WHITEPAPER

Pondering Parameterization

Papers and discussions from Verilab consultants on the verification of parameterized and configurable designs

Collected and edited by Jonathan Bromley, Verilab Ltd

March 2020

Contents

1. Motivation

1.1 Parameterize Like a Pro

Verilab consultants Jeff Montesano and Paul Marriott presented a tutorial workshop ***Parameterize Like a Pro*** at DVCon-US in March 2020. That workshop drew on Verilab's collective experience to present a wide-ranging collection of techniques, tips and recommendations for creating verification environments that can meet the tricky challenges presented by highly configurable and parameterized designs and design IP. To accompany the workshop, this whitepaper offers a curated collection of papers and other material from several Verilab consultants with extensive experience of these challenges.

1.2 Using this material

As you look through this whitepaper, it's important to remember that – like everything we do in Verilab – it represents an ever-growing depth of experience. The single-issue snapshot presented in a typical conference paper or tutorial is never the end of a story. Our techniques and our understanding of best practice are continually evolving in the light of new experience, vibrant internal discussion among colleagues, and invaluable input from our clients and other friends in the industry.

To try to capture our passion for continual improvement, this whitepaper tells the story of our growing understanding – a story told through written material prepared by our consultants, the internal discussions within Verilab that have challenged and refined those ideas, and the experience of applying them on real projects.

As a result, this is **not** a cookbook. Think of it more as a series of challenges, ideas, suggestions and occasional warnings. Use it to question and inform your own thinking. As verification engineers we are privileged to work in a field where we must continually evaluate, think and create, in the company of smart colleagues, working on exciting and challenging projects. In that spirit of curiosity and exploration we hope you will enjoy sharing our exploration of configurability and how to handle it. Feel free to tell us if you think we got anything wrong.

1.3 A word about confidentiality and acknowledgement

Everything in this whitepaper is based on real-world experience, usually on our clients' projects. To protect their confidentiality, some information and examples have been simplified and edited. Wherever possible we have tried to acknowledge the valuable input we have gained from sources outside Verilab, but if there's anything we have missed, please let us know and we'll be happy to correct the oversight.

2. Classes and RTL Parameters Don't Play Nicely

A long, long time ago, when I was first writing a driver and monitor (back in the days of OVM!), I thought it would be a good idea to parameterize my classes to accommodate different sizes of data bus:

```
class my_monitor #(parameter DATA_WIDTH=16) extends uvm_monitor;
  ...
  protected bit [DATA_WIDTH-1:0] data_value;
  ...
```

Experience – the best, but the hardest teacher – soon showed that this was A Very Bad Idea. To create a parameterized monitor component instance within our agent, the agent class needs to be parameterized too – otherwise there's no way to set the parameter on our monitor instance. And if the agent is parameterized, the environment that instantiates that agent needs a parameter… and so on, up the tree of components. In Verilab we have come to think of this as the "parameter ripple effect".

Our senior consultant **Mark Litterick** explains this issue, and many more relating to the hazards of using parameterized classes in a verification environment, in his tutorial **Parameterized Classes, Interfaces and Registers** presented at DVCon-Europe 2015:

verilab.com/files/verilab_dvcon_eu2015_6_params.pdf

In the vast majority of our work we find that the dynamic nature of class-based verification code means that we can handle parameterized bus widths, and similar RTL scale considerations, *without* using parameterized classes or interfaces. Instead we create "maximum footprint" connections in our interfaces, allowing the interfaces themselves to be free of parameters. That makes it very much easier to take virtual interface references to them. However, it leaves us with the problem of getting each parameter value into our class-based verification environment where we can use it -for example – to mask unwanted bits of the data bus. Over the years we have experimented with numerous approaches to solve this problem, but it was a while before we found a truly complete and satisfying solution. More on that in the next section.

3. A Reuse Game Changer: the UVM Harness

At Verilab we get very excited about reuse of verification assets. We talk of "horizontal" and "vertical" reuse.

- Horizontal reuse means that a single piece of code (such as a verification IP block) can be used in many different but similar situations, possibly more than once within a given testbench.
- Vertical reuse describes how a block of verification code – possibly even a complete testbench – can be used, without modification, as a sub-component of a larger verification environment.

Both kinds of reuse are essential for verification productivity, and we place great value on achieving reusability in almost everything we do. Horizontal reuse typically demands a high degree of configurability, so that the verification IP can be configured to match the situation into which it is placed. Is it to be an active driver, an active slave (responder), or a purely passive monitor? Which versions of its protocol are permissible in this DUT? How many lanes does this SERDES have? Vertical reuse may also call for some configurability, but most importantly it calls for reliability and robustness – it's not very helpful to deploy an unreliable, untrustworthy verification subsystem in a larger environment.

