Novelty-Driven Verification: Using Machine Learning to Identify Novel Stimuli and Close Coverage

Tim Blackmore and Rhys Hodson
Infineon Technologies UK Ltd, Hunts Ground Road, Bristol, UK
Email: firtsname.lastname@infineon.com

Sebastian Schaal
Luminovo GmbH, Nymphenburger Str. 84, Munich, Germany
Email: sebastian@luminovo.ai

***Abstract*- Constrained-random generation is the main method used for test-case automation for pre-silicon digital block verification. Constrained-random generator and simulator performance means that it is usually possible to close coverage using this method on a small or medium-sized block within a week or a month with little manual intervention. However, the redundancy inherent in constrained-random generation means that it can take many months and considerable manual effort to achieve 100% functional coverage for large and complex blocks. Unsupervised machine learning techniques, such as anomaly detection, look for unusual or novel behaviour. This raises the possibility of using such techniques to reduce the redundancy in simulated tests. Here we report positive results in closing coverage on a large and complex block using these techniques. The results suggest that for this block the time for coverage closure can be reduced from six months to less than three months. This still allows for considerable soak testing and, moreover, the tests used for soak testing will be different from each other. As a result of this work the Infineon Automotive Microcontroller Division is using machine learning in the verification of complex designs. We believe the techniques described here have real potential for wider practical application.**

I. Introduction

Constrained-random generation has been the workhorse of pre-silicon digital block level verification for the last twenty years or so. It enables automatic test generation which both speeds up coverage closure and allows for soak testing to uncover bugs not related to coverage items. For many blocks coverage can be closed by simulating thousands or tens of thousands of generated tests. The performance of tools for test generation and simulation mean that in such cases coverage closure can usually be completed within a week or a month. However for some blocks, typically with complex coverage models containing many items that are difficult to hit, coverage closure can take the simulation of hundreds of thousands or millions of generated tests and many months. The coverage model itself may contain only thousands of items. To speed coverage closure, coverage items that haven't been hit are analyzed and constraints are written to guide the constrained-random generation towards hitting these coverage items. For complex coverage models, the engineering effort involved in doing this can be considerable.

It is easy to hypothesize that the reason it takes millions of randomly generated tests to hit thousands of coverage items is that randomness inherently leads to a lot of redundancy. This certainly seems true when redundancy is measured with respect to the coverage model. This then leads naturally to the idea of applying some intelligence to the test generation process, and in particular to the use of machine learning. This has been explored in a number of publications – a good overview is given in [1]. The reported use of machine learning for coverage closure has typically been for coverage models that can be filled quickly and it is difficult to know whether the reported techniques would generalize to the more interesting case of a coverage model that may take months to fill.

Here we report on positive results achieved using machine learning to close coverage for a block with a complex coverage model. The machine learning was used in the verification of the RADAR Signal Processing Unit (RSPU), which is a large and highly configurable block at the heart of the Advanced Driver Assistance Systems (ADAS) subsystem in Infineon's Aurix family of microcontrollers. Achieving 100% functional coverage for this block typically requires simulating around two million tests generated using an industry-standard constrained-random generator. With an average simulation time of 2 hours this consumes around 1000 machines and licenses almost continuously for a period of six months. During this six month coverage closure phase many person months of effort are spent writing constraints to steer the generation towards hitting coverage holes. However, even with these

constraints, there is a high degree of redundancy in the test generation meaning that, using an industry standard ranking algorithm, a ranked regression of only around 3000 tests is needed to achieve 100% coverage. (Of course the simulation of all two million tests is needed in order to identify which tests to put into the ranked regression.)

In order to assess whether machine learning could be used to reduce this redundancy an experiment was conducted using around 82500 tests generated during the course of the project augmented with the 3000 ranked tests. The speed of coverage closure was compared when selecting tests randomly and when selecting tests using various machine learning techniques from these 85500 tests. The best results achieved using machine learning showed that coverage could be closed, or at least very nearly closed, with 60% fewer tests than random selection. If this was realised on an executing project this would save over a million simulations and potentially three months or more of schedule as well as many months of engineering effort. As a result of this work the Infineon Automotive Microcontroller Division is now using machine learning in the verification of complex designs. We believe the machine learning techniques described here have real potential for practical application.

