2021 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE
OCTOBER 26-27, 2021

No Country For Old Men – A Modern Take on Metrics Driven Verification

A simple mix of Python/C++/SQL/javascript/SV to improve our lives

Svetlomir Hristozkov, Graphcore, svetlomirh@graphcore.ai
James Pallister, Graphcore, jamesp@graphcore.ai
Richard Porter, Graphcore, richardp@graphcore.ai

***Abstract*— While Metrics Driven Verification (MDV) is universally accepted as the go-to methodology in verification, it has not kept up with the speed of innovation in exponentially more complex designs. The effectiveness of MDV is held back by the requirement to use SystemVerilog, a language not designed for large-scale data management. Innovation around EDA vendor-provided MDV flows are held back by the need to support legacy code, and they inevitably drive users towards reliance on proprietary APIs. In this paper we present an approach centred around an in-house Python implementation that allows us to fully utilise the power of MDV. We outline new workflows and extended tools which are enabled through this implementation as well as productivity and performance improvements. As a result of utilising this infrastructure we have measured in real life examples up to 40% code compute time performance improvement, up to 30% latency to result improvements as well as reductions in total code size.**

Keywords—metrics driven verification; methodology; productivity; coverage; tools; python; c++;

I. INTRODUCTION

The typical Metrics Driven Verification flow employed today usually consists of an engineer designing a Verification Plan based on an Architectural Specification. The Verification Plan is then translated to metrics collecting code i.e. functional coverage. The two are linked together, usually via a 3rd party EDA tool, and a final report produced and reviewed. This process is iterated until the team has enough confidence to tapeout the design.

If the specification were to change or mistakes were to be discovered, there are multiple changes to be made – in the Verification Plan, in SV coverage and in any EDA tool scripting linking the two together. This approach involves editing multiple files, in different languages and is time expensive and error prone.

We have tried to address that deficiency by going for a "single source" approach. Our design is centred on the use of a Python Golden Reference to build a Verification Plan, define coverage collection and design extended tools for reporting and reviewing results. This approach is outlined in Figure 1.

The Architectural Specification feeds into the definition of a "coverage model" in python. That includes motivations and descriptions of features to be verified as well as the actual coverage that will be verified. When run through our infrastructure, a reStructuredText (RST) output is produced and compiled into a PDF and HTML format for review. Note that the Verification Plan and Coverage Definition are produced (and can be reviewed) without having to define any of the coverage collection infrastructure and can happen well ahead of a testbench even existing.

Coverage is collected from a simulator via the DPI. A thin standardised and scalable C++ layer converts SV packets to python dictionaries which are then processed by individual coverpoints in the equivalent of a "sample" method. The connection between C++ and Python is done via the boost::python library and callbacks but other options are available also.

The backend infrastructure counting hits per coverpoint/cross is implemented in C++ for performance reasons and we will demonstrate that it is more performant and flexible than coverage in SV via an EDA tool.

The final results are uploaded to a relational database. The speed of SQL queries allows us to build multiple tools to better review and debug our data with some extended capabilities not currently present in solutions provided by EDA vendors.

