GraphCov: RTL Graph Based Test Biasing for Exploring Uncharted Coverage Landscape

Debarshi Chatterjee, Spandan Kachhadia, Chen Luo, Kumar Kushal, Siddhanth Dhodhi
Nvidia Corporation
Santa Clara, CA - 95051

***Abstract*- Cover Property (CP) can be used to describe functional coverage statements within Register Transfer Logic (RTL) design code. In this paper, we propose GraphCov - a heuristic for hitting CPs which remained uncovered in end-of-project coverage reports. To achieve this, we bias a set of tests towards a CP that was hit before (albeit infrequently) and is most closely linked to a cluster of unhit CPs in the RTL design. We parse RTL code and extract signal driver information. This is used to generate a Trace Driver Graph (TDG) which encapsulates a high-level abstraction of how each CP is driven by signals in the design and how each of these signals are in turn driven by other signals. Based on that, we compute association scores of each Covered CP (CCP) to other Uncovered CPs (UCPs) in its neighborhood. These scores eventually help us decide which CCPs are ideal candidates for test-biasing to maximize the likelihood of hitting UCPs. We tested our methodology on a GPU unit level design at Nvidia. GraphCov based test-biasing (using pre-existing runtime command-line arguments in testbench) was able to hit UCPs which remained uncovered at the end of a project.**

I. Introduction

Coverage closure is undoubtedly one of the most tedious phases in Design Verification (DV). At the beginning of the project, as we run more and more Constrained Random tests, the coverage score gradually increases. Towards the end of the project, the coverage numbers usually asymptotically converge to some number less than 100%. Going this last mile is the hardest part, since at this point, running more and more tests do not move the coverage needle substantially. At this point, DV engineers usually go through the tedious process of writing directed tests or fixing TestBench (TB) bugs to fill coverage holes. Due to time constraints, sometimes uncovered CPs are waived when the risk of coverage hole caused by such waivers are deemed reasonable. The question we want to ask is: Is there an automated way in which we could hit at least a subset of CPs that were uncovered at the end of a project?

II. Previous Work

Considerable prior work exists on accelerating coverage closure for Simulation Based Verification. Guzey et al. [1] propose a generic technique which learns (from many signal traces) how to control the input signals to generate a specific value for an internal signal. The authors then use this learning to modify TB constraints to guide tests towards a coverage target that has not been hit before. Chatterjee et. al [2] proposes a technique to hit uncovered bins by detecting and removing over-constraints in a TB.

Vasudevan et. al [3] uses RTL Control Data Flow Graph (CDFG) to generate a deep-learning based model that predicts test coverage based on input knobs. The authors then use this model to predict which knob combination will generate high functional coverage. Nazi et. al. [4] extracts the graph showing how random variables in a TB control interface coverage . The authors then leverage this dependency information for adaptive test stimuli generation. Chatterjee et. al. [5] uses information from FIFO Dependency Graph (FDG) to accelerate the coverage closure of stalling regressions.

While there is a rich literature on automatic test generation for coverage closure, most prior works that are scalable to industry level design complexity focusses on Machine Learning (ML) based techniques that rely on collecting huge amounts of data to build a model. With design changing rapidly, it becomes challenging to keep the model updated. Moreover, learnings from such training are not readily transferable from one design unit to another. In this paper, we present a simple and intuitive heuristic to hit a subset of uncovered CPs without training models with huge amounts of data.

```
always_ff @posedge(clk)begin
  if(X == 1)begin
    Y <= Z;
  end else begin
    Y <= 1;
  end
end

`ifndef SV_COVER_OFF
  wire W = (Y == 1);
  svt CP1(.clk(clk),.tp(W));
`endif
```

```
module svt(clk,tp);
  input clk,tp;
  wire clk,tp;
  property p1;
    @(posedge clk)tp;
  endproperty
  c1: cover property(p1);
endmodule
```

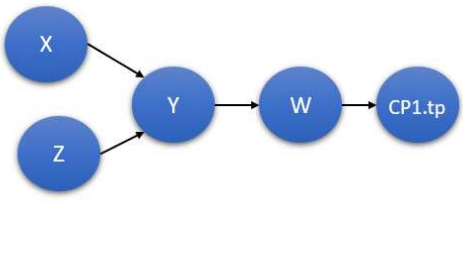

Figure 1. RTL code (on left) and corresponding TDG (on right). Note: Z is a driver for Y. X is a guard condition for the driver for Y. A change in either X or Z can potentially cause a change in Y. CP1 is an instance of svt module which implements the cover property.

