Efficient Bug-Hunting Techniques Using Graph-Based Stimulus Models

Nguyen Le, Microsoft Corp: ngls@microsoft.com
Mike Andrews, Mentor Graphics Corp: mike_andrews@mentor.com

***Abstract*- This paper describes how the emergence of graph-based stimulus modeling, as an alternative to constrained random generation, can be leveraged to apply certain bug hunting strategies in a simple and automated manner. These strategies, should, if the theoretical benefits are realized, provide verification engineers with a way to improve their bug detection rates while improving the efficiency of their regression runs by one or more orders of magnitude.**

The paper combines a brief introduction to graph-based stimulus with some examples of promising bug hunting strategies, and then describe a series of experiments run on a real design to see if these strategies deliver the promised efficiency improvements in that case.

I. Introduction

Traditional verification projects have, in recent years, employed SystemVerilog constrained random stimulus for early bug hunting, once the RTL for the device under test is mostly functional and simple directed 'sanity' tests are working. The expectation is that the random stimulus will broadly exercise the functionality of the DUT and quickly find a fairly large number of defects. This process generally occurs well before coverage goals are determined and implemented in SystemVerilog functional coverage models.

The availability of more advanced stimulus description languages and tools has introduced more effective options for this early bug-hunting procedure, allowing the initial defect discovery rate to be increased significantly. This can obviously result in valuable time savings if more bugs are found at the block level before system integration begins in earnest, and especially if they are found very early.

Graph-based stimulus models, that have the ability for their random generation to be guided based on user-defined goals, have been used quite effectively for functional coverage closure since such tools & methodologies have been available. Recent growth in the use of graph-based stimulus in general has also generated interest in defining a new domain-specific Accellera standard, which has been given the name 'Portable Stimulus', since it is independent of any particular verification language. As usage gains traction in the industry, new applications are, of course, being devised. For example, users of such models have found that the same technology can be used to create far more efficient bug hunting strategies that are based on guiding the random distribution of the stimulus space to follow known good strategies, such as the method of all-pairs (aka pairwise) testing.

Pairwise testing is a common technique in the software realm, but it can also be applied to hardware testing, where, in SV/UVM functional verification terms, applied to an interface, it would effectively be the crosses of each sequence item field with each other field.

Another technique that has been employed is the creation of very large cross coverage strategies which would target the cross of all fields in an item, using suitable binning to bring the number of combinations to a manageable size - perhaps in the 7-8 figure range, which is well within the reach of graph-based techniques.

A third technique uses strategies that target transition coverage of key item fields or groups of fields, helping to expose bugs that are dependent on specific sequences of activity.

This paper describes these techniques in more detail and also presents case studies and actual data on their relative effectiveness and cost in terms of time and other resources.

II. Graph-based Stimulus Models

Graph-based stimulus is a primarily declarative format that describes legal stimulus scenarios in an abstract manner. It combines elements of Backus-Naur Form (BNF) with other constructs more familiar to verification engineers such as algebraic constraints, object hierarchy, variable typing, and the binning of variables into values or ranges. The specifics of any syntax, which differ across the various graph-based languages in the industry, are not important to this paper. The key difference exploited here between the graph-based description and traditional stimulus models comes from the BNF nature, where the graph, like the BNF for a language syntax, can be used to

define the rules for what constitutes a legal example of a scenario (or a syntax compliant program in the language case).

Figure 1 shows a simple example of a rule and its graph representation. In the absence of any defined standard, the language syntax used is borrowed from the Accellera Portable Stimulus working group, and is merely intended to be illustrative of the concept, in preference to using one or more available proprietary languages. For easy recognition, keywords of this language are colored red.

In the example, an object called an **action** is given the name scenario. The action object contains instances of a lower-level object that are of type pkt, with instance names pkt1 and pkt2. It also contains a declaration of its topology, specified here as its **graph**.

The **repeat** keyword declares that the scenario is to continue performing one of the options specified, and each time through the loop one of the paths is **select**ed.

