Adaptive Test Generation for Fast Functional Coverage Closure

Azade Nazi[‡], Qijing Huang[††], Hamid Shojaei[†], Hodjat Asghari Esfeden[†], Azalia Mirhosseini[‡], Richard Ho[†]
[‡]*Google Research*, [†]*Google*, [††]*UC Berkeley*
{azade, hamids, hodjat, azalia, riho}@google.com, [††]qijing.huang@berkeley.edu

Abstract

Coverage-driven test generation (CDG) is a well-studied approach to accelerate design verification. However, existing CDG methods rely on verification experts to extrapolate coverage-to-input relationships or use costly data-driven techniques to learn these dependencies, both of which introduce significant overhead to the verification process. In this work, we propose $\mathcal{CDG}$4CDG, a low-overhead framework that automatically extracts the coverage dependency graph associated with a design, and leverages the dependency information for adaptive test stimuli generation. The test generation is performed by maximum likelihood estimation method in $\mathcal{CDG}$4CDG that uses the dependency graph as prior information and adaptively changes the distribution of the random input stimuli to exercise the design more comprehensively towards higher coverage. We integrated $\mathcal{CDG}$4CDG with standard CRV flow. Our evaluations on open-source as well as large-scale real-world designs of a complex block in tensor processing unit (TPU) show that our approach significantly speeds up the coverage convergence with no human effort required.

I. Introduction

Design Verification (DV) is a critical step in chip development to guarantee the correctness of hardware designs [1], [2], [3]. One of the most widely adopted techniques in DV is running hardware simulations to evaluate the designs and uncovers bugs [4]. A typical way to generate the test input stimulus is through constrained random verification (CRV). In CRV, a testbench consists of random variables, constraints, and coverage metrics. Random variables are testbench variables whose values are stochastically generated subject to user-specified constraints. These constraints can be used to enforce the ranges and distributions of the random variable values (a.k.a. test stimuli). After simulating the design with CRV-generated test stimuli, the quality of the test is measured by the coverage metrics. To improve the speed and efficiency of verification, many prior works [5], [6], [7], [8], [9], [10], [11] use coverage feedback to guide the test stimulus generation. Such techniques are referred to as coverage-directed test generation (CDG).

Two main approaches to CDG are model-based CDG and data-driven CDG. In model-based CDG [5], [6], [7], [8], a model of the design under verification (DUV) is used to generate test-cases targeting desired coverage events. Creating a simple and accurate model is challenging and requires manual efforts. An example of these models is the abstract finite-state machine (FSM), which suffers from scalability issues on complex designs. Existing data-driven CDG techniques [9], [10], [11] aim to capture the relationship between coverage and stimuli directives directly from the observations gathered from simulation. These feedback-based techniques either rely on a significant amount of domain knowledge [11] or require a considerable number of simulations to obtain enough training data [9], [10].

These drawbacks make practical implementation of CDG tools for modern large-scale designs very challenging as data collection and model training may take several months. In this paper, we introduce an automated flow to generate tests for fast coverage closure with minimal verification overhead. Unlike prior CDG techniques, ours does not require data collection and model training, and is applicable to complex large-scale designs. Specifically, we leverage the design information parsed from off-the-shelf design tools and build a *Coverage Dependency Graph* ($\mathcal{CDG}$) based on the automatically extracted information. Our flow uses the coverage-to-random-variable dependency from $\mathcal{CDG}$ to optimize the test stimuli generation. We call our approach Coverage Dependency Graph for Coverage Driven Test Generation ($\mathcal{CDG}$4CDG) as we generate the tests by exploiting the information extracted from the $\mathcal{CDG}$. In $\mathcal{CDG}$4CDG, we formulate the problem of assigning proper distributions to the test stimuli as a maximum likelihood estimation problem (MLE). We then propose an algorithm that increases the probability of hitting uncovered events by adaptively tuning the distribution constraints over random variables. We prove that this strategy is in fact optimal to achieve coverage closure. In this work, we optimize on coverpoints that in $\mathcal{CDG}$ can be traced back to random variables. Coverpoints with an unsolvable mapping to input random variables are out of our scope, and for their coverage, we rely on random stimuli generation.

We implemented the $\mathcal{CDG}$4CDG flow in a standard CRV pipeline and evaluated its performance on an open-source RISC-V processor design [12] and a complex block in tensor processing unit (TPU) [13] which is an industry custom supercomputer for machine learning. The baseline approach is a common industry practice where the test input stimuli are drawn from a default distribution over random variables. To enable more exploration over the search space and capture the corner cases, we also compared our approach against random search where the distribution constraints are changed randomly over the simulation runs. Comparing to the existing CDG-based approaches is not practical because they require domain knowledge about the designs and are not openly available. We show that under certain scenarios, when coverpoints are uniformly distributed, our

baselines can be competitive. But in real-world designs where the corner cases must be exercised and the underlying distribution is skewed, $\mathcal{CDG}$4CDG significantly outperforms the baselines.

In summary, we made the following contributions:

- We introduce a framework, $\mathcal{CDG}$4CDG, that automatically handles the coverage dependency extraction and distribution constraints generation in CRV.
- We propose a method to automatically build the coverage-to-variable dependency graph ($\mathcal{CDG}$) by leveraging the design information extracted by off-the-shelf tools.
- We formulate the distribution constraints assignment problem for random test stimuli generation as a maximum likelihood estimation (MLE) problem and devise an optimal algorithm to maximize coverage.
- Our experiments on industrial TPU block show that $\mathcal{CDG}$4CDG consistently achieves higher coverage on all designs.
  - $\mathcal{CDG}$4CDG speeds up coverage closure by up to 21.7× compared to the baseline.
  - $\mathcal{CDG}$4CDG achieves 100% coverage (coverage closure) with less than 66 iterations while the baseline approach fails to reach converge closure within 500 simulation runs.