III. Technical Solution

Our solution to hit Uncovered CPs (UCPs) focuses on identifying Covered CP (CCPs) that meet three criterion: 1) Overall hits to the CCP in End-Of-Project (EOP) coverage report is relatively low compared to other CCPs in the design, which means the neighborhood of this CCP has not been well-explored 2) The CCP has large number of UCPs in its neighborhood 3) The CCP is closely tied to the UCPs in its neighborhood. The last two criteria ensure exploring this CCP neighborhood increases the likelihood of hitting some UCPs. For now, let's assume we have some way to quantify how good a CCP is based on the above criteria by assigning a score to each CCP. We will elaborate on how to assign such scores in section V. The CCP with highest score will be our first *Target-CP* (TCP). The high-level idea is: *If we can bias a set of tests towards the Target-CP, then we could possibly hit UCPs in its neighborhood*.

IV. Trace Driver Graph (TDG)

To implement the idea discussed in section III, we need to quantify a) What is meant by neighborhood of a CCP? b) How do we measure the number of UCPs in the neighborhood of a CCP? and c) How do we measure how closely associated a CCP is to all the UCPs in its neighborhood? For this, we need an abstraction of the underlying RTL. Several such abstractions are possible. Here we introduce a simple abstraction that we call the Trace Driver Graph.

We parse the RTL to create the Trace Driver Graph (TDG). Every node in TDG is either a CP or a Signal. A directed edge from source node (X) to a destination node (Y) in TDG, implies one of the following: 1) X is a driver for Y 2) X is a guard condition for the driver for Y. Fig. 1 shows sample RTL code and corresponding TDG. Note, we abstract out all timing statements, constant operands (whether the constants are used in the guard statements or are part of the assignment expression) from the TDG. We use the Novas Programming Interface (NPI) commands provided by Synopsys (as part of the Verdi package) to extract the raw signal driver information. We then use that data to generate the TDG in NetworkX (a Python Package). For better visualization, we augment TDG with coverage data to assign one of the three colors to each node: Green for CCPs, Red for UCPs and Grey for Signals that are not CPs. The TDG as defined above has several properties listed below: 1) The TDG is in general a directed cyclic graph. Loops could exist in the TDG due to various logic. For example, counters in the design would create a self-loop. 2) A change in the source node value does not necessarily imply a change in the destination node value. For example, if the source and destination nodes are flop input and output signals respectively, then a change in former does not necessarily cause in a change in the latter (since TDG abstracts out timing statements). 3) In the absence of a cycle, a change in destination node value will always be attributed to a change in the source node value. 4) Every CP is a leaf node in the TDG, but every leaf node is not a CP.

Now that we have defined TDG and some of its properties, let's define what do we mean by *N-hop Neighborhood* of a CCP? We define the N-hop neighborhood of a CCP to be the set of nodes (Signals and UCPs) that can be reached within N consecutive hops, starting from the CCP in the TDG. These hops are edge direction agnostic, i.e., the hops could be made in (or against) the edge direction. Another way to think about this neighborhood is as follows: Starting from the CCP, if one is allowed to make N clicks to either Trace-Load or Trace-Driver button on Verdi, then all the signals and UCPs that can be possibly reached, belongs to the N-hop neighborhood of the CCP. We obviously don't trace load/drivers on Verdi to determine the N-hop neighborhood. This is done using algorithms on the TDG. Once we determine the N-hop neighborhood of a CCP, we could easily assign a score to the CCP. This score will be a

function of 2 attributes – 1) The number of UCPs in the neighborhood of the CCP and 2) A measure of how closely tied the CCP is to all UCPs in its neighborhood.