One of the path options is to populate the pkt1 instance, with consideration to an inline constraint {flags == 0;}, followed by populating the pkt2 instance with a different inline constraint {flags != 0;}. The other path option switches the inline constraints between the two instances.

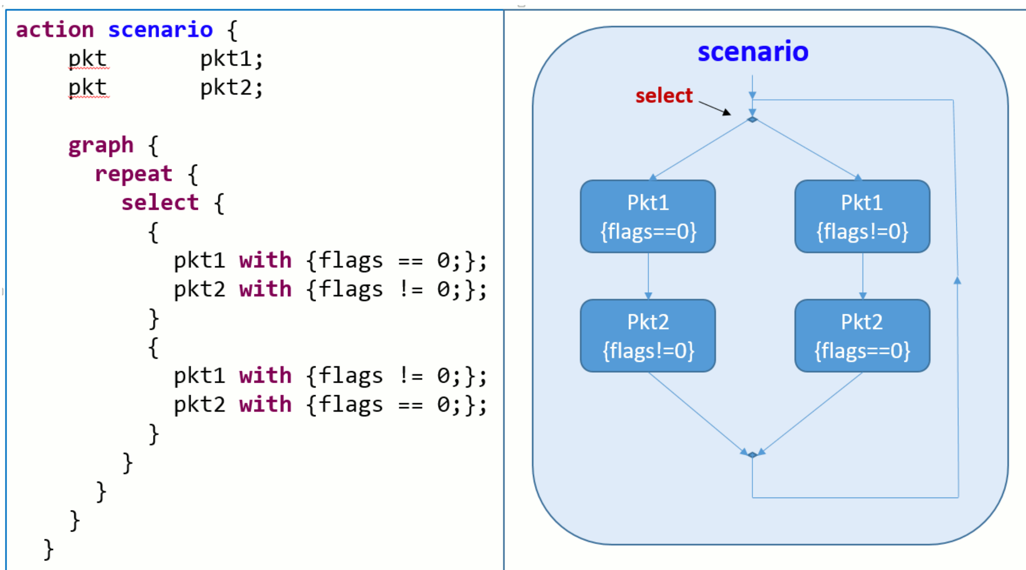

Figure 1: Example Rule Graph

What it means to 'populate an instance' requires that we look a little deeper into the hierarchy. Figure 2 show the definition of the object named pkt.

```
struct pkt {
    rand bit[11:0]       sz;
    rand bit[7:0]        payload[1500];
    rand bit[3:0]        flags;

    constraint {
      if (flags == 0) {
        sz <= 256;
      }
    }
  }
```

Figure 2: Example of a Randomize-able Graph Object

This example uses the keyword struct to define an object of type pkt, that, much like a SystemVerilog class can contain at least typed fields and constraints. So, in this case, to populate one of the instances of pkt means to assign

values to its rand fields, obeying the internal and inline 'with' constraints. This is very familiar to SystemVerilog constrained random users of course, analogous to the class.randomize() call, with an optional 'with {..}' clause.

The action called scenario is therefore conceptually like a SystemVerilog sequence, except that it is actually many possible sequences, all of which must follow the declared rules.

III. Coverage Goals as an Input

Functional coverage is normally considered to be an independent tally of the effectiveness of a set of stimulus, normally used with constrained random stimulus generation. Recent advances in tools developed to accelerate coverage closure have employed coverage goals as an additional input to the stimulus model, with the intention that this will impact the stimulus generation process in some way.

The coverage goals expressed in this way can be high level, i.e. "capture the data flow through the chip and the interaction between the various IP blocks and subsystems that make up the overall chip" [1]. Or these goals can resemble the common SystemVerilog functional coverage covergroups, with coverpoints and crosses on random fields in TLM stimulus objects or on signal values detected in monitors that probe into the state of lower level RTL signals.

