Dynamically Optimized Test Generation Using Machine Learning

Rajarshi Roy, Mukhdeep Singh Benipal, Saad Godil
NVIDIA
2788 San Tomas Expressway
Santa Clara, CA 95051

***Abstract*- Modern constrained random testbenches for hardware verification often require engineers to manually optimize test randomization constraints to target different use cases such as feature bringup, bug hunting and coverage closure. This paper presents a practical solution that automates the process of optimizing constrained random test generation by monitoring feedback from the design under test (DUT) and dynamically optimizing randomization constraints. We leverage Bayesian optimization (BayesOpt) with the gradient boosted regression trees (GBRT) machine learning models to optimize constraints in (1) batched offline flows and (2) live online flows. Experimental results in various testbenches demonstrate that the offline flow can boost coverage counts across coverpoints in a design by up to 6x and improve an interface's activity by 6.3x. The online flow is shown to improve the occupancy of a complex FIFO in the design by 7x.**

I. Introduction

Constrained random verification methodology [1] is a common technique for hardware design verification. With increasingly complex hardware architectures, constrained random testbenches and test generators often have constraints at various levels to control the randomness of test stimulus. Examples of such constraints include interface stall injection constraints, data packet distribution constraints etc. Typically, these constraints are manually optimized by design verification engineers while monitoring how tuning them affects test stimulus. However, the optimal stimuli that effectively cover the design under test (DUT) often have complex, probabilistic dependencies on multiple constraints. As a result, the manually tuned constraints are often suboptimal and adding new sets of constraints may overwhelm or restrict other stimulus, resulting in features that are not exercised.

Machine learning has been leveraged for increasing the effectiveness of constrained random testing in several test selection approaches [2-4]. These approaches train machine learning models to predict the effectiveness of generated tests and utilize these models to select an effective subset of generated tests to be run on the DUT. However, the effectiveness of the selected subset is limited by the quality of test generation. Our approach directly optimizes test randomization constraints based on feedback from the DUT, such as interface and internal signal monitors, to improve the quality of test generation. Other approaches for directly optimizing test stimulus have utilized specially designed Bayesian networks [5] or representations for genetic algorithms [6]. Our BayesOpt solution, in comparison, can be applied to various verification scenarios with minimal modification. In the context of hardware design, a similar approach was effective at optimizing the micro-architectural constraints of a processor [7].

II. Background

A. Bayesian Optimization

Bayesian optimization (BayesOpt) [8] is a machine-learning based optimization algorithm that is focused on finding the parameters x* that optimizes an objective function f(x). In the context of test generation, any objective that we wish to maximize, such as coverage hit counts, is a function f(x) that depends on test randomization constraints x.
BayesOpt specializes in the use-cases where:

- f(x) is black-box: there is no known analytical equation for f(x), so numerical optimization is not possible. This is the typical scenario in complex testbenches where the test objective cannot be analytically expressed as a function of test randomization constraints.
- f(x) may be noisy: f(x) may yield different values for the same value of x. This is a common scenario in constrained random testbenches where the test objective varies across different randomization seeds for the same randomization constraints.
- f(x) is expensive: evaluating f(x) has computational requirements that limit the approach of determining x* by evaluating every possible value of x. In testbenches, evaluating f(x) involves running tests and measuring test objectives. The test run-time limits the possibility of evaluating all possible randomization constraint combinations.

- f(x) may be multidimensional: x is a vector of values. Test generation objectives usually depend on the interaction between various randomization constraints.

BayesOpt is a Sequential Model-Based Optimization (SMBO) algorithm that trains and utilizes a machine-learning model of the objective in an iterative sequence. The machine-learning model, is a "surrogate" (Section II-B) of the expensive objective function f(x) that models the value and uncertainty of f(x), given an input x. The BayesOpt iteration involves:

1. Evaluating the objective f(x) for an initial random set of x values $[x_0:x_{n-1}]$
2. Training the surrogate model with data $[(x_0, f(x_0)):(x_{n-1}, f(x_{n-1}))]$ collected from the evaluation
3. Determining the next set of x to evaluate using an acquisition function (Section II-C) on the surrogate model
4. Evaluating the objective for the new set of x values: f(x) for x in $[x_n:x_{2n-1}]$
5. Starting the next iteration at Step 2 with the additional data collected from the evaluation

BayesOpt iteratively evaluates promising samples of x based on the current surrogate model. It aims to gather as much information as possible about the objective function and the location of its optimums. The acquisition function tries to balance exploration (x where the objective is most uncertain) and exploitation (x where the objective is likely optimal). Fig. 1 illustrates the BayesOpt process of minimizing a simple 1-dimensional objective with multiple minimas. The exploration property of the acquisition function lets BayesOpt find and converge at the global minima.