V. Identifying The Target-CPs

We only consider low-hit CCPs (bottom quartile in terms of hit-rate) to be possible candidates for a Target-CP. We run the GraphCov algorithm to assign scores to each of these low-hit CCPs that are possible candidates for Target-CPs. The score formula is designed in a manner that a CCP score will be higher if: 1) The number of UCPs in its neighborhood is higher; 2) The association between a CCP and its surrounding UCPs is higher. We define a CCP to be closely associated with a UCP if : 1) The (CCP, UCP) pair share a greater number of common ancestral nodes in the N-hop neighborhood of the CCP in the TDG 2) The average distance of the UCP from the shared ancestral nodes (between CCP and UCP) is less. The equation below shows the formula for assigning score to a CCP.

$$\text{Score}_{\text{CCP}_i} = \sum_{j=1}^{n} \frac{A(CCP_i, UCP_j)}{d(CCP_i, UCP_j)}$$

The term $A(CCP_i, UCP_j)$ refers to the number of shared ancestral node between the node pair (CCP_i, UCP_j). The term $d(CCP_i, UCP_j)$ refers to the average distance of the UCP_j from shared ancestral nodes between the pair (CCP_i, UCP_j). The overall score assigned to a CCP, denoted by $Score_{CCP}$, is the sum of the A/d values for all UCPs in its neighborhood. One could think of A/d value as the association metric between a CCP-UCP pair. From that perspective, the overall CCP score is a quantification of the cumulative association of a CCP to all UCPs in its neighborhood. Fig. 2 illustrates two sample 2-hop Neighborhood around 2 CCPs (C1 and C2) in TDG. Table I describes the GraphCov algorithm for score assignment for each of the CCPs in Fig. 2. From the scores in Table I it is clear that C2 is more closely associated with UCPs in its neighborhood than C1, although both C1 and C2 have 2 UCPs in their neighborhood. In general, we pick the CCP with the highest score and call it Target-CP1. We then remove the Target-CP1 and all UCPs in its N-hop neighborhood from the TDG and rerun the GraphCov algorithm on updated TDG to re-assign scores to the remaining CCPs. We select the CCP with the highest score in the second run and call it Target-CP2. This process is repeated until one of the following happens: 1) We run out of low-hit CCPs 2) We run out of UCPs in the TDG 3) Hit a max count of Target-CPs user wants to extract 4) Last found Target-CP score is less than user-defined threshold.

Intuitively, if we can direct tests towards the Target-CPs, then there is a greater likelihood of hitting UCPs that were never hit in earlier regression runs. This intuition is further backed by the following statistical analysis using coverage data and TDG, shown in Fig. 3. Every point in the scatter plot in Fig. 3 is a CP. The x-coordinate of the CP being represented by a point shows the sum total number of hits to all CPs in its vicinity. The y-coordinate of the CP being represented by a point is the percentage of CPs that are covered in its vicinity. Here we define the vicinity as 6-Hop neighborhood. Fig. 3 validates our intuition that a high percentage of CP coverage in the vicinity of a CP is correlated with a high total number of hits to all CPs in the vicinity of that CP. Hence, we aim to boost the coverage percentage (or hit uncovered CPs) by biasing tests to CCPs with low hit rate and high score values.

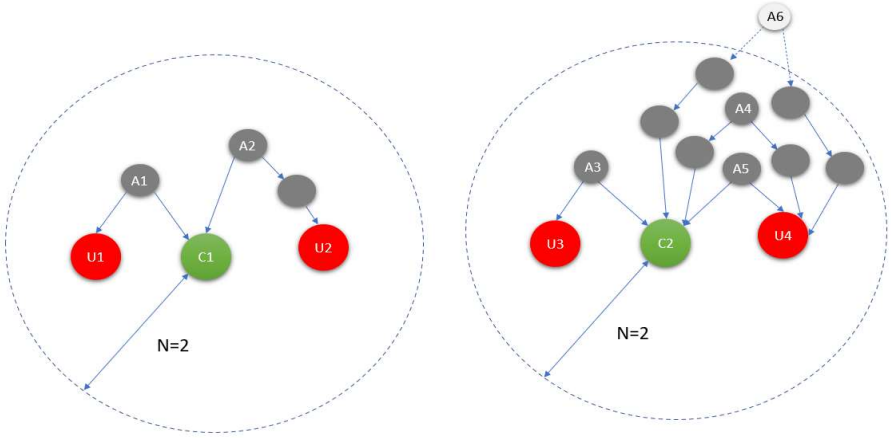

Figure 2. Sample N-hop (N=2) neighborhood around 2 different CCPs (C1 and C2) in TDG. U1-U4 are UCPs; A1-A6 are shared ancestral nodes. Dotted circle shows the 2-Hop neighborhood around C1 and C2 respectively.