Given this information, there are various techniques used by tools that help with achieving coverage closure. These techniques depend on the ability for an application to 'walk the graph' and then to apply the scenario defined by that traversal. The way in which the graph is traversed again differs depending on the implementation, since it could be random or systematic, and the graph itself could have loops or recursion or just be a simple tree.

As an example, figure 3 below shows a graph that defines the behavior of a UART. This graph could be traversed continuously, and, as long as the lines and arrows are followed, the behavior exhibited stays within the specification. In this case there is no definite end, and the start node is only visited once.

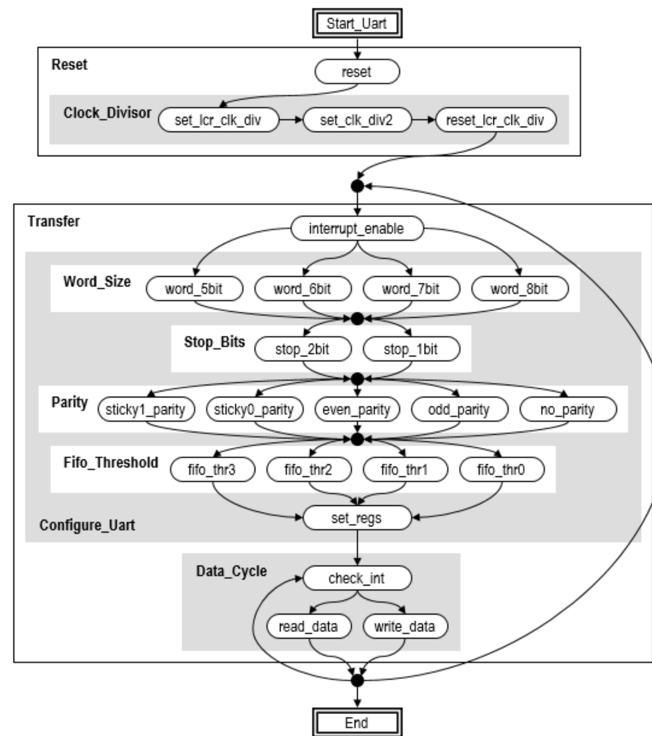

Figure 3: Graph Defining the Operation of a UART

If an application could keep track of the paths it has traversed through the 'Configure_Uart' section of the graph, then it could always choose a unique path, with 160 options to choose from.

The tree form would, in contrast, always start at the top and end at a leaf, with each traversal corresponding to a test.

For the purposes of aiding the user with the task of coverage closure, there is an assumption that an application can somehow keep track of which scenarios have been applied, thereby enabling the desired goals to be run through in some systematic fashion, either via automated processes or manually. Note: Even if the closure is assisted by manual intervention, the tools involved still typically provide value in providing the definition of the desired scenario and or by automated generation of the code required.

If the graph language used includes both rules on sequencing and syntax elements to define algebraic constraints, then walking the graph can include solving for consistent values of any randomizable quantities that are relevant to the scenario description. One of the benefits of this combination of both scenario description and constraints is that a solver can benefit from more context information, allowing it to know which of a set of variables is currently relevant. This means that the model already implicitly defines the interesting and legal state space, whereas a traditional constrained random model does not know which of its random variables are of relevance. This is often very helpful in applications such as instruction set generation where a "SystemVerilog Constrained-Random (CR) 'single class with constraints on all possible fields approach' can be unwieldy and hard for solvers to handle" [2].