The machine learning techniques used are geared towards the identification of novelty in a test. Aside from closing coverage faster, simulating novel tests may be beneficial in other ways. Redundancy in constrained-random generation has already been mentioned, but coverage-driven verification suffers from some other limitations - in particular, the derivation of a functional coverage model is subjective and the implementation is error-prone, with false positives (i.e. incorrectly coded coverage items that are hit) going unexamined. The correlation between novelty and coverage established here shows that novelty, as determined by the machine learning techniques, is meaningful, but its meaning may go beyond coverage closure and novelty may be a useful property of tests to target in order to drive verification forward. This is not explored in detail in this paper but suggestions for future work to investigate this possibility are presented.

The rest of the paper is structured as follows. Section 2 gives an introduction to test selection based on novelty and how it was used in the verification of the RSPU. Section 3 describes and discusses the machine learning techniques used for novelty detection in more depth. Section 4 gives a detailed description of the experiment executed, including an analysis of the suitability of the data, and the results of the experiment. Section 5 concludes on the experimental results and discusses future work.

II. Overview

In this section we give an overview of the RADAR Signal Processing Unit (RSPU) and its verification. We describe how machine learning can be used in principle to the speed up coverage closure and why the verification of the RSPU is suited to the use of machine learning.

The RSPU is at the heart of the Advanced Driver Assistance Systems (ADAS) subsystem in Infineon's Aurix family of microcontrollers. It receives sampled RADAR data from one or more off-chip sensors for processing before writing the processed data to memory. The written data can be read from memory across different dimensions to be re-processed multiple times to infer properties such as distance, acceleration and angle of arrival of an identified object. The RSPU can be re-configured between each such data pass. There are multiple processing stages within the RSPU each able to apply many mathematical functions and each processing stage is highly configurable.

The verification of the RSPU is coverage-driven using constrained-random generation of test cases. The generated test cases consist of the configurations of the RSPU and the data to be processed. As well as various code coverage metrics, the targeted coverage includes an extensive functional coverage model derived from various sources, including input from concept and design engineering, and using the various methods listed for test case derivation in Table 10 of Part 5 of the ISO26262:2018 standard, [2]. The functional coverage model is divided into configuration coverage and white-box coverage. The configuration coverage ensures that all identified interesting configurations have been verified and that a good spread of configurations have been verified. The white-box coverage requires monitoring of internal design state and includes coverage of all identified implementation corner cases. For the RSPU, achieving 100% functional coverage is much more time-consuming than achieving 100% code coverage. Whilst closing configuration coverage is straightforward, since a test writer can directly constrain values of configuration fields, closing white-box coverage is more difficult. Although the configuration will typically determine which white-box coverage items a test case hits, the relationship between configuration and white-box coverage is varied and complex and takes time for engineers to understand.

It usually requires the simulation of around two million test cases to close coverage on the RSPU. With an average simulation time of two hours, this uses 1000 machines and licenses almost continually for a period of six months. During this six month coverage closure phase many person months of effort are spent analysing coverage holes and writing constraints to steer the generation towards hitting coverage holes.

