Survey of Machine Learning for Software-assisted Hardware Design Verification: Past, Present, and Prospect

NAN WU, George Washington University, USA
YINGJIE LI, University of Maryland, USA
HANG YANG, Georgia Institute of Technology, USA
HANQIU CHEN, Georgia Institute of Technology, USA
STEVE DAI, NVIDIA Corporation, USA
CONG HAO, Georgia Institute of Technology, USA
CUNXI YU, University of Maryland, USA
YUAN XIE, Hong Kong University of Science and Technology, China

With the ever-increasing hardware design complexity comes the realization that efforts required for hardware verification increase at an even faster rate. Driven by the push from the desired verification productivity boost and the pull from leap-ahead capabilities of machine learning (ML), recent years have witnessed the emergence of exploiting ML-based techniques to improve the efficiency of hardware verification. In this paper, we present a panoramic view of how ML-based techniques are embraced in hardware design verification, from formal verification to simulation-based verification, from academia to industry, and from current progress to future prospects. We envision that the adoption of ML-based techniques will pave the road for more scalable, more intelligent, and more productive hardware verification.

Additional Key Words and Phrases: hardware verification, machine learning, formal verification, simulation-based verification

ACM Reference Format:
Nan Wu, Yingjie Li, Hang Yang, Hanqiu Chen, Steve Dai, Cong Hao, Cunxi Yu, and Yuan Xie. 2024. Survey of Machine Learning for Software-assisted Hardware Design Verification: Past, Present, and Prospect. *ACM Trans. Des. Autom. Electron. Syst.* 1, 1 (July 2024), 42 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Hardware verification is at least comparably important with hardware design, if not more, in terms of labor, costs [191, 217], and project time [93] consumed in chip development cycles: the number of average engineers required for verification is commensurate with that of design [93]; Figure 1 shows the development efforts for advanced circuit designs, where verification roughly occupies half of the cost for hardware development; Figure 2 illustrates that verification obviously dominates the project time. With the growing complexity in design and verification, nearly 70% of application-specific integrated circuit (ASIC) or field-programmable gate array (FPGA) projects are completed behind schedule [93]. This indication is appealing for innovations in not only design but also verification.

Machine learning (ML) has demonstrated its versatility in hardware design optimization and evaluation [267]. This brings up one key question: **whether ML-based techniques can improve**

Authors' addresses: Nan Wu, Department of Electrical and Computer Engineering, George Washington University, Washington, District of Columbia, 20052, USA, nan.wu@gwu.edu; Yingjie Li, Department of Electrical and Computer Engineering, University of Maryland, College Park, Maryland, 20740, USA, yingjiel@umd.edu; Hang Yang, School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, Georgia, 30332, USA, hyang628@gatech.edu; Hanqiu Chen, School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, Georgia, 30332, USA, hchen799@gatech.edu; Steve Dai, 2701 San Tomas Expressway, NVIDIA Corporation, Santa Clara, California, 95050, USA, sdai@nvidia.com; Cong Hao, School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, Georgia, 30332, USA, callie.hao@ece.gatech.edu; Cunxi Yu, Department of Electrical and Computer Engineering, University of Maryland, College Park, Maryland, 20740, USA, cunxiyu@umd.edu; Yuan Xie, Department of Electrical and Computer Engineering, Hong Kong University of Science and Technology, Hong Kong, China, yuanxie@ust.hk.

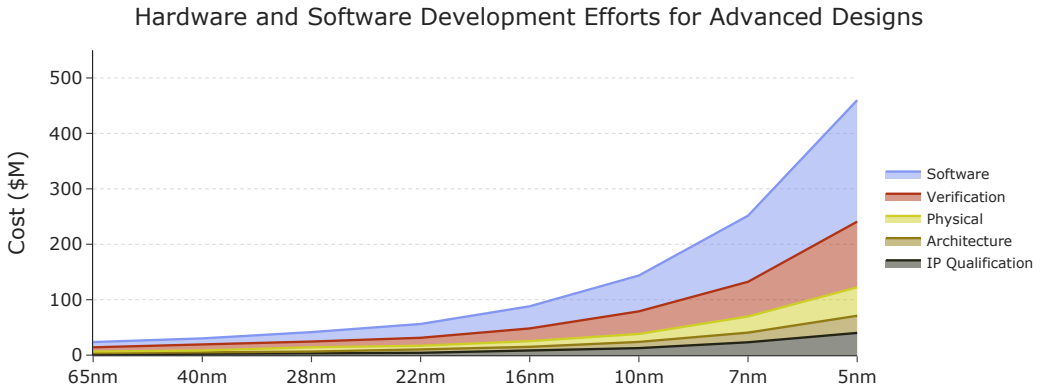

Fig. 1. Development efforts for advanced designs under different technology nodes, which is redrawn based on data from Cadence [217].