IV. The Emergence of Graph--Based 'Portable Stimulus'

The use of random stimulus for early bug hunting in a verification project has been prevalent for many years, and is now over a decade old [3]. Of course no paper on verification methodology can get away without men-tioning the inevitable increase in complexity of the verification problem, and so it is not surprising that many in the industry are claiming that that a newer approach is now needed. This is of course a significant force behind the recent interest in standardization of a new domain specific stimulus model, which is being termed "Portable Stimulus" or sometimes "Graph-Based Portable Stimulus"; since the main requirement is that such a model is independent of any specific verification environment. "With this proposed standard, user companies will be able to specify the behaviors once, from which multiple implementations may be derived." [4]

V. Bug Hunting in a Typical Flow

Any advanced verification environment will typically use some amount of randomization, and in the early part of the verification project this may be over-constrained in order to produce a basic sanity test, or it may be under constrained if some limitations and details of the design specification are not yet finalized, or if the full stimulus model itself is still in development. In either case, one or more simple random tests can be created that evolve with the device under test (DUT), and this will provide a way to validate new drops of the DUT with a reasonable number of vectors.

As the complexity of the variables plus the constraints increases, this does however suffer from the same redundancy issues that are seen later in the verification project, when coverage closure becomes the main concern [5]. If the state space is very large, and the constraints complex enough to skew the random distribution, then while the size of the net that is cast may be large, there may be many significant holes that diminish the effectiveness, leaving many bugs undetected until the work of coverage metrics definition is underway. Of course that process in itself can be time consuming and has its own series of iterations through the code -> test -> debug -> modify loop.

The solution to this, as with final coverage closure, is to augment the automated stimulus generation with manual tweaks and some number of highly directed tests.

It is also common for a project to take advantage of verification IP that comes with various stimulus options, such as a variety of sequences that together would give a broad coverage of a particular interface and its protocol. It is often the case though, that these 'off the shelf' sequences are created to cover the full specification of the interface, and may take some time to constrain down to the needed subset for a particular application. Furthermore, the built-in sequences, setup to spread the effort and time over the broader area, may not probe deep enough into the areas that are critical to this particular DUT, again requiring some modifications.

VI. Bug Hunting Strategies Beyond Pure Randomization

Obviously the best verification process possible would be to cover the exhaustive set of possible DUT inputs, which in SystemVerilog terms would amount to crossing every variable together, with all possible values in their own bin. However, just as obviously, exhaustive coverage is impractical except in the most trivial of cases for most modern projects that require automated stimulus generation. Reducing the sheer quantity of the goals therefore, while maximizing the chance of uncovering bugs, is the subject of much research.

One simple approach I have seen employed is to attempt to at least cover all possible values of key control or configuration variables, using perhaps the SystemVerilog 'randc' modifier. Use of 'randc' however can bring in other issues for the randomization process (see page 4 of [6]).

Another scheme that has been suggested as a good compromise is to use a pairwise approach, where each variable in a random set of stimulus is crossed with each other variable. The theory has been explored quite extensively in the software domain, where in one study a very high percentage of failures were triggered by the interaction of the values of two variables, as high as 97% [7]. There have also been attempts to layer this capability onto SystemVerilog, at least for a limited scope of application. The level of benefit gained from this approach is obviously very application dependent, but the assumption that seems to hold true in most cases is that such an approach is: a) far fewer combinations than the exhaustive set, and; b) more likely to uncover more bugs than the same number of purely randomly selected combinations across a variable set.

For example, in one analysis, the pairwise approach applied to 5 fields of an 8-bit register amounted to only 18 combinations when crossing field 1 x field 2, field 1 x field 3 etc., when the exhaustive set is, of course 256 values [8].

A variant on the pairwise approach uses binning of the large variables to reduce the state space to well below the exhaustive set, while still crossing all the key variables that the verification engineer suspects will have an impact.

In the last two cases, the number of unique combinations of input variables needed to achieve the goal of the strategy can be tuned by the selection of variables to include, and also by the coarseness of the bins defined for those variables. In both cases, the unique combinations applied are typically 2 or more orders of magnitude higher than the pure random approach. It is the intent of this paper, as previously stated, to generate real-world data on how that translates to verification effectiveness, using a combination of bug detection rate and code coverage metrics as the metrics being recorded. The paper will also give an indication of the effort required to employ these strategies, with the assumption that a pure random run requires no additional coding effort from the engineer, and that must also be considered.