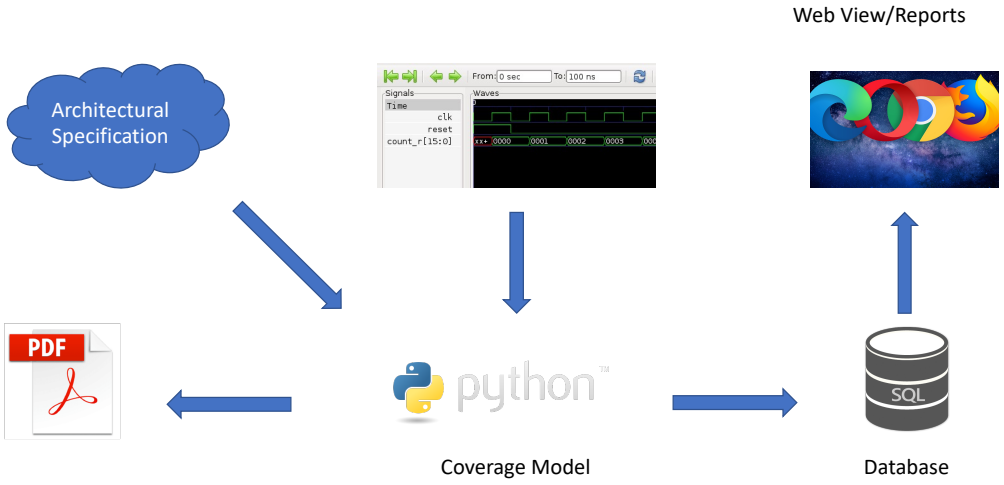

Figure 1 – Graphcore's Single Golden Source Approach to Metrics Driven Verification. The coverage model is at the root of verification documentation, coverage collection and visualisation.

II. New Workflows and Productivity Improvements

The fact that our coverage definition and collection is decoupled from a simulator means that some untraditional workflows become available.

A. Accelereated development of constrained random stimuli that hit our coverage goals

Coverpoints only require a python dictionary to collect data. Thus, we can generate stimuli entirely offline (ie outside of a simulator) and run it through our DUT reference models. Coverage can then be collected on the set of stimuli produced and a decision made on whether it's worth it running the seed in question.

The process in question is shown in Figure 2. The main difference with respect to a more conventional UVM-based setup is that the reference model is implemented in C++ and coverage collection itself in Python. The notional stimulus is generated offline and by querying the very same reference model whenever needed. Stimulus and constraints are first developed in the quicker EDA-independent testgen step and are later replayed within a real simulation.

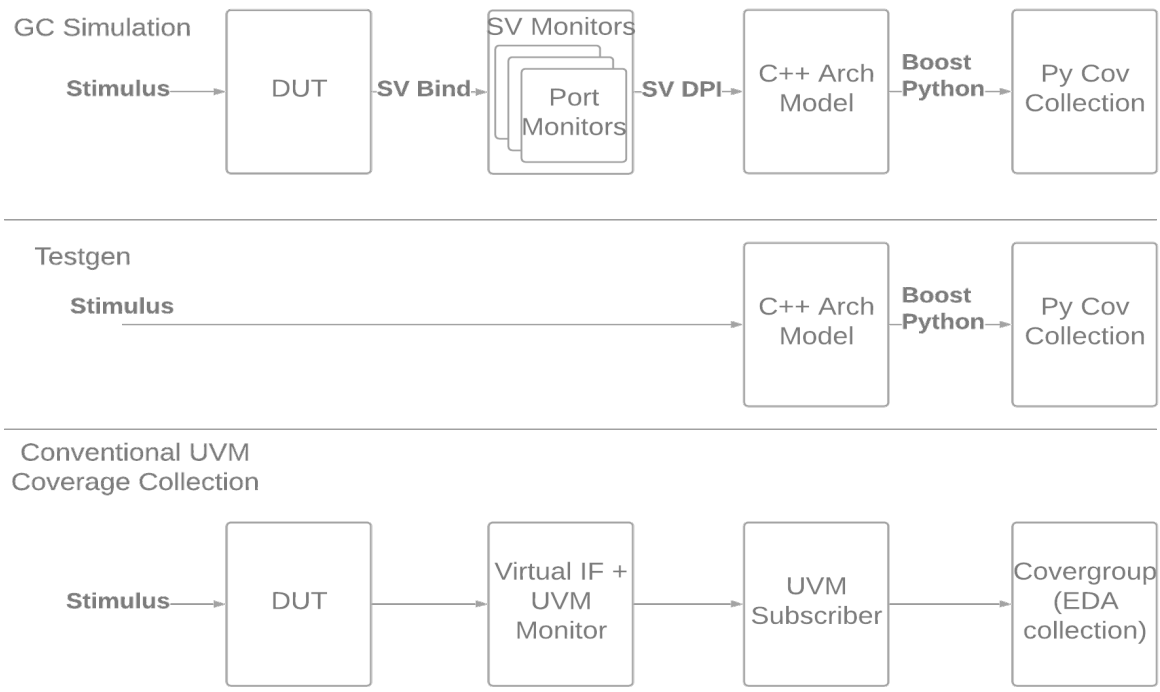

Figure 2 - Graphcore's distinct simulation and test generation steps compared to a more conventional UVM-based approach to coverage collection. Stimulus constraints can be tuned in the quicker EDA-independent testgen step before replaying on simulation

This greatly accelerates the development of constraints whilst closing down coverage at later stages of a project. This is shown in Table 1 below where the runtime of collecting coverage on the same test set is shown for a very real testbench within our current project.