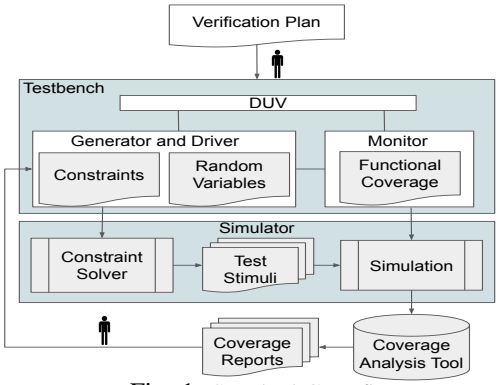


Fig. 1: Standard CRV flow

```
rand bit [1:0] State;
constraint con_State
    { State dist {
        idle := 10,
        start := 10,
        read := 10,
        write := 10
    };}
```

(a)

```
rand bit [4:0] Rs1;
constraint con_Rs1
    { Rs1 dist {
        [0 : 7] := 1,
        [8 : 15] := 25,
        [16 : 23] := 10,
        [24 : 31] := 32
    };}
```

(b)

Fig. 2: Distribution constraint examples in CRV. $\mathcal{CDG}$4CDG automatically generates and applies the constraints for different random variables.

II. Preliminaries

A. Constrained Random Verification (CRV)

Fig. 1 shows the standard CRV flow. Given a verification plan, verification engineers create a testbench to simulate the designs at the code level. The key components of a testbench are generator, driver, and monitor. The generator creates a stimulus, which is sent to DUV by the driver. The driver translates the operations produced by the generator into the actual design inputs. The monitor component has a checker that verifies the design output against the modeled output. It also contains all the definitions related to the functional coverage that need to be tested according to the verification plan.

In order to enable the constraint random stimulus generation, *random variables* and *constraints* are defined in the testbench. Constraints defined on the random variables direct the random stimulus generation. There are three types of constraints in a testbench: legalization constraints, ordering constraints, and distribution constraints. Legalization constraints impose user-specified valid value ranges on random variables. Ordering constraints are used to specify the ordering of constraint solving for different random variables. Distribution constraints help to bias the uniform value distribution over a random variable to a user-defined distribution. Values in a distribution constraint range with higher weights will get assigned more often to a random variable. Our main focus in this work is on distribution constraints. Fig. 2a and 2b show distribution constraints on two random variables *state*, and *Rs1* in SystemVerilog, where distributions are defined over variable values and a range of values respectively. Our goal is to guide the test stimulus generation toward uncovered events via distribution constraints.

As shown in Fig. 1, simulator in CRV runs the constraint solver to generate random test stimuli that satisfy all the constraints. A Register-Transfer Level (RTL) simulation is then run with the generated stimuli as input to exercise the DUV and report the verification coverage. The verification team can attain the statistics of the unexplored events from the coverage analysis to direct the simulation towards the uncovered events, test parameters and constraints can be changed, or even a new test can be added. This process continues in order to improve the overall quality of the generated test scenarios toward coverage closure. However, the current practice is very lengthy and requires a large amount of human effort in the loop.

There has been a lot of work on coverage-directed generation that focused on finding ways to automate this procedure and reduce the coverage gap. In this work, our goal is to automate this loop by focusing on distribution constraints and the functional coverage in the testbench. The main idea is to leverage the predefined coverage in the testbench in order to generate the proper distribution constraints to improve the verification quality and efficiency. Next, we provide a brief background on existing techniques to motivate our proposed approach.

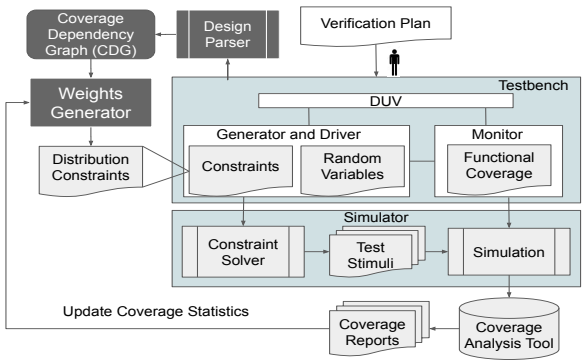


Fig. 3: $\mathcal{CDG}$4CDG flow. This flow augments design parser to automatically generate a coverage dependency graph $\mathcal{CDG}$ and then uses maximum likelihood estimation to automatically update the distribution constraints.

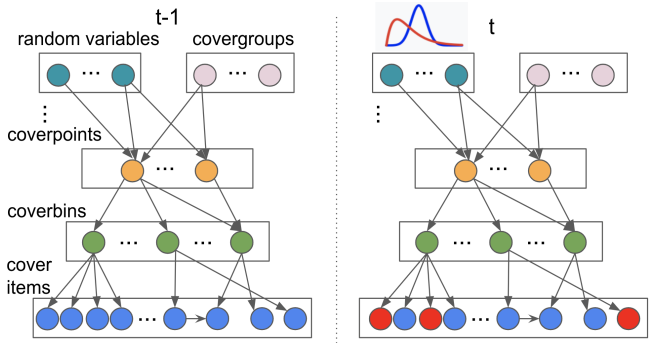


Fig. 4: Distribution constraints generation using Coverage Dependency Graph ($\mathcal{CDG}$) over multiple simulation runs. At step t, the red nodes are the covered events and the blue ones are those that are still uncovered .

B. Coverage Directed Generation

A large body of research on coverage-directed generation (CDG) has aimed at automating the process of leveraging feedback from coverage analysis to improve stimuli generation [5], [6], [14], [7], [9], [10]. Many prior works [5], [6], [14], [7] rely on model-based CDG where the DUV is modeled with an abstract finite-state machine (FSM). The FSM is used to generate directives or test cases to hit desired coverage events. Such approach often suffers from the state space explosion problem. Another line of work [8] describes the behavior of the DUV as constraints over data members of various objects in the DUV. Both approaches require a significant amount of human effort to create and maintain an accurate model.

