# Learning Semantic Representations to Verify Hardware Designs

**Anonymous Author(s)**
Affiliation
Address
email

## Abstract

Verification is a serious bottleneck in the industrial hardware design cycle, routinely requiring person-years of effort. Practical verification relies on a "best effort" process that simulates the design on test inputs. This suggests a new research question: Can this simulation data be exploited to learn a continuous representation of a hardware design that allows us to predict its functionality? As a first approach to this new problem, we introduce Design2Vec, a deep architecture that learns semantic abstractions of hardware designs. The key idea is to work at a higher level of abstraction than the gate or the bit level, namely the Register Transfer Level (RTL), which is somewhat analogous to software source code, and can be represented by a graph that incorporates control and data flow. This allows us to learn representations of RTL syntax and semantics using a graph neural network. We apply these representations to several tasks within verification, including predicting what cover points of the design will be exercised by a test, and generating new tests that will exercise desired cover points. We evaluate Design2Vec on three real-world hardware designs, including an industrial chip used in commercial datacenters. Our results demonstrate that Design2Vec dramatically outperforms baseline approaches that do not incorporate the RTL semantics, scales to industrial designs, and can generate tests that exercise design points that are currently difficult to reach with manually written tests by design verification experts.

## 1 Introduction

Hardware designs are verified to check if a design implements the architectural specification correctly. Verification is widely considered the most serious bottleneck in the contemporary industrial hardware design cycle [2, 16, 15]. It requires 60-75% of the time, compute, and human resources during the design phase, routinely taking multiple person-years of effort. Verification is a complex problem because modern hardware has a massive number of inputs and flops (on the order of billions), and the number of states is exponential in the number of flops and inputs. Checking every state of the system is infeasible due to a combinatorial state space explosion. Despite decades of research in automatic hardware verification involving formal verification and static analysis, their applicability in practice is limited by scale. Instead, practical design verification sacrifices automation and completeness for a "best effort, risk reducing" process based on simulating the design on test inputs.[1]

Traditionally, formal verification methods analyze synchronous hardware designs through reachability analysis of a gate level state transition graph. Each node in this graph corresponds to a single value of bit-level state of all registers, and edges correspond to the legal changes in state that the design can make in a single clock cycle. It is well known in the formal methods community that most questions

[1] All future references to verification will imply simulation based design verification as used in industry, not formal verification.

Submitted to 35th Conference on Neural Information Processing Systems (NeurIPS 2021). Do not distribute.

about hardware functionality can be posed as reachability questions in the gate-level graph. This suggests a fundamental new research question in representation learning: *Can we learn a continuous representation of a hardware design that allows us to predict its functionality?*

This paper presents a first approach to this problem. Since designs are practically verified by simulations using millions of test inputs, this gives us a ready source of training data. To avoid the combinatorial explosion at the gate level, we approach the problem at a higher level of abstraction, the Register Transfer Level (RTL) that describes hardware at the bit-vector or integer level. RTL is described in a Hardware Description Language (Verilog RTL [3]) that is syntactically similar to the source code of software, while modeling the concurrent, non-deterministic, non-terminating, and reactive semantics of hardware. Despite the higher level of abstraction, RTL static analysis approaches for reachability analysis [4] and test input generation [30] do not scale to even reasonably sized practical designs. So our research question can be restated as: *Can we use simulation data to learn a tractable continuous representation that can predict the answers to the state reachability queries in hardware?*

In this paper, we introduce Design2Vec, an architecture for learning continuous representations that capture the semantics of a hardware design. We choose to represent the design as a control data flow graph (CDFG) at the RTL level. Based on the CDFG, we use graph neural networks (GNN) to learn representations that predict states reached by the design when simulated on test inputs. While standard GNN architectures do work well, we achieve improved performance by introducing a new propagation layer that specifically incorporates the concurrent and non-terminating semantics of RTL. Design2Vec is trained to predict if, given a simulation test input, a particular branch (like a case statement in software) will be covered. The CDFG is an abstraction of the gate-level state space since one edge in the CDFG maps to many edges in the gate-level transition graph. We can therefore interpret Design2Vec as learning an *abstraction* of the intractable gate-level design state space.

We apply Design2Vec to two practical problems in hardware verification: coverage prediction and test generation. The coverage prediction model can act as a proxy simulator, and an engineer can query it to estimate coverage in seconds, instead of waiting for a night of simulations. Our test generation method uses a gradient-based search over the trained Design2Vec model to generate new tests. We empirically observe that our approach can generate tests for *hard to cover* points, which can take many nights of simulations to cover using black-box optimization or manual approaches.

We demonstrate Design2Vec is able to successfully learn representations of multiple designs: (i) IBEX v1 [1], a RISCV design (ii) IBEX v2, IBEX enhanced with security features, and (iii) FastML[2], an industrial scale commercial infrastructure ML accelerator chip, and performs dramatically better than black-box baselines that are uninformed by knowledge of the design. Our key contributions follow.

- We introduce the problem of learning continuous representations (abstractions) of hardware semantics that can be used for various tasks in hardware verification. We present Design2Vec, a model that encodes the syntactic and semantic structure of an RTL design using a GNN based architecture.
- We propose the RTL IPA-GNN, which extends a recent architecture for learning to execute software [6] to model the concurrent and non-terminating semantics of RTL.
- We introduce a test generation algorithm that uses Design2Vec to generate focused tests for a subset of cover points. The tests generated by Design2Vec are able to successfully cover hard to cover points, while requiring significantly fewer simulations than a black-box method.
- We demonstrate the scalability of the approach to large industrial scale designs (Table 7). Our approach is practical in industrial settings.