Table 1 – Regression runtime when collecting coverage on the same 200 seeds on a simulator vs within Graphcore's testgen (offline) setup

	Simulation	Offline Test Generation
Runtime (s)	40493.0	3718.0s
Speedup	X1.0	X10.9

Generally, the slower a testbench is the greater the advantage of using this technique. The example above is a rather small testbench where only a relatively modest gain of x10 is realised but we have seen this number go up to x1000.

Additionally, when soaking seeds to make full use of compute hours available, it means we can only run seeds that are likely to hit state space locations we have not yet explored.

It should be noted that if the model is not timing accurate, this technique will only apply to coverage that does not rely on cycle accurate information – i.e. architectural coverage. If looking at particular signals or timing between events this is likely to be insufficient but there should still be a strong correlation.

B. *Unit testing coverage*

Functional coverage may often involve complex sampling functions that incorporate logic, preserve state etc. Having that sample method in Python allows the user to leverage a rich Unit Test Ecosystem. Conversely, only SV Unit comes close but still requires a simulation license, a compilation/elaboration step and is generally a lot more work to setup (quantified purely in terms of total lines of code).

C. *Defining and reviewing coverage in conjunction with the Verification Plan*

Ordinarily, a verification plan should be produced far ahead of any verification environment or any coverage having been defined. Coverage is only then, at a later point in the project, defined and mapped to the verification plan.

Since python has a wealth of libraries to interface and edit documents, one can produce verification documentation that is centred around a coverpoint. This is shown in Figure 3 below. The “pkt_cp” class defines the coverpoint by using Xpath to read the architectural specification. Extracted are field sizes as well as enumerations allowing us to define functional coverage for said protocol. Finally, the resulting page in the verification plan is shown. This can later be annotated with collected coverage data.