One of the greatest barriers to effective vertical reuse is the challenge of establishing the right signal connections between our verification components and the RTL world of the device under test. Thanks to a seminal whitepaper by David Larson of Synapse Design Automation, we learnt of the *UVM Harness* technique. Since then, Verilab consultants have continued to refine and extend this technique and we now use it wherever we can. Kevin Johnston, Jeff Montesano and Jeff Vance gathered much of our accumulated knowledge in a paper presented at SNUG Austin in 2017. It's available here:

https://www.verilab.com/resources/papers-and-presentations/#snugau2017verifprowess

It's difficult to over-state the impact of adopting this approach. It almost magically simplifies the task of connecting a verification component to a new DUT or a new location within a DUT. Better still, it facilitates vertical reuse by making it trivially easy to deploy a verification block as a sub-environment of a bigger testbench.

Almost as they say on TV, that paper contains ideas that some readers may find unsettling. You'll find familiar SystemVerilog language features such as `input` ports and `bind` being used in surprising ways. Stick with it, though – the pay-off is impressive.

4. Connections Again

Not satisfied with the UVM Harness solution, our "gang of three" (boosted to four with the addition of Kevin Vasconcellos) then took on the challenge of automating the virtual-interface linkage between connection interfaces in the Harness and their corresponding verification agents.

Their solution calls for structuring the testbench to have a component hierarchy that mirrors the RTL design hierarchy. This sounds onerous, but in practice it is not – as you can see for yourself by reading their 2018 DVCon paper **My Testbench Used to Break, Now it Bends!**

https://www.verilab.com/resources/papers-and-presentations/#DVCon2018Bend

The central idea in that paper was that interfaces within a UVM Harness can deposit their own virtual interface references into the UVM configuration database in such a way that each agent in the UVM testbench can pick out the virtual interface it requires, with the whole setup reconfiguring automatically as sub-environments are deployed in various ways for both vertical and horizontal reuse.

This issue isn't going away. We continue to have vigorous discussions within Verilab reviewing different ways to make the virtual interface connection between a UVM environment and the module-centric world of the DUT and its test harness. Recently Jeff Vance summarized the relative merits of three solutions:

> ***Explicit deposit of virtual interface references into the UVM's configuration database***
> *PRO: can work with any possible UVM environment setup. No restrictions on *how* you structure environments.*
> *CON: requires maintenance at tb_top and arguably less scalable.*
>
> ***Interfaces self-publish to the configuration DB, using agent-name keys***
> *PRO: No maintenance at tb_top and scales to parameterized number of agents.*
> *PRO: Moderate flexibility in env hierarchy conventions.*
> *CON: No horizontal reuse between projects if agents change names.*
> *CON: Only supports a single env hierarchy. Does not work if parameters change number of envs (or can it?? Need to think about this more).*
>
> ***Interfaces self-publish to the configuration DB, using keys derived from %m module path***
> *PRO: The most flexible and scalable. Supports generating different env hierarchies in the same testbench. Needed if parameters change number of env's (?? I think ??).*
> *CON: project *must* adopt the mirrored module/env hierarchy convention.*

Without worrying about those details, the most interesting point here is the simple observation that *almost every architecture decision has consequences* and there is no substitute for active, engaged discussion within a creative project team to evaluate any proposed innovation and to investigate its possible side-effects.

5. A Different Kind of Connection: Integrating SVA Into Your UVM Testbench

5.1 SVA, too good to miss

SystemVerilog Assertions (SVA) are awesome. They provide correctness checking that has sophisticated trigger conditions and awareness of behavior over time, using a concise but intuitive language and with some very powerful features. They allow you to identify a complex sequence of activity and take action whenever it occurs.

If you enjoy using a powerful text editor, or script languages such as Perl, Python or Tcl, you surely have used regular expressions to search through text looking for anything that matches your chosen pattern. SVA provides you with patterns very similar to regular expressions that search through a set of waveforms seeking a match, and it allows you to check (assert) that your pattern has occurred in the right place and has followed all your rules. It's an enormously valuable addition to your verification toolkit.

5.2 SVA doesn't play nicely with UVM

So why don't we see SVA used more extensively in UVM testbenches? Because – frustratingly – SVA cannot exist in the UVM's dynamic world of classes and objects. SVA can be created only within the static world of modules and interfaces. There's no arguing with that; it's baked into the structure of SystemVerilog.