Data-driven CDG, rather than modeling the behavior of the design, learns the relationships between coverage and stimuli directives from the simulation feedback. [11] uses expert systems to capture such relations and requires lots of domain knowledge to analyze the feedback and design. [9], [10] use observations to build and train a Bayesian network to model the casualty between input variables and coverage. The observations are the directives to the stimuli generator and the coverage data from the simulation. The performance of these models heavily relies on the training data that takes a considerable number of simulations to obtain. It also requires an offline training phase to establish the basis for future online decision-making. These limitations make implementing and comparing against such approaches impractical.

This paper introduces an automated flow to generate tests for fast coverage closure with minimal verification overhead. Specifically, we leverage the design information parsed from off-the-shelf design tools and build a *Coverage Dependency Graph* ($\mathcal{CDG}$) based on the automatically extracted information. Instead of building and training a model, we directly leverage $\mathcal{CDG}$ and maximize the use of dependency information to improve the coverage.

III. Proposed Verification Framework

Fig.3 shows our $\mathcal{CDG}$4CDG framework, where it augments design parser to automatically generate a coverage dependency graph and guide the test stimuli generation in the constrained random verification (CRV) toward coverage closure. Fig.4 illustrates the idea behind $\mathcal{CDG}$4CDG. The blue nodes at the lowest level of $\mathcal{CDG}$ are the uncovered coveritems and the red ones are those that are covered. Distribution constraints at iteration t will be updated based on dependency graph at iteration $t-1$. Next, we discuss the details of our proposed frameworks.

A. Coverage Dependency Graph ($\mathcal{CDG}$) Extraction

As we discussed in section I, many existing CDG tools [9], [10] aim to leverage coverage-to-random-variable dependency. But, capturing such dependencies rely on expert knowledge and demands a large number of simulations and training. To tackle this problem, we propose a new approach that automates this process by extracting the coverage dependency graph $\mathcal{CDG}$ from the top-level testbench module.

As shown in Fig.4, $\mathcal{CDG}$ is a directed acyclic graph with hierarchical structure that captures the dependency of covereitem to random variables. For simplicity, we introduce a running example to illustrate the details of our $\mathcal{CDG}$ extraction method. Fig. 5 shows an example SystemVerilog functional coverage (highlighted blue box) and its corresponding coverage dependency graph $\mathcal{CDG}$ that we generate. Specifically, this example shows a covergroup called `cg_memory` that gets sampled at every positive edge of the clock. Each covergroup contains at least one coverpoint that is defined over the value of the random variables. In this example, `state_cp` is a coverpoint in `cg_memory`, that depends on the value of the random variable `State`. The bins in coverpoint specify scenarios that the test aims to target e.g. `valid_states`, `valid_trans`, and `reset_trans`. In standard CRV flow Fig.1, such functional coverage is required to be exercised according to the verification plan defined in the testbench by verification engineers.

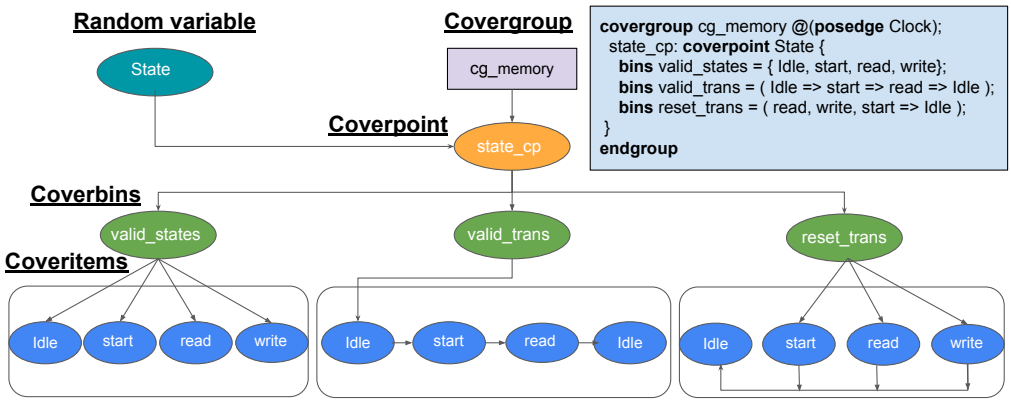


Fig. 5: Coverage Dependency Graph ($\mathcal{CDG}$) example for a memory controller design

In $\mathcal{CDG}$4CDG flow (Fig.3), we query and analyze the top-level testbench module to automatically extract $\mathcal{CDG}$. More specifically as shown in Fig. 5, for each coverpoint in a covergroup, we back trace until we reach a random variable. Then, for each coverbin, the coveritems are the values of the random variables required to be sampled in a test. The directed edges between the coveritems enforce the ordering of values to be observed in a simulation. In Fig. 5, the coverbin `valid_states` is used to show if all possible values for the random variable `State` have been exercised. The other two coverbins aim to show if particular sequences of `state` values have been tested. In this graph, a coverbin is deemed covered only when all of its coveritems are exercised in the test. Note that each coverbin can be associated with more than one variable as shown in Fig.4 and our approach still applies to coverpoints with multiple input variables.

In our experiments, we used Verific SystemVerilog parser tool [15] to query testbench modules and automatically extract the $\mathcal{CDG}$ graph. $\mathcal{CDG}$4CDG optimizes on coverpoints that can be traced back to random variables. The flow resorts to uniform random stimuli generation for coverpoints with an unsolvable mapping to input random variables. Next, we discuss how $\mathcal{CDG}$4CDG uses the current coverage feedback and dependency graph to generate weights and the distribution constraints for the next iteration.