In the experience on one author, a good companion strategy to the pairwise approach is to also consider some sequential interaction, by defining the scenario description in a rule graph to be a pair of transactions, in which case, the transition coverage of certain variables can be targeted.

An example is shown below in figure 4, where two instances of a transaction follow each other, with an additional scenario variable 'DELAY' in between.

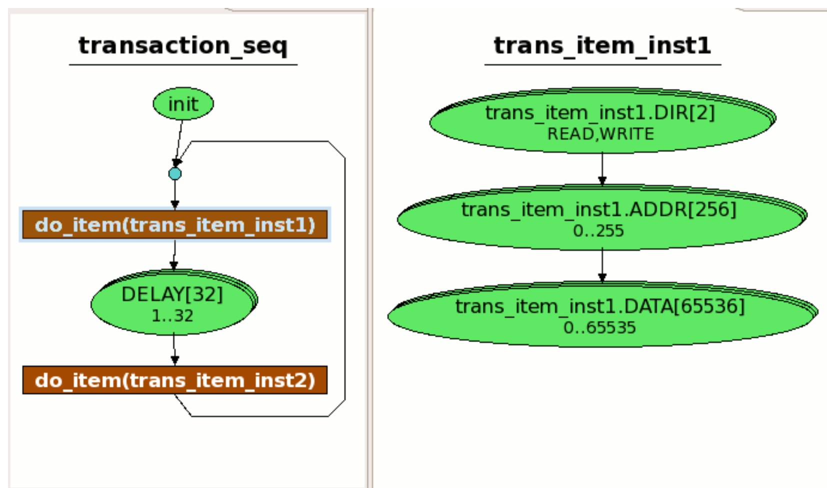

Figure 4. Graph Defining a Sequence of Two Transactions with Delay

The graph in this case starts with an initialization step, then repeats pairs of transactions with a value of DELAY between 1 and 32 inserted in between each. On the right of the figure is a sub-graph, showing the three fields of the first instance. In this example, there are seven total choices in the scenario, all of which can be independently made each time around the loop. The scenario could be enhanced by constraining one of the transactions, for example the constraint '{trans_item_inst2.DIR == READ}' means the second transaction is always a read operation.

VII. Overview of the Design and Methodology Used

In this case study, we are using a design for a secure compute block. This block is used to encrypt data before sending out to the outside world (data center) so that no one can see its content. On the way back from the outside world, data is decrypted and sent back to the local server. The design supports four different packet formats and a separate configuration set for each logical flow.

There are two sequences used to mimic traffic from the local network server to the data center network and back. Each sequence attempts to mimic the standard work load from ten logical network flows. The number of network flows is a balanced tradeoff between design concurrency stress and simulation runtime.

Note: the original verification project started with general random stimulus, and when the testbench self-checking had become fully functional we switched to graph-based stimulus for full cross of key fields.

Code coverage was used as a key metric of test effectiveness, and will be used in this paper as a reasonable proxy metric for bug detection rate in this case. The existing RTL has been stable in the field for quite some time now, so repeating the final 100% code coverage would demonstrate high confidence of the test suite.

The graph-based tool used in this experiment is Questa inFact from Mentor Graphics.

VIII. Validating the Effectiveness by Experiment – Setting the Baseline

The theoretical benefits suggested earlier in this paper have been tested by experiment using the following approach.

The secure compute block design described in the last section, and its original testbench was used in this experiment, with tests focusing on exercising a specific subset of the DUT code.

The baseline we will use for comparison is a test that randomizes the contents of two sequence items within the two traffic sequences previously described.

Each sequence item has approximately 50 random fields, and the exhaustive legal space is, as is typically the case, a number longer than 100 digits.

The baseline random regression run five parallel simulations, and in each simulation the random sequences used generate and apply, on average, six hundred and seventy five packets to their associated interfaces. To gauge the effectiveness of this run, a code coverage report is generated for the subset of the DUT code that is exercised.