the quality and productivity of hardware design verification? Hardware verification technologies can be roughly classified into two categories: hardware-assisted and software-assisted. The former includes emulation and prototyping, and the latter encompasses formal verification and oracle-based verification. Hardware-assisted verification has been gaining an expanding market share [141], especially advantageous for large-scale hardware designs. Despite its growth, the application of ML-based techniques in hardware-assisted verification is still in its early stages, with limited research studies available. For example, Siemens EDA employs ML to analyze the knowledge database built from previous compilations and emulation runs, making the hardware emulator smarter and more cost-effective over time [213]. To improve the compilation flow of multi-FPGA-based emulation systems, ML models can be applied to predict whether a netlist is easy or hard to compile, identify effective compilation strategies, estimate compilation time, and offer recommendations for modifying hard-to-compile netlist partitions into easier ones [6]; evaluation on a large-scale industry system-on-chip design demonstrates that the ML-based recommendation flow achieves an additional 15% reduction in compilation time. It is highly expected that more endeavors will be made in the future to leverage ML to enhance hardware-assisted verification. In light of the wealth of research integrating ML with software-assisted verification, our emphasis in this survey is on exploring how ML complements software-assisted design verification, specifically within the realms of formal and oracle-based verification.

Formal verification uses static analysis to mathematically prove or disprove the functional correctness of a system with respect to certain formal specifications or properties [220]. Several popular formal methods include the Boolean satisfiability (SAT) problem and equivalence checking. As the SAT problem is NP-complete [66] and equivalence checking is coNP-complete [113], they become computationally impractical with complex systems or hardware designs. **Oracle-based verification** exercises the designs with valid input stimulus to compare whether the outputs match the oracle provided by a golden reference model [40]. Different coverage metrics are defined to assess if the design-under-verification (DUV) or design-under-test (DUT) has been adequately examined. Though oracle-based verification has been the major technique for hardware verification, it still suffers from the scalability issue caused by the growing design size and more complex system-level integration; the tremendous amount of data generated during simulation further complicates the problem. These challenges all call for more productive verification by incorporating more intelligence. Driven by both the push from the desired verification productivity boost and the