B. Distribution Constraint Generation

The configuration of distribution constraints is critical to the coverage convergence as it directs the constraint solver to generate a test stimulus that targets the next set of coveritems. We formulate the problem of finding proper distribution constraints that maximize the probability P of hitting more coveritems as a maximum likelihood estimation. At every simulation run, given the unseen coveritems x, the goal is to find distribution constraint θ, that maximizes $P(x;\theta)$. We first propose algorithm 1 to show how $\mathcal{CDG}$4CDG generates distribution constraints. Then we prove that the distributions generated by our algorithm are, in fact, the exact solutions to MLE formulation.

From the simulator coverage feedback, we can get all coverage statistics in different hierarchical levels of our $\mathcal{CDG}$ graph, including the covergroups (cg), coverpoints (cp), coverbins (cb), and coveritems (v_i). The coverage statistics can be represented as a multi level dictionary $\mathcal{S}\{cg{:}\{cp{:}\{cb{:}\{v_i{:}0/1\}\}\}\}$. A coveritem is covered if $\mathcal{S}[cg[cp[cb[v_i]]]]$ is one. In every simulation cycle, after getting the feedback and updating the coverage statistics ($\mathcal{S}$) $\mathcal{CDG}$4CDG runs Algorithm 1 to generate new a set of weights for the next cycle.

The inputs to our Algorithm 1 are the coverage statistics $\mathcal{S}$, and our coverage dependency graph $\mathcal{CDG}$. The output is distribution weight constraints $\mathcal{W} = \{v{:}\{v_i{:}w_i\}\}$, where v is a random variable that takes values v_i with weight w_i. For example, $\mathcal{W} = \{State{:}\{idle{:}10,\ idle{:}10,\ start{:}10,\ read{:}10\}\}$ is equivalent to distribution constraint example in Fig. 2a. If a coverpoints is related to more than one variable, all of them will be added to $\mathcal{W}$. As shown in lines 4-6, it traverses the covergroups to find the unhit coverpoints. For each uncovered coverpoint, the algorithm looks up the dependency graph $\mathcal{CDG}$ to get all related random variables V and the coveritem nodes v_i. It then checks the coverage statistics and if v_i is not covered yet ($\mathcal{S}[cg][cp][cb][v_i] == 0$) it increments the weight $\mathcal{W}[v][v_i]$ by 1. In $\mathcal{CDG}$, coverpoints and coverbins will be marked as covered when all their coveritems are covered. Doing so reduces the graph traversal time as more coverpoints get covered. The output of this algorithm represents the distribution constraints for the next simulation. Note that later in CRV, normalized weights will be used for test generation.

Next, we show that Algorithm 1 is the optimal strategy that maximizes the probability of hitting target coveritems.

1) Algorithm Analysis

For a random variable v, Algorithm 1, counts the number of times (w_i) that coveritems v_i are not hit yet. If the next simulation run returns an observation set x, where random variable value v_i appears w_i times, then with a high probability, the overall coverage is increased. The goal is to find a distribution θ over random variable values v_i that maximizes the likelihood of occurrence of x. More formally:

$$\theta_{MLE} = argmax_{\theta} P(x;\theta) \tag{1}$$

Algorithm 1 Distribution constraint generation

Input: Coverage dependency graph $\mathcal{CDG}$, coverage statistics $\mathcal{S}$
Output: Distribution constraint weights $\mathcal{W} = \{v{:}\{v_i{:}w_i\}\}$
Initialize constraint weights $\mathcal{W} = \{v{:}\{v_i{:}w_i{=}0\}\}$, where v is the random variable, and v_i is its value.
for Uncovered coverpoint $cp \in \mathcal{CDG}$ **do**
    Get all cp's related related random variables
    for coveritems $v_i \in$ coverpoint cp **do**
        if $\mathcal{S}[cg][cp][cb][v_i] == 0$ **then** Increment the weight $\mathcal{W}[v][v_i]$+=1
        Coverpoint cp is covered if all its coveritems v_i are covered.
return $\mathcal{W} = \{v{:}\{v_i{:}w_i\}\}$

Given the distribution constraint θ and the dependency information from $\mathcal{CDG}$, we can calculate the log likelihood of observing x given the distribution using Equation 2, where n is the size of observation set x, ($|x| = \sum w_i$).

$$\begin{aligned} P(x|\theta) &= \frac{n!}{\prod_i w_i!} \prod \theta_i^{w_i} \\ log(P(x|\theta)) &= \log(n!) - \sum_i \log(w_i!) + \sum_i w_i \log \theta_i \end{aligned} \tag{2}$$

Given that θ is a distribution, a constraint to MLE is $\sum_i \theta_i = 1$. We use the Lagrange multipliers with this constraint in Equation 3 to derive the closed form solution of MLE that maximizes the likelihood $p(x|\theta)$.

$$\begin{aligned} \mathcal{L}(\theta, \lambda) &= \log(n!) - \sum_i \log(w_i!) + \sum_i w_i \log \theta_i + \lambda(1 - \sum_i \theta_i) \\ \frac{\partial \mathcal{L}}{\partial \theta_i} &= \frac{w_i}{\theta_i} - \lambda = 0, \quad \implies \quad \theta_i = \frac{w_i}{\lambda} \\ \frac{\partial \mathcal{L}}{\partial \lambda} &= 1 - \sum_i \theta_i = 0, \implies \quad \sum_i \theta_i = 1 \end{aligned} \tag{3}$$

Therefore, $\sum_i \frac{w_i}{\lambda} = 1$, i.e, $\lambda = \sum_i w_i = n$ and $\theta_i = \frac{w_i}{n}$. In other words, the optimal distribution θ_i for random variable value v_i is proportional to the number of times that coveritems v_i are not hit. The $n = \sum_i w_i$ is the normalization factor. Algorithm 1 returns weights w_i and normalization happens in CRV.