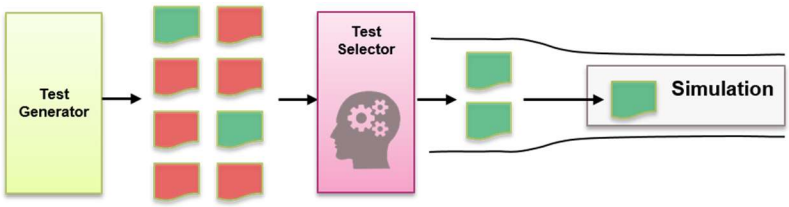

Figure 1 Test Selector between Generator and Simulator

As the two million tests are simulated any test that hits coverage that has not been hit before is added to a golden regression. The golden regression is run frequently across design changes and is used at the end of the project to sign-off that 100% coverage has been achieved. For the RSPU it is usual for there to be around 3000 tests in the golden regression. This gives an indication of the large amount of 'coverage-redundancy' in the generated tests cases – on average over 550 test cases have to be simulated for one to be added to the golden regression. It's worth noting the configuration space of the RSPU is large enough that it is unlikely that any two tests are generated with the same configuration. Also, the reported redundancy is despite the effort put into constraining the generation towards hitting coverage holes.

For the first version of the RSPU coverage closure was particularly problematic. As the verification progressed many failing test cases were identified as being due to configurations that had not been explicitly excluded in the specification but were not supported by the design nor required by the customer. This would lead to an update to the specification and additions to the constraints used for the constrained-random generation. The additional constraints introduced new relationships between configuration fields - it changed the connected field sets - and meant that test cases that had previously added to coverage, and so had been added to the golden regression, were now generated very differently and were no longer adding to coverage. The solution was to de-couple the generation of the test case and its simulation. The test case is written to a file (or a database) after generation and the file is read in by the simulator. This means that test cases do not need to be regenerated across changes in constraints since they are stored as files. For the most part stored test cases would remain legal across changes to the specification or require only a small modification and would continue to hit the same coverage. This is described in more detail in [3].

Whilst not common, the decoupling of the generation and simulation has a number of other benefits, including allowing for post-processing of the generated tests. One possible post-processing step is selecting which test cases to simulate, as shown in Fig. 1. Of course the test selection has no prior knowledge of the simulated results, such as the coverage achieved by the test case or whether the test case passes or fails. Machine learning can be used for test selection as suggested in [4]. The machine learning would be based on features of the generated test cases and, potentially, feedback from previous simulation results. This is described in more detail in the next section. Choosing which features of a generated test case to use for the machine learning may be difficult but the RSPU has an obvious feature set – the values of the configuration fields. These were considered to have a far larger influence on the coverage achieved by a test case than the data.

The goal of the test selector in Fig. 1 is to remove redundant, or uninteresting, test cases (however this is regarded) from the set of generated tests. Depending on the notion of redundancy used, it is unlikely that the test selector will be perfect and it is likely to remove some interesting tests. So typically it will be necessary to generate more tests than would be generated without the test selector. The extra time spent in generation and in test selection should reduce the overall time spent simulating but the time spent simulating should be more effective in achieving the verification targets such as closing coverage and soak testing the design (in order to find bugs). The net benefit of using a test selector is the time saved simulating to achieve these targets minus both the extra time spent in generation and the time spent in test selection, as well as any engineering effort saved in analysing coverage holes and writing constraints. The verification of the RSPU has one final property that lend itself well to this approach – the generation time is very quick (seconds per test case) and the simulation time is slow (hours per test case). This makes generating a very large pool of tests to feed to a test selector very cheap and means the impact of a good test selector is considerable.

In summary, the verification of the RSPU was identified as a good candidate for the use of machine learning for coverage closure because

- The generation and simulation of test cases is decoupled, allowing for the introduction of a test selector. A test selector is probably the simplest way to introduce machine learning into a verification flow. (We note that the decoupling of generation and simulation does not usually hinder verification. Reactive generation

can normally be replaced by reactive driving. For example, where a verification environment needs to handle errors or exceptions triggered in the design such handling can be generated along with the test and driven by the simulation environment when necessary.)