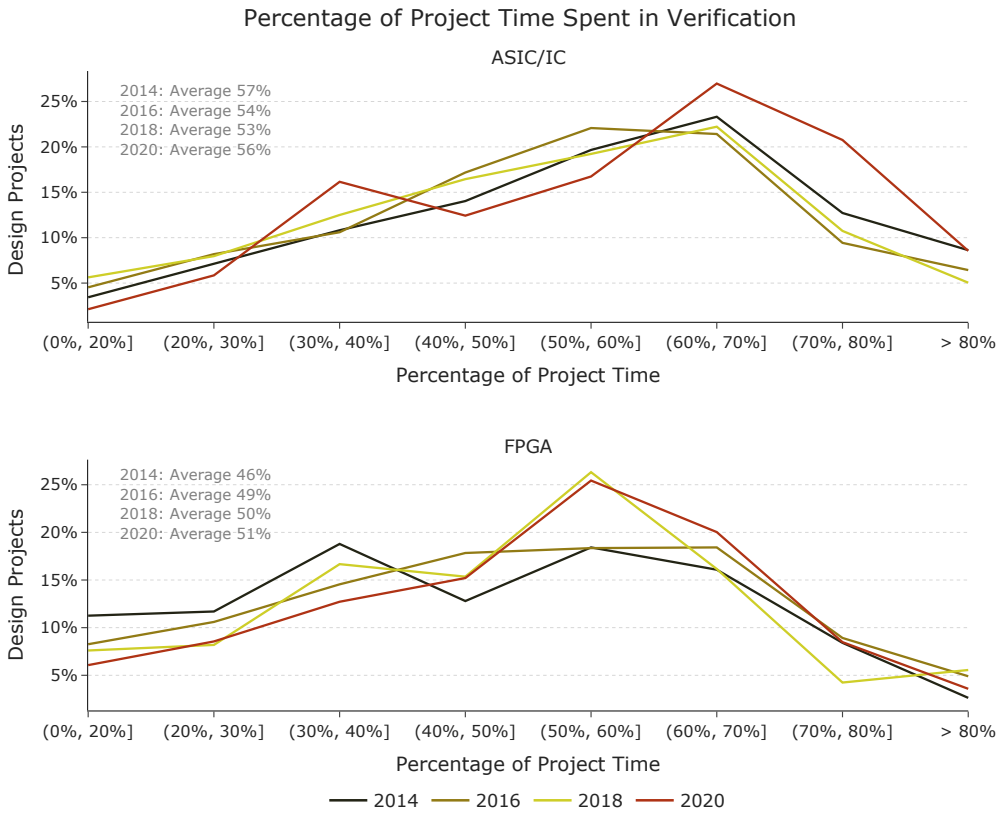

Fig. 2. Percentage of ASIC/IC and FPGA project time spent in verification, which is redrawn based on data from the 2020 Wilson Research Group Functional Verification Study [93]. Similar trends are observed in the study in 2022 [93]. Note that verification time dominates project time on average.

pull from leap-ahead capabilities of ML-based techniques, **this question has a positive answer**, as follows.

- In formal verification:
  - graph-based learning approaches are naturally suitable to improve the efficiency of SAT solvers, since Boolean formulas can be represented as graphs;
  - the natural language processing (NLP)-based techniques largely benefit assertion generation across natural languages, verification languages, and hardware description languages;
  - various types of classifiers enable efficient identification of equivalence across different levels of abstractions, facilitate fast generation and verification of properties, and aid in selecting the optimal premises and heuristics used in theorem proving as well as the most suitable formal engine for diverse verification problems.
- In oracle-based verification:
  - fast and accurate ML-based predictive models can significantly reduce the long simulation time of DUTs and simplify exhaustive coverage analysis;
  - ML-based optimizers are capable to eliminate the long search time for stressful and comprehensive test sets and accelerate coverage closure;

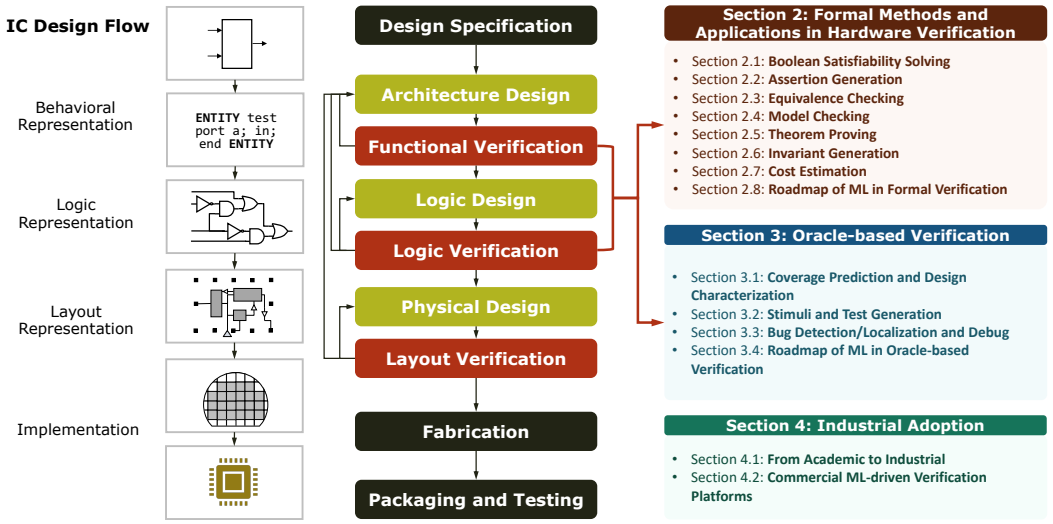

Fig. 3. Overall organization of this paper. Note that hardware verification is an indispensable step to guarantee desired functionalities.