IV. Experimental Results

We implemented the $\mathcal{CDG}$4CDG flow in a standard CRV pipeline and evaluated its performance on two SystemVerilog designs, an open-source processor design and an industrial custom supercomputer for machine learning. We compare $\mathcal{CDG}$4CDG against a common practice where the CRV is run with existing distribution constraints in testbench (if any), to demonstrate the impact of weight changing on the coverage. We also compare $\mathcal{CDG}$4CDG against random weight assignments to show the empirical difference between the two weight assignment strategies. The detailed experimental results are discussed in section IV-A. We run two ablation studies (in section IV-B) to evaluate the impact of binning size on coverage and investigate the side effect of $\mathcal{CDG}$4CDG on code coverage even though our flow is targeting functional coverage. Finally, in section IV-C, we provide a theoretical bound for the performance of the baseline when we do not change distribution constraints, to better explain its behavior in our experimental results.

Design Details: Two hardware verification designs are presented for evaluation.

- ***RISCV-Ibex***: The first testbench we target is a processor design implementing RISCV ISA called RISCV-Ibex [12]. This testbench leverages RISCV-DV [16], an open-source RISCV instruction generator, to produce random sequences of compiled instruction binaries as test input. It then simulates the processor to run this input program and compares the execution trace log with a golden reference output to check for correctness.
- ***SML***: The second design is a complex block in tensor processing unit (TPU) [13] which is a hyper-scale industrial custom accelerator for machine learning. The TPU block consists of multiple large and complex designs. Each design has an independent verification environment with predefined coverage. We considered three designs in this TPU block (SML1-3).

Table I shows the number of random variables, coverpoints, and weights, defined in each design. There are four random variables (3 operands and 1 opcode) and ∼5K coverpoints in RISCV-Ibex. We generate 111 distribution constraints for this testbench, including 63 for opcode and 48 for operands (16 weights per operand). For SML designs, the number of random variables ranges from 10 to 129. The total number of coverpoints defined for design SML1-3 is 371, 572, and 1129, respectively. The total number of weights is 32, 52, and 84.

Design	Number of random variables	Number of coverpoints	Number of weights
RISCV-Ibex	4	4942	111
SML1	10	371	32
SML2	25	574	52
SML3	129	1129	84

TABLE I: Testbench information in RISCV-Ibex and three designs in tensor processing unit (SML1-3).

Baseline: We compare the performance of our proposed $\mathcal{CDG}$4CDG approach against two baselines *default_dist*, and *rand_dist*. In *default_dist*, we use existing distribution constraints which are defined by verification engineer in testbench (if any) and we do not change them during the simulation, while in *rand_dist*, we change the distribution constraints randomly to enable exploring the search space. Note saying that even though there are prior CDG techniques [9], [10]; however, the comparison against them is not practical since they rely on expert knowledge and require simulation runs to collect training data. Besides, there is no public dataset available. Preparing such a dataset is expensive and relies on expert knowledge.

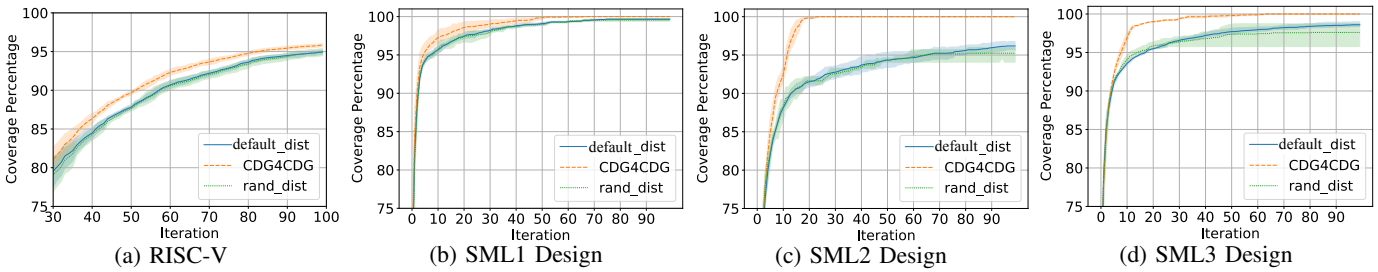
Experimental Setup: In our experiment, we use Verific [15] for parsing the designs in SystemVerilog and run the simulation with VCS [17] to collect coverage statistics. For VCS and UCAPI, we use Synopsys-VCS-2019-06-sp1, and 2019.06-1 versions respectively. For each experiment, we run the simulation for 100 iterations, each with new algorithm-suggested distribution constraints, and we collect the aggregated coverage after each iteration. Our goal is to increase the overall coverage while also reducing the number of simulation runs. Each experiment is run five times with different random seeds. The seeds in different iterations are different in CRV. However, for a fair comparison, we used the same random seed among different approaches in every simulation.

Constraint Binning: Since the value range of a random variable in a design can be extremely large, sometimes assigning one distribution constraint per value can introduce overhead in the simulation. Therefore, to reduce the total number of constraints, we group multiple values together via binning and assign one distribution constraint over the bins instead of per value. The values in one bin have an equal probability of getting sampled. We run an ablation study to empirically investigate the effect of binning size on performance.