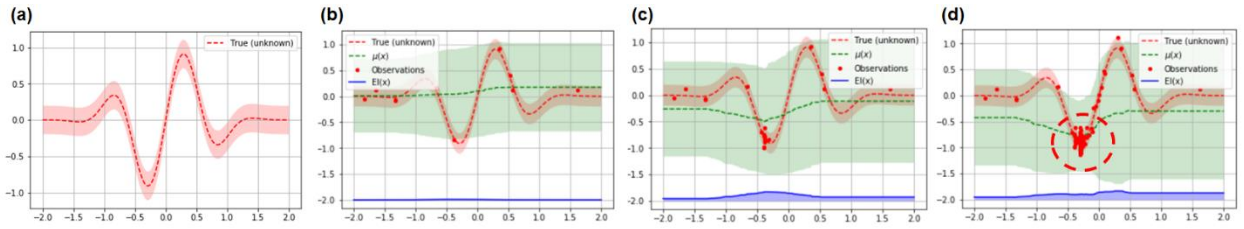

Figure 1. With the optimization goal of minimizing an unknown 1D function (a), the BayesOpt iteration simultaneously constructs a model by sampling points on the function and optimizing by directing new samples to the minimum over 10 (b), 20 (c) and 50 (d) iteration steps [9].

B. *Surrogate Models*

The surrogate model of the objective is utilized by the acquisition function to determine the next set of samples to evaluate in the BayesOpt iteration. Based on previously collected (x,f(x)) pair of data collected from the expensive objective f(x), the model is expected to predict the value of the objective at any x, and an uncertainty estimate of the prediction. We evaluate the following machine-learning surrogate models:

- Extra trees (ET) [9] use an ensemble of independently trained decision tree regressor models that are combined.
- Gradient-boosted regression trees (GBRT) [10] use an ensemble of decision trees that are trained jointly on iteratively re-weighted versions of the dataset.
- Gaussian processes (GP) [11] use kernel function distributions that maximize the log-likelihood of the dataset.

Deep neural networks (DNN) [12] models have recently gained popularity for their powerful modelling capabilities of high-dimensional data such as images and language. However, DNNs are less practical since their model architecture, input pre-processing, training parameters must be fine-tuned for different datasets.

C. *Acquisition Functions*

Acquisition functions are heuristics that evaluate the usefulness of samples for maximizing the objective function f(x). The expensive objective f(x) is evaluated at the selected samples x so the function must trade off exploitation and exploration. Exploitation means sampling where the surrogate model predicts a high value of the objective and exploration means sampling where the prediction uncertainty is high. We evaluate the following acquisition functions:

- Expected improvement (EI) [8] samples x with the highest expected improvement based on the surrogate model, over the best observed value of the objective $f(x_t^+)$
$$-EI(x) = -E[f(x) - f(x_t^+)] \qquad (1)$$
- Probability of improvement (PI) [8] samples x with the highest probability of improvement based on the surrogate model, over the best observed value of the objective $f(x_t^+)$
$$-PI(x) = -P(f(x)) \geq f(x_t^+)) + \kappa \qquad (2)$$
- Lower/Upper confidence bound (LCB/UCB) [8] samples x with the lowest/highest sum of value and scaled uncertainty (standard deviation) when minimizing/maximizing the objective
$$LCB(x) = \mu(x) + \kappa\sigma(x)) \qquad (3)$$

The acquisition functions provide constraints (e.g., κ) for controlling the exploration-exploitation trade-off.

III. Methodology

D. *Optimizer Interface*

We utilized the BayesOpt implementation in the Scikit-Optimize [13] Python library for this work. This library provides ET, GBRT and GP surrogate models and EI, PI and LCB acquisition functions. For example, if we want to optimize the utilization of an interface by controlling the following randomization constraints:

- PacketType ∈ {A, B, C} A categorical constraint of 3 categories
- MultiCycleDataEnable ∈ {True, False} A boolean constraint treated as categorical variable of 2 categories
- StallPushProbability ∈ $\{x \mid 0.0 \le x \le 100.0\}$ A numeric variable with a continuous range
- StallPopProbability ∈ $\{x \mid 0.0 \le x \le 100.0\}$ A numeric variable with a continuous range
- StallPushCyclesMax ∈ $\{x \mid 1 \le x \le 4\}$ A number variable with an integer range of 4 values
- StallPopCyclesMax ∈ $\{x \mid 1 \le x \le 4\}$ A numeric variable with an integer range of 4 values