– various clustering and classification methods enable the automation of bug detection, localization, and debug from a tremendous amount of failure information, which is nearly impossible for human engineers to well parse..

While there exist several short surveys about ML applications in hardware verification [70, 145, 251], they mostly focus on simulation-based verification, which is insufficient in practice, and provide detailed discussions on a few case studies, lacking a panorama of both static and dynamic verification, both academia and industry, and both current progress and future prospects. In this paper, we present a comprehensive overview of ML for hardware verification. From the static perspective (Section 2), we first briefly introduce the mainstream formal methods and their applications in hardware verification; then we summarize the problems in formal verification that can be solved by ML techniques, the common ML techniques adopted to resolve each of them, and the reason why ML techniques are powerful in these scenarios. From the dynamic perspective (Section 3), we review studies associated with three major steps in oracle-based verification: coverage analysis, test generation, and debug; we further discuss the role that ML techniques play, either a predictive model, an optimizer, or a combination of these two. We also provide a future vision of challenges, opportunities, and prospects of applying ML for hardware verification, aiming to pave the road for integrating more intelligence into next-generation verification (Section 2.8 and Section 3.4). In addition to attention to academic studies, we conduct sketchy discussions on how ML techniques are embraced in industrial productions or commercial verification tools (Section 4).

2 FORMAL METHODS AND APPLICATIONS IN HARDWARE VERIFICATION

The use of formal approaches in hardware verification is widespread. Contrary to testing-based verification approaches, formal verification makes use of rigorous formal methods and representations to formulate the specification and verification objectives of systems and to determine, without the need for extensive testing, whether the finalized design satisfies the specification at all properties [154]. To be more precise, formal approaches in hardware verification cover a wide range of specifications, from high-level specifications (such as SystemC, behavior HDL) to low-level

logic or physical representations (e.g., netlists). Depending on the nature of the design specification representation and its functionality, the requirements could be shown as block diagrams, look-up tables, or temporal logic. While there are a large variety of hardware formal verification problems, the fundamentals are to model the design properties and hardware implementation into rigorous mathematical representations and to obtain the verification objectives using desired verification techniques and solving mechanisms. In particular for hardware verification, the majority of the verification problems are formulated and solved with model checking [61], equivalence checking [159, 249], assertion-based formal verification [92, 97], and theorem proving [21, 74, 167]. To prove the mathematical models introduced by such verification techniques, solving mechanisms based on *canonical diagrams* [38, 39], *Boolean satisfiability* (SAT) [28, 175, 183], *Satisfiability Modulo Theories* (SMT) [22, 29, 75, 184], theorem proving, *computer algebra* [58, 59, 69, 173, 212, 273], etc., have been designed to prove or disprove with counterexamples. Even though formal verification can complement and eliminate extensive simulation or testing processes by rigorous mathematical reasoning, their scalability in practical verification problems has been the biggest challenge due to its runtime complexity and/or memory usage [96]. Hence, there have been increasing efforts in harnessing ML techniques to improve the scalability and reduce the runtime of different formal verification categories from various aspects. This section reviews the recent progress in SAT solving (Section 2.1), assertion-based formal verification (Section 2.2), equivalence checking (Section 2.3), model checking (Section 2.4), theorem proving (Section 2.5), invariant generation (Section 2.6), and runtime prediction and optimization (Section 2.7).

2.1 Boolean Satisfiability Solving

The SAT problem (i.e., Boolean satisfiability problem) is a decision problem: given a Boolean propositional formula, if there exists at least one valuation making the formula true, this formula is claimed to be satisfiable (SAT); otherwise, this formula is unsatisfiable (UNSAT). Boolean formulas are usually represented in the Conjunctive Normal Form (CNF): a formula is expressed as a conjunction ($\wedge$) of clauses; a clause is expressed as a disjunction ($\vee$) of literals that can be either a variable or its negation ($\neg$).