The overall code coverage reported for this initial random regression is 95.9% (by files) and 97.6% (by instance). The area with the least coverage is of the type Finite State Machine (FSM) Transitions, which is a key indicator of how thoroughly this run has exercised the functionality of the DUT. The coverage reported for this category is 85.8%.

In this case there are a handful of variables that we predict, from an understanding of the DUT, to have the most correlation with broader coverage, and these are corrupt_data_en, length, m_dtls_type, m_seq_flow_type, table_index & m_Flags. The range of legal values for these variables varies from a 0 or 1 setting for a single bit, up to 1,388 for the length field. The full cross of all legal values would be ~100k combinations, requiring at least that many packets to be simulated to cover them all. Statistically, a typical constrained random simulation would take more than 1.2M (N x ln(N)) random packets to cover these. For the purpose of these experiments some bins have been defined for the larger variables to define a cross coverage goal of about 10% of that size (10,800 bins), with the expectation that full code coverage is likely to be achieved by applying all of these combinations.

IX. Migrating to a Graph-based Stimulus Model

With a baseline established, corresponding to the traditional methodology used, the first experiment is to construct a graph-based stimulus model that can assign legal values to the fields of the sequence items used (in this case it was automatically translated from the source SystemVerilog code). The stimulus model is then extended to incorporate the coverage goal just described, and is integrated into a new SystemVerilog sequence as an alternate to the original SV constrained random one. A new regression can now be run, also using five parallel simulations, where the ability of the graph model to systematically apply each of the 10,800 required combinations is leveraged. In this environment, these are applied in a pseudo-random manner, roughly split equivalently across the five simulations, using a dynamic auto-distribution capability.

The same code coverage report is generated and compared to the baseline. The total code coverage is measured at 100% across the board, with a total of 13,666 packets being applied across all the simulations to achieve this. This first experiment therefore represents the best achievable coverage using the intelligence of the verification engineer

to determine the key random fields, and the appropriate binning for these. This strategy will be named the 'BigCross' strategy.

X. EXPERIMENTS IN FURTHER EFFICIENCY IMPROVEMENTS

For the second experiment, the goal to cross the six key variables, and the manually selected binning, is replaced by a strategy that doesn't target any cross combinations of the item fields, and uses a default binning strategy. The default binning strategy in this case is to have one 'edge' bin for the lowest and highest possible values for the field, and then divides the reminder into a maximum of 256 bins, resulting in 258 bins for the largest variable, and for all other variables every legal value is considered. Note: this is conceptually akin to how 'randc' is sometimes used in SystemVerilog, but without many of its limitations. We will call this strategy the 'Fields' strategy.

Once again, five parallel simulations are run, and the code coverage results and number of packets applied is recorded. In this case we see total code coverage of 97.1% (by files) and 97.8% (by instance). The area with the least coverage is again of the type FSM Transitions, with coverage of 86.7%. All of these figures are slightly higher than our initial random baseline, but the total number of packets simulated is only 439, which is about 13% of the packets applied in the original constrained random sequence. So the use of the 'Fields' strategy, gave us slightly improved coverage with approximately 1/8th of the simulation effort, and, equally importantly, required no additional insight into the DUT or any manual testbench code development.

As mentioned previously, the coverage category 'FSM Transitions' is an important one, since it reflects how thoroughly the DUT logic has been exercised. That metric is still below 90%, so we should probably not be content at that level, since the priority here is to find the most bugs we can at this early stage in the verification cycle, while still allowing rapid regressions of the frequent code changes.

The third experiment run will test whether the pairwise approach is, as the research generally shows, a highly effective compromise between the first two strategies. In this case, each of the six key fields is crossed with each other one in a pairwise manner, resulting in this case twenty five individual cross coverage goals of various sizes. Initially, we elected to try setting the maximum number of bins to 66 (2 'edge' bins, plus 64 distributed across the remainder). The largest of the pairwise goals in this case turned out to be 660 combination bins. Note: this is about 2.5x the largest individual goal in the 'Fields' strategy, so we should expect the same proportional change in the number of packets required to be applied.