We use the following interfaces of the BayOpt optimizer:

- `opt = Optimizer(dimensions, base_estimator,  acq_func)`
  Creates an optimizer with the base_estimator surrogate model and acq_func acquisition function. `dimensions` describe the possible values of the testbench randomization constraints:
  `[[A,B,C], [T,F], (0.0,100.0), (0.0,100.0), (1,4), (1,4)]`
- `xs = opt.ask(n_points)`
  Asks the optimizer for a set of n randomization constraint combinations as chosen by the acquisition function. the samples `xs` (with `n_points=2`) are initially random:
  `[[B, F, 30.5, 89.1, 2, 1], [C, T, 90.2, 20.3, 4, 3]]`
- `opt.tell(x, y)`
  Tells the optimizer feedback from the design under test (DUT) of the objective corresponding to the samples xs. For example, if the observed interface utilizations were 20% and 83% corresponding to the two constraint combinations:
  `opt.tell(x=[[B, F, 30.5, 89.1, 2, 1], [C, T, 90.2, 20.3, 4, 3]], y=[0.20, 0.83])`
  Every tell() call adds on more observation to the optimizer state without deleting previous observations. Every tell() call also fits the surrogate model to all the data observed.

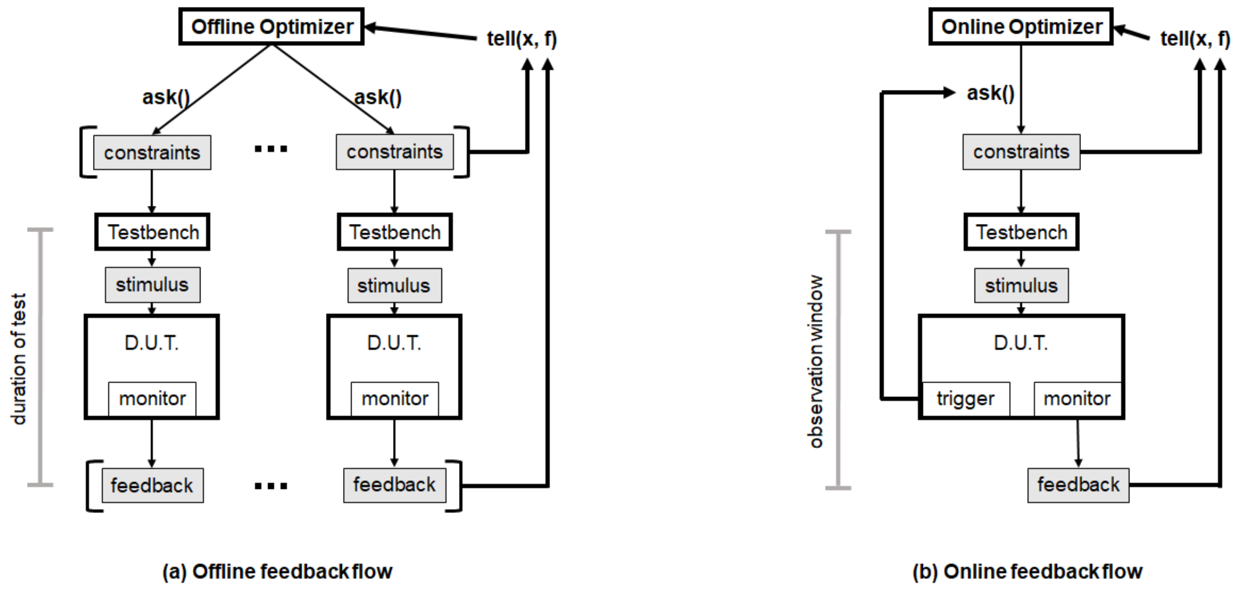

Figure 2. The offline flow (a) asks the optimizer for a batch of constraint and provides the batch of feedback after simulating a test per setting. Multiple iteration of the loop occurs over batches of tests. The online flow (b) asks the optimizer for a short-term constraint upon a trigger from the testbench, the effect of the setting is monitored over a short observation window and fed back to the optimizer. Multiple iterations of the loop occur live while a test is running.