TABLE I
GRAPHCOV ALGORITHM FOR COMPUTING SCORE FOR CCPS IN FIG. 2

Steps	Score Calculations for CCP (C1)	Score Calculations for CCP (C2)
Find pairs of the CCP with every other UCP in its neighborhood TDG	Found pairs (C1,U1) and (C1,U2)	Found pairs (C2,U3) and (C2,U4)
For the first pair, find the number of shared ancestors A	A(C1, U1) = 1, since there is only one common ancestor A1	A(C2, U3) = 1, since there is only one common ancestor A3
For the first pair, find the average distance of the UCP from the common ancestors	D(C1, U1) = 1, since distance between A1 and U1 is 1	D(C2, U3) = 1, since the distance between A3 and U3 is 1
For the 2nd pair, find the number of shared ancestors A	A(C1, U2) = 1, since there is only one common ancestor A2	A(C2, U4) = 2, since there are 2 common ancestors A4 and A5. Note, we don't count common ancestor A6 outside the 2-hop neighborhood
For the 2nd pair, find the average distance of the UCP from the common ancestors	D(C1, U2) = 2, since the distance between A2 and U2 is 2	D(C2, U4) = (d(A4,U4) + d(A5, U4))/2 = (2+1)/2 = 1.5
Compute Score for CCP using formula $\text{Score}_{\text{CCP}_i} = \sum_{j=1}^{n} \frac{A(CCP_i, UCP_j)}{d(CCP_i, UCP_j)}$	Score_CCP1 = (A(C1,U1)/ D(C1, U1) + (A(C1, U2)/ D(C1, U2)) = (1/1) + (1/2) = 1.5	Score_CCP2 = (A(C2, U3)/ D(C2, U3)) + (A(C2, U4)/ D(C2, U4)) = (1/1) + (2/1.5) = 2.33

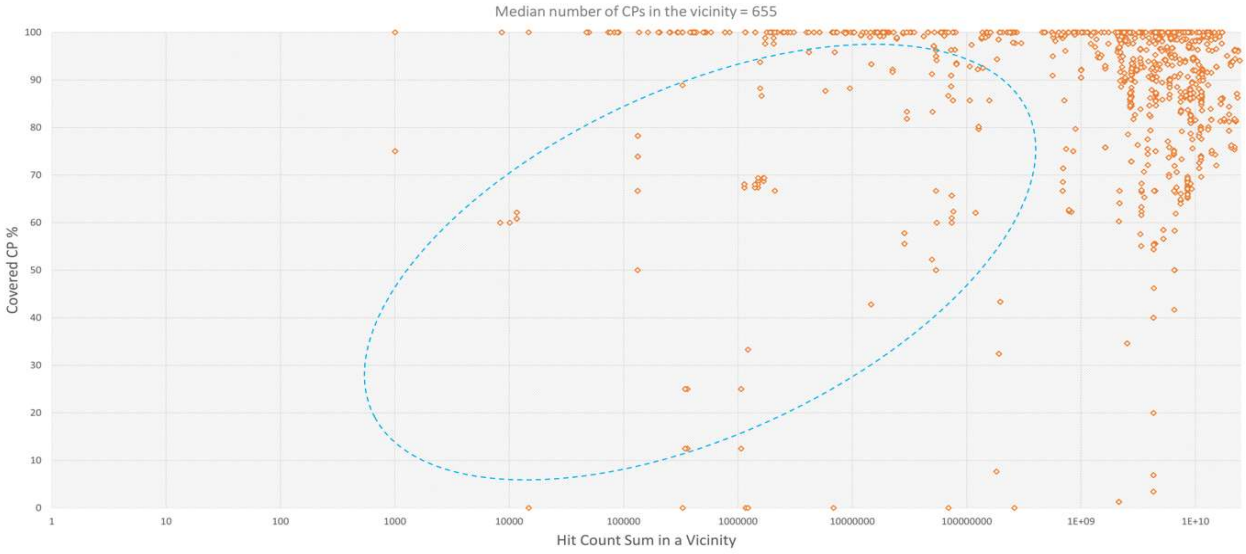

Figure 3. Covered CPs% vs Total Hit count in the vicinity of a CPs. Note: The CPs enclosed within blue ellipse are possible good candidates for test biasing, since they have low total hit count and less than 100% CP coverage in their vicinity.

