A Survey of Machine Learning Applications in Functional Verification

Dan Yu, Harry Foster, and Tom Fitzpatrick
dan.yu@siemens.com, harry.foster@siemens.com, tom.fitzpatrick@siemens.com
46871 Bayside Pkwy,
Fremont, CA 94538

***Abstract* - Functional verification (FV) is computationally and data-intensive by nature, making it a natural target of machine learning (ML) applications. This paper provides a comprehensive and up-to-date analysis of FV problems addressable by ML. Among the various ML techniques and algorithms, several emerging ones have demonstrated outstanding potential in FV. Despite the promising research results, the paper identified critical challenges of applying ML in the industrial EDA environment. Finally, the authors indicate that the unavailability of high-quality verification data is impeding the research of ML in FV and calling for contributions to open verification datasets.**

I. Introduction

Ever-increasing design complexity and shortening design-to-market time has demanded faster and more accurate functional verification. Industry surveys indicate that design engineers spend about half of their time on Functional Verification (FV), and the situation has not improved over the years [1][2]. By its nature, FV is computation- and data-intensive, which makes it an ideal field to apply machine learning (ML) techniques in EDA.

ML development has witnessed exponential growth in the past decades [3][4][5]. An abundance of ML techniques has been devised to address challenges in data analytics of every data modality, including numbers, text, audio, image, video, graph, and combinations of multiple modalities. Depending on the data label's availability and the problem's nature, supervised, unsupervised, semi-supervised, or reinforcement learning can be applied to analyze the data. Many ML algorithms were also developed for these data analysis tasks, including regression, instance-based, regularization, decision tree, Bayesian, support vector machine artificial neural network (ANN), deep learning, dimensionality reduction, and many more emerging algorithms.

Many ML algorithms have been experimented with in different areas of FV with various degrees of success. This paper contributes to the overall study of applying ML to FV by identifying problems addressable by ML, listing new techniques and algorithms that are applicable algorithms, outlining the industrial perspective, and discussing the challenge of data unavailability.

II. Topics of ML in Functional Verification

This paper outlines the important topics ML applies to in FV. Several existing publications have offered perspectives on the general problem landscape. Huang, Guyue, et al. give a general overview of ML's application in EDA, with limited coverage on verification about test set redundancy reduction and test complexity reduction [6]. Additionally, Ismail and Abd El Ghany provide insight into the automation of troubleshooting, automation of formal engine selection, root cause analysis, and environment modeling [7]. In addition to the selected topics mentioned above, Cristescu also presents the topic of automated code generation [8].

This paper gathers a comprehensive list of ML topics applicable to FV when including all functional verification topics in the general perspective of programming code verification. Table 1 summarizes the topics and subtopics and includes a column for the related section, which follows the table. Italicized text indicates subtopics that were unexplored in other general survey publications.

TABLE I
TOPICS OF ML APPLICATIONS IN FV

Topic	Subtopic and Explanations	Section
Requirement engineering	Code generation from requirement specs in constrained NL text [8]	1.1
	Code comprehension, summarization, and translation to natural language	1.2
	Specification mining from runtime traces [7]	1.3
Static code analysis	*Code smell detection and code quality assessment*	2.1
	Automated refactoring and automatic correction	2.2
	Code completion and semantic code search	2.3
Verification Acceleration	Automated selection of formal prover [7]	3.1
	Automated simulation parallelization	3.2
	Complex module approximation with ML models	3.2
Coverage Closure	ML-guided random test generation [6][7][8]	4
	Test complexity reduction	4
Bug detection and localization	*Assisted root cause analysis*	5
	Automated troubleshooting [7]	5

1. Requirement Engineering

Requirement engineering in FV is the process of defining, documenting, and maintaining verification requirements, which is critical to ensure the excellent quality of the underlying IC design.

1.1. Requirement definition

Requirement definition involves translating ambiguous natural language (NL) verification objectives into verification specs with formality and precision. The quality of the translation directly dictates the correctness of the verification. Traditionally this process is laborious and consumes substantial design cycles with several iterations of manual proofing to ensure quality.