- The resource needed to close coverage is dominated by simulation times and engineering effort so not using generated tests has only a small cost. Using a test selector may still be beneficial if the generation times are comparable to the simulation times but the benefit will not be as large
- There is no clear correlation between a generated test and the white-box functional coverage that it hits meaning a simple deterministic test selector would not be suitable and the use of machine learning for a test selector for coverage closure is appropriate
- The generated test cases have an obvious feature set to use for machine learning. This means that time could be focused on trying different machine learning approaches to see which is most effective rather than on trying different feature sets (although it is possible that the results reported here could be improved on with more careful consideration of the feature set – for example by considering a subset of the configuration fields)
- Coverage closure - in particular white-box functional coverage closure - takes a significant amount of resource. The use of machine learning to successfully close coverage more efficiently has significant benefit
- There is a large amount of coverage-redundancy in the simulated test cases. Reducing this redundancy by 50% or more would leave a large number of simulated test cases with which to soak test the design, especially if the simulated test cases were of a higher quality, meaning all such savings would be realised as saved effort

III. Machine Learning Techniques for Novelty Detection

Since the advent of new neural networks based methods in the early 2010s [5], Machine Learning (ML) has received increased attention not just by research but also by industry. It has shown to be useful for a wide range of applications, including the approximation of complex simulations. ML is often split into the three sub-domains Supervised Learning (SL), Unsupervised Learning (UL), and Reinforcement Learning (RL).

Given the nature of the task of using ML as a test selector, we decided to disregard both SL and RL for this research. SL works well if we have a dataset at hand with a meaningful input output relationship and a reasonable amount of examples for each output class. However, the task of finding novel tests embodies that the predictions we really care about are those on previously unseen classes, where our SL model cannot do a lot better than random. In theory, one could imagine an RL system operating as a test selector that gets the added coverage as a reward signal and the batch of already simulated tests as the state. However, with the vast number of tests adding no coverage, this reward signal would be very sparse and the model hard to train. In addition, moving such an RL system to production is rather complex and it is unclear how well a trained agent generalizes to a new environment. Therefore, we will only focus on UL for this research.

In UL, the data from the input space is used to find meaningful patterns and relationships. One of the most common applications is clustering, where the machine learning models try to learn a transformation of the input space to minimize a distance function, characterizing what proximity means in a specific cluster. Anomaly detection can be seen as s special case of clustering, where all normal events are clustered into a dense region and all abnormal events are spread across the rest of the transformed input space. When mapping these characteristics to the use case of a test selector for coverage closure, the meaning of these algorithms changes. In test selection, we are assuming that selecting a diverse set of tests will be beneficial when trying to close coverage. Thereby, finding abnormalities becomes the target, since outliers can be seen as novel. Therefore, we are referring to the exploitation of anomaly detection algorithms to find novel instances as *novelty detection*.

For novelty detection, in theory, all anomaly detection algorithms are suitable - from linear models such as *Principal Component Analysis*, proximity-based models as *K Nearest Neighbors*, outliers ensembles as *Isolation Forests* to neural network-based methods such as *Autoencendors*. PyOD [6], a comprehensive and scalable Python toolkit for detecting outlying objects, provides a great overview of all methods and was used to conduct this research. When considering different methods, besides accuracy, the training and inference time also becomes a significant factor. Feed-forward neural networks provide a lot of model capacity and can easily be parallelized, especially when run on a GPU. Based on these criteria, the Autoencoder was chosen for further investigation.

An Autoencoder is a type of artificial neural network used to learn efficient data codings in an unsupervised manner, by trying to copy its input to its output, [7]. It is constituted by two main parts: an encoder that maps the input into the code, and a decoder that maps the code to a reconstruction of the original input. The middle layer holding the code is constructed as a bottleneck, i.e. holding significantly fewer neurons than the input layer. Thereby, the network is incentivized to efficiently encode common patterns to store them in a reduced space. The difference between the original input and the decoded output is referred to as the reconstruction loss. During training, it is the guiding loss

function. For the use in novelty detection, we do not care about the encoding itself, but we use the reconstruction loss during inference as a measure for divergence.