## 2 Design2Vec: Representation learning of design abstractions

### 2.1 Modeling of RTL designs as CDFGs

Hardware is designed using RTL source code as non-terminating, continuous, reactive "programs". It is structured as a modular hierarchy, with multiple concurrent blocks executing within a module. We consider the synthesizable subset of behavioral RTL that can be synthesized into gate level designs [3].

[2]Name withheld for blind review.

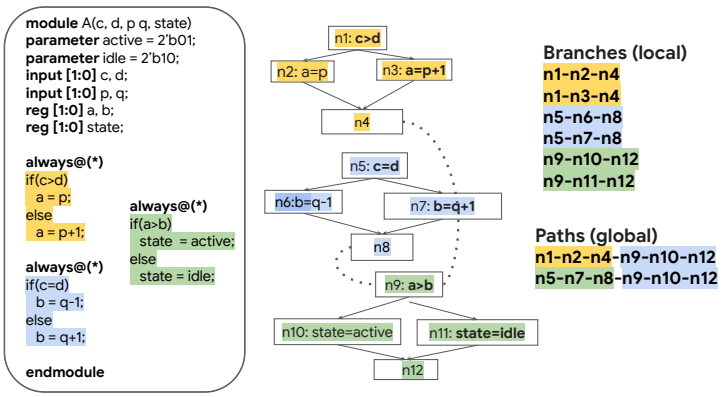


Figure 1: An example snippet of RTL source code module and the corresponding CDFG.

Figure 1 shows an example snippet of Verilog RTL source code. A "module" corresponds to a hardware construct (eg. a decoder or ALU). A module can instantiate other modules (eg. a cache module might instantiate a prefetch module). Each "variable" in the RTL corresponds to either an input signal (input), an output signal (output), a register that stores values (reg) or a temporary variable (wire). The bit width of each such variable is declared. In the example, `a,b,state` are 2 bit registers, `c,d,p,q` are 2 bit input signals.

This design has three concurrent (always) blocks that execute when any of the conditions within the sensitivity list (*) like reset, are triggered. For synchronous hardware, an external clock signal triggers execution of each concurrent block per *clock cycle*. Each always block is executed on every clock cycle and one statement within an always block is executed. In subsequent cycles, statements are executed in sequence within an always block. Each statement in the current cycle executes with *variable values from the previous cycle.* There is an implicit loop back from the end to the start of each block, to simulate the non-terminating nature of hardware. In any cycle, the order in which the always blocks are executed is *non-deterministic*. In practice, it is as determined by an RTL simulator.0

In the example, at a given clock cycle `t`, the values of `a` and `b` from the previous clock cycle `t-1` will be used for evaluating the condition `a > b` in the green always block and the corresponding branch will be executed in that cycle. In the other two always blocks, in cycle `t`, `b` and `a` will be assigned values based on values of `c` and `d` from previous cycle `t-1`. In the next cycle `t+1`, `a` and `b` will get values from `a` and `b` from cycle `t`.

While the typical mode of using RTL designs is to simulate using an RTL design simulator, we build a control data flow graph (CDFG) for Verilog RTL to represent its simulation semantics. This is with the purpose of using it for ML-based analysis. Figure 4 shows the execution (simulation) over a three cycle window of this example. The CDFG has heterogeneous node types and edge types. Nodes correspond to statements in code. Edges corresponds to data and control flow. Dotted lines show data dependencies between concurrently executing blocks. Branches denote localized control edge sequences, while paths are global control edge sequences from root node to a given node. The designs that we consider contain branches, but not loops, which are atypical in synthesizable RTL.

Branch coverage of a test refers to branches executed by that test. Each branch is referred to as a *cover point*. A cover point over the RTL CDFG is computed as a sequence of nodes corresponding to a local branch. Two local branch cover points lie along the same global path if they have ancestor/descendant relationship. Figure 5 in the appendix shows the branches taken for the given input stimulus.

A *test* is a set of high level parameters, each of which can be Boolean, integer or categorical. Each parameterized test triggers some constraining and randomization of inputs through the testbench. The output of the testbench is Boolean input stimuli applied to the RTL design under test. These stimuli execute different design paths. When a path is executed, all the local branches along that global path are said to be covered.

### 2.2 Architecture

We use a GNN-based architecture for Design2Vec (shown in Figure 2) and train it on the supervised task of coverage prediction. Our training data involves tests (set of parameters and constraints) and coverage vector of all code cover points with covered/not covered. For evaluation, we find if the

coverage of a cover point that has never been covered/seen in the training data (always 0) can be predicted accurately. This is the quantity that is being predicted as is_hit(cover_point, test).