Two groups of classic approaches have been proposed to automate translation. One group of approaches is to introduce constrained natural language (CNL) to formalize the specification drafting, followed by a template-based translation engine. This approach requires significant upfront investment in developing a powerful CNL syntax and a comprehensive compiler/template system to ensure it is powerful enough to address most of the requirements encountered in FV. Furthermore, it burdens developers with learning an additional language, which prevents the idea from becoming widely accepted [9].

The other group leverages the classic natural language process (NLP) to parse NL specifications and extract relevant key elements to formulate formal specifications, represented in [10].

The advance of ML translation in the NL domain has made fully-automated machine translation commercially feasible and sometimes exceeds the performance of average human translators. It has kindled the hope of leveraging large-scale trained NL models with up to billions of parameters to directly translate NL specifications to verification specs in SystemVerilog Assertions (SVA), Property Specification Language (PSL), or other languages. Several attempts to do a successful end-to-end translation have been observed, but none have been made production-ready. The major hurdle to this approach is the scarcity of available training datasets that pair NL specifications with their formal translation. The most extensive datasets are merely about 100 sentence pairs [11]. The number pales in comparison to their NL peers, which routinely come in millions or even billions of sentence pairs.

1.2. Code summarization and translation to NL

Converse to requirement definition, summarization looks at the code and translates it into a human-understandable NL summary. It assists developers in reading less ideally maintained code or understanding complex logic. An ideally implemented code summarization can insert inline documentation into code blocks or generate separate documentation. With its help, the maintainability and documentation of the code can be significantly improved.

The application of ML on code summarization has been experimented with in more popular computer languages, e.g., Python and JavaScript [12][13]. Several groups of approaches have experimented with various degrees of success.

Information Retrieval (IR) based approaches focus on applying NLP to source code and looking for similar code with existing summarization in place. This group of approaches heavily relies on the quality of the existing code with summaries. Its use is only possible within a close organization where many existing code repositories are readily available. Heuristics-based approaches instead try to define specific rules based on heuristics identified in a module's definition, e.g., a module with many submodules of basic read / write command lines might be considered a memory module. Therefore, a summary can be constructed from a predefined pattern for the memory module.

At the time of this writing, code summarization in IC design verification has not yet been reported in any literature. It is reasonable to be optimistic that the success of other languages can be realized in IC design and verification, which has yet to be confirmed by the research community. In particular, recent progress with cross-language models might help transfer learned knowledge from other programming languages to IC design. However, in addition to the challenges general to ML on code summarization, the intrinsic temporal parallelism in IC design and verification code can present challenges uncommon in other programming languages.

1.3. Specification mining

Specification mining, as described by Dallmeier, Valentin, et al. and Wenchao, Forin, and Seshia, has been a long-term software engineering topic [14][15]. As an alternative to manually drafting specifications, it extracts specifications indirectly from the design under test (DUT) execution. ML can be applied to mine recurring patterns from simulation traces. It can help automate either simulation-based coverage closure or formal verification. It is assumed that commonly recurring patterns might be the expected behavior of the DUT. Alternatively, an event pattern rarely occurs in the traces can be regarded as an anomaly; therefore, it can be used for diagnostic and debugging purposes [16].

ML has been applied in pattern discovery and anomaly detection across many domains where temporal data of a complex system are available. Azeem et al. propose a general software engineering approach where ML is used to discover formal specifications from observing protocol traces and finding the possible problematical implementation of the protocol [17]. Successful experiments cited in the paper have inspired interesting follow-up research projects in specification mining with ML.

2. Static code analysis

As the cost of fixing a bug grows exponentially along the stages of IC development, static code analysis offers an attractive option to improve code quality and maintainability at an earlier stage of design development.

2.1. Code smell detection and quality assessment

Code smell refers to suboptimal design patterns in source code, which might be syntactically and semantically correct, but violate best practices and can lead to poor code maintainability. A particular example is code duplication, where the same function is implemented multiple times across a project or the entire code base. Some copies can have a particular bug fixed in a relatively short period, while the same bug goes unnoticed in other copies.

