 Step 6: Scoreboarding

Description

Scoreboarding is the self-checking technique of predicting, storing and comparing data. Typically, it includes three parts.

- The first part is a transfer function to generate expected data.
- The second part is some sort of data structure to store the data generated in part one (this is often called a scoreboard).
- The third part is a comparison function that is used to compare actual data with expected data.

Input data is applied to the transfer function (and the DUT) and the resulting expected data is stored into a suitable data structure. The comparison function removes data from the data structure at the appropriate time, and compares it to the expected data that comes out of the device under verification and/or monitors.

The data structure choice is highly dependant upon the comparison functions needed to handle ordering or latency issues. Also, the comparison function must determine success or failure either on the fly or as a post process.

The nature of the transfer function, the comparison function and the data structure is highly design dependent and often is the part of the *e*VI that takes the most effort to set up and maintain.

Scoreboards typically make use of *e*'s list construct and the associated list operators and pseudo-methods. Lists must be used carefully, because often there are large memory use ramifications. On-the-fly scoreboarding is generally better, because it is more memory efficient, and it can stop on an error, leaving the DUT in its current state.

8.1 How To Build a Scoreboard

- The scoreboard and its components (transfer function, data structure, comparison function) should be separate units.
- First, come up with an overall approach. Use on-the-fly autocheck mechanism or a post-processing mechanism.

  How should the data (input, intermediate, output) be structured? How much data are we talking? The nature and amount of the data will dictate the approach. What indicates good/bad? When do we sample data? What events trigger data? Do we need to remove the data from the scoreboard?
- Next, define the transfer function.

  This is a behavioral model of the part. It does not need to emulate the structure or latency of the actual device. It needs only to mimic the interfaces (what is going in and what is going out).

  Some systems are more easily modeled than others, but regular design engineers typically have never experienced the quick development or tangible use of a good behavioral model.
- Then, come up with a data structure to handle the data.

  Use lists to make mailboxes to handle data through the part. You may need a multi-dimensional list, which *e* cannot do. A struct containing a list can be used to make a list of lists. What methods (push, pop) will the mailboxes need? Consider using keyed lists for memories. Can you trigger the coverage buckets to hold data? Can you make generic mailboxes and then instantiate them as needed (per channel)? How will the comparison function access data? What methods are needed? What data is essential?
- Now, determine the comparison function.

  On-the-fly or post-processing? What data to check? When to check? What event triggers the comparison? How do you handle latency and ordering differences? You may have several comparison functions that compare along the way and move data from mailbox to mailbox using hook or handshaking methods. When the test is done, verify that the scoreboard is empty.
- It is often difficult to trace packets through a design. Add information to the payload portion of the data packet that will help identify and trace this packet.

9.0 Step 7: Coverage

Description

Coverage is the feedback mechanism. It should flow naturally from your random generation and scoreboarding setups. The interesting scenario and error injection list generated in Steps 3-6 can be used as a framework to guide the coverage brainstorming. You need to come up with coverage goals and then tangible data mechanisms that will let you see if the goals were met.

If the coverage goals are too nebulous, then you will be unable to provide meaningful metric information to gauge how the verification is going. The coverage information should be tangible. We will be able to see which testbases and seeds are most effective, which are redundant, and if any holes exist. Use those seeds that were most effective for regressions.

Continue to run with random seeds to cover the holes or create new directed tests to cover the holes (sniper). Functional coverage does not tell you when you are done verifying. It tells you what has not yet been verified. Coverage reports will point to new constraints and new testcases.

9.1 How To Develop Functional Coverage

Remember that we are talking functional coverage here, not code coverage.

- Brainstorm on metrics and goals.

  Gather the design engineer, the architect, the verification engineers, any protocol experts, and so on. Gather documentation, CVP, feature list, design specification, protocol specification, and so on. Start with the coverage information added on the yellow stickies during the feature extraction.
- In general, go over this shopping list: basic functionality, major state machines, any interesting event or scenario, major chip modes, any established protocols, all op codes, all address and data, interrupts, types of packets, relevant traffic patterns, etc.. To drive your brainstorming, review each interface.
- Examine when, what and how on each one.

  Basic or exhaustive iterations first, then interesting combinations. This is highly design dependant. Come up with coverage categories. For a large range of values, use ranges (or buckets) to capture the metrics. Decide which coverage items should be crossed, for example, in port crossed with out port. Use transitional coverage to check for valid and invalid states.
- For each coverage category, translate the goals into coverage groups and items. Examine how these functional coverage groups would be integrated in the testbase. It may suggest changes to testbase or BFMs to facilitate the coverage mechanisms.
- Run some example coverage setups and get the coverage reports. Then, use the results to tweak the whole coverage setup.