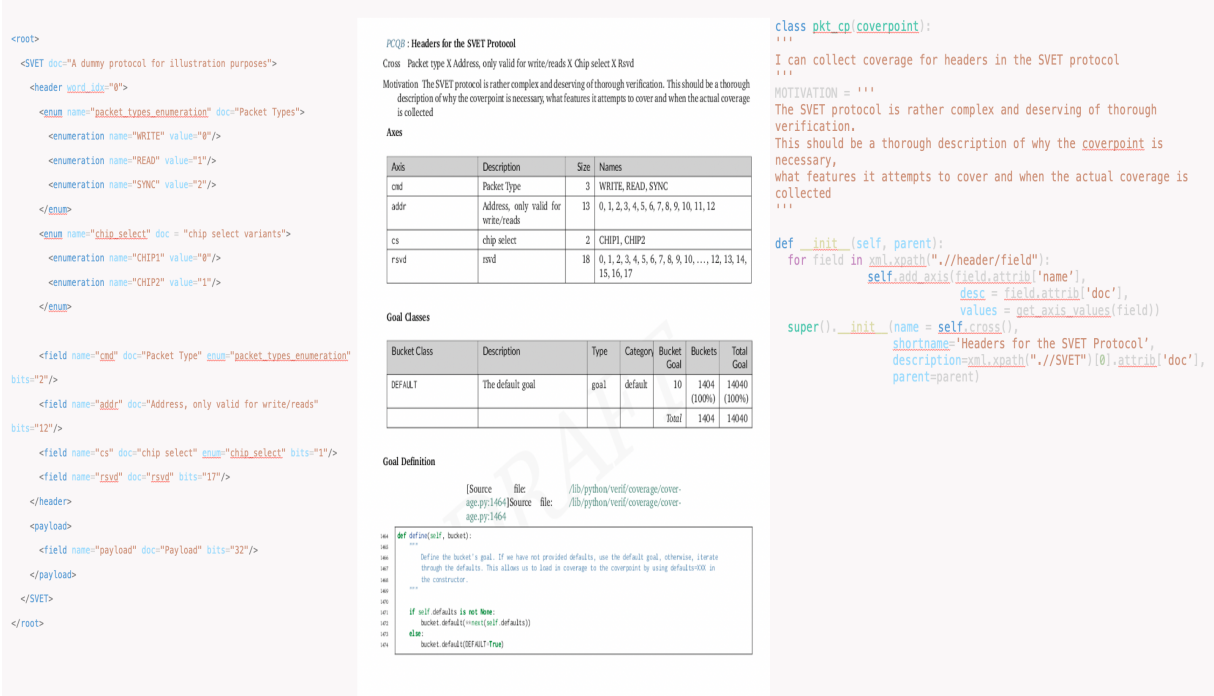

Figure 3 - An Example Architectural Specification, an associated coverpoint definition showing and the resulting page in the verification plan

D. *Maintainability of Verification Plan and Coverage*

Python is a widely used language and has a much easier time integrating with other tools. One particularly nice application is being able to process a machine readable description of IP and ensure there is less maintenance work in the future.

For example, a specification in IP-XACT or a proprietary XML format can easily be read out via the help of tools such as XML Xpath. Coverpoints can then use the information read to ensuring the amount of work to maintain verification models up to date with spec changes is minimised. A sample coverpoint defined via XPATH can be seen in Figure 3 - An Example Architectural Specification, an associated coverpoint definition showing and the resulting page in the verification plan.

III. EXTENDED TOOLS

The verification community has come to expect and rely on certain tasks performed by EDA verification management tools. For example, coverage merge in a regression is a pretty standard feature. That is naturally supported by our solution as well (in fact even with improved performance as outlined below). However, having

full control of the infrastructure has enabled us to come up with an extended set of tools to further improve our productivity.

A. Web Dashboard

Mainly used as a viewing aid, we have leveraged the speed of SQL queries and advancements in web technology to improve the user experience when viewing and comparing regression results. This is often difficult to do using conventional verification management tools due to capacity issues of file systems.

Figure 4 below illustrates a top-down view of the full hierarchy of a coverage model for a particular IP. The inner-most layer represent the top container (covergroup in SV terms). Each container can contain more containers or coverpoints. The outermost leaf nodes will always be coverpoints. Every node is colour-coded with the coverage collected for said hierarchy. A more conventional table view where individual bins can be queried is also available.

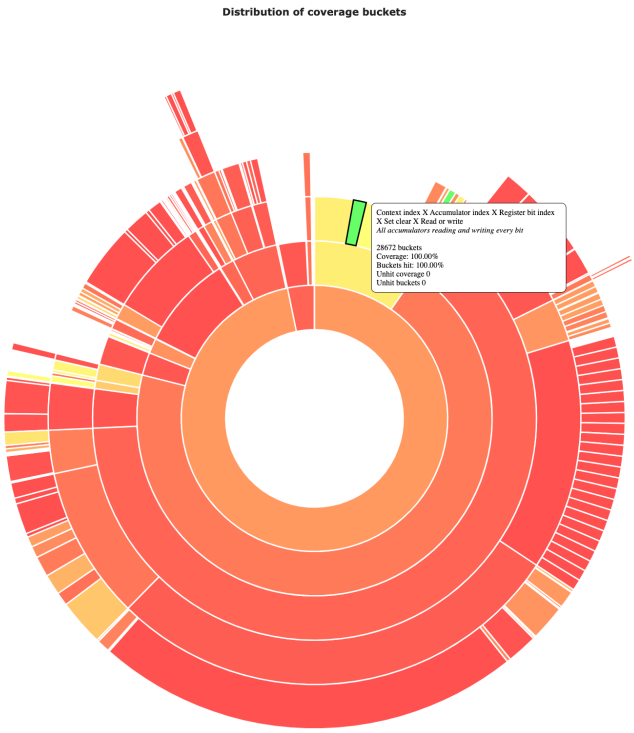

Figure 4 - Coverage "doughnut" outlining top-down view of the distribution of hits. Each layer represents a level of hierarchy and is colour-coded with collected coverage.