A. Experimental Results

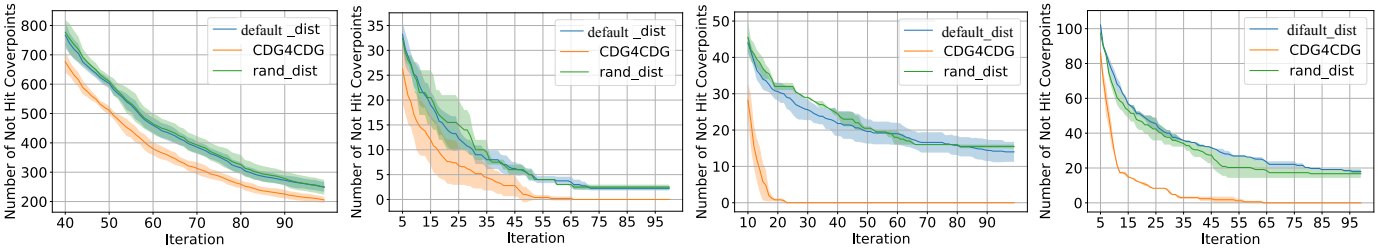
1) Open-source Processor Implementation RISCV-Ibex

We first evaluate our flow on the RISCV-Ibex testbench with four random variables. Our coverage convergence curve in Fig. 6a shows that $\mathcal{CDG}$4CDG outperforms *default_dist* and *rand_dist* at every iteration. Fig. 7a shows that, towards the last 800 not hit coverpoints, it takes $\mathcal{CDG}$4CDG 5 to 10 fewer iterations to reach the same coverage compared to the other approaches. With the same number of simulations, $\mathcal{CDG}$4CDG covers more than 50 additional coverpoints compared to the baseline. Note that most of the coveritems in this testbench are uniformly specified among all random variable values, and the performance of *default_dist* with uniform weights is competitive for the uniform coveritems. Our theoretical bound in the following Sec. IV-C



(a) RISC-V (b) SML1 Design (c) SML2 Design (d) SML3 Design

Fig. 6: Functional coverage convergence comparison among *default_dist*, $\mathcal{CDG}$4CDG, and *rand_dist*.



(a) RISCV-Ibex (b) SML1 Design (c) SML2 Design (d) SML3 Design

Fig. 7: Statistics of not hit coverpoints towards the last 10% coverage.

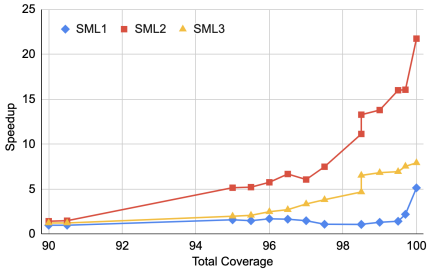


Fig. 8: Convergence speed up of $\mathcal{CDG}$4CDG over *default_dist* targeting different coverage.

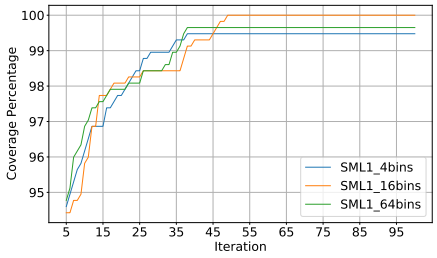


Fig. 9: Impact of the binning size on coverage on Design SML1.

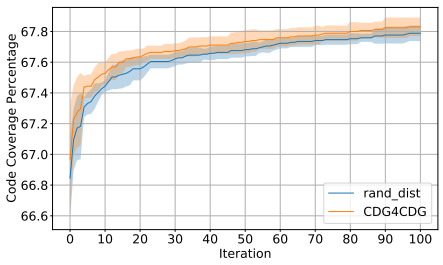


Fig. 10: Indirect impact of $\mathcal{CDG}$4CDG flow on Code Coverage on the RISCV-Ibex design.

also justifies that *default_dist* can perform well in this scenario. Nonetheless, $\mathcal{CDG}$4CDG achieves better coverage results over *default_dist*, which motivates us to run it on more challenging testbenches from a complex block in TPU.

2) Industrial Accelerator SML: TPU block

Fig. 6b, Fig. 6c, and Fig. 6d show that, compared with *default_dist* and *rand_dist*, $\mathcal{CDG}$4CDG flow converges faster and consistently achieves higher coverage over the 100 iterations (simulation runs) on all designs. *rand_dist* demonstrates similar convergence rate compared to *default_dist*, which suggests that solely changing the weights does not lead to better coverage. The statistics of the not hit coverpoints towards the last 10% coverage for design SML1-3 are shown in Fig. 7b, Fig. 7c, and Fig. 7d. SML2 result in Fig. 7c more prominently manifests the advantages of $\mathcal{CDG}$4CDG over *default_dist* and *rand_dist*. $\mathcal{CDG}$4CDG takes only 2 iterations (iteration 10-12) on this design to cover 5 coverpoints (from 25 to 20 uncovered coverpoints) while it takes the other approaches around 10 iterations (iteration 30-40). This again illustrates a 5× test speedup for coverage convergence. Note that it is more difficult to improve the coverage towards the last 10% coverpoints as they usually are hard-to-hit cases.

In design SML1, we observe a less significant coverage improvement from $\mathcal{CDG}$4CDG. It is because sometimes the weight distribution assignment in this design can violate other constraints. For example, we may assign weight 0 to the *add* operation while there is an existing constraint to ensure there is at least one *add* operation appear in the test. Such constraint violation can result in nullification of the specific distribution constraints in CRV but does not lead to test failures. With more weight constraints successfully applied in SML2 and SML3, we observe better coverage performance from $\mathcal{CDG}$4CDG.

Since after 100 iterations, the baseline fails to reach full coverage, we increased the number of simulation runs to 500. As shown in Table II, on average $\mathcal{CDG}$4CDG reaches 100% coverage at iteration 66, 23, and 63 respectively on design SML1-3 while *default_dist* converges at iteration 340 on SML1 and it does not converge within 500 iterations on SML2 and SML3. These results show a minimal 5.15×, 21.7×, and 7.9× verification time reduction towards the coverage closure using $\mathcal{CDG}$4CDG.

	SML1	**SML2**	**SML3**
$\mathcal{CDG}$4CDG	5.15× (66 vs. 340)	>21.7× (23 vs. 500+)	>7.9× (63 vs. 500+)

TABLE II: Speedup in coverage closure over *default_dist*. We observe at least 5.15×, 21.7×, 7.9× speedup from $\mathcal{CDG}$4CDG over the *default_dist* baseline, respectively, on designs SML1-3.