VI. BIASING TOWARDS TARGET CPS

We have identified the top-M Target-CPs. But how do we automatically direct a batch of tests towards the Target-CP? For this, we use an existing Nvidia in-house tool which uses Bayesian Optimization (BayOpt) [6]. This tool takes a set of TestBench (TB) knobs and the Target-CP as its input. The tool eventually spits out a set of knob combinations and knob values that increases the number of tests that hit the Target-CP (compared to what would happen if default knob combinations were used from the regression). In the absence of BayOpt we have seen that it is also possible to manually review the Target-CP and have DV engineers suggest knobs that would bias the test towards it. However, BayOpt is usually able to boost hit rates further by fine tuning the knobs.

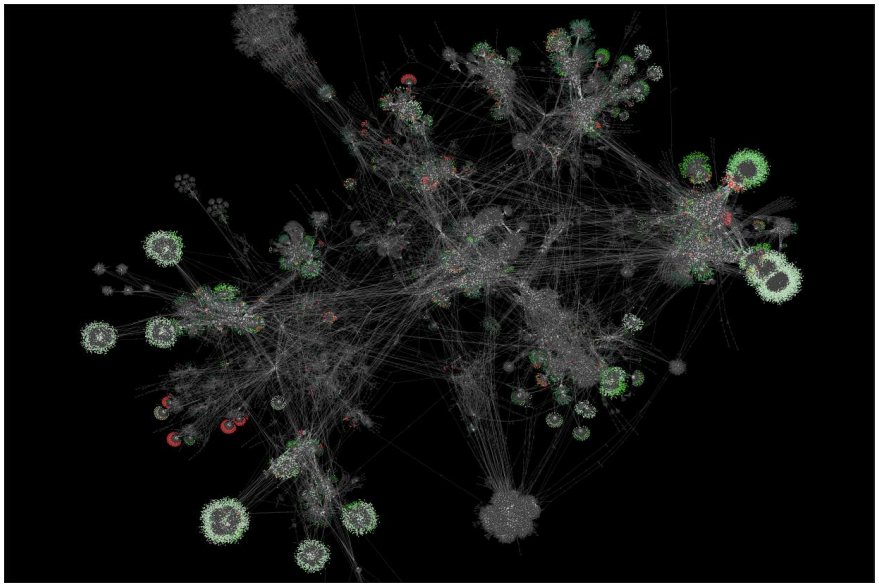

Figure 4. TDG for a specific design unit augmented with coverage data. CCPs, UCPs and Signals are colored Green, Red and Grey respectively. Darker shades of Green imply high-hit rate for the CCPs

TABLE II
RUNTIMES FOR EXTRACTING AND RUNNING TDG

Steps	Runtime
Extracting Raw Data file for TDG using Verdi NPI	~1hr
Raw Driver Information to TDG in Python NetworkX	~3min
Run GraphCov to find top-M Target-CP	~5min
Sum Total Time	~1hr8min

TABLE III
RESULTS FROM BIASING TESTS TOWARDS TARGET-CP

Target-CPs	Tests Run	Boost		Number of UCPs hit	Number of UCPs hit in 6-hop of Target-CP
		CP Hit per test	Overall CP hit		
Target-CP1	10000	23x	1.78x	7	6
Target-CP2	5000	390x	15x	24	22
Target-CP3	4000	20x	0.6x	15	6

VII. RESULTS

We generated TDG using a specific unit level RTL code. There were some common signals (like clocks and reset) which became the common ancestral node to most (CCP,UCP) Pair. After pruning clocks and reset signals, the generated graph had ~75K nodes (75,161 nodes) which include ~15K CPs (15,489). We then colored the TDG nodes using merged coverage data from 5 nightly regression (~130K tests). We found 847 UCPs (Red nodes), 14,642 CCPs (Green nodes) and 59,672 signals (Grey nodes). 238 out of 847 UCPs were waived as unreachable in EOP coverage report. Using GraphCov, we attempt to hit a subset of 847-238=609 UCPs. Fig. 4 shows a sample colored TDG from an actual design unit. Fig. 5 shows location of a Target-CP1 in the TDG and Fig. 6 shows the 6-hop Neighborhood of Target-CP1 in TDG. These graphs were rendered using the Force-directed graph drawing algorithm in *Cytoscape*[7]. Cytoscape is an open-source platform used for biological research, complex network analysis and visualization.