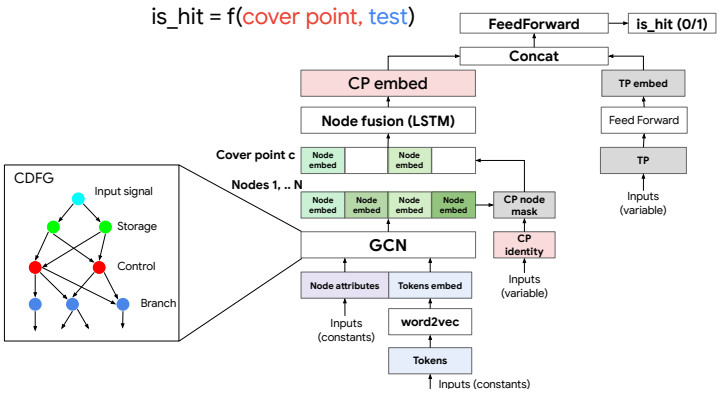


Figure 2: The Design2Vec architecture that takes as input a cover point and an input test vector, and predicts the corresponding coverage. CDFG: Control and data flow graph. TP: test parameter. CP: cover point.

**Cover point embedding:** We use a graph convolution network (GCN) over the Verilog CDFG. Each node $n \in N$ has attributes/features denoted by $f_j^n$ such as node identifier, node type, fanin and fanout. In addition, the RTL token sequence $s^n =< s_1^n, s_2^n, \cdots, s_k^n >$ in the program statement is also considered as an additional attribute. We first perform an embedding of the token sequence $\phi(s^n)$ using an LSTM (or alternatively using a pre-trained word2Vec module):

$$\phi(s^n) = \mathsf{LSTM}(< s_1^n, s_2^n, \cdots, s_k^n >) \tag{1}$$

The initial embedding of a node $\phi^{(0)}(n)$ is computed as concatenation of the node attribute values and the token sequence embedding.

$$\phi^{(0)}(n) = \mathsf{Concat}(< \phi(s^n), f_1^n, \cdots, f_l^n >) \tag{2}$$

Let the embedding of a graph $G$ at a step $t$ be denoted by $\psi^{(t)}(G) = \{\phi^{(t)}(n) | \forall n \in N\}$. We use a GCN (with learnable parameters $\theta$) to perform a sequence of graph convolution steps to update the node embeddings:

$$\psi^{(t+1)}(G) = f_\theta(\psi^{(t)}(G)) \qquad \forall t \in \{0, 1, \cdots, T\} \tag{3}$$

A cover point $C =< n_1, n_2, \cdots, n_m >$ is a sequence of nodes. The cover point identity is used to select the corresponding sequence of node embeddings of the cover point nodes. Since these sequences are of varying length, an LSTM layer is used to learn the embeddings of the cover points.

$$\phi^{(T)}(C) = \mathsf{LSTM}(< \phi^{(T)}(n_1), \phi^{(T)}(n_2), \cdots, \phi^{(T)}(n_m) >) \tag{4}$$

**Test parameter embedding**: For the input test parameters $I$, we learn an embedding for the parameters by passing them through a feed forward MLP layer. The test parameters are either integer or enum data types.

$$\phi(I) = \mathsf{MLP}(\mathsf{Concat}(i_1, \cdots, i_p)) \tag{5}$$

**MLP layer**: Finally, the test parameter $\phi(I)$ and cover point embeddings $\phi^{(T)}(C)$ are concatenated and provided to a feed forward layer with sigmoid activation to predict is_hit() for each cover point.

$$\mathsf{is_hit}(C) = \mathsf{MLP}_\sigma(\mathsf{Concat}(\phi^{(T)}(C), \phi(I))) \tag{6}$$

Given a supervised dataset of test inputs and the corresponding coverage information of different cover points in the RTL design, the network is trained using binary cross entropy loss. We find that our model learns more from local paths than global paths (root to cover point). A reason for this is that GNNs are not proficient at learning long path information. In the case of a control flow, tracing and learning global paths is relevant. We use shortcut edges, where we add edges along every kth node along a path, to reinforce the relationship of long paths (results in Appendix D). We use a variation of the GNN, called the GNN-MLP [7] to propagate edge information. We also used gating layers and residual layers in the GGNN.

**Algorithm 1** Test Input Generation

Input: A set of cover points $C = \{c_1, \cdots, c_m\}$
$\texttt{uncov} \leftarrow C$
$i \leftarrow \texttt{random}(); I = \{i\}$
**while** $\texttt{uncov} \neq \emptyset$ **do**
  $\texttt{obj} \leftarrow \Sigma_C \texttt{Design2Vec}(i)$
  $i' \leftarrow i + \nabla_i \texttt{obj}$
  $I \leftarrow I.\texttt{append}(i'); i \leftarrow i'$
  $\texttt{cov} \leftarrow \texttt{GetCov}(\texttt{Design2Vec}(i'))$
  $\texttt{uncov} \leftarrow \texttt{uncov} \setminus \texttt{cov}$
**end while**
**return** $I$

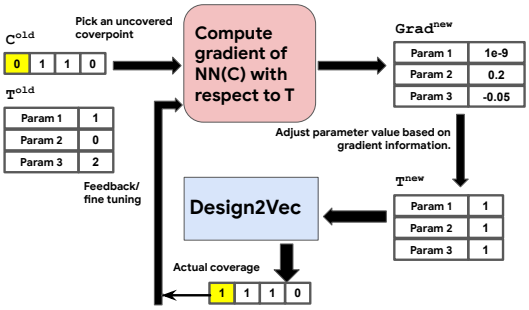


Figure 3: The overall workflow to use Design2Vec in generative loop to generate input test parameters.