E. *Offline Flow*

In this setting (Fig. 2a), a batch of M tests are generated with M corresponding constraints as produced by the `ask(n_points=M)` call to the optimizer. After the M tests are run and feedback on the objective is gathered, the M (constraint, objective) pairs are reported to the optimizer with the `tell()` call. The optimizer `ask()` call then provides constraints for the next iteration of tests.

The batched offline flow is useful for exploring a large set of constraints and optimizing for multiple test objectives simultaneously. This flow is especially effective when batches of multiple tests can run on every iteration of BayesOpt.

F. *Online Flow*

In this setting (Fig. 2b), during test simulation, a signal from the testbench triggers an `ask(n_points=1)` call to the optimizer to generate constraints. The constraints control the test stimulus over a certain duration and the feedback from the DUT is captured during this observation window. At the end of the window, the (constraint, objective) pair is reported to the optimizer with the `tell()` call. The testbench signal then triggers the `ask()` call that provides the constraints for the next observation window.

Online feedback is suitable for long simulation runs when the optimization agent can explore the constraints for a particular test objective as the test is running. The optimization agent can continuously optimize the test stimulus with live feedback, hence removing the need to run multiple tests to get optimal results.

IV. Experimental Results and Discussion

G. *Interface Activity Optimization using Offline Feedback*

In this usecase, the testbench controlled 8 instruction properties (categorical constraints of [2,2,2,2,2,2,6,12] categories). A downstream interface activity, that is to be maximized, was affected by the interaction of the selected instruction properties and other dynamic test stimulus. We used this environment to study the performance of various BayesOpt surrogate models and acquisition functions.

Across all the surrogate model (GBRT, ET, GP) and acquisition function (EI, LCB, PI) settings, 130 iterations of the offline BayesOpt flow was run with a batch of 1 test per iteration. We observed that the PI and EI acquisition functions perform well (Fig. 4a) and improve the interface activity ~6.3x from 15% to 95-100% over 120 iterations of the optimization. The LCB acquisition function was slightly worse and improves the interface activity to 83%. The GBRT and GP surrogate models performed the best and reached 95% interface activity whereas the ET model performed slightly worse and reached 90% interface activity (Fig. 4b). However, the GP model optimization time (max 42.29s) was significantly higher than GBRT (max 0.64s) and ET (max 1.21s) (Fig. 4c). Thus, the GBRT model was found to be the best performing surrogate model.

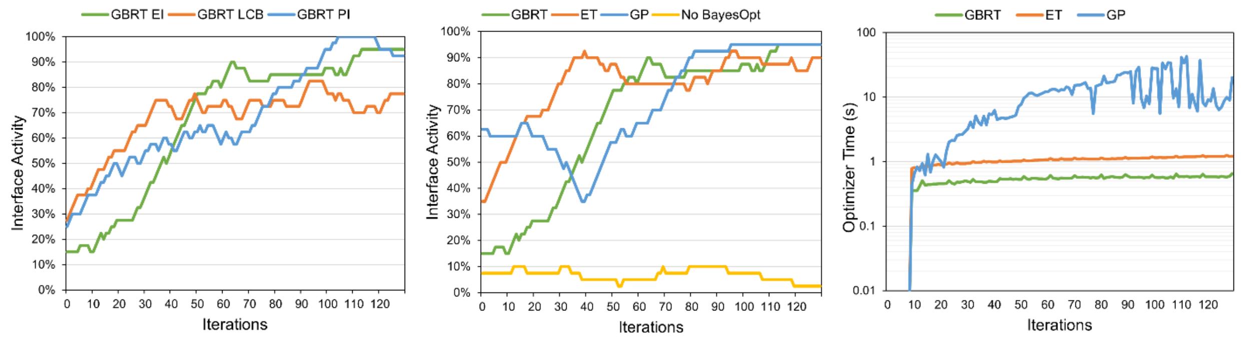

Figure 4. Interface activity improvement with GBRT and across acquisition functions (a), with EI and across surrogate models (b), and runtime of optimizers across surrogate models (c).

H. *Coverpoint Optimization using Offline Feedback*

In this usecase, the testbench controlled 27 interface stall constraints (integer constraints: 3 with range of 5, 3 with range of 10, 12 with range of 15, 9 with range of 26) to optimize for the coverage counts of 150 coverpoints. 150 individual optimizers were used corresponding to each coverpoint. 40 iteration were run with a batch of 600 tests per iteration. Even though tests from 4 constraint settings were generated and run per optimizer, the results from all 600 tests were fed back to all optimizers. A variety of other stimulus was interacting with stall randomization, so the probabilistic nature of Bayesian optimization was ideal for this usecase.