Many practical problems can be reduced to the SAT problem, a proven NP-complete problem [66]. Given its broad applications, there has been consistent research attention on this hard combinatorial problem, and one promising direction is employing ML-based techniques to advance combinatorial optimization [24]. Targeting the SAT problem, ML-based techniques can be adopted to ❶ **build classifiers that provide fast SAT/UNSAT decisions based on input formulas** [15, 41, 81, 198, 222, 224], or ❷ **improve bottleneck components in existing SAT solvers** [163, 221, 265, 272, 276], such as Conflict-Driven Clause Learning (CDCL) solvers and Stochastic Local Search (SLS) solvers. For more reference, there is a comprehensive survey of ML for SAT solving [121].

Classification for fast SAT/UNSAT decisions. The SAT problem can be formulated as a binary classification based on input Boolean formulas. A Boolean formula in CNF can be represented as a bipartite graph, referred to as a literal-clause incidence graph (LIG), which contains nodes for each clause and each literal in the formula as well as edges connecting a clause node and a literal node if the clause contains this literal. When the literal nodes of the same variable in a LIG are connected, it is defined as LIG$^+$. The LIG-based representation makes it natural to exploit graph neural networks (GNNs) to classify satisfiability [41]. For example, NeuroSAT [222] consists of two parts: an encoder with a customized GNN that takes in the LIG$^+$ representation of a formula and outputs an embedding for each literal, and an aggregator that maps each literal embedding to a scalar vote and aggregates them into the final prediction of satisfiability. NeuroSAT reaches an accuracy of 85% when evaluating on a synthetic random SAT generator with 40 variables per instance. Following NeuroSAT, Shi et al. [224] employs the transformer architecture to realize parallel speedup, which uses self-attention

Table 1. Formal verification with ML-based techniques. Both formal methods (SAT solving, assertion generation, equivalence checking, model checking, theorem proving, and invariant generation) and cost estimation can be improved by different ML-based techniques. S, U, and R stand for supervised, unsupervised, and reinforcement learning, respectively.

Formal Method	Problem Formulation	FWK	ML Model/Technique
Boolean Satisfiability Solving (Section 2.1)	Binary classification for satisfiability	S	GNN [222]
		S	Transformer [224]
		S	GNN + Contrast learning [81]
	Generating satisfiable assignments	U	Deep-gated recursive GNN [15]
		U	Recurrent GNN [198]
	Improving branching heuristics in CDCL	S	GNN [221]
		R	DQN + GNN [163]
	Improving variable initialization in CDCL	S	Logistic regression [265]
	Improving variable selection in SLS	R	REINFORCE + GNN [272]
	Improving variable initialization in SLS	S	GGCN [276]
Assertion Generation (Section 2.2)	RTL + simulation trace → assertion	S	Binary decision tree [166, 223, 254]
		S	Breadth-first decision tree [126]
	NL → SAL	U	NLP [108]
	NL → SVA	U	E-GRIDS [127], constituency tree [199]
		S	Transformer [50]
		S	GPT-3 [5]
	NL → RTL + UVM, SVA → NL	S	LSTM + machine translation [118]
Equivalence Checking (Section 2.3)	System-level modeling & RTL equivalence	S	SVM [134, 135]
	RTL & Gate-level signal correlation	S	Decision tree [13]
	Boolean functions & hardware implementation	U	RNN [232]
Model Checking (Section 2.4)	Generating properties	R	GAN with RNN+CNN [102]
	Directly verifying properties by classification	S	Boosted tree, random forest, decision tree, and logistic regression [278]
		S	GNN [185]
	Finding counter-examples	R	Q-learning [17, 23, 269]
	Refining abstraction based on counter-examples	S	Decision tree [60, 62]
Theorem Proving (Section 2.5)	Premise selection	S	Naive Bayes/kernel-based learning [98, 151]
		S	CNN, RNN [143]
		U	Transformer [18]
		U	GNN [91]
	Heuristic configuration adaptation	S	Gradient-based methods [98]
		S	Gaussian-kernel-based regression [160]
	Heuristic selection	S	Kernel-based learning [36, 37]
	Autoformalization	S + R	Imitation learning, REINFORCE, and GAN [263]
		S	LLM [268]
Invariant Generation (Section 2.6)	Generating numerical and quantified invariants	S	Decision tree [103, 104]
	Generating loop invariants	R	GNN, LSTM, and policy gradient [228]
Cost Estimation (Section 2.7)	Runtime estimation for formal verification	S	Linear regression [84]
	Formal engine selection for verification problems	S	Decision tree [85]
	Time and memory usage	S	MLP [247]