### 2.3 RTL IPA-GNN architecture

We enhance the recently proposed Instruction Pointer Attention Graph Neural Network (IPA-GNN) architecture [6] that explicitly models control flow in programs. Our novel RTL IPA-GNN architecture can model the concurrent, non-terminating semantics of RTL. We make the following architectural decisions beyond that of the original IPA-GNN: Since RTL hardware is highly parallel, the RTL IPA-GNN maintains a separate instruction pointer for each always block in the hardware design. To model non-termination, we introduce a control edge from the end of each always block to the start of the always block. Data flow in an RTL design is also more complex than in a single-threaded program. We model data flow in the RTL IPA-GNN architecture by propagating messages between nodes along data flow edges at each step of the model. The control flow conditions found in RTL are also more complex than those in the prior work, and accordingly we modify the IPA-GNN model to support these conditions. A formal description of the RTL IPA-GNN is provided in Appendix B. By incorporating RTL semantics in our modeling decisions, in some settings coverage prediction performance improves compared with domain independent graph neural networks.

### 2.4 Gradient-based search for test generation

We now present an algorithm to generate input tests for an RTL design that covers different cover points using a trained Design2Vec model. The overall workflow of the algorithm is shown in Figure 3. The key idea of the algorithm is to first define an objective function as the sum of the Design2Vec model prediction probabilities of reaching the desired cover points, and then use a gradient based search to update the test parameters to maximize the objective function.

The test generation algorithm is shown in Algorithm 1. The algorithm takes as input a set of `uncovered` cover points for which we would like to generate test inputs. It first predicts the RTL coverage on an initial set of test parameters $I_0$ using the trained Design2Vec model. The algorithm then computes the objective function as the sum of probabilities of covering the input cover points, and computes gradients of the objective function with respect to each test parameter that are further used to update the test parameter values $I_1$. The new test parameters are then used to predict the new coverage vector and all the cover points that are predicted to be covered are removed from the `uncovered` set. This process is repeated until the `uncovered` set becomes empty. Finally, the tests with highest predicted coverage are run through an RTL simulator to provide ground truth.

## 3 Context and impact of Design2Vec solutions in verification

In this section, we detail the verification flow, and highlight the value proposition brought by the Design2Vec solutions to this problem. Figure 6 in the appendix shows the industrial verification flow.

The current approaches for simulating such RTL designs follow a constrained random verification [9] approach. This involves a complex program called the testbench, through which some inputs can be constrained to legal values, while others can be randomized, to widely sample the input space. Developing the testbench, generating input constraints, as well as parameterizing the testbench into

high level tests are all manual and resource intensive tasks that take many person months. In state of practice, verification engineers write a high level parameterized test, which triggers some constraints in the testbench, and randomizes other inputs. The testbench output is a Boolean level stimulus that is applied to the primary inputs of the RTL design. The simulation takes around a night, and is called nightly regression. Bug reports and coverage are assessed at the end of a nightly regression. This process is iterated upon until there are 0 bugs and 100% of the design cover points are covered. Simulator calls are costly and need to be minimized.

We select two verification tasks for Design2Vec: (i) coverage prediction, or predicting which cover points in the design would be covered by a given test. Such a predictive model can serve as a proxy simulator, and an engineer can query it to estimate coverage instantaneously, instead of waiting for a night of simulation. (ii) test parameter generation, or automating tests towards faster coverage closure. We expect the test generation to be useful for covering hard-to-cover points in the verification cycle. Design2Vec performs two functions in these tasks. First, it summarizes the state space of the RTL design through the RTL CDFG, and learns how to reach different states (nodes) of the design (graph), given an input test stimulus. Second, *it appproximates the functionality of the testbench*, and bypasses it by directly correlating the high level test parameters to design coverage.

# 4 Coverage prediction experiments

We first evaluate the Design2Vec architecture on the task of coverage prediction. Our key research questions are (i) to evaluate whether Design2Vec is able to exploit information from the RTL hardware design to improve predictions and (ii) to evaluate whether the performance of Design2Vec can add practical value to industrial-scale designs.

We evaluate our methods on the three real-world designs from Table 7. For each design, we obtain training data by generating random tests and sampling each test parameter uniformly. For each test, we use the testbench to randomly sample input test stimulus, and a Verilog RTL simulator to obtain ground-truth labels of whether a cover point is covered by that test or not, in the form of a coverage vector. We sample 1696 tests for IBEX v1, 1938 for IBEX v2, and 1995 for FastML. Some cover points are very easy to predict, e.g., they are covered by almost 0% or nearly 100% of the random tests. To avoid evaluating our models on these trivial cases, we only include cover points in the data if they are covered by between 10% and 90% of the random tests. After this filtering, there are 160 cover points in the data set for the IBEX v1 design, 177 for IBEX v2, and 3781 for FastML.

Furthermore, there is a particular type of generalization that is vital to applying coverage prediction in practice. Recall that when a design engineer uses a coverage prediction model, it is because they are proposing a new test that should exercise a cover point that is not exercised by the existing test. Therefore, in practice the primary concern is how well the model predicts whether a test that it has never seen during training will exercise a cover point *that the model has never seen how to cover during training*. For this reason, we divide the data into training and test point by cover point, such that no cover point and no test occurs in both the training and validation set for the learned models.[3] All results are the median over three random train-validation splits.