Classic code smell detection relies on defined heuristic rules to identify patterns in the source code. Instead of manually developing these rules and metrics in the static code analysis tools, an ML-based approach can be trained on a large amount of available source code to identify code smells. Research detailed by Fontana et al. and Aniche et al. has proven that smell detection with ML can lead to universal code smell detection and significantly fewer pattern implementation efforts [18][19]. The resulting smell score can then be used for code quality assessment and help developers improve product quality consistently. Furthermore, ML-based code refactoring might provide helpful hints on improving code smell or even further some candidate changes.

The application of ML in FV is not yet visible, and the unavailability of large training datasets has prevented existing research from fully exploiting this solution's potential.

2.2. Coding assistance

Developers working on IC design can be most productive when the proper tools are provided. Simple code completion is a standard feature in the modern Integrated Development Environment (IDE). However, more advanced techniques involving deep learning were proposed [21] and are maturing quickly [22]. It is now possible to train ANNs with billions of parameters from many large-scale open-source code repositories to give reasonable recommendations of code snippets from developers' implementation intent or the context.

ML might also help IC developers stay productive with semantic code search, which allows retrieving relevant code by NL queries [23]. As code is usually full of various abbreviations and technical jargon, semantic searches can be more effective in finding relevant code snippets without correctly spelling the key variable, function, or module names. While similar to semantic search in many existing search engines, semantic code search is able to help find abbreviated and highly technical code with vague concepts. The Mean Reciprocal Rank of the best model evaluated in [23] can already achieve usable scores of 70%.

Although theoretically, the same ML techniques applied to other programming languages can be applied to IC design, no research has been published yet on coding assistance.

3. Verification Acceleration

The surveys in [1][2] indicate that functional verification is still the most time-consuming step in IC design, and functional and logic errors are still the most important cause of a respin. Any improvement in the speed of FV will significantly impact the quality and productivity of IC design. ML has been used in both formal and simulation-based verification for their acceleration.

3.1. Formal verification

Formal verification uses formal mathematic algorithms to prove the correctness of a design. Modern formal verification orchestration employs formal algorithms to target designs of different sizes, types, and complexities. Experience and heuristics can help developers select the most appropriate algorithms from the library for a specific problem.

As a statistical method, ML cannot directly address formal verification problems. However, it has been proven to be very helpful in formal orchestration. With its prediction of computing resources and probability of solving a problem, it is possible to schedule formal solvers to best use these resources to shorten the verification time [25] by first scheduling the most promising solvers with lower compute resource consumption. The Ada-boost decision tree-based classifier can improve the ratio of solved instances from the baseline orchestration from 95% to 97%, with an average speedup of 1.85. The other experiment in [25] is able to predict resource requirements of formal verification with a 32% average error. It iteratively applies feature engineering to carefully selects features from DUT, properties, and formal constraints, which are then used to train a multiple linear regression model for resource requirements prediction.

3.2. Simulation-based verification

Contrary to formal verification, simulation-based verification usually cannot ensure complete correctness in the design. Instead, the design is put under a test bench with certain random or fixed-pattern input stimuli applied, while the outputs are compared to the reference outputs to verify if the design's behavior is expected. While simulation is the bread and butter of FV, simulation-based verification can also suffer from long verification times. It is not uncommon for the verification of a complex design to take weeks to complete.

A promising idea being discussed and experimented with is to use of ML to model and predict the behavior of a complex system. The Universal Approximation Theorem proves that a multi-layer perceptron (MLP), a feed-forward ANN with at least one hidden layer, can approximate any continuous function with arbitrary accuracy [28]. Whereas normalized Recurrent Neural Networks (RNNs), a specialized form of ANN, are proven to approximate any dynamic system with memory [29]. Advanced ML accelerator hardware has made it possible that ANNs can model the behaviors of some IC design modules to accelerate their simulations. Significant acceleration may be achieved depending on the capability of AI accelerators and the complexity of the ML models.