In Fig. 8, we plot the speedup of $\mathcal{CDG}$4CDG over *default_dist* for converging to different coverage values. For example, in SML2, $\mathcal{CDG}$4CDG takes 9 iterations to reach 90%, while for *default_dist* it takes 13 iterations, showing, speedup of 1.4×. The higher the target coverage is, the harder the test is. As shown in this figure, the highest speedup for SML1-3 all achieved toward the 100% coverage. This indicates that our $\mathcal{CDG}$4CDG is perfect for hard-to-hit coverpoints and can offer a greater advantage in such cases.

B. Ablation Studies

1) Impact of binning size on performance

In this ablation study, we investigate the impact of the binning size on coverage. On the SML1 design, we evaluate the coverage performance with 4, 16, 64 values per bin for the distribution constraints. Results in Fig. 9 show that, from 4 bins to 16 bins, there is a significant improvement in coverage, but when we increase the number of bins to 64, the coverage is reduced. Fewer values per bin allow the algorithm to more precisely control the random variable values in CRV. However, more control also will translate to more distribution constraints. This increases the probability that our introduced constraints conflict with the existing ones in the testbench. Such constraint violation can result in nullification of the specific distribution constraints in CRV, but does not lead to test failures.

2) Impact on Code Coverage

Code Coverage is another important metric used to measure the quality of verification. Code coverage, as indicated by its name, reflects the completeness of a test over HDL code. Though $\mathcal{CDG}$4CDG is designed to improve the functional coverage, it alters the variable values and the execution paths of the simulation, and subsequently may affect code coverage as well. In this study, we investigate the impact of $\mathcal{CDG}$4CDG on code coverage on the RISCV-Ibex design. Fig. 10 shows that there is also improvement in code coverage when $\mathcal{CDG}$4CDG is deployed. One explanation is that our function-coverage-driven approach more systematically exercises different control variables' values and leads to the coverage of more code segments.

C. Theoretical Bound on default_dist Baseline

In our experimental evaluations we observe that in some scenarios *default_dist* can be as competitive as $\mathcal{CDG}$4CDG (see Fig. 7a), while in others it is far behind our approach (see Fig. 7c and Fig. 7d). This observation motivated us to do further theoretical analysis to understand this baseline. In this section, we discuss our findings on the best-case and worst-case scenarios that may happen if we do not change the distributions at all. Let us assume total number of coveritems that need to be covered are m and in worst case every simulation run can cover at most one coverpoint. Note that the same coveritem may be covered repetitively. The question is that how many times we should run the simulator to cover all of the m coveritems.

Theorem 1. *For the fixed distribution constraints, the lower bound of the expected number of simulation runs to cover all m coveritems is $O(m \log m)$, while the upper bound is infinite.*

Proof. Let us start by analyzing the upper bound. If the underlying distribution of the coveritems is heavy-tailed, the convergence is unbounded. In other words, there exists hard-to-reach coveritems, and the probability of reaching them is very small. Thus, for the fixed distributions, the expected number of simulation runs to cover all m coveritems tend to infinity.
The best scenario is when the distribution constraint is uniform and coveritems are equally likely to be covered (with probability $\frac{1}{m}$). In this case, the problem is reduced to an instance of well known *Coupon collector's problem*, where there are m coupon types with equal probability of occurrence. At each run, we get one sample and the goal is to collect all m types of coupons. [18], [19] showed that the expected number of coupons that we need to sample to complete the collection is $O(m \log m)$. □

The above analysis provides clear explanation of our results as follows: Most of the coveritems in RISCV-Ibex are uniformly specified among all random variable values, making it a great fit for the *default_dist* approach. Our experimental results also show that *default_dist* can be as competitive as $\mathcal{CDG}$4CDG (Fig. 7a) for RISCV-Ibex. On the other hand the worst case scenario usually happens in real practice, where the goal is to test corner cases and the reachability of the coverpoints are different. Based on Theorem 1, *default_dist* may fail since the expected number of simulation runs to cover all m coveritems tends to infinity. Our experiments in Fig. 6 and Table II also confirm that *default_dist* has difficulty in reaching 100% coverage for our industrial SML designs.

V. Related Work

Coverage-directed test generation (CDG) is an effective and widely adopted technique for hardware verification. There are two main categories of CDG techniques: model-based [5], [6], [7], [8] and data-driven [9], [10]. Model-based CDG hardly scales to real industry designs due to the state explosion problem. Meanwhile, data-driven CDG incurs significant training overhead and still requires expert knowledge of the DUT. StressTest [20] relies on user-defined templates for constructing a Markov model to direct the instruction sequences generation for processor verification. To support new design, the existing approaches all require a good amount of expert time. In our work, we aim to eliminate the human effort in the process by leveraging the coverage to input information that can be directly generated from automatic System Verilog parser. Our tool extracts the coverage dependency information for any arbitrary hardware designs without training, and automatically adjust the input for the reachable cover point without human intervention.

Some approaches treat the coverage function as black-box and use machine learning algorithms [21], [22], [23], [24], [25], [26], [27] to drive the input generation. We argue that the tool should harness the known information to the maximum extent to reduce the training overhead. RFuzz [28] and DirectFuzz [29] leverage various input mutation techniques from software fuzzing at the Bealoon level to exercise DUT. RFuzz introduces RTL transformations to generate FPGA-emulatable designs for accelerating the verification process. DirectFuzz [30] performs static analysis on the DUT to determine the priority of inputs for fuzzing. In this work, we target the weight constraints in constrained random verification (CRV) instead of directly mutating boolean input values. Our approach is similar to the fuzzing-based techniques in the way that it is DUT-agnostic and does not rely on human knowledge, yet offering the benefits of CDG. However, instead of fuzzing for mux coverage, CRV targets specific user-defined cases that are critical for functional correctness.