We ran GraphCov algorithm on the TDG to identify Top-10 Target-CPs. For every Target-CP, the script also spits out all UCPs in its neighborhood. Table II provides the breakdown and total runtimes to execute the entire flow to extract Target-CPs. After we generated the Top-10 Target-CPs, we manually select 3 out of those which we believe are controllable via TB knobs. For those Target-CPs, we then identify the knobs that could steer the tests towards them and any knobs that might be useful for hitting the neighboring UCPs. We use these knob combinations to direct tests towards the Target-CPs. Sometimes BayOpt tool helped finetune these knob combinations and values (which further increased the hit rate for the Target-CP). Experimental results are tabulated in Table III. Note that in all these experiments, we did not change TB code to hit the UCPs, nor did we add additional knobs to control tests. We only use existing runtime arguments to steer tests towards a Target-CP.

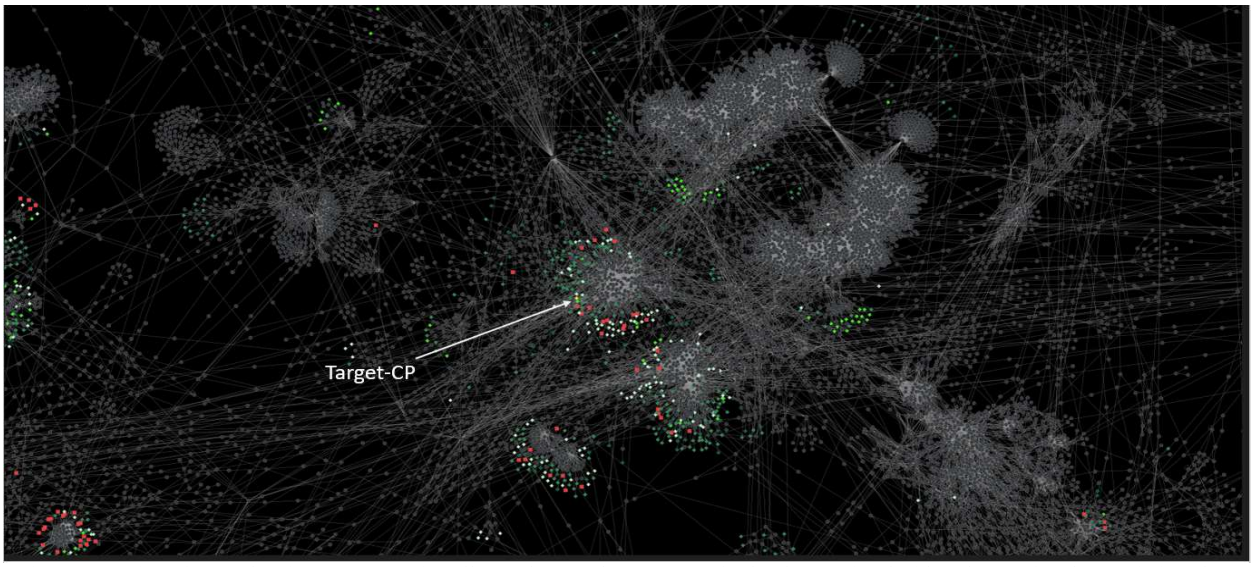

Figure 5. Portion of TDG showing Target-CP (in yellow) Observe, the Target-CP is closely associated with a cluster of Red UCPs in its 6-hop neighborhood