4. Test generation and coverage closure

Besides manually defined test patterns, standard techniques employed in simulation-based verification include random test generation and graph-based intelligent testbench automation. Due to the "long tail" nature of coverage closure, even a tiny efficiency improvement can easily result in significantly reduced simulation time. Much research on the application of ML to FV has focused on this area.

Extensive ML studies have demonstrated that they can do better than random test generation. Most research employs a "black box model," assuming that a DUT is a black box whose inputs can be controlled and outputs can be monitored. Optionally, some testing points can be observed. The research does not seek to understand the behavior of the DUT. Instead, the focus is spent on reducing unnecessary tests. They employ various ML techniques to learn from historical

input/output/observation data to tune the random test generators or eliminate tests that are unlikely to be useful. In a recent development in [31], a Reinforcement Learning (RL) based model is used to learn from a DUT's output and predict the most probable tests for a cache controller. When the reward given to the ML model is the FIFO depths, the experiments are able to learn from historical results and hit the full target FIFO depths in several iterations, while the random test generation-based approach is still struggling to hit more than 1. However, the paper does not further experiment with how it performs for a more complex design where the reward targets are not easily definable. [28] introduces an ML architecture with much finer granularity, where an ML model needs to be trained for every cover point. A ternary classifier is also employed to help decide if a test shall be simulated, discarded, or used to retrain a model further. Support Vector Machine (SVM), Random Forest, and deep neural network are all experimented on a CPU design. It is able to close 100% coverage with 3x to 5x fewer tests. Further experiments on FSM and non-FSM designs have demonstrated 69% and 72% reductions compared to directed sequence generation. However, most of these results still suffer from the limitation of the statistical nature of ML. A more comprehensive review of ML-based Coverage-Directed test generation (CDG) in [32] gives an overview of several ML models and their experiment results. Bayesian Network [41] genetic algorithms and genetic programming approaches, Markov model, data mining, and inductive logic programming are all experiments with various degrees of success.

In all the approaches discussed, an ML model can make a prediction based on the learning from historical data it has gathered but has the minimal capability to predict the future, i.e., which test might be a more promising option for hitting an uncovered test target. As this kind of information is not yet available, the best they can do is pick the tests that are the most irrelevant to the historical tests. A promising experiment conducted in [33] explored a different approach, where the DUT is regarded as a white box, and the code is analyzed and converted to a Control / Data Flow Graph (CDFG). A gradient-based search on a trained Graph Neural Network (GNN) [40] is used to generate tests for a predefined test target. The experiments on IBEX v1, v2, and TPU achieved 74%, 73%, and 90% accuracies at coverage prediction when trained with 50% cover points. Several additional experiments also confirm that the gradient search method employed is insensitive to the GNN architecture.

It is noted that due to the unavailability of training data, most of these ML approaches only learn from each design without exploiting any prior knowledge from other similar designs.

5. Bug analysis

Bug analysis aims to identify potential bugs, localize the code blocks containing them, and give fix suggestions. The surveys [1][2] found that verification of an IC spends roughly the same amount of time as it does in design and that functional bugs contribute to about 50% of respins for an ASIC design. Therefore, it is critically important that these bugs can be identified and fixed in the early functional verification stage. ML has been employed to help developers detect bugs in designs and find bugs faster.

Three progressive problems need to be solved to speed up bug hunting in FV, namely, clustering of bugs by their root causes, classification of root causes, and suggestion of fixes. Most research focuses on the first 2, with no research results on the third one available yet.

In research conducted in [34], semi-structured simulation log files are used for bug analysis. It extracts 616 different features from metadata and message lines from log files of undisclosed designs. The experiment on clustering achieved Adjusted Mutual Information (AMI) of 0.543 with K-means and agglomerative clustering and 0.593 with DBSCAN even after feature dimensionality reduction, far from ideal clustering when AMI achieves 1.0. Various classification algorithms were also tested to determine their accuracy in solving problem 2. All the algorithms, including random forest, Support Vector Classification (SVC), decision tree, logistic regression, K-neighbors, and naïve Bayes, are compared on their power to predict root causes. The best score was achieved by random forest with 90.7% prediction accuracy and 0.913 F1 scores. Raffel et al. propose to use a labeled dataset from code commit to train a gradient boosting [40] model, where more than 100 features about authors, revisions, codes, and projects were tested until 36 were selected for the algorithm [35]. The experiments show that it is possible to predict which commits are most likely to contain buggy code and potentially reduce manual bug-hunting time significantly.