and cross-attention to capture interactions among homogeneous and heterogeneous nodes in LIG-based representations, respectively. To relax the requirement of labeling a large number of SAT instances in supervised models, Duan *et al.* [81] combine label-preserving augmentations with contrast learning, which achieves comparable prediction accuracy to NeuroSAT with 100× reduction in the number of required labels. Aiming to solve the SAT problem by directly generating a solution, Amizadeh et al. [15] propose a differentiable framework mimicking reinforcement learning (RL), which consists of a solver network and an evaluator network: the solver network, with the deep-gated recursive GNN structure, takes in Boolean circuits represented in directed acyclic graphs (DAGs) and directly generates an assignment of input variables; the evaluator

network evaluates whether the predicted assignment is SAT; the optimization goal is to push the solver network to generate an assignment that yields a higher satisfiability value in the evaluator network. QuerySAT [198] adopts recurrent GNNs with a query mechanism; for each query of a variable assignment, a differentiable unsupervised loss function is designed by relaxing input variables from discrete to continuous values, which can be directly optimized toward finding a satisfying assignment.

Even though ML-based techniques can make fast decisions of SAT/UNSAT, they are approximating the heuristics employed in conventional SAT solvers by stochastic NN models, without a theoretical guarantee of correctness. Thus, these ML-assisted SAT decisions are expected to quickly filter out obviously UNSAT solutions at early search stages instead of completely replacing conventional SAT solvers. In addition, several studies introduce unsupervised techniques to get rid of the efforts of labeling data, however, it requires high expertise to craft their differentiable loss functions [81, 198]. So, there is a trade-off between labeling efforts and required expertise in loss functions, suggesting the adoption of semi-supervised techniques in future work.

Improving conventional SAT solvers. For CDCL-based solvers, NeuroCore [221] incorporates a simplified NeuroSAT [222] into several high-performance SAT solvers that use the Exponential Variable State-Independent Decaying Sum (EVSIDS) as the branching heuristic [183], which maintains activity scores for every variable and branches on the free variable with the highest score. The goal is to complement the branching heuristics used in existing solvers by periodically querying NeuroSAT and resetting the variable activity scores according to NeuroSAT's predictions of how likely the variables appear in an unsatisfiable core. Graph-Q-SAT [163] exploits deep Q-network (DQN) to improve the branching heuristic in CDCL-based solvers. The SAT formula is represented in a LIG, and the DQN agent equipped with a GNN decides whether to set each unassigned variable as true or false. Wu [265] focuses on finding the preferred initial value of each Boolean variable in a CDCL-based solver, where a logistic regression model is employed to predict SAT/UNSAT of Boolean formulas after fixing the values of certain variables.

For SLS-based solvers, Yolcu and Póczos [272] incorporate GNNs with REINFORCE to act as the variable selection heuristic. The GNN takes in LIG-based representations of Boolean formulas together with an assignment to all variables, and outputs a probability vector of flipping the value of each variable. NLocalSAT [276] uses a gated graph convolutional network (GGCN) to directly predict solutions to SAT instances based on LIG-represented formulas, providing guidance to the variable initialization heuristics.

Integrating ML-based techniques with conventional SAT solvers is a conservative yet more feasible way. Despite the performance boost in the solvers, one thing worth noting is the employment of ML models may introduce new sources of biases [180]. For instance, a common issue is the significant imbalance in data samples from SAT and UNSAT categories, potentially leading to sampling bias; even with representative data, human experts perform feature engineering and ML model selection, such as certain optimization objectives, regularizations, and constraints, based on their understanding of the nature of verification problems, which may include flawed or stale information that can bias the outcome of ML algorithms.

2.2 Assertion Generation

In formal verification, assertions are used to express desirable properties to be proven by formal methods and to define constraints in the verification environment for DUTs. The effectiveness and completeness of formal verification rely heavily on the generated assertions, however, assertion generation requires heavy manual efforts, such as multiple iterations and man-months, to achieve the minimal but effective assertions with high coverage [97]. This motivates automatic assertion generation using ML techniques. In general, assertions can be generated from two major sources: ❶

design-related information, such as RTL designs and simulation traces [126, 166, 223, 254], and ❷ **design specification documents in natural languages** [5, 50, 108, 118, 127, 199].