When using a novelty detection system during a continuous process, one has to decide on how often the system gets retrained. The simplest approach is one-shot learning, where the system gets trained once and is then used to score novel instances. Another setup is an iterative approach, where the novelty detection system is always trained on a specific set of instances to evaluate the novelty of some unseen instances. Since re-training the Autoencoder is not costly compared to the test simulation costs, we will use an iterative approach for this research, always re-training the Autoencoder on all the simulated tests once a new batch is added to the set. We then use this dense representation of the already seen instances to score the novelty of the remaining, non-simulated tests.

In addition to the selection of the learning algorithm, one has to decide on the strategy of preprocessing and feature engineering. Neural networks usually expect a normalized input, however, input data mostly comes as a mix of categorical and numerical variables. Pre-processing steps include deleting zero variance variables, converting pseudo numerical into categorical, normalizing numerical variables, and one-hot-encoding categoricals. Feature engineering describes the task of applying prior knowledge to transform the input prior to being fed into the machine learning system. For this research, we applied custom transformation on the binary codes of some configuration fields to reduce the input space.

Normally, it is best practice in ML to define one success metric that is used to be optimized for. In this setup, it is a little different, since the training success metrics is unrelated to the problem success metric. For the training, we are trying to minimize the reconstruction loss, but on a global scale, we are trying to minimize the amount of tests needed for a specific coverage. Thereby, no actually data set split was performed for this research

IV. Experimental Set-up and Results

This section details an experiment conducted on real life data from the production RSPU verification environment. Artificial coverage closure is carried out on a subset of the full coverage model using a large generated test pool. Machine learning techniques are used to order the test pool in terms of novelty with respect to the whole test pool. By simulating the most novel tests first we show favorable results in reducing time required to close coverage.

A. Data Generation

Emulating the coverage closure task requires the creation of a coverage model, a test pool and a mapping between the two. The test pool will be used to train our machine learning models while the mapped coverage for each test will be used in evaluating model performance. The generation of these will be explained in this section.

It is crucial that the choice of coverage model fairly represents the effort required in practical coverage closure. The validity of the resulting advantages from using machine learning techniques is tied to the quality of the coverage model. These techniques may not generalise well to other IP if the coverage model is easy to complete. The existing extensive RSPU coverage model is used for this reason. As mentioned in Section 2 the RSPU coverage model contains both code and functional coverage metrics which show the completeness of verification and guides remaining effort [8]. Achieving 100% code coverage, while clearly necessary, is not sufficient for sign off of a complex IP. Because of this the RSPU coverage model contains a large portion of functional coverage which aims to ensure all parts of the hardware design code have been exercised within the context that the code is written for. In other words, through achieving 100% functional coverage we expect to also complete code coverage. Any leftover code coverage signifies a gap in the functional coverage model and is subsequently filled. With modern automated design tools we get code coverage implementation for free with easily interpretable results, so most time is spent implementing, understanding and closing functional coverage. For these reasons the verification of the RSPU focuses on functional coverage and subsequently the coverage model for this experiment only includes functional coverage. Configuration coverage items are also emitted from the experimental coverage model since, as explained in Section 2, most of the test redundancy is derived from attempting to fill white box functional coverage items. With these omissions our experimental coverage model still totals 5,992 coverage items. For the remainder of this article: coverage refers to white box functional coverage.

For training data an artificial test pool is produced containing tests from 2 sources. Since closure has already been reached on the existing RSPU coverage model we pull in 3,076 tests already generated from this period that ensure 100% coverage is reached no matter what order the test pool as a whole is simulated. In practice ~2 million simulations and 6 months were required to obtain these 3,076 tests. Continuous ranking over the course of the 6 months discarded tests that do not uniquely contribute to coverage helping to preserve a relatively small ranked regression. In the experimental test pool we emulate this redundancy by adding 82,664 tests produced by constrained

random generation making a total of 85,470 tests. Constrained random generation is performed by the RSPU generation environment written in Specman 'e' language [9]. Specman comes with a sophisticated constraint solver; randomly seeding the solver ensures we explore a good portion of the legal constraint space with our test pool whilst also allowing for repetition of patterns within the test configurations for novelty detection to work on.