For this experiment the result achieved were: 98.9% (by files), 98.1% (by instance), and 96.4 for our FSM Transitions category. The total number of packets applied was 1,026. So this experiment would tend to show that, at least in this case, the pairwise approach yields us a significant coverage improvement of 10% for the key metric, with 30% of the effort expended in the baseline approach. This strategy we will name the 'Pwise' strategy.

The fourth experiment we ran raised the maximum bin limit for the pairwise approach to 258, resulting in the largest cross combination being 2,580 bins. The results for this experiment were: 100% (by files), 99.8% (by instance), and 100% for the FSM Transitions category. The total number of packets applied was 4,440 in this case, meaning that it yielded practically full code coverage with only 33% more effort than the baseline random approach, once again agreeing with the hypothesis that, in absence of any specific functional coverage goals, a pairwise strategy is highly effective as a bug hunting technique.

The pairwise approach is of course just a specific case of the more general 'ntuple' concept with n=2. Using the same automated approach a fifth experiment was run with n=3, and the maximum number of bins per field reduced once more to 66. The largest cross combination in this case, with 3 fields included, totaled 1,320 bins. The code coverage results achieved from this strategy were: 99.7 (by files), 99.8 (by instance), and 100% FSM Transitions coverage. The total number of packets applied across all five simulations was 2,682. We named this strategy as 'Triples'.

These results are summarized in the table below.

Table 1 – Strategy Comparisons vs Baseline

Strategy Name	Maximum Bins/fields	Largest Goal	Num Packets	Total (Files)	Total (Instance)	FSM Trans
Rand	N/A	N/A	3371	95.9	97.7	85.8
BigCross	N/A	10,800	13,666	100	100	100
Fields	258	258	439	97.1	97.8	86.7
Pwise	66	660	1,026	98.9	98.1	96.4

PwiseLrg	258	2,580	4,440	100	99.8	100
Triples	66	1,320	2,682	99.7	99.8	100

Broader conclusions will be described later, but there are already some straightforward observations that can be made from this.

The relative efficiency of both the 'Fields' and 'Pwise' strategies vs the original random approach shows the fundamental benefit of the systematic graph traversal capability.

Secondly, the improvement in coverage between the Pwise and PwiseLrg strategies is to be expected in this case. The value for the maximum number of bins primarily affects the binning of the packet length, and in this type of network application we know that the design is very sensitive to packet length due to the encrypt/decrypt operation and alignment at the packet tail end. In the original DV project, we spent a lot of time trying to cover the different packet lengths with the following criteria

- Min size packet and the next 16 increment
- Max size packet and the next 16 decrement
- All values equally spaced in between
- The total bin count is 151

This clearly show that the original Pwise Maximum Bins setting of 66 is an under-sampling of the input space for this critical variable. The PwiseLrg is a better fit with 258 bins => this is almost 2x the ideal hand-crafted bin. Loosely speaking with this double sampling we almost satisfy Nyquyst's criterion.

XI. CONSIDERING SEQUENTIAL DEPENDENCIES

The experiments performed so far rely on little manual effort being applied, with each taking only a few minutes to setup, and full automation of each possible via some simple scripting. None of the strategies applied so far though consider the possibility that the code coverage metrics being used might have some dependence on the transition coverage between certain sequence item fields.

The final experiment we conducted took further advantage of the descriptive capability of the graph-based model, extending the scenario from the simple application of a single sequence item, to two consecutive items. This can simply be achieved by declaring two instances, say 'm_item' and 'm_item2' and placing these one after the other in the graph's repeat loop.