Pure random constraints led to 25x-200x hitcount for 10% of coverpoints over the human-set fixed constraints, but fared worse for 40% of coverpoints than human-set fixed constraints. This demonstrated the need for automated tuning. Best tuned constraints with offline feedback optimization were better than random constraints and fixed constraints for all the coverpoints. A 2x-6x further coverage count increase was observed with tuned constraints over random constraints for 20% of coverpoints (Fig. 5).

For each coverpoint, the crucial constraints converged on narrow ranges while noncrucial constraints remained randomized over wider ranges. This had an unintentional benefit of highlighting to design verification engineers how various parts of the design are affected by various stimulus.

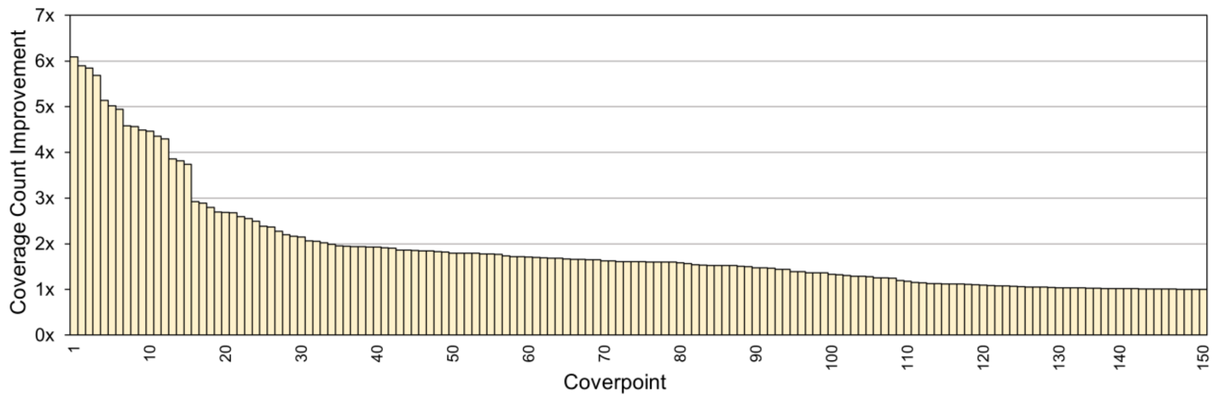

Figure 5. Improvement in coverage count with optimized constraints over random constraints across 150 coverpoints.

I. *FIFO Occupancy Improvements using Online Feedback*

In this usecase, the design under test (DUT) contained a combination of interconnected FIFO structures with complex state machine logic at their fill and drain points. The verification objective was to improve the occupancy of a target FIFO that was underutilized. The testbench driver had control of 5 stall constraints (categorical constraints of [8,8,8,8,8] categories) that indirectly affected the state machine logic and thus the FIFO occupancy.

Pure random stall combinations resulted in an average fifo occupancy of ~26%. This problem perfectly fit into the online optimization approach. With longer simulation runs, the optimization agent explored the stall constraints space and converged to optimal constraint combinations by gathering live feedback from the DUT. An observation window of 30 clock cycles was utilized to observe the FIFO occupancy changes from selected stall constraint combinations.

The Bayesian optimizer tuned test resulted in average fifo occupancy of around 61%(~2.3x improvement over random settings). Furthermore, the number of tests covering the FIFO full condition increased by 7x (Fig. 6).

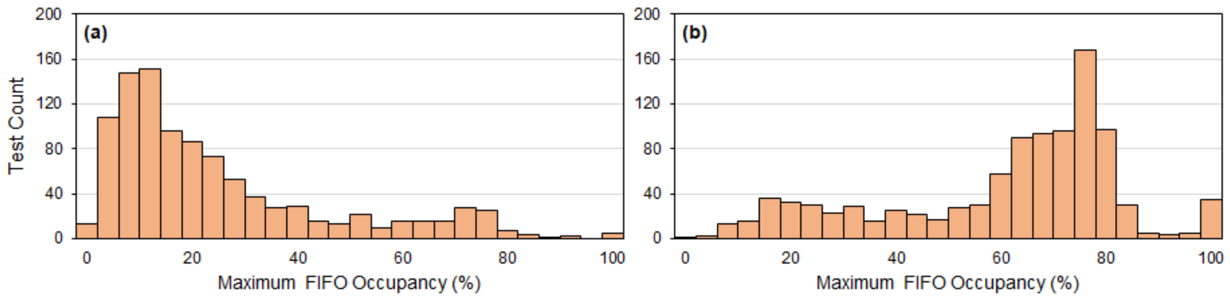

Figure 6. Distribution of maximum FIFO occupancy observed across 1000 tests with (a) random and (b) optimized stall injection constraints.