Generation from design-related information. RTL designs and their simulation traces are good sources to generate hardware assertions using data mining techniques. GoldMine [254] is a methodology to automatically generate RTL assertions using data mining and static analysis. It consists of five major components: (1) data generator, which simulates RTL designs with random vectors to produce dynamic behavioral data; (2) lightweight static analyzer that extracts domain-specific information about the design; (3) A-miner that derives knowledge and information from simulation traces and design constraints, where a binary decision tree is employed to produce a set of ranked candidate assertions; (4) formal verifier, which verifies the candidate assertions produced by A-miner to filter out spurious assertions and retain system invariants; (5) A-Val, which compares the machine-generated assertions to those generated by a human and provides feedback to refine GoldMine accordingly. To achieve a higher coverage and generate more succinct assertions, Sheridan et al. [223] improve GoldMine by proposing a coverage-guided assertion miner with a combination of association rule learning and greedy set covering. Based on the GoldMine framework that generates bit-level assertions, Liu et al. [166] enhance the quality of assertion mining by generating word-level assertions that have higher expressiveness and readability than their counterparts in the bit-level. Hanafy et al. [126] extend the binary decision tree adopted in GoldMine to a breadth-first decision tree for assertion generation and refinement.

Generation from design specification documents. Hardware assertions can be directly extracted from design specification documents by translation from natural languages (NL) to assertion languages. ARSENAL [108] automatically translates NL requirements into a unified symbolic analysis laboratory (SAL) model with associated properties, through two stages: the NLP stage that extracts relations from texts using semantic parsing and interprets NL sentences to logical formulas, and the formal methods stage that converts the generated logical formulas to a SAL-based formal model. GLAst [127] transforms specification sentences in English to formal SystemVerilog assertions (SVA) by extending the E-GRIDS algorithm [204], which creates a custom formal grammar to capture the writing style and sentence structures of specifications to facilitate the automatic translation. With a similar goal, SpecToSVA [199] employs a random forest classifier to decide which sentence in specifications should be translated, an NLP package, spaCy [129], to detect name entity recognition, and a constituency tree with a self-attentive encoder [155] for final translation from design specifications in English to SVA. The evaluation shows an average precision of 64% on a dataset created from proprietary IC specification documents. Notably, SpecToSVA can only translate single and self-contained sentences to SVA, and cannot handle cases where multiple sentences need to be collectively considered and translated into a single SVA expression. Gulliya et al. [118] give attention to the conversion between English and SVA in register verification: from English to SVA, a long short-term memory (LSTM)-based model is trained to classify the type and functionality of different registers, and the prediction results are processed to generate proper RTL and universal verification methodology (UVM) code; from SVA to English, rule-based machine translation is employed. Recently, Transformer-based models, which are the state-of-the-art models of NLP, outperform conventional one-step attention RNNs in machine translation from English to SVA [50]. Though DAVE, a fine-tuned GPT-2 (the second generation Generative Pre-trained Transformer) model, majorly aims to automatically translate NL into Verilog snippets [202], it considers the assertion generation as the future work. Aditi et al. [5] propose a hybrid approach that utilizes GPT-3 to extract the entity information from specification sentences, after which a set of hand-crafted rules is applied for the final translation. This approach successfully generates SVAs for 1712 out of the 2000 statements (~85% success rate).

Automatic assertion generation enabled by ML-based techniques largely benefits verification productivity. The process of identifying assertions from either design itself or specification documents is analogous to neural machine translation, revealing numerous opportunities, especially in the era of large language models (LLMs). Though these ML-based models may not always generate non-trivial assertions, they can offer potential candidates that can be easily refined by human experts [5, 50].