10.0 Step 8: Testcases

Description

At this point we have created a testbase with smart random generation, error injection, scenarios, scoreboarding and coverage. If we did it right, this last part is easy. We let it run and randomly make testcases for us (the shotgun shoots a series of strategically aimed bursts).

Next, examine the coverage reports and see what is still not covered.

Then, we create additional testcases to sniper in on any holes. The testcase file is the top-most level -- a simple set of instructions to set up the control panel. It is simply going to contain a list of contraints and custom extensions. A typical verification engineer would extend and constrain values in this top file only, leaving all the underlining files alone. In fact, the engineer need not know about most of the knobs underneath, only a few of the scenarios. He or she picks a scenario that is closest to what the engineer wants to do. It sets up a majority of the constraints automatically. Then, the engineer extends and modifies at this top level just enough to drive the testcase to cover the hole.

10.1 How To Create Testcases

- Start with a template file that contains comments that list the fields they can play with and what the ranges/values are.
- The engineer imports the top files along with any bug-related files, and turns on coverage and error injector.
- The engineer sets them to perform a specific test. He or she might even hard code a seed in there to reproduce a bug.
- The engineer tweaks this file, which enforces the concept of data hiding. If you wanted to get fancy, this file could be the output of a GUI or a more user-friendly/ automated program (platform independent).
- For this to work, there must be some documentation of the VI. This document details the scenarios and the associated methods and constraints enough so that a testcase engineer can write directed testcases easily. This public document need not go overboard in describing all the private details of the environment. The document needs only enough detail to guide the testcase writer.

11.0 Conclusion

There has been a paradigm shift in verification. Just like going from test-vector-based verification to self-checking testbenches, this new shift allows DUTs to be tested more quickly and with greater confidence. Market forces have pressured the design time to be shorter, while chip complexity has increased.

To meet the new demands, often one of the first things to be cut is the verification time. New tools and methodology must be embraced to stay competitive and maintain verification quality. HVLs and random verification have become the weapons of choice for verification engineers. Once a verification engineer trades in his or her pea shooter for a shotgun and a rifle, that engineer won't go back.

Good *e*VIs do not just happen. They must be planned and built. The eight-step methodology spelled out here injects the random approach up front and all along the way. Code is built strategically and in a layered fashion. This facilitates both ease of extending and constraining, as well as future reuse of the verification components.

The random *e*VI is run on autopilot to shotgun most of the features and find most of the bugs. Coverage is used to give feedback on any holes discovered. Constrained testcases are then aimed into these holes. This random shotgun approach often is something that needs to be experienced by a verification team before they see the benefit. In reality, it will happen over time, often over several iterations of a project. But, it is important for them to quickly see the benefit of this new approach to drive the transformation.

Naturally, some super *e* coders will develop. They will learn to love their shotguns and become great at writing shotgun code. Others will become snipers, not needing to know all the details of the shotgun code, just the ability to write sniper testcases. Both are necessary and both will team up to verifying with more confidence, more accuracy, and in a shorter time frame.

Examples of Shotgun *E*

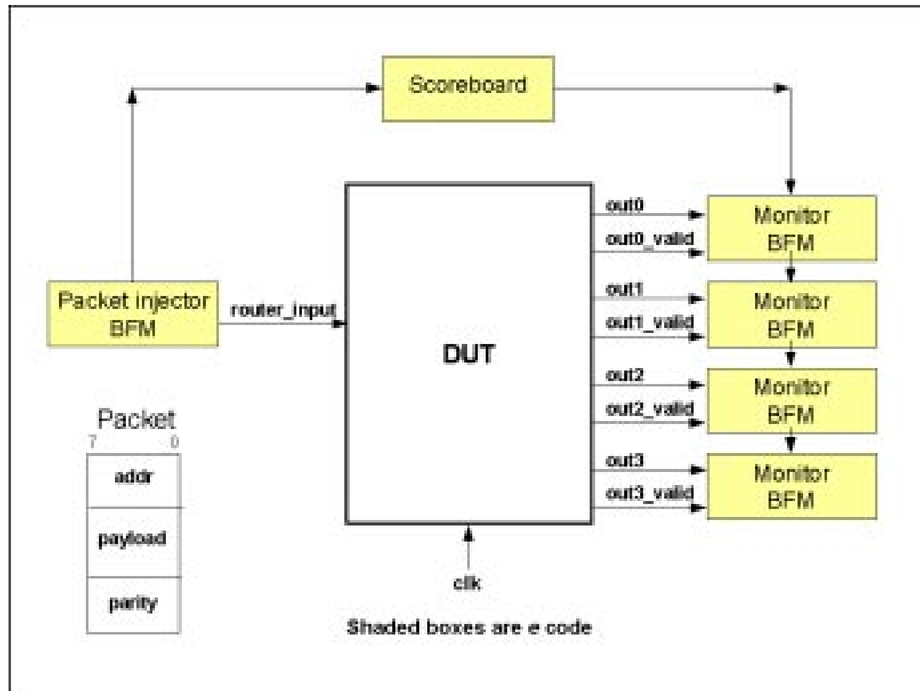
An Appendix Supporting the Eight-Step Approach to Random Verification