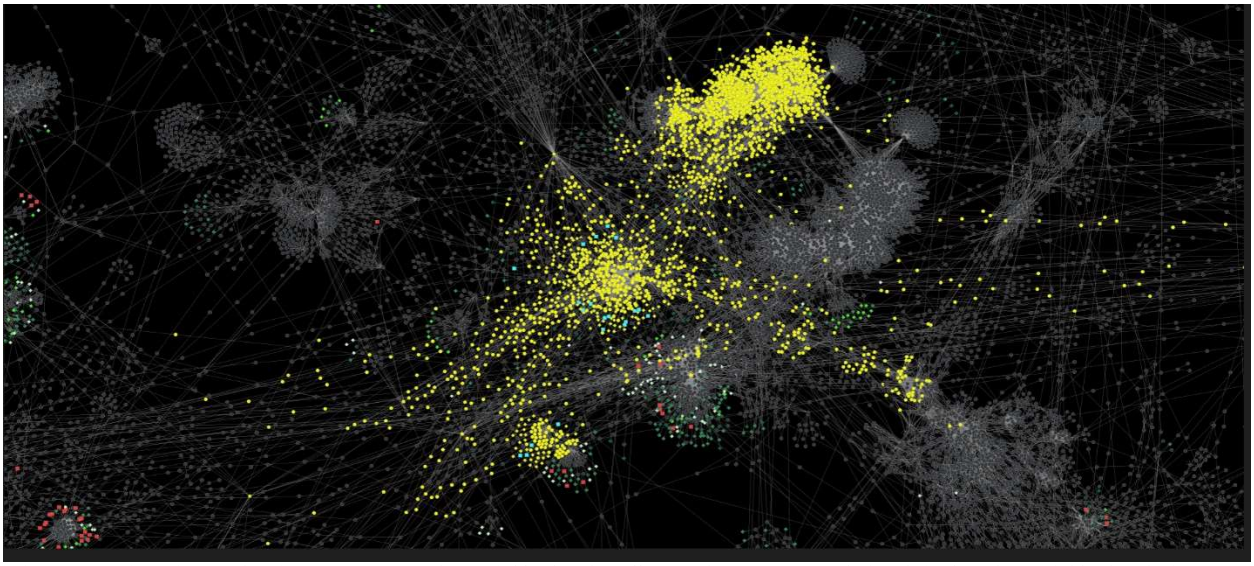
Figure 6. Portion of TDG showing Target-CP's 6-hop neighborhood (Yellow Nodes are CCPs and Signals in the 6-hop neighborhood of Target-CP; Blue nodes are UCPs in the 6-hop neighborhood of Target-CP; Red/Green/Grey Nodes are UCP/CCP/Signals outside 6-Hop neighborhood of Target-CP respectively)

For the first experiment with Target-CP1, we ran 10K tests with some directed knobs. These 10K tests were able to boost the hit-rate of the Target-CP1. Target-CP1 was hit 738K cycles in 130K tests in EOP coverage report. In our GraphCov experiment, we were able to hit Target-CP1 1307K cycles in just 10K tests, i.e., a 1.78X boost in overall hits with 13X less tests. In this experiment, we were able to hit 7 UCPs. Interestingly, 6 out of 7 UCPs that were hit were in the 6-hop neighborhood of Target-CP1 in the TDG.

For the second experiment with Target-CP2, we ran 5K tests with some directed knobs. Target-CP2 was hit 407K cycles in 5K tests compared to 27K cycles in 130K tests in original EOP coverage results. Overall hits to Target-CP2 were boosted 15X times with 26X less tests. In this experiment we hit 24 UCPs. 22 out of these 24 UCPs belonged to the 6-hop neighborhood of the Target-CP. A total of 53 UCPs were there in the 6-hop neighborhood of the Target-CP2. After analyzing why other UCPs in the neighborhood are not getting hit, we found that many of these 53 were related to a specific feature "F" which can be enabled through a PRI register. Turned out that this feature "F" was no

longer supported and hence the PRI was never enabled. So ideally those UCPs should have been waived. GraphCov helped us hit the UCPs as well as identify a cluster of UCPs which should be waived in the neighborhood of the Target-CP2.

For the third experiment, we ran 4K tests and boosted the Target-CP3 hit rate per test by 20x. We hit a total of 15 UCPs. 6 out these 15 UCPs where in Target-CP3 neighborhood. Overall, there were 19 UCPs in the neighborhood of Target-CP3. On analyzing we found that 6 out of 19 were reachable and remaining 13 UCPs need to be waived as unreachable.

Note there were no overlap in the UCPs that were hit across the three runs. In other words, 3 experiments found 3 sets of unique UCPs. In summary, we found that with 3 GraphCov runs, we were able to hit 46 UCPs and waive-off 44 UCPs. The number of CPs which were deemed unhit and reachable went down from 609 to 519, a 15% reduction.

VIII. Common Questions

Why do we remove the UCPs in the neighborhood of Target-CP1 and recompute the scores to determine Target-CP2? Why not select the CCP with second highest score as Target-CP2? If we do not remove the UCPs in the neighborhood of Target-CP1 and select the CCP with the second highest score as Target-CP2, then it is highly likely that Target-CP1 and Target-CP2 neighborhood will contain a lot of common UCPs. In other words, Target-CP1 and Target-CP2 in that case could be located in the same cluster in the graph where a lot of red-nodes (UCPs) are clustered around some green nodes (CCPs). By removing the UCPs around Target-CP1 and recomputing the scores, we make sure Target-CP2 has a lot of closely associated UCPs around it (even after ignoring the contributions from UCPs around Target-CP1, if any).