Figure 5 is mouse-over tooltip used to give a quick indication of how well a particular coverage container (or covergroup in more conventional verification terminology) is covered.

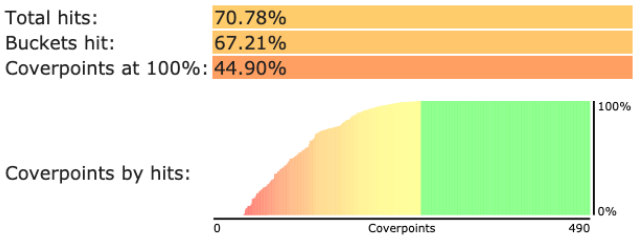

Figure 5 – Compressed bar chart illustrating the state of coverage for all coverpoints under a coverage hierarchy

These are just some of the many extended tools that can be custom built to suit the particular team's needs without having to rely on API/support from EDA vendors.

B. *Coverage Diff and Regression Fail*

While coverage merging is a very familiar feature – a diff may be less so. It allows us to produce a differential between 2 regressions outlining the particular bins that are seeing reduced coverage.

This can be further leveraged in CI runs - a run can be automatically failed if a particular coverage regression is introduced as a result of work on constraints.

IV. PRODUCTIVITY AND PERFORMANCE IMPROVEMENT RESULTS

A. *Coverage Pruning means simulations get faster further in the project to counteract diminishing returns of constrained random*

Coverage closure with constrained random stimuli typically exhibits a non-linear behaviour. It raises exponentially at first, but it starts to exhibit diminishing returns later on in the project. Conversely, the time penalty for instrumenting a simulation run with coverage roughly scales linearly with the number of bins being collected.

One of the undesired effects is that some crosses are fully hit but continue to "contribute" to the time penalty exhibited when instrumenting a simulation with coverage. We have built infrastructure that allows us to prime a simulation run with the results of a previous regression and "prune" away coverpoints that are already hit.

Figure 6 - Coverage Pruning Results

Shown in Figure 6 are the results of using coverage pruning on the same seeded regression. Coverpoints that are fully hit in a previous run are removed and no longer collect – thus the overall coverage is slightly lower with respect to the same seed without any pruning. However, the compute time is reduced from 46.6 hours to 38.2 hours.

Effectively, the pruning feature allows us to trade essentially useless data for performance. Additionally, it means simulations speed up the later in the project you are thereby counteracting the diminishing returns effect discussed above.

B. *Multi-threaded coverage runs reduce the latency to result when actively debugging*

Since coverage is collected via a call that goes through C++ and into Python, the verification engineer is in full control of all available cores on a run machine as opposed to when coverage is collected via the EDA simulator tool. This is shown in Figure 7 and Figure 8.

* Identify applicable sponsor/s here. If no sponsors, delete this text box (*sponsors*).

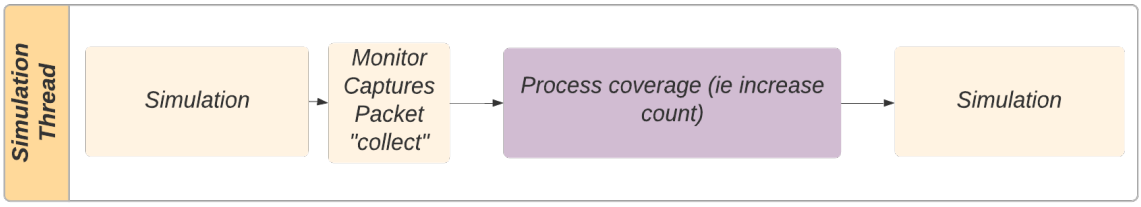

Figure 7 – A timeline diagram showing coverage collection using a single thread process