Our key question is whether the graph neural networks allow Design2Vec to generalize from cover points in the training set to related cover points, e.g., nearby in the RTL CDFG, which were not covered. Therefore, as a baseline we compare to a multilayer perceptron (MLP) that treats the design as a black box, and does not take into account any information from the RTL. The MLP takes as input the test parameters, represented the same as in Design2Vec, and the numeric index of the cover point, represented as a one-hot vector. Hyperparameters including number of layers, learning rates, and embedding dimensions are reported in Appendix C.

First, Table 1 compares Design2Vec to the black-box MLP across all three designs, varying the size of the training set. In these results, Design2Vec uses the RTL IPA-GNN (Section 2.3) as the GNN layer. Notably, the MLP performs catastrophically poorly on the test data, performing close to chance. Indeed, on the largest design, the industrial FastML design, Design2Vec has an accuracy of 47% higher *in absolute terms* than the MLP. This is because the MLP overfits. We observe that the training accuracy of MLP is high (above 95%), so the model is failing to generalize to cover points outside the

[3] We avoid using the term "test set" to refer to the data on which we evaluate our machine learning methods, to avoid confusion with tests of the RTL design.

Table 1: Accuracy at coverage prediction for Design2Vec (with the RTL IPA-GNN layer) compared to a black-box multi-layer perceptron (MLP), which does not have information about the design.

| | IBEX v1 | | | IBEX v2 | | | FastML | | |
|---|---|---|---|---|---|---|---|---|---|
| Train cover points | 50% | 80% | 90% | 50% | 80% | 90% | 50% | 80% | 90% |
| Design2Vec | **74.2** | **77.3** | **77.8** | **73.4** | **78.0** | **80.3** | **90.5** | **90.6** | **91.1** |
| MLP | 57.5 | 56.8 | 56.8 | 58.7 | 58.0 | 58.2 | 42.79 | 42.46 | 34.7 |

Table 2: Coverage prediction accuracies of different GNN architectures within Design2Vec.

| | | IBEX v1 | | | IBEX v2 | | | FastML | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Node fusion | 50% | 80% | 90% | 50% | 80% | 90% | 50% | 80% | 90% |
| GCN | lstm | 74.0 | 73.0 | 73.2 | 70.4 | 74.5 | 73.9 | 90.9 | 90.5 | 91.1 |
| | mean | 74.1 | 75.8 | 74.0 | 69.0 | 73.9 | 72.5 | — | — | — |
| GGNN | lstm | **75.0** | 75.9 | 74.5 | 71.4 | 77.0 | 76.0 | **91.1** | 90.4 | **91.2** |
| | mean | 73.8 | 75.6 | 76.4 | 70.0 | 75.1 | 72.3 | — | — | — |
| GNN-MLP | lstm | 73.8 | 76.8 | **78.5** | 70.7 | 74.5 | 71.2 | 91.0 | 90.1 | 91.1 |
| | mean | 73.0 | 76.4 | 74.4 | 70.1 | 75.1 | 72.0 | — | — | — |
| RTL IPA-GNN | lstm | 74.2 | **77.3** | 77.8 | **73.4** | **78.0** | **80.3** | 90.5 | **90.6** | 91.1 |
| | mean | 73.6 | 77.2 | 76.3 | 72.2 | 75.8 | 78.9 | — | — | — |

training set. This is expected, because the model has no information about which of the training cover points are most close to the cover points in validation set. Such information can only be obtained from the design. The MLP using cover point identities has no access to it. The MLP prediction is based only on test parameter features. The Design2Vec model, on the other hand, is able to use the RTL design to generalize effectively to cover points unseen during training. This pattern holds across all three designs. On IBEX v1 and v2, the difference between the two models is smaller (although still around 15% absolute). This is due to the irregular structure of the Ibex designs, with fewer repeated hardware modules, causing the control flow path to be harder to predict. However, on the FastML industrial design is the largest and most complex of the three, and on this design, Design2Vec has over 90% accuracy. This can be attributed to the highly regular structure of FastML with a relatively easier to predict control flow than Ibex. Given that FastML is a typical ML accelerator chip whose architecture will have similar properties across generations, this shows the potential of Design2Vec to serve as a proxy simulator.

For Design2Vec, we also compare several variants of graph neural networks: graph convolution networks [24], gated graph neural networks [27], GNN-MLP [7], and the RTL IPA-GNN (Section 2.3). These results are shown in Table 2. Additionally, we vary the method by which representations of CDFG nodes are aggregated to represent cover points. Recall from Section 2.2 that to represent a cover point, we take the final node embeddings from the GNN, apply a mask to obtain only a few relevant CDFG, and then aggregate them. We compare using an LSTM, as in (4), to simply taking the mean. We compare this for all of the different GNN architectures, except for the FastML architecture, because experiments were more computationally expensive to run, we noted that aggregation methods performed similarly on IBEX, so we evaluated only LSTM aggregation on FastML. Overall, all of the GNN variants performed similarly. RTL IPA-GNN performs similarly to the other GNN architectures on IBEX v1 and FAST ML, but was significantly better on IBEX v2, showing that in some designs, a GNN architecture can be improved by taking into account more information about RTL semantics. Because the RTL IPA-GNN performs similarly or better to other GNNs across the three designs, we use this as the main Design2Vec model in Table 1. We also tuned several other features of the architecture, including presence of the residual connections, embedding size, and label smoothing (details in Appendix C). We also tested whether the results are sensitive to the details of the experimental design. Cover points are related to each other, in the sense that cover points from the same nested branch provide much more information about each other than points in two different always blocks. To show our conclusions are not dependent on how the cover points in the training set are sampled, we compare two different sampling methods: random, and hiding every $k$-th cover point (order is based on the syntax tree of the RTL design). We found that difference in performance was not significant, and that they do not affect our overall conclusions (see supplementary).