However, due to the relative simplicity of the ML techniques adopted, they are not able to train ML models that can consider rich semantics in code or learn from historical bug fixes. Therefore, they cannot explain why and how the bugs occur nor suggest revising the code to eliminate bugs automatically or semi-automatically.

III. Emerging ML Techniques and Models Applicable to FV

Significant breakthroughs in ML techniques, models, and algorithms have been witnessed in recent years. Our paper found that very few of these emerging techniques are adopted by FV research, and we optimistically believe great success will be possible once they are used to tackle challenging problems in FV.

Transformer-based large-scale NL models with billions of parameters trained on the enormous amounts of text corpus achieved near-human or exceeding-human level performance in various NL tasks, e.g., question answering, machine translation, text classification, abstractive summarization, and others. The application of these research results in code analysis has also demonstrated these models' great potential and versatility [12][24]. It has been demonstrated that these models can truly ingest a large corpus of training data, structure the learned knowledge and provide easy accessibility. This capability is instrumental in static code analysis, requirement engineering, and coding assistance for several popular programming languages. Given enough training data, it is reasonable to believe that these techniques can train ML models for various FV tasks.

Until recently, it was hard to apply ML to graph data due to the complexity of their structure. Graph Neural Network (GNN) advances have promised a new opportunity for FV. The research in [33] converts a design into a code/data flow graph, which is then further used to train a GNN to help predict the coverage closure of a test. This kind of white box approach promises previously unavailable insight into the control and data flow in a design, which can generate directed tests to fill potential coverage holes. Graphs can represent rich relational, structural, and semantic information encountered in verification. The rich information from training an ML model on graphs can afford many new possible FV tasks, e.g., bug hunting and coverage closure.

IV. Urgency of Large-Scale Verification Open Datasets

Despite many encouraging research results on applying ML in FV, many challenges must be addressed to realize mature and large-scale success. Among them, the biggest challenge is the unavailability of training datasets. Due to the lack of large datasets, much research has to settle for relatively primitive ML techniques that demand only small training datasets with hundreds of samples. The situation has prevented advanced ML techniques and algorithms from being applied. Historically, IC designs are considered highly valued property of their respective owners. The designs and the associated verification data, including the verification plan, simulation settings, assertions, regression results, simulation logs, coverage data, trace files, revisions, and their commits, are also carefully guarded by their owners.

If that were not bad enough, an internal survey has indicated that a typical ML project will spend up to 70% of the team's time on data preparation. It is also confirmed by an independent study [34] that general data preparation, including data cleansing, reporting, and visualization, takes 75% of data professionals' time. This is the time they spend before any exploration of an original idea.

The unavailability of quality training data and expensive data preparation is a substantial hurdle for any team to research ML for FV. It impedes research advancement and prevents research communities from learning from each other's success. After spending so much effort and time to curate their own data for the research, very few researchers are ready to share their data, code, or ML models publicly, making their research results hard to repeat or verify by other researchers. Consequently, it seriously reduces the value of the ML research results.

Standardization of data format and active contributions to this kind of verification datasets by the community will significantly accelerate the development of AI/ML for FV, and Siemens is actively working with different consortiums and partners to make this happens.

If there is anything to be learned from ML in Computer Vision (CV) community, it is that the abundance of training data helps boost the research significantly. Published in 2009, ImageNet now contains 14,197,122 images in 21,841 categories as of August 2022. Its availability has helped researchers invent unthinkably deep ML models and compete on a comparable basis, which directly contributed to the practical application of deep learning. Challenges hosted by ImageNet help build a healthy community where researchers can afford to focus on learning, inventing, and training their models instead of spending efforts to curate a small private dataset. According to [42], ImageNet has changed the thinking of the ML community from paying attention to models to paying attention to data, which consequently helps redefine how models are developed.