The generation environment produces output files containing information on how to configure the RSPU in the simulation environment. There are 290 configuration parameters in total for each test that are either numerical, categorical or binary in nature. The algorithms used in this experiment will aim to find novelty within these 290 test features.

After the creation of the test pool each test can be simulated and the white box functional coverage achieved can be extracted from the resulting coverage files. The mappings between test and coverage bins hit are stored in a SQLite relational table. Querying the relational table infrastructure circumvents expensive simulation providing a fast way of evaluating the test ordering and by extension the performance of the machine learning algorithm.

B. Data Analysis, Autoencoder Architecture and Test Selection Procedure

For the experimental setup, we combined the techniques outlined in chapter III: preprocessing, feature engineering and the training of a neural network-based model.

In the preprocessing step, we took the 290 configuration parameters, containing categoricals, Booleans and integers in binary format and transformed them into a format suitable for the neural network. Pseudo-categoricals, i.e. quantized integers up to a maximum level of 10 categories, were transformed into categoricals. In addition, one-hot encoding was performed on all categoricals, all zero-variance features were eliminated, and remaining integers where normalized between 0 and 1, leaving us with 285 input features.

For feature engineering, we incorporated expert knowledge to transform the fields. This included

- Zeroing data fields when a related enable field is zero
- Or-reducing a multibit field to a single bit i.e. regarding a multi-bit field as being only zero or non-zero
- Mapping fields with the same leading one to the same category, i.e. ignoring the trailing bits
- Ignoring fields completely

Lastly, we report on the choice of the autoencoder architecture and the training setup. In order to achieve an improvement over the random test selection, we have to make sure that the autoencoder learns a meaningful representation of the input space to successfully identify novelty. Therefore, we are facing a trade-off between giving the autoencoder enough capacity to learn patterns in the input features of the data used for training whilst ensuring that the network cannot memorize the input space or learn the identity function. To achieve this, it is recommended to use regularization techniques such as an L2 regularizer and dropout, as well as to choose a non-shallow architecture since this enables the autoencoder to learn complex functions without memorizing. A good source is the chapter on autoencoders in [10]. We chose a vanilla Autoencoder with hidden layers of size 64, 32, 32, and 64 neurons and Relu activations, dropout of 0.2, and Sigmod output activations. We used a mean-squared error reconstruction loss with an L2 regularizer of 0.1. We trained for 600 epochs with an Adam optimizer and a batch-size of 32

By comparing the one-shot method with the iterative approach, we could prove that step-wise retraining of the Autoencoder is helpful in identifying new tests. We chose a re-training cycle of 1000 tests for our experiment. The training times were around one hour on a single CPU.

In summary, the analysis was conducted according to the following procedure: 1) convert all raw tests configuration files into a database, 2) apply transforms for feature engineering 3) run preprocessing 4) store modified input features in database 5) train initial Autoencoder on full dataset 6) run inference and chose 1000 biggest "outliers", i.e. diverse tests, for initial simulation 7) re-train Autoencoder only with already chosen tests 8) run inference and choose 1000 most novel tests for simulation 9) repeat from 7) until level for manual intervention is reached.

We note that we are using a relatively small set of data to train the autoencoder. Typically, unsupervised learning requires fewer training samples than supervised learning. For this reason, we would expect this approach to scale to projects with larger feature sets and larger coverage models.

C. Results

An overview of the experimental results is given in Table 1. The results describe coverage curves: coverage percentage achieved after a certain amount of tests simulated. The table contains coverage curve data when intelligently selecting tests based on their assigned novelty as well as when they are selected randomly from the test pool. Random selection is analogous to the old approach. Figure 2 plots the random selection coverage curve and compares this with results achieved by novelty selection. There is a chance a particular test ordering can result in a

local minima when selecting randomly. To mitigate this we perform coverage closure with random selection ten times and the best-case results are given in the Random Selection column of Table 1.