Table 3: Comparison of Design2Vec and black-box optimizer tests for covering hard to cover points: IBEX v1.

| | Cover probability (separate dataset) | Number of tests (simulator calls) to cover | |
|---|---|---|---|
| Hard cover point | Random 1700 tests | Vizier | Design2Vec |
| 526 | 0.0059 | Not covered | **12** |
| 879 | 0.0071 | 42 | **14** |
| 882 | 0.0059 | 24 | **3** |
| 886 | 0.0018 | 93 | **25** |

# 5 Test generation using Design2Vec

We use the Design2Vec model to generate a test for a given set of input cover points and focus on *hard to cover points*, since covering them using constrained random stimulus takes very long (100s of calls to the simulator and person-hours trying to write constraints). We define *hard to cover points* as those identified by human engineers as taking more than 5 nights of simulation.

The two RTL testing approaches are directed testing and random testing. Directed testing refers to targeted testing of specific functionality in RTL, whereas random testing refers to random perturbation of inputs with the goal of covering the design space maximally. Although Design2Vec based test generation is directed, we did not find a comparable directed testing tool in open source (or commercially). We compare tests generated by Design2Vec with a random testing tool based on Vizier [18] that uses black-box optimization to maximize total coverage all cover points with every test. This comparison is inherently unequal due to the difference in their objectives.

We compare the two tools by i) providing them with the same datasets and ii) measuring the total coverage achieved against the number of simulator calls made. Given that simulator calls are very expensive, this is a relevant metric to compare. Vizier is allowed to run a certain number of trials. In each trial, it generates a test for the total set of cover points in the design, and calls the simulator. The simulator provides actual coverage of the test, which is used as feedback by Vizier. Vizier uses Bayesian optimization and coverage feedback to generate tests with progressively higher coverage.

We intercept Vizier after running for a number of trials $N$ ($N$=200 for IBEX v1, =243 for FastML). We provide the same $N$ sample tests collected by Vizier and the corresponding coverage vector per test as training data for Design2Vec. We identify the set of unseen cover points by Vizier $N$ trials as $C_{\text{uncov}}$. We provided them as inputs to Design2Vec test generation algorithm described in Table 1. Design2Vec generates multiple test recommendations, of which we select the top test and send it to the simulator for evaluating actual coverage. If the target cover point is uncovered, we generate a different test using Design2Vec until it is covered. To compare to Vizier, we continue to run Vizier beyond $N$ trials and record the coverage at every trial. For cover points in $C_{\text{uncov}}$, we measure the number of simulator calls (i.e., tests) taken by Vizier to cover it for the first time, and measure the number of simulator calls used by Design2Vec to cover it. The points in $C_{\text{uncov}}$ tend to be rare and hard to cover, as evidenced by their cover probabilities on a randomly sampled dataset. Note that we do not train on this separate dataset, but use it to indicate the rareness of the points.

Table 3 shows the results of this comparison. Cover point 526 is covered by Design2Vec in just 12 simulator calls (12 tests), whereas Vizier is never able to cover this point. The cover probability of this point in a random set of 1700 tests is very low. Similarly, for 882, due to its low probability, Vizier needs 24 tests to cover it, while Design2Vec uses only 3 tests to cover.

Interestingly, when we provide multiple neighboring cover points with the same branch index (sibling nodes in the CDFG), the hard to cover points benefit from the easy to predict cover points in their locality. Table 5 demonstrates this important insight in our test generation. When the input set includes multiple cover points with the same branch index, it results in highly successful tests. This is a consequence of GNN-based learning that is highly effective in the local neighborhood of the design.

Table 4 shows similar results on the hard to cover points in the FastML design. These cover points are of practically 0 probability in a randomly sampled dataset. Design2Vec can still cover them in as low as 2 simulator calls. **This result shows the power of representation learning in Design2Vec. Despite the lack of data and few examples of hard to cover points, our Design2Vec model is able to generate a test to cover these cover points with orders of magnitude fewer simulator calls than the black-box optimizer and random testing.**

Table 4: Comparison of Design2Vec and black-box optimizer tests for covering hard to cover points: FastML

| | Cover probability (separate dataset) | Number of tests (simulator calls) to cover | |
|---|---|---|---|
| Hard cover point | Random 1995 tests | Vizier | Design2Vec |
| 35793 | 0.0 | 22 | **17** |
| 36996 | 0.0 | 90 | **2** |
| 36372 | 0.0 | 78 | **17** |

Table 5: Example tests generated by Design2Vec for hard to cover points. Multiple cover points that are local neighbors of the target point are provided as input to Design2Vec.

| Cover points | Top test recommendations |
|---|---|
| $\{521, \ldots \mathbf{526}, \ldots, 531\}$ | `+instr_cnt=17600 +illegal_instr_ratio=35`<br>`+hint_instr_ratio=45 ... +enable_unaligned_load_store=0`<br>`+disable_compressed_instr=1 +randomize_csr=0 +no_wfi=1` |
| $\{881, \ldots \mathbf{882}, \ldots, 891\}$ | `+instr_cnt=17400 +illegal_instr_ratio=25`<br>`+hint_instr_ratio=35 ... +enable_unaligned_load_store=0`<br>`+disable_compressed_instr=1 +randomize_csr=0 +no_wfi=1` |