Presented in this Appendix are key examples that illustrate using the eight-step approach to random verification.

12.0 Step 1: Feature Extraction Example

The device under test is a simple four-port router.

Figure 2. DUT is a four-port router



Based on our router, the features would be:

- Send in all types of packets scenarios
- Interesting packet payload size (long (> 128 bytes), short (< 128 bytes), zero)
- Interesting delays between packets (long delay (> 20 clocks), short delay (< 20 clocks), zero delay)
- Inject errors into all scenarios
- Compare output packets to expected

Coverage metrics:

- Verify that all types of packets were sent to all four ports
- Verify that errors were injected in all packet types

13.0 Step 2: Testbase Extraction Example

The simplicity of this router example, as well as its short feature list, make the testbase extraction very easy. We need only one testbase that instantiates:

- Packet injector (1)
- Packet monitor/receiver (4)
- Method to corrupt parity
- Scoreboard

We will use a scenario descriptor to handle the different packet length. We will use a smart injector, scoreboard and monitor combination to handle the error injections. All features will be verified using this one testbase. If the DUT had other interfaces (CPU programming interface, proprietary interfaces, other standard interfaces), as well as other modes to run in, then we might have more testbases.

14.0 Step 3: Random Generation Example

The device under test is a simple 4-port router.

There are two primary interfaces:

1. The packet injector
2. The router output (4 copies of the same interface)

Abstract of injector operation:

1. Create a packet.
2. Pack the packet.
3. Inject the packet (one bit per clk).
4. Wait some delay and start again.

The injector needs to create a data packet. So we create a simple data packet struct:

Sample 3
Simple data packet struct

```
struct pkt {
     !addr     : byte;
          -- use ports 0-3
          keep soft addr < 4;
     !length   : byte;
     !payload  : list of byte;
     !parity : byte;

     calc_parity() : byte is {
         result = addr ^ length;
         for each (word) in payload {
            result = result ^ word;
         };
      };
 };
```

Next, create the injector. In this example, we are not going to look at internal signals.

Sample 4
Creating the injector

```
unit pkt_injector {
    !my_pkt     : pkt;
    !packed_pkt : list of bit;
     event rclk is rise('clk')@sim;

     inject() @rclk is {
         gen my_pkt keeping { it.parity == sys.pkt.calc_parity() };
         packed_pkt = pack(packing.high, my_pkt);
         for each (single_bit) in packed_pkt {
             'router_input' = single_bit;
             wait [1] * cycle;
         };
      };
};
```

Next, create the monitors. Abstract of monitor operation:

1. Wait until rising edge of valid signal.
2. Sample data coming out of port until valid drops.
3. Emit a flag to signal that a word was received.
4. Loop to step one and wait for next word.

In the following code, only one of the monitors is shown. The other three monitors would be similar.

Sample 5 Creating the monitor

```
unit monitor {
     !out0   : list of bit;
      ....
      event rclk is rise('clk')@sim;
      event out0_done;
      ....

      mon0() @rclk is {
         while TRUE {
            wait true('out0_valid' == 1);
            out0.clear();
            out0.add('out0');
            while ('out0_valid' == 1) {
               wait [1] * cycle;
               out0.add('out0');
            };
            emit out0_done;
         };
      };
....
 };
```

Finally, we create a loop that calls the injector, then delays. The delay will be one of our knobs on the control panel:

Sample 6 Creating a loop that calls the injector and then delays

```
unit top {
    event rclk is rise('clk')@sim;

    main()@rclk is {
        var max_loop : uint;
        var clk2wait : uint;

        gen max_loop;
        gen clk2wait;
        for i from 1 to max_loop {
            sys.pkt_injector.inject();
            wait [clk2wait] * cycle;
        };
     };
 };
```

At this point, consider constraining the design. Focusing on the top struct, the number of loops (max_loop) and number of clks to wait (clk2wait) are unconstrained. We could constrain them internally to the main() method via:

Sample 7 Constraining the design internally to the main() method

```
unit top {
   event rclk is rise('clk')@sim;

   main()@rclk is {
       var max_loop : uint;
       var clk2wait : uint;

       gen max_loop keeping { it in [5..10] };
       gen clk2wait keeping { it in [0..20] };
       for i from 1 to max_loop {
           sys.pkt_injector.inject();
           wait [clk2wait] * cycle;
       };
    };
};
```

However, the constrains cannot be modified/extended externally. A better way to write the constraint would be have the values in their own struct. That way they can be constrained via extends:

Sample 8 Constraining the design with values in the struct enables you to constrain via extends

```
struct top_max_loop_value {
     value : uint;
};

struct top_clk2wait_value {
     value : uint;
};

unit top {
    event rclk is rise('clk')@sim;

    main()@rclk is {
       var max_loop : top_max_loop_value;
       var clk2wait : top_clk2wait_value;

       gen max_loop;
       gen clk2wait;
       for i from 1 to max_loop.value {
            sys.pkt_injector.inject();
            wait [clk2wait.value] * cycle;
        };
    };
};
```

Now the values can be externally constrained via extends:

Sample 9 Constraining the design via extends

```
extend top_max_loop_value {
     keep value in [5..10];
 };

extend top_clk2wait_value {
     keep value in [0..20];
 };
```

Other changes could be made to optimize the code.

15.0 Step 4: Scenarios Example

Taking the router example from the previous section, we are going to create three scenarios:

- Norman (unconstrained) packet
- Long packet with zero delay
- Short packet with a large delay

A long packet will have >128 bytes of payload; the short packet will have <128 bytes payload. To do this, start by creating an enumerated type for the scenario types. Then in a new struct, use when statements to create the proper set up for the different scenarios:

Sample 10 Creating an enumerated type for the scenario types

```
type scenario_types : [normal, long_zero, short_big];

unit scenario {
     scenario_type : scenario_types;

     when long_zero scenario {
        keep sys.pkt.payload.size() > 128;
        keep sys.top_clk2wait.value == 0;

     };

     when short_big scenario {
        keep sys.pkt.payload.size() < 128;
        keep sys.top_clk2wait.value == 20;

     };
};
```

As additional scenarios are identified, they can be added. For example, assume we need a packet with zero payload. This could be added via:

Sample 11 Adding scenarios as needed

```
extend scenario_types : [zero_pload];

extend scenario {

     when zero_pload scenario {
        keep sys.pkt.payload.size() == 0;
      };
 };
```

16.0 Step 5: Error Injection Example

At this point, we are going to inject parity errors.

First thing we need to do is create another enumerated type to specify the type of error to be injected. Next, focus on the actual error code.

There are a couple of different ways we could do this. The first and most obvious way is to extend the calc_parity() method to create a bad parity byte. This is shown via:

Sample 12 Extending the calc_parity() method to create a bad parity byte

```
type error_types : [none, parity];

extend pkt {
    error_type : error_types;

    when parity pkt {
        calc_parity() " byte is also {
           gen result keeping { it != result };
        };
    };
};
```

Now assume that it wasn't possible to change the parity via a simple is first or is also extension. Instead, change parity by adding a hook to an empty method internal to the inject method, as shown here:

Sample 13 Changing parity by adding a hook to an empty method

```
unit pkt_injector {
     !my_pkt     : pkt;
     !packed_pkt : list of bit;
      event rclk is rise('clk')@sim;

     inject() @rclk is {
         gen my_pkt keeping { it.parity == sys.pkt.calc_parity() };
         post_gen_my_pkt_hook(my_pkt);      ---  ****
         packed_pkt = pack(packing.high, my_pkt);
         for each (single_bit) in packed_pkt {
             'router_input' = single_bit;
             wait [1] * cycle;
         };
      };

     post_gen_my_pkt_hook(my_pkt : *pkt) is empty;
 };
```

Then we could extend the hook method via an error scenario:

Sample 14 Extending the hook via an error scenario

```
type error_types : [none, parity];

extend pkt_injector {
      error_type : error_types;

     when parity pkt_injector {
         post_gen_my_pkt_hook(my_pkt : *pkt) is also {
            var tmp_pkt : pkt;
            gen tmp_pkt keeping { it != my_pkt};
            my_pkt = tmp_pkt;
      };
 };
```

17.0 Step 6: Scoreboarding Example

The scoreboard is easy to write, but it is often the most time consuming in terms of debugging.

Up to this point, we have created an injector and four monitors.

Next, use the following steps to create a scoreboard:

1. Create the overall approach used for the scoreboard.

For the transfer function:

- Capture the data that is injected into the router.
- Calculate the expected output port based on the address bits.
- Add it to a list of expected packets.