Do we use waiver information in GraphCov? UCPs can be classified into 3 categories – 1) UCPs waived as unreachable – these are CPs that are architecturally impossible to hit. Many autogenerated CPs fall in this category 2) UCPs waived as judged – such UCPs are reachable theoretically but waived because they are expected to be covered in other TBs and hence deemed less risky even if they remain uncovered 3) UCPs that are not waived. For our implementation, we removed the first category of UCPs (those that are waived as unreachable) from the TDG for score computation.

In section V, for score computation, why $d(CCP_i, UCP_j)$ refers to the average distance of the UCP_j from shared ancestral nodes between the pair (CCP_i, UCP_j)? Why not measure the average distance between (CCP_i, UCP_j) via all paths through common ancestral nodes? If we bias tests towards Target-CP and are able to boost the Target-CP hit rate, then all nodes/signals leading to the Target-CP (including ancestral nodes) must have been exercised with the appropriate values in tests that hit the Target-CP. So, we really need to see how far the UCPs are w.r.t to shared ancestral nodes. We tried the alternative approach as well and found that the mean distance of UCP from shared ancestral node gives a more meaningful result than mean distance of UCP from the CCP via shared ancestral nodes.

Can the approach scale to larger graphs? We expect the algorithm to scale to larger graphs (10x-100x size) very easily. Since the score computation is done at N-hop neighborhood level, the overall size of the graph has no impact on score assignment for a particular CCP.

What is reason for using N=6 for defining neighborhood and score assignment? As we increase the N, we increase the size of the neighborhood. Which means more and more UCPs will now be included in the neighborhood of a CCP. We want N to be large enough so that it includes reasonable number of UCPs from logic related to the CCP. However, too large N will basically include UCPs from unrelated logic (such logic might be far from the sub-unit where the CCP resides). We visually inspected the TDG and CCP neighborhoods. From that we found N=6 to be a reasonable choice. We can think of N as a hyper-parameter in the GraphCov method.

VIII. Conclusion

In conclusion, we would like to say that this approach shows promise in covering CPs which remained unhit due to random regressions not having sufficient bias towards specific areas in the design space. This can arise because test-lists might not have certain important knob combinations which exercise some crucial areas of the design (for which CPs have been coded by designers). It is common for most TBs to have 100 -1000s of runtime knobs. Since number of knob combinations grow exponentially with number of knobs, it is not possible to cover all such combinations

exhaustively. Although the method has demonstrated success in hitting hard to hit CPs, it cannot hit CPs which remain unhit to inherent limitations in TB code such over-constraints, in-built timing limitations in sequences etc.

ACKNOWLEDGMENT

The authors would like to thank Chris Wilson at Nvidia for providing the script which uses NPI to generate the raw trace driver data; Rajarshi Roy and Mukhdeep Benipal for providing BayOpt infrastructure; and Mikhail Ryzhov for providing help in coverage infrastructure to collect coverage data.

REFERENCES

[1] Guzey, Onur, and Li-C. Wang. "Coverage-directed test generation through automatic constraint extraction." 2007 IEEE International High Level Design Validation and Test Workshop. IEEE, 2007.

[2] Debarshi Chatterjee, Spandan Kachhadia, Ismet Bayraktaroglu, Siddhanth Dhodhi, “Systematic Constraint Relaxation (SCR): Hunting for Over Constrained Stimulus”, DVCON USA 2022

[3] Vasudevan S, Jiang WJ, Bieber D, Singh R, Ho CR, Sutton C. Learning semantic representations to verify hardware designs. Advances in Neural Information Processing Systems. 2021 Dec 6;34:23491-504.

[4] Nazi A, Huang Q, Shojaei H, Esfeden HA, Mirhosseini A, Ho R. Adaptive Test Generation for Fast Functional Coverage Closure. DVCON USA 2022.

[5] Debarshi Chatterjee, P. Lathigara, S. Dhodhi and C. Parsons, "FIFO Topology Aware Stalling for Accelerating Coverage Convergence of Stalling Regressions," *2022 IEEE 40th VLSI Test Symposium (VTS)*, 2022, pp. 1-7, doi: 10.1109/VTS52500.2021.9794164.

[6] Rajarshi Roy, Mukhdeep Singh Benipal, Saad Godil, “Dynamically Optimized Test Generation Using Machine Learning”, DVCON USA 2021

[7] https://cytoscape.org/