Both sets of results exhibit a typical logistic curve however the novelty selection curve is consistently faster to all coverage percentage landmarks. A test saving of 60% is consistently achieved up to 99.5% coverage before tailing off as the remaining, hardest to hit coverage will no doubt start to dominate. Happily a large saving in tests simulated of 40% is still seen all the way up to 99.95%. The remaining 0.05% of coverage represents only 3 coverage items out of the 5,992 items originally in our coverage model. This is an easily manageable number for a verification engineer to manually intervene on and sign off on an IP.

TABLE I
NUMBER OF TESTS TO REACH CERTAIN COVERAGE LEVELS WITH BOTH RANDOM AND NOVELTY DETECTION DETERMINED TEST ORDERINGS.

Coverage	Number of Tests with Random Selection	Number of Tests with Novelty Selection	Saving in Number of Tests Simulated	% Saving in Number of Tests Simulated
87%	3000	1200	-1800	-60%
95%	13600	5400	-8200	-60%
97%	23700	9950	-13750	-58%
99%	52350	21300	-31050	-59%
99.5%	63500	25400	-38100	-60%
99.95%	85150	51800	-33350	-40%

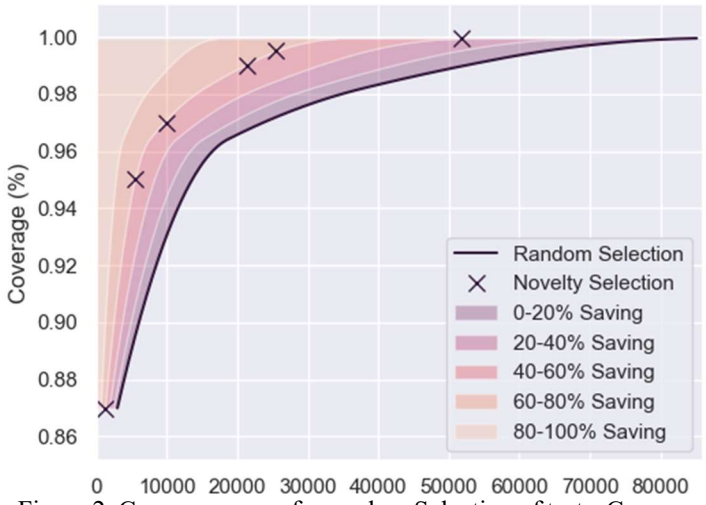

Figure 2: Coverage curve for random Selection of tests. Coverage landmarks for novelty based selection are shown as black crosses.

V. DISCUSSION AND FUTURE WORK

The reported experimental results show that novelty detection using an Autoencoder can be used to select tests that are more likely to add to coverage. Moreover, that the correlation between novelty and coverage can persist until the last few not-hit coverage items. The experimental results have led to the introduction of test selector based on test novelty in the verification of the RSPU. We are also exploring the use of such a test selector in the verification of other blocks for which the generation and simulation of tests is de-coupled, including the verification of microprocessors. In the rest of this section we explore other possible benefits for the use of novelty detection for test selection.

One significant challenge in coverage-driven verification is ensuring the quality of the coverage model. The derivation of the coverage model is subjective and the implementation may be error-prone. It is usual to involve many stakeholders in deriving and reviewing a coverage model but a different set of stakeholders would be likely to come up with a different coverage model. The extent of the coverage model may be tested against code coverage but with any reasonable coverage model it is usual that 100% code coverage is reached before 100% functional coverage. This does not mean that the functional coverage model could not be improved. Moreover, an incorrectly coded coverage item that is hit may go unnoticed. We look at how novelty detection could potentially address these limitations with coverage-driven verification.