The opportunity of FV also lies in such kind of open datasets. Our search on GitHub, RISC-V and OpenCores conducted in May 2022 also indicates that limited design data and even fewer verification data are available in the public domain. The table below summarizes our search results. We estimate that there are about 10 million lines of code and comments altogether. This scale is hardly comparable to millions of projects and billions of lines of source code in other popular programming languages. Even more, the verification-related data are very sparse. Even if it is possible to generate verification data from the sets of source code, tremendous effort must be invested by individual research teams to decode and use the data effectively due to the lack of published and agreed standard upon common data schemas.

Making these data usable for ML tasks requires substantial effort to eliminate those with restrictive licenses and those with simple forks, followed by further deep cleansing to ensure the quality and usability of the final dataset. The paper did not identify any announced activities to curate open-source verification datasets and make them available to the public.

TABLE 2.
NUMBER OF OPEN SOURCE PROJECTS HOSTED ON GITHUB, RISC-V, AND OPENCORES

Number of Projects	VHDL	Verilog	SystemVerilog	Other
GitHub	19,861	16,352	3,212	-
RISC-V	59	11	25	16
OpenCores	-	470	467	30

V. INDUSTRIAL PERSPECTIVE

In addition to the scarcity of data, several other challenges must be addressed to ensure the successful application of ML in FV.

Model generalization is one of the most prominent challenges in many ML research results. An ML model's generalization capability measures how easy it is to apply the model trained from training data to problems that new problems have never seen before. A model trained on data from specific types of designs, coding styles, certain projects, or some niche types of data might not perform well for other types of designs, other coding styles, other projects, or other types of data. The hard-to-generalize research results will only have limited value in an industrial production environment, as the cost of training and managing these models can easily negate the value they bring to the solutions.

Model scalability is another challenge in applying these research results in a production environment. A model might perform well for a relatively simple design. However, there is no guarantee it would work equally well when applied to a large-scale design with billions of RTL gates with code from multiple development teams spanning multiple continents. To benefit from these models, additional investment in further model compression, additional computing resources, e.g., ML accelerators, or establishing MLops [38] might be inevitable. The decision to invest in this effort must be balanced against the expected return from applying ML, which is not yet evident in many cases.

The third challenge is related to data governance in ML applications. In real-world applications, the owner of the datasets used to train the ML model, the FV tool developer, and the tool user might belong to different organizations. In combination with the distribution of computing power and availability of MLOps expertise, they may jointly dictate the selection of ML models. For example, a user who cannot divulge her data out of her organization and has no ML infrastructure may not be able to use an ML model that requires much computing for the training on her data to achieve optimal performance. Alternatively, an FV tool developer with mature MLops workflow, cloud infrastructure, and knowledge may be able to offer attractive FV tools hosted in the cloud, with easily accessible RESTful APIs a user may integrate into their FV workflow.

VI. SUMMARY

Abundant data are available along a design's life cycle. ML is one of the best tools to extract values from the data for functional verification.

This paper has presented a comprehensive survey of how ML may help address almost every aspect of FV. These typical applications and their state-of-the-art achievements are then summarized and discussed. Despite diverse ML techniques applied, most research still employs primitive ML techniques and is restricted by the scale of the training data. It is not dissimilar to the early days of ML applications in many advanced domains and prompts that ML applications in FV are still in their early days. Many advanced techniques and models are yet to be applied to exploit ML's potential fully. Much semantic, relational and structural information has not yet been fully utilized in today's ML applications.

Working experience and review of many research results have indicated that the application of ML in FV cannot advance at a much faster pace without substantial, quality datasets being made available. The paper calls for efforts to build more extensive datasets to train more complex big ML models. Such datasets will afford the FV community to make research results more comparable, repeatable, and credible.