Since our strategy goals can span all objects in the graph model, we can form a cross combination strategy that encompasses fields from each instance. For this next experiment we presume that the most likely fields to exhibit sequential dependencies are the length and the two 'type' fields. We therefore combined the six graph fields - m_item.length, m_item.m_dtls_type, m_item.m_seq_flow_type from the first instance and m_item2.length, m_item2.m_dtls_type, m_item2.m_seq_flow_type from the second – to form a six-field 'transition' cross coverage goal. To keep the total number of bins to a reasonable value we tried three variants where the length field in each instance was divided into 12, 7 or 5 bins, making the goal size 2,304, 784 and 400 respectively.

The results of these three variants are recorded below in a similar table, with the 'Rand' baseline used for comparison.

Table 2 – Transitions Strategies vs Baseline

Strategy Name	Number of Length Bins	Largest Goal	Num Packets	Total (Files)	Total (Instance)	FSM Trans
Rand	N/A	N/A	3371	95.9	97.7	85.8
Trans_12	12	2,304	8751	99.9	99.8	100
Trans_7	7	784	3,033	99.7	99.8	100
Trans_5	5	400	1,508	99.6	99.6	99.1

Once again, the more targeted stimulus generation clearly outperforms the basic random approach, at least for the code coverage metric we are using.

XII. CONCLUSIONS

We can draw three simple conclusions from this analysis. The first is that a verification engineer's insight into the design is still the best way to get full coverage, i.e. 100% across the board, at least in the context of this methodology. Typically that insight is expressed by the designers and verification team in terms of various functional coverage metrics. In this case, it is expressed in terms of the key variables selected to be included in the coverage strategy used with the graph-based stimulus model, along with the manually crafted bins.

The second conclusion we can draw is that, at least in this case, there is significant efficiency to be gained by applying some basic automated strategies, such as the simple 'Fields' strategy. This can be used, for example, as a very quick sanity test when making frequent changes to either the DUT or the testbench, leaving the larger strategies for longer runs, such as a nightly regression.

The third conclusion is that, again in this case, the theorized efficiency of the pairwise strategy seems to be realized here, with only a minor difference in coverage from the full cross of all six variables. Also, we can see that dialing up the maximum number of bins gives an easy way to trade off time taken versus the coverage achieved.

From a verification engineer's perspective, the flexibility in verification time/quality metric can be a powerful asset in DV project planning/execution in general. Since we can run with the shortest strategies and improve our stimulus as DV team gains deeper knowledge in the design. We can also use Pwise to expand our search into other variables in the input spaces that might be too expensive to explore in the full cross.

XIII. FURTHER EXPLORATION

It would be interesting to also experiment with combinations of these strategies. Verification management tools could be used to merge and look at the combined coverage results of a combination such as the smaller pairwise and the medium sized transition coverage strategies. Since their application can be automated once the graph based stimulus model is created, two or three of the most efficient ones can be determined early on in the verification project, and then applied independently to suit the need at hand, and perhaps combined to form the nightly regression test suite.

XIII. REFERENCES

[1]. Adnan Hamid. Complex Chip Designs Need System-Level Scenario Coverage. Electronic Design, December 2014.

[2]. Staffan Berg, Mike Andrews. A New Stimulus Model for CPU Instruction Sets. Verification Horizons, November 2015

[3]. Janick Begeron & others. Verification Methodology Manual for SystemVerilog. 2005

[4]. Portable Stimulus Working Group. Scope Definition. http://www.accellera.org/activities/working-groups/portable-stimulus

[5]. Matthew Balance. Trading Cards and the Art of Verification. Electronic Engineering Journal, January 2011.

[6]. Ahmed Yehia. The Top Most Common SystemVerilog Constrained Random Gotchas. DVCon Europe 2014.

[7]. D. Richard Kuhn, "Practical Combinatorial Testing" http://dx.doi.org/10.6028/NIST.SP.800-142

[8]. Johnathan Bromley, Kevin Johnston, "Is Your Testing N-wise or Unwise? Pairwise and N-wise Patterns in SystemVerilog for Efficient Test Configuration and Stimulus". SNUG 2015.