The experimental results suggest that coverage on the RSPU could be closed – or very nearly closed – using novelty-selection whilst simulating 60% fewer tests than with constrained-random generation. If this translated directly to project execution this would mean the simulation of eight hundred thousand rather than two million tests to close coverage. The golden regression would likely still consist of around 3000 tests meaning that there is still considerable coverage-redundancy in the novelty-selected test set. However, this may not be true redundancy and may still explore design behaviour that has not been (correctly) coded as a coverage item. The question then is whether the novelty-selected tests are better at exploring design behaviour than the randomly generated tests. More formally we could regard two tests as equivalent if they explore the same design behaviour. For example, tests that just differ in the

number of wait states applied by the verification environment after the design has correctly reached a quiescent state are equivalent. The set of all possible tests could be partitioned into equivalence classes and we would ideally select one test to simulate from each equivalence class. Of course determining such an equivalence relation is not practical and even if it were we may end up with more tests than could be simulated in a reasonable amount of time. The question though would be whether the novelty-selected tests have representatives from more equivalence classes than the randomly-generated tests, even though there are fewer of them. This, of course, is a difficult question to answer, although it may be tractable on a small design – perhaps even one that could not be easily exhaustively verified by simulation.

Perhaps an easier question to answer is whether novelty-selected tests are better at finding bugs. An obvious way to check this with mutation testing. For example it could be seen whether novelty-selected tests find all mutations more quickly than randomly selected tests. It is not clear though whether the mutations are representative of actual bugs found in a project. It would be better to replay a project across all bug fixes and see which approach needs fewer tests to find the bugs.

Another way that novelty detection could potentially help address the limitations of coverage-driven verification is by manually analysing the most novel tests that do not hit coverage. This may be an indication of missing or incorrectly coded coverage items. Of course such a review may help uncover more missing or incorrectly coded coverage items than those hit by the test.

In summary, the paper helps to establish that machine-learnt novelty is a good (but not too good) predictor of coverage. It hypothesizes that the usefulness of novelty may be go beyond closing coverage and allow more of the design space to be explored more quickly and bugs to be found more quickly and suggests future work to explore this hypothesis. If such a hypothesis is true then, in future, novelty may complement coverage in being used to determine the value of a test set. With the decoupling of generation and simulation, novelty may in future be used to drive verification forward.

REFERENCES

[1] Saumil Gogri, Jiang Hu, Aakash Tyagi, Mike Quinn, Swati Ramachandran, Fazia Batool, Amrutha Jagadees, “Machine Learning-Guided Stimulus Generation for Functional Verification” Design Verification Conference, 2020

[2] “ISO 26262-5:2018 Road Vehicles - Functional Safety,” International Standardization Organization, 2018

[3] Tim Blackmore, Genevieve Bartlett and Jeganath Rajamohan, “Replay Mode for Coverage Closure and Efficient Debug,” Design Automation Conference, 2018

[4] Chen, WenRay, Sandip Abadir, Magdy Bhadra, Jayanta and Wang, Li-C. “Challenges and Trends in Modern SoC Design Verification” IEEE Design and Test. PP. 1-1. 10.1109/MDAT.2017.2735383, 2017.

[5] A.Krizhevsky, I.Sutskever and G.E.Hinton,“Imagenet classification with deep convolutional neural networks,” in Advances in Neural Information Processing Systems, 2012, pp. 1097–1105.

[6] Zhao, Y., Nasrullah, Z. and Li, Z., 2019. PyOD: A Python Toolbox for Scalable Outlier Detection. Journal of machine learning research (JMLR), 20(96), pp.1-7.

[7] Charu C Aggarwal. Outlier analysis. In Data mining, 75–79. Springer, 2015.

[8] S. Tasiran and K. Keutzer, "Coverage metrics for functional validation of hardware designs," in *IEEE Design & Test of Computers*, vol. 18, no. 4, pp. 36-45, July-Aug. 2001, doi: 10.1109/54.936247.

[9] "IEEE Standard for the Functional Verification Language e," in *IEEE Std 1647-2019 (Revision of IEEE Std 1647-2016)* , vol., no., pp.1-622, 9 Aug. 2019, doi: 10.1109/IEEESTD.2019.8793253.

[10] Ian Goodfellow, Yoshua Bengio and Aaron Courville, “Deep Learning”, MIT Press, 2016.