# 6 Related work

**Automated test generation for RTL:** Most previous techniques for RTL test generation generate Boolean level stimuli at the inputs of the RTL design [35, 30, 31, 26, 8]. By virtue of operating at a higher abstraction level of parameterized tests, it is orders of magnitude more scalable than the Boolean input stimulus generation techniques. Static analysis based approaches for test generation [33, 19, 5] are based on traversing the state transition graph at a logic gate level, or the RTL design have inherent scalability limitations. Approaches that combine static and dynamic analysis like HYBRO [28] and concolic testing [35, 30, 31] in RTL rely on SMT/SAT solvers, which are also limited by scale. Other approaches random forests and decision trees with static analysis and formal verification [29, 14] require manual feature engineering and handcrafted algorithms.
**Neural program testing:** While our approach learns the semantics of a design, learning based fuzzing [17] does not take program semantics into account. In contrast to our approach that models semantics of an RTL design, Neuzz [34] approximates a program using only the input-output behavior of the program, which is similar to our blacklbox MLP baseline. GMETAEXP [11] models test generation as a reinforcement learning problem with the objective of maximizing total program coverage, where programs are represented using a GGNN.
**Learning to Execute:** Among methods that model program semantics, [37] present an approach to use LSTM to embed a program as input and generate the output as the output sequence. IPA-GNN [6] represents the program execution using an RNN augmented with a differentiable mechanism to represent next instruction after a statement execution. Unlike these works that learn to generate program output, we tackle the problem of reachability, or learning to reach a state/node in a graph.
**Abstractions for verification:** Abstraction techniques have been used to scale design verification for many decades. Some techniques are property specific [10, 25], design specific [20], data abstractions (word level abstractions) [23, 36], language based (abstract interpretation) [12, 22], execution or structure based [13]. These abstractions are typically defined manually and are challenging to create in a precise way to both scale the verification as well as maintain desired precision. In contrast, GNN based abstractions are task-specific, in the form of continuous representations of the CDFG nodes, that generalizes both across tasks, as well as designs.

# 7 Conclusion

We present an approach that is able to learn representations of hardware designs that can predict their functionality. With Design2Vec, we demonstrate, for the first time, the ability of deep learning in creating abstractions of the hardware design state space. These abstractions outperform black-box baselines and human baselines (state of practice) in verification by orders of magnitude. They can also generalize across verification tasks, and be applied to different designs. Since Design2Vec learns reachability over the RTL design graph, it can also potentially generalize to other verification tasks like debugging, property generation and root causing. More broadly, this work shows the power and potential of deep learning to make a quantum leap in progress in the area of verification, which is considered a longstanding practical and scientific challenge in computing.

## References


[1] IBEX RTL source. https://github.com/lowRISC/ibex.

[2] Functional Verification study. https://blogs.sw.siemens.com/verificationhorizons/2021/01/06/part-8-the-2020-wilson-research-group-functional-verificatio

[3] Ieee standard for systemverilog–unified hardware design, specification, and verification language. *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, pages 1–1315, 2018. doi: 10.1109/IEEESTD.2018.8299595.

[4] Viraj Athavale, Sai Ma, Samuel Hertz, and Shobha Vasudevan. Code coverage of assertions using rtl source code analysis. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2014. doi: 10.1145/2593069.2593108.

[5] K. Bansal and M. S. Hsiao. Optimization of mutant space for rtl test generation. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*, pages 472–475, 2018.

[6] David Bieber, Charles Sutton, Hugo Larochelle, and Daniel Tarlow. Learning to execute programs with instruction pointer attention graph neural networks. *CoRR*, abs/2010.12621, 2020. URL https://arxiv.org/abs/2010.12621.

[7] Marc Brockschmidt. GNN-FiLM: Graph neural networks with feature-wise linear modulation. In *International Conference on Machine Learning (ICML)*, 2020.

[8] Eris Furkan Taylor Michael Bedford Egele Manuel Canakci Sadullah, Delshadtehrani Leila and Joshi Ajay. Directfuzz: Automated test generation for rtl designs using directed graybox fuzzing. In *58th IEEE/ACM Design Automation Conference Proceedings, 2021*, 2021.

[9] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. Balancing scalability and uniformity in SAT witness generator. *CoRR*, abs/1403.6246, 2014. URL http://arxiv.org/abs/1403.6246.

[10] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, September 2003.

[11] Hanjun Dai, Yujia Li, Chenglong Wang, Rishabh Singh, Po-Sen Huang, and Pushmeet Kohli. Learning transferable graph exploration. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *NeurIPS*, pages 2514–2525, 2019.

[12] Dennis Dams, Rob Gerth, and Orna Grumberg. Abstract interpretation of reactive systems. *ACM Trans. Program. Lang. Syst.*, 19(2):253–291, 1997.

[13] Sistla A.P. Emerson, E.A. Symmetry and model checking. In *Formal Methods in System Design*, pages 105–131, 1996.

[14] Monica Farkash, Bryan Hickerson, and Michael Behm. Coverage learned targeted validation for incremental hw changes. In *Proceedings of the 51st Annual Design Automation Conference*, page 1–6, 2014.