Consequently, we need to put our assertions out in the static world of modules and interfaces, and then find some way to connect them with the class-based dynamic world of our UVM testbench. Without that connection, our SVA will be unaware of testbench configuration and UVM phasing, and it won't fit in with the rest of our verification strategy.

5.3 SVA encapsulation

Dealing with this SVA/classes mismatch, and making the connections, is not in itself especially difficult. On the other hand, without a consistent systematic approach to the problem, engineers will find themselves working with ill-disciplined ad-hoc solutions that differ from project to project and even among verification IP blocks within the same project. To address this concern, our Senior Consultant Mark Litterick put in the effort to construct a consistent, reusable collection of coding patterns and recommendations that allow SVA and a UVM testbench to interoperate in a flexible way. You can find his easy-to-follow methodology in his DVCon-US 2013 paper at this link.

6. Defusing the Parameter Combinations Bomb

6.1 Too many parameter combinations

We all know that it's impossible to simulate all possible patterns of activity in any realistic-size DUT. There are just too many combinations of activity on the DUT's various ports and internal state machines. Nevertheless, we have a powerful and project-proven battery of techniques for establishing enough confidence in our verification that we can say "yes, verification is done and we can sign off". We use functional coverage, code coverage, mutation analysis, exhaustive formal verification of critical sub-blocks, and – most of all – we build our systems by combining thoroughly verified and field-proven IP blocks.

What happens, though, when you have a DUT that is highly parameterized? Each simulation run – indeed, each complete regression run – can operate only on one specific parameterization of the DUT. Yet our customers will use a huge variety of different parameterizations, especially if our product is a configurable subsystem such as a network-on-chip or a memory controller. How can we be sure that **all** legitimate parameterizations will work correctly? The answer is simple – we can't. For any realistic parameterizable design, there are simply too many parameter combinations for us to be able to test them all. With regression runs on a single parameterization taking hours of compute-farm time, the cost of testing even a modest number of different parameterizations is alarmingly high.

6.2 Tools for the job

The first challenge for verification engineers is to ensure that their testbench can easily cope with any single parameterization of the DUT. This is harder than it might at first seem, because the compile-time nature of parameters doesn't fit comfortably with the dynamic configuration we generally expect from a UVM testbench. In their paper presented at SNUG Canada 2017, Alex Melikian and Paul Marriott surveyed a battery of techniques that a UVM testbench can use to find out about the parameterization of its connected DUT, and bring those parameters into the configuration/resource database so that the testbench can read and make use of them.

6.3 Getting the problem under control

Once we have a testbench that's sufficiently flexible to work with any parameterization of the DUT, the next step is to determine exactly which parameterizations we should test. Should we choose them randomly for each regression run, in the hope of finding corner-case bugs that affect some weird combination of parameter values? Should we put massive effort into testing the parameterizations that we know our early key customers will be

using? (Yes, definitely!) And is there a better way of choosing the sets of parameter values that need to be tested?

6.4 Pairwise testing

A few years ago, one of our clients introduced us to an approach that's long been used in software testing. It's called **pairwise** and it is based on a simple idea that has some surprising consequences.

It's obvious that we need to check that our DUT works correctly with every possible value of every parameter. That is probably not very difficult, as each test run can use a new value for **every** parameter and so we can work through all the possible values fairly quickly. This, though, doesn't deal with all possible parameter **combinations**, and we know from bitter experience that bugs will lurk in some specific combinations.

Pairwise testing is a compromise that tests every possible combination, not of all parameters – that would be impossible in practice - but of every **pair** of parameters. Kevin Johnston and Jonathan Bromley further explored this idea in a 2015 paper that looks at the theory of pairwise parameter choice through some simplified examples, and provides plenty of references to papers from the software world exploring the pros and cons of pairwise testing. They also had some fun building a prototype SystemVerilog implementation of pairwise parameter generation, making advanced use of macros to allow pairwise generation to be added to almost any SystemVerilog class. That generator code is probably not very useful in practice – there are many other pairwise generators freely available, and the paper references some of them – but it was an interesting exercise and allowed us to discover afresh that SystemVerilog macros are not all bad.

7. Bringing It All Together

To wrap up all this exploration of the challenges of verifying parameterized designs, it seemed natural for us to present a summary in the form of a short tutorial. That led to our contribution to DVCon-US 2020, **Parameterize Like a Pro**. You can find the tutorial materials (slides with notes) on the papers and presentations page of our website.

While you're there, take a look around at the other content. We hope you will find something to pique your interest. And don't hesitate to get in touch with us if you think we can help you with your next verification challenge, parameterized or not.