For the comparator:

- Wait until the output monitor emits the event for a particular port.
- Compare the value that came out to the expected value.

2. Create the transfer function:

Sample 15 Creating the transfer function

```
unit scoreboard {
      event rclk is rise('clk')@sim;
      !mailbox0 : list of pkt;
      !mailbox1 : list of pkt;
      !mailbox2 : list of pkt;
      !mailbox3 : list of pkt;

     transfer(my_pkt : pkt) is {
         var tmp_pkt : pkt;

        if my_pkt.addr == 0 {
            gen tmp_pkt keeping { it == my_pkt };
            mailbox0.add(tmp_pkt);
         } else if my_pkt.addr == 1 {
            gen tmp_pkt keeping { it == my_pkt };
            mailbox1.add(tmp_pkt);
         } else if my_pkt.addr == 2 {
            gen tmp_pkt keeping { it == my_pkt };
            mailbox2.add(tmp_pkt);
         } else {
            gen tmp_pkt keeping { it == my_pkt };
            mailbox3.add(tmp_pkt);
         };
      };
};
```

3. Hook up the scoreboard so the transfer function is started correctly.

   In this case, we want to capture the data just before it is injected to the router. We already added a hook for an empty method back in Step 5. So we can extend the method to capture the data:

Sample 16 Extending the method to capture the data before it's injected to the router

```
extend pkt_injector {
     post_gen_my_pkt_hook(my_pkt : *pkt) is also {
        sys.scoreboard.transfer(my_pkt);
     };
};
```

4. Create the comparison function.

   Like the monitors, this will be implemented as four separate comparison functions. Only one is shown:

Sample 17 Creating the comparison function

```
extend scoreboard {

    compare0() @sys.monitor.out0_done is {
        var unpack_pkt   : pkt;
        var expected_pkt : pkt;

       while TRUE {
           unpack(packing.high,
                  sys.monitor.out0,
                  unpack_pkt.addr,
                  unpack_pkt.length,
                  unpack_pkt.payload,
                  unpack_pkt.parity);
           expected_pkt = mailbox0.pop0();
           check that unpack_pkt == expected_pkt
              else dut_error("Mismatch on port 0");
           wait [1] * cycle;
        };
     };
    ....
};
```

18.0 Step 7: Coverage Example

Next we are going to implement functional coverage. First, you must decide on which metrics you want to cover. For this example, we want to verify that we have every type of scenario (normal, long_zero, short_big) on all four ports.

Next, we need to decide when we want to sample the data. It makes sense to sample it right before we inject it. We already have a hook method we can extend on. We are also going to sample the items we are performing coverage on. That way all of our data is in one struct, making it very easy to do coverage on it:

Sample 18 Sampling data before injecting it

```
extend pkt_injector {
      event inject_data;
      cover_addr : byte;
      cover_scenario : scenario_types;

     post_gen_my_pkt_hook(my_pkt : *pkt) is also {
         cover_addr = my_pkt.addr;
         cover_scenario = sys.scenario.scenario_type;
         emit inject_data;
      };
 };
```

Next create the coverage code:

Sample 19 Creating the coverage code

```
extend pkt_injector {

     cover inject_data is {
          item cover_addr;
          item cover_scenario;
          cross cover_addr, cover_scenario;
      };
 };
```

Finally, we need to turn coverage on:

Sample 20 Turning on the coverage code

```
extend global {
     setup_test() is also {
        set_config(cover, mode, normal);
     };
 };
```

19.0 Step 8: Testcases Example

Finally, we can create the testcase template file that everyone is using. We are going to create a basic template that others can customize to steer testcases:

Sample 21 Testcase template file

```
-- Uncomment next line to turn coverage on
-- import coverage;
extend top_main_values {
       -- Set the max # of packets to send into the router
        keep max_loop_value == 30;

        -- Set the delay between the packets
        keep clk2wait_value in [0..20];
};

 extend scenario {
       -- Set the packet type and frequency of packet.  Choices are:
           -- normal
           -- long_zero
           -- short_big
           -- zero_pload
        keep soft scenario_type == select {
           50 : normal;
           20 : long_zero;
           25 : short_big;
            5 : zero_pload;
        };
       -- Or could create new type
 };
extend pkt_injector {
       -- Set the error type and frequency of packet.  Choices are:
           -- none
           -- parity
        keep soft error_type == select {
            1 : parity;
            9 : none;
        };


       -- Uncomment next line to turn off all errors
       -- keep error_type == none;
       -- Or create new types
 };
```