The paper also presented other critical challenges to be overcome to release the potential of ML in FV, and

Generalizability, scalability, and data governance are the three main challenges every industrial researcher has to consider applying ML in real-world production. Recognizing their implication will also help identify the suitable ML models applicable to many possible diverse use cases.

The paper's authors are optimistic that FV is intrinsically a data analytics problem and ML is a powerful tool to extract value from the abundance of FV data. We look forward to seeing more contributions from the community in developing quality datasets, applying advanced ML techniques, and successfully adopting ML in FV tools.

REFERENCES

[1] Harry Foster, "2020 Functional Verification Study," Wilson Research Group and Siemens EDA https://blogs.sw.siemens.com/verificationhorizons/2020/10/27/prologue-the-2020-wilson-research-group-functional-verification-study/, retrieved on Nov. 18, 2022.

[2] Harry Foster, "2022 Functional Verification Study," Wilson Research Group and Siemens EDA, https://blogs.sw.siemens.com/verificationhorizons/2022/10/10/prologue-the-2022-wilson-research-group-functional-verification-study/, retrieved on Nov. 18, 2022

[3] James, Gareth, et al. *An introduction to statistical learning*. Vol. 112. New York: Springer, 2013.

[4] Qiu, Junfei, et al. "A survey of machine learning for big data processing." *EURASIP Journal on Advances in Signal Processing* 2016.1 (2016): 1-16.

[5] Das, Kajaree, and Rabi Narayan Behera. "A survey on machine learning: concept, algorithms and applications." *International Journal of Innovative Research in Computer and Communication Engineering* 5.2 (2017): 1301-1309.

[6] Huang, Guyue, et al. "Machine learning for electronic design automation: A survey." ACM Transactions on Design Automation of Electronic Systems (TODAES) 26.5 (2021): 1-46.

[7] Ismail, Khaled A., and Mohamed A. Abd El Ghany. "Survey on Machine Learning Algorithms Enhancing the Functional Verification Process." *Electronics* 10.21 (2021): 2688.

[8] Cristescu, Mihai-Corneliu. "Machine Learning Techniques for Improving the Performance Metrics of Functional Verification." *Sci. Technol* 24 (2021): 99-116.

[9] Zhao, Junchen, and Ian G. Harris. "Automatic assertion generation from natural language specifications using subtree analysis." *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019.

[10] S. Ghosh, D. Elenius, W. Li, P. Lincoln, N. Shankar, and W. Steiner, "ARSENAL: automatic requirements specification extraction from natural language," in NASA Formal Methods Symposium. Springer, 2016, pp. 41–46.

[11] Kansas State University CIS Department, Laboratory for Specification, Analysis, and Transformation of Software (SAnToS Laboratory), Property Pattern Mappings for LTL. Accessed: April 2019. URL: http://patterns.projects.cs.ksu.edu/.

[12] Allamanis, Miltiadis, et al. "A survey of machine learning for big code and naturalness." *ACM Computing Surveys (CSUR)* 51.4 (2018): 1-37.

[13] Zhang, Chunyan, et al. "A Survey of Automatic Source Code Summarization." Symmetry 14.3 (2022): 471.

[14] Lemieux, Caroline, Dennis Park, and Ivan Beschastnikh. "General LTL specification mining (T)." *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015.

[15] Dallmeier, Valentin, et al. "Generating test cases for specification mining." Proceedings of the 19th international symposium on Software testing and analysis. 2010.

[16] Li, Wenchao, Alessandro Forin, and Sanjit A. Seshia. "Scalable specification mining for verification and diagnosis." *Design Automation Conference*. IEEE, 2010.

[17] Ammons, Glenn, Rastislav Bodik, and James R. Larus. "Mining specifications." *ACM Sigplan Notices* 37.1 (2002): 4-16.

[18] Azeem, Muhammad Ilyas, et al. "Machine learning techniques for code smell detection: A systematic literature review and meta-analysis." *Information and Software Technology* 108 (2019): 115-138.