[15] Harry Foster. Why the design productivity gap never happened. In Jörg Henkel, editor, *The IEEE/ACM International Conference on Computer-Aided Design, ICCAD'13, San Jose, CA, USA, November 18-21, 2013*, pages 581–584. IEEE, 2013. doi: 10.1109/ICCAD.2013.6691175. URL https://doi.org/10.1109/ICCAD.2013.6691175.

[16] Harry D. Foster. Trends in functional verification: a 2014 industry study. In *Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, June 7-11, 2015*, pages 48:1–48:6. ACM, 2015. doi: 10.1145/2744769.2744921. URL https://doi.org/10.1145/2744769.2744921.

[17] Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&fuzz: machine learning for input fuzzing. In Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen, editors, *ASE*, pages 50–59. IEEE Computer Society, 2017.

[18] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Elliot Karro, and D. Sculley, editors. *Google Vizier: A Service for Black-Box Optimization*, 2017. URL http://www.kdd.org/kdd2017/papers/view/google-vizier-a-service-for-black-box-optimization.

[19] Yang Guo, Wanxia Qu, Tun Li, and Sikun Li. Coverage driven test generation framework for rtl functional verification. In *2007 10th IEEE International Conference on Computer-Aided Design and Computer Graphics*, pages 321–326, 2007.

[20] Rajiv Gupta, Mary Jean, Harrold Mary, and Lou Soffa. Program slicing-based regression testing techniques+. *Journal of Software Testing, Verification, and Reliability*, 6:83–112, 1996.

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016. doi: 10.1109/CVPR.2016.90.

[22] Charles Hymans. Design and implementation of an abstract interpreter for vhdl. In Daniel Geist and Enrico Tronci, editors, *Correct Hardware Design and Verification Methods*, pages 263–269, 2003.

[23] Pnueli A. Kesten, Y. Control and data abstraction: the cornerstones of practical formal verification. *International Journal of Software Tools for Technology Transfer*, 2:328–342, 2000.

[24] Thomas N Kipf and Max Welling. Semi-Supervised classification with graph convolutional networks. In *International Conference on Learning Representations (ICLR)*, 2017.

[25] Daniel Kroening, Alex Groce, and Edmund Clarke. Counterexample guided abstraction refinement via program execution. In *International Conference on Formal Engineering Methods*, volume 3308, 08 2004.

[26] Kevin Laeufer, Jack Koenig, Donggyu Kim, Jonathan Bachrach, and Koushik Sen. Rfuzz: Coverage-directed fuzz testing of rtl on fpgas. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, 2018.

[27] Yujia Li, Richard Zemel, Marc Brockschmidt, and Daniel Tarlow. Gated graph sequence neural networks. In *International Conference on Learning Representations (ICLR)*, 2016.

[28] Lingyi Liu and Shobha Vasudevan. Efficient validation input generation in rtl by hybridized source code analysis. In *2011 Design, Automation Test in Europe*, pages 1–6, 2011.

[29] Lingyi Liu, David Sheridan, William Tuohy, and Shobha Vasudevan. A technique for test coverage closure using goldmine. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 31(5):790–803, 2012.

[30] Yangdi Lyu and Prabhat Mishra. Automated test generation for activation of assertions in RTL models. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, page 223–228, 2020.

[31] Yangdi Lyu, Xiaoke Qin, Mingsong Chen, and Prabhat Mishra. Directed test generation for validation of cache coherence protocols. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(1):163–176, 2019.

[32] Rafael Müller, Simon Kornblith, and Geoffrey E. Hinton. When does label smoothing help? In *Advances in Neural Information Processing Systems*, pages 4696–4705, 2019.

[33] Jen-Chieh Ou, Daniel G. Saab, Qiang Qiang, and Jacob A. Abraham. Reducing verification overhead with rtl slicing. In *Proceedings of the 17th ACM Great Lakes Symposium on VLSI*, page 399–404, 2007.

[34] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. NEUZZ: efficient fuzzing with neural program smoothing. In *2019 IEEE Symposium on Security and Privacy, SP 2019,*, pages 803–817. IEEE, 2019.

[35] Hasini Witharana, Yangdi Lyu, and Prabhat Mishra. Directed test generation for activation of security assertions in rtl models. *ACM Transactions on Design Automation of Electronic Systems*, 26(4), 2021.

[36] Cunxi Yu and Maciej Ciesielski. Automatic word-level abstraction of datapath. In *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1718–1721, 2016.

[37] Wojciech Zaremba and Ilya Sutskever. Learning to execute. *CoRR*, abs/1410.4615, 2014.

# Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes] We discuss overall challenges and limitations of verification in the introduction; although our work aims to improve current processes, it is not immune to these overall limitations of the field.
   (c) Did you discuss any potential negative societal impacts of your work? [N/A] This paper considers a technological problem that is several layers removed from direct social impact, so it is difficult to identify specific direct negative impacts.
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [N/A]
   (b) Did you include complete proofs of all theoretical results? [N/A]
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] Although the code is proprietary, we provide details about the model and architectures required to reproduce the main experimental results. Furthermore, we evaluate on the open source IBEX v1 and IBEX v2 designs, so that others can reproduce our work.
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] See Section C.
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [No] We run experiments multiple times and report the median over random seeds.
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] See Appendix C.
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes] We use existing real-world hardware designs, which we cite.
   (b) Did you mention the license of the assets? [N/A]
   (c) Did you include any new assets either in the supplemental material or as a URL? [N/A]
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]