VI. Conclusion

In this work, we introduce an automated test generation framework, $\mathcal{CDG}$4CDG, to accelerate the functional coverage convergence in CRV. Our approach first automatically extracts coverage-to-constraint dependency information and then uses this information to devise an optimal weight assignment strategy that maximizes the likelihood of hitting uncovered values. We implement $\mathcal{CDG}$4CDG for both open-source and real-world complex hardware designs and empirically show that $\mathcal{CDG}$4CDG achieves consistent coverage improvement over the original flow. Furthermore, it reaches the coverage closure significantly faster with up to $21.7\times$ speedup in a fully automated flow.

REFERENCES

[1] J. Bergeron, *Writing Testbenches: Functional Verification of HDL Models*. Kluwer Academic Publishers, 2003.

[2] B. Wile, J. Goss, and W. Roesner, *Comprehensive Functional Verification: The Complete Industry Cycle (Systems on Silicon)*. Morgan Kaufmann Publishers Inc., 2005.

[3] H. Foster, "Wilson research group functional verification study: Ic/asic functional verification trend report," 2020. [Online]. Available: https://blogs.sw.siemens.com/verificationhorizons/2021/01/14/part-9-the-2020-wilson-research-group-functional-verification-study/

[4] S. Tasiran, F. Fallah, D. G. Chinnery, S. J. Weber, and K. Keutzer, "A functional validation technique: biased-random simulation guided by observability-based coverage," in *Proceedings IEEE International Conference on Computer Design: VLSI in Computers and Processors.*, 2001.

[5] ——, "A functional validation technique: Biased-random simulation guided by observability-based coverage," in *International Conference on Computer Design (ICCD)*, 2001.

[6] D. Van Campenhout, T. Mudge, and J. P. Hayes, "High-level test generation for design verification of pipelined microprocessors," in *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference*, ser. DAC '99. Association for Computing Machinery, 1999.

[7] H. Iwashita, S. Kowatari, T. Nakata, and F. Hirose, "Automatic test program generation for pipelined processors," in *Proceedings of IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, ser. ICCAD '94. IEEE Computer Society Press, 1994.

[8] A. Adir, E. Bin, O. Peled, and A. Ziv, "Piparazzi: a test program generator for micro-architecture flow verification," in *Eighth IEEE International High-Level Design Validation and Test Workshop*, 2003.

[9] S. Fine and A. Ziv, "Coverage directed test generation for functional verification using bayesian networks," in *Proceedings of Design Automation Conference (DAC)*, 2003.

[10] S. Fine, L. Fournier, and A. Ziv, "Using bayesian networks and virtual coverage to hit hard-to-reach events," *International journal on software tools for technology transfer*, 2009.

[11] G. Nativ, S. Mittermaier, S. Ur, and A. Ziv, "Cost evaluation of coverage directed test generation for the IBM mainframe," in *Proceedings IEEE International Test Conference*, 2001.

[12] P. D. Schiavone, F. Conti, D. Rossi, M. Gautschi, A. Pullini, E. Flamand, and L. Benini, "Slow and steady wins the race? a comparison of ultra-low-power risc-v cores for internet-of-things applications," in *International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, 2017.

[13] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*. Association for Computing Machinery, 2017.

[14] S. Ur and Y. Yadin, "Micro architecture coverage directed generation of test programs," in *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference*, 1999.

[15] http://www.verific.com.

[16] *SV/UVM based instruction generator for RISC-V processor verification*, 2020. [Online]. Available: https://github.com/google/riscv-dv

[17] https://www.synopsys.com/verification/simulation/vcs.html.

[18] P. Flajolet, D. Gardy, and L. Thimonier, "Birthday paradox, coupon collectors, caching algorithms and self-organizing search," *Discrete Appl. Math.*, 1992.

[19] P. Erdős, "On a classical problem of probability theory," 1961.

[20] I. Wagner, V. Bertacco, and T. Austin, "Stresstest: an automatic approach to test generation via activity monitors," in *Proceedings of Design Automation Conference (DAC)*, 2005.

[21] G. Squillero, "Microgp—an evolutionary assembly program generator," *Genetic Programming and Evolvable Machines*, 2005.

[22] Z. Kotásek *et al.*, "Automation and optimization of coverage-driven verification," in *Euromicro Conference on Digital System Design*, 2015.

[23] T. Menzies and C. Pecheur, "Verification and validation and artificial intelligence," *Advances in computers*, 2005.

[24] F. Hutter, D. Babic, H. H. Hoos, and A. J. Hu, "Boosting verification by automatic tuning of decision procedures," in *Formal Methods in Computer Aided Design (FMCAD)*, 2007.

[25] Y. Naveh, M. Rimon, I. Jaeger, Y. Katz, M. Vinov, E. s Marcu, and G. Shurek, "Constraint-based random stimuli generation for hardware verification," *AI magazine*, vol. 28, no. 3, 2007.

[26] A. Dinu and P. Ogrutan, "Opportunities of using artificial intelligence in hardware verification," in *International Symposium for Design and Technology in Electronic Packaging (SIITME)*, 2019.

[27] W. Hughes, S. Srinivasan, R. Suvarna, and M. Kulkarni, "Optimizing design verification using machine learning: Doing better than random," *arXiv preprint arXiv:1909.13168*, 2019.

[28] K. Laeufer, J. Koenig, D. Kim, J. Bachrach, and K. Sen, "Rfuzz: Coverage-directed fuzz testing of rtl on fpgas," in *International Conference on Computer-Aided Design (ICCAD)*, 2018.

[29] S. Canakci, L. Delshadtehrani, F. Eris, M. B. Taylor, M. Egele, and A. Joshi, "Directfuzz: Automated test generation for rtl designs using directed graybox fuzzing," 2021.

[30] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," *IEEE Transactions on Software Engineering*, 2017.