[19] Fontana, Francesca Arcelli, et al. "Code smell detection: Towards a machine learning-based approach." *2013 IEEE international conference on software maintenance*. IEEE, 2013.

[20] Aniche, Mauricio, et al. "The effectiveness of supervised machine learning algorithms in predicting software refactoring." *IEEE Transactions on Software Engineering* (2020).

[21] Bruch, Marcel, Martin Monperrus, and Mira Mezini. "Learning from examples to improve code completion systems." *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering*. 2009.
[22] Svyatkovskiy, Alexey, et al. "Intellicode compose: Code generation using transformer." *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2020.
[23] Husain, Hamel, et al. "Codesearchnet challenge: Evaluating the state of semantic code search." arXiv preprint arXiv:1909.09436 (2019).
[24] Feng, Zhangyin, et al. "Codebert: A pre-trained model for programming and natural languages." *arXiv preprint arXiv:2002.08155* (2020).
[25] Elmandouh, Eman M., and Amr G. Wassal. "Guiding formal verification orchestration using machine learning methods." *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 23.5 (2018): 1-33.
[26] El Mandouh, Eman, and Amr G. Wassal. "Estimation of formal verification cost using regression machine learning." *2016 IEEE International High Level Design Validation and Test Workshop (HLDVT)*. IEEE, 2016.
[27] Seshia, Sanjit A. "Combining induction, deduction, and structure for verification and synthesis." Proceedings of the IEEE 103.11 (2015): 2036-2051.
[28] Cybenko, George. "Approximation by superpositions of a sigmoidal function." *Mathematics of control, signals and systems* 2.4 (1989): 303-314.
[29] Schäfer, Anton Maximilian, and Hans-Georg Zimmermann. "Recurrent neural networks are universal approximators." *International journal of neural systems* 17.04 (2007): 253-263.
[30] Gogri, Saumil, et al. "Machine Learning-Guided Stimulus Generation for Functional Verification." *Proceedings of the Design and Verification Conference (DVCON-USA), Virtual Conference*. 2020.
[31] Hughes, William, et al. "Optimizing design verification using machine learning: Doing better than random." *arXiv preprint arXiv:1909.13168* (2019).
[32] Ioannides, Charalambos, and Kerstin I. Eder. "Coverage-directed test generation automated by machine learning--a review." *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 17.1 (2012): 1-21.
[33] Vasudevan, Shobha, et al. "Learning semantic representations to verify hardware designs." *Advances in Neural Information Processing Systems* 34 (2021): 23491-23504.
[34] Truong, Andy, et al. "Clustering and Classification of UVM Test Failures Using Machine Learning Techniques." Proceedings of the Design and Verification Conference (DVCON), San Jose, CA, USA. Vol. 26. 2018.
[35] Werneman, Oscar. "Predicting Bugs to Reduce Debugging Time.", Master Thesis, 2021
[36] Raffel, Colin, et al. "Exploring the limits of transfer learning with a unified text-to-text transformer." J. Mach. Learn. Res. 21.140 (2020): 1-67.
[37] Anaconda, "2022 State of Data Science", https://www.anaconda.com/state-of-data-science-report-2022, accessed in Oct. 2022
[38] Mäkinen, Sasu, et al. "Who needs MLOps: What data scientists seek to accomplish and how can MLOps help?." *2021 IEEE/ACM 1st Workshop on AI Engineering-Software Engineering for AI (WAIN)*. IEEE, 2021.
[39] Scarselli, Franco, et al. "The graph neural network model." *IEEE transactions on neural networks* 20.1 (2008): 61-80.
[40] Friedman, Jerome H. "Stochastic gradient boosting." *Computational statistics & data analysis* 38.4 (2002): 367-378.
[41] Friedman, Nir, Dan Geiger, and Moises Goldszmidt. "Bayesian network classifiers." *Machine learning* 29.2 (1997): 131-163.
[42] Gershgorn, Dave, "The data that transformed AI research—and possibly the world." https://qz.com/emails/daily-brief/1849725171/shrinking-gdp-and-expanding-inflation, retrieved in Oct. 2022