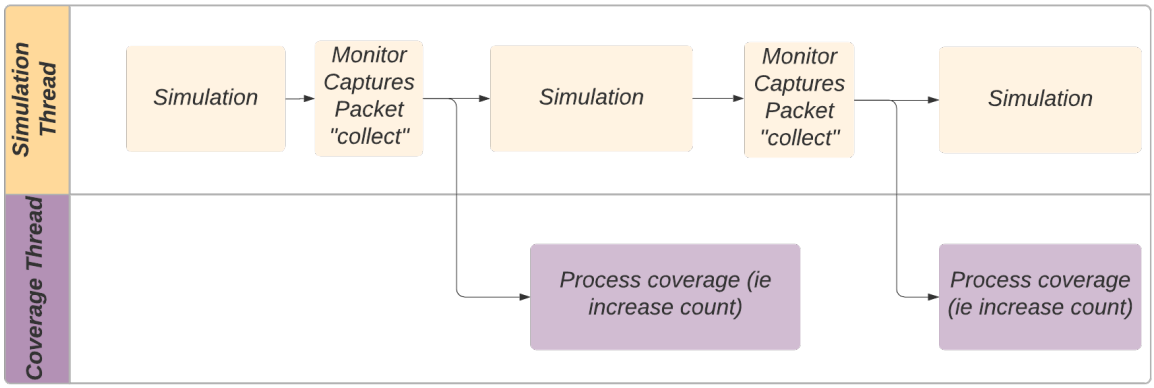

Figure 8 – A timeline diagram showing coverage collection using multi-threading

The penalty associated with running a simulation with coverage collection enabled stems from the sample method. This is the time to locate a particular bin within what is usually a rather large data structure of all bins and the incrementing the associated count. In an EDA simulator all of this happens under the hood and outside of a verification engineer's control.

Conversely, having implemented this in C++/Python, we can leverage standard libraries to spawn a separate processing thread. This way, one can call the sample routine and immediately return to the simulation leaving the coverage processing to happen asynchronously in the background.

Using this technique we can reduce the latency, or time to result, when actively debugging coverage results. Real world results from a Graphcore testbench are shown in **Error! Reference source not found.**.

Table 2 - Summary of a Execution when using Single vs Double Thread and Executing Coverage

Type	System Time to Result(s)	Processor Load Average
Single Thread Execution	322.3	0.98
Double Thread Execution	234.7	1.41

The example above achieves a latency improvement of about 30% and this roughly correlates with the Processor Load Average observed. The benefit of this technique will vary depending on how good a test is at hitting coverage – the more sample calls that are made the better the results will be.

Note that the time to result is different from total compute time. This technique can give lower latency when debugging interactively but does not impact total compute requirements.

C. Productivity improvements

One of the big productivity gains stems from the fact that Python as a language is far more expressive than System Verilog. For example, defining "illegal bins" or "don't care bins" is less verbose, more readable and easily unit-testable in Python.

```
def define(self, bucket):
  '''Make every 5th and 6th value of the address coverpoint/axis illegal. All others are legal'''
  if bucket.axis.addr % 7 in [1, 2]
      bucket.default(ILLEGAL=True)
  else:
      bucket.default(DEFAULT=True)
```

Figure 9 - Sample Programmatic Definition of Bins in Python

Shown in Figure 10 is the bins definition of an arbitrary specification for a coverpoint. It illustrates how the expressiveness of the Python is very well suited to defining arbitrary logic. In fact, illegal/don't care definitions can even be fetched from the Architecture definition.

V. CONCLUSIONS

A new take on Metrics Driven Verification was presented. It demonstrated that the Metrics Driven Verification flow can be made flexible in order to improve on productivity of both people and machines used in the verification of a design.

Using the demonstrated flow yield EDA independence which means there would be no work involved in porting between vendors should a team decide that is in their best interest as well as enable simulators outside of the "Big 3" e.g. Verilator.

It attempts to address some of the common grievances involved in the verification process. Namely, the need to maintain multiple documents and look after an ever changing architectural specification.

We demonstrated how this methodology can enable new workflows such as running stimulus on an architectural model, collecting coverage and debugging constraints all without launching a simulation.

It was further demonstrated how being in full control of the coverage infrastructure can enable the user to optimise performance further and develop debug and aid tools not readily available as part of EDA packages.

Finally, we've observed a productivity improvement in terms total code produced that is due to the expressiveness of Python and C++ for the application at hand.

All of this, naturally, comes at a cost. Namely, the need to maintain and develop all of this extra infrastructure. Additionally, experienced verification engineers have often only had limited exposure to C++/Python which means switching to a new verification methodology introduces a learning curve.