J. *Testbench Suitability of Dynamically Optimized Test Generation*

In Sections F-H we observed that dynamically optimized test generation is well suited to the constrained random testing of various designs. However, we would like to highlight certain properties of these testbenches that are well suited our approach.

- The feedback from the DUT should be fine-grained. A better feedback than whether a coverpoint is hit or not, is the hitcount of the coverpoint in a test, or what fraction of tests hit that coverpoint. For example, in Section IV-F, the feedback counts the fraction of tests that observed the interface activity.
- Constraints cannot be directly optimized towards an objective that has never been reached. The optimizer does receive any useful feedback until the objective has been hit at least once. Feedback that indirectly leads the optimizer towards the objective may be useful in such a scenario.
- The test generation constraints should affect the verification objective either directly or indirectly. For example, we observed that the coverpoints unaffected by the stall constraint optimizations in Section IV-G (Fig. 5), were unrelated in functionality to interface stalls altogether.

V. Conclusion And Future Work

In this paper, we applied Bayesian optimization (BayesOpt) to dynamically optimize test generation constraints with feedback from the design under test (DUT). Results from various testbenches and test generation settings demonstrate that BayesOpt is a practical and effective approach for optimizing test generation towards significant improvements in hardware verification objectives. We study various BayesOpt model and acquisition function choices and conclude that the gradient-boosted regression trees (GBRT) model is efficient, and most acquisition functions are reliable. We hope that the ease of applying BayesOpt allows other design verification teams to adopt this technique for feature bringup (online flow) or bug hunting and coverage closure (offline flow). We plan to open-source example designs and testbenches with BayesOpt integration for easy adoption.

Acknowledgment

We would like to thank our employer, NVIDIA Corporation for supporting this research. We would also like to thank the organizers of the Design and Verification Conference (DVCON) for their constructive feedback and the opportunity to share our work.

References

[1] J. Yuan, C. Pixley, A. Aziz, and K. Albin, "A framework for constrained functional verification," in *ICCAD-2003. International Conference on Computer Aided Design (IEEE Cat. No. 03CH37486)*. IEEE, 2003, pp. 142-145.

[2] O. Guzey, L.-C. Wang, J. Levitt, and H. Foster, "Functional test selection based on unsupervised support vector analysis," in *2008 45th ACM/IEEE Design Automation Conference*. IEEE, 2008, pp. 262-267.

[3] P. H. Chang, D. Drmanac, and L. C. Wang, "Online selection of effective functional test programs based on novelty detection," in *2010 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2010, pp. 762-769.

[4] F. Wang, H. Zhu, P. Popli, Y. Xiao, P. Bodgan, and S. Nazarian, "Accelerating coverage directed test generation for functional verification: A neural network-based framework," in *Proceedings of the 2018 on Great Lakes Symposium on VLSI*, 2018, pp. 207-212.

[5] S. Fine and A. Ziv, "Coverage directed test generation for functional verification using bayesian networks," in *Proceedings of the 40th Annual Design Automation Conference*, 2003, pp. 286-291.

[6] K. Salah, "New trends in RTL verification: Bug localization scan-chain-based methodology GA-based test generation," in *Proc. of DVCON*, 2015.

[7] O. DeMasi, J. Gonzalez, B. Recht, and J. Demmel, "Using bayesian optimization for hardware design," in *NIPS Workshop on Bayesian Optimization*, 2014, pp. 1-5.

[8] E. Brochu, V. M. Cora, and N. De Freitas, "A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning," *arXiv preprintarXiv:1012.2599*, 2010.

[9] P. Geurts, D. Ernst, and L. Wehenkel, "Extremely randomized trees," *Machine learning*, vol. 63, no. 1, pp. 3-42, 2006.

[10] J. H. Friedman, "Greedy function approximation: a gradient boosting machine," *Annals of statistics*, pp. 1189-1232, 2001.

[11] C. E. Rasmussen, "Gaussian processes in machine learning," in *Summer School on Machine Learning*. Springer, 2003, pp. 63-71.

[12] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436-444, 2015.

[13] M. Kumar, T. Head, G. Louppe, I. Shcherbatyi, H. Nahrstaedt, and contributors, "Scikit-optimize," 2017. [Online]. Available: http://scikit-optimize.github.io