2.3 Equivalence Checking

Equivalence checking guarantees design integrity by using mathematical modeling techniques to show that two representations from different design levels of the DUT exhibit the same behaviors. Practically, fast and accurate equivalence checking saves significant time and effort in engineering change order (ECO). With the help of ML-based techniques, the features and embeddings from one design level can be well captured and accurately mapped to another design level, greatly accelerating equivalence checking. For cross-design-level translations, we discuss the equivalence checking ❶ **from system-level to RTL** [134, 135], ❷ **from RTL to gate-level**, and ❸ **from Boolean functions to hardware implementation** [232].

System-level → RTL. Modern hardware designs usually start with behavioral descriptions in high-level languages, such as C/C++ and SystemC, which are then compiled into RTL designs by synthesis tools or manual transformation. This translation step is time-consuming and error-prone. Thus, Hu et al. [135] employ support vector machine (SVM) to improve the efficiency of finite state machines with datapath (FSMD)-based equivalence checking between the system-level modeling (SLM) and its generated RTL design. Both SLM and RTL designs are converted into FSMD representations. The SVM model then learns from the SLM states to predict the corresponding RTL states, which helps identify the path pairs between SLM and RTL and drastically reduces the time complexity by avoiding arbitrary path selection. The selected path pairs will be further checked using symbolic simulation and the SMT solver. They also analyze multiple ML models to examine the trade-offs between effectiveness and efficiency [134].

RTL → gate-level. Designers usually optimize RTL designs for ECO, which can change the design hierarchy and transform signal names, making it hard to correlate RTL design signals with their counterparts in gate-level implementation [65]. Alhaddad et al. [13] explore seven ML models, including decision tree, random forest, K-nearest neighbors, linear support vector classification (SVC), linear regression, naive Bayes, and linear discriminant analysis, to identify each pair of signals from RTL and gate-level are equivalent or not, based on name features, structural features, signal proprieties, and functional similarity features. Evaluation on the OpenCores [195] benchmark shows that the decision tree outperforms other models, with the best accuracy of 89%.

Boolean functions → hardware implementation. Singireddy et al. [232] focus on the equivalence checking between flow-based computing systems in memristor crossbars and reference Boolean functions. The crossbar design is first recast into an RNN, which allows equivalence checking to be efficiently performed by NN inference, and then instances of Boolean variables are verified by passing input vectors to the RNN and checking the output against the reference Boolean function. Evaluation on circuits from the RevLib [264] and MCNC [271] suites show that the proposed method can verify the correctness of a design 166× faster than the state-of-the-art method based on graph reachability on average.

Despite the speedup achieved by ML-based techniques, there are several observations on existing approaches. First, feature engineering plays an important role in building effective ML models [13, 135]. However, this process can become cumbersome, particularly in scenarios involving large feature spaces. Second, the trained ML models seek to imitate the rules in equivalence checking,

aiming to provide rapid suggestions. Their full potential can be realized by integrating them with solvers that offer theoretical guarantees [13, 232].

2.4 Model Checking

Model checking involves abstracting the execution of a concrete system into a finite-state automaton and expressing the properties of interest by temporal logics. The procedure of model checking exhaustively explores the (abstract) state space, and either validates the properties or provides a counter-example. In practice, exhaustive exploration is challenging in face of the state explosion problem [63], which prompts the development of various techniques aiming at reducing the state space or enhancing its exploration through path exploration heuristics. We discuss how ML-based techniques contribute to ❶ **automatically generating properties** [102], ❷ **directly verifying properties** [185, 278], ❸ **finding counter-examples** [17, 23, 269], and ❹ **refining abstraction based on counter-examples** [60, 62].

Generating properties. To automate the laborious process of manually generating verification properties, Gao et al. [102] present a GAN-based property generation approach, which employs an LSTM-based model as the generator and a CNN-based classifier as the discriminator. Existing verification properties are represented as sequences of word vectors, serving as the initial inputs for the SeqGAN [274] algorithm to generate new verification properties. As this approach focuses on generating new verification properties using computational tree logic (CTL) formulas and/or Kripke structures, and given that hardware designs can be converted to Kripke structures and design specifications can be translated into CTL formulas, it is easily transferrable to hardware program property generation by adjusting the training data and corresponding labeling efforts.

Directly verifying properties. Linear temporal logic (LTL) model checking suffers from the state explosion problem and the exponentially growing compute complexity [63]. To alleviate these issues, several studies formulate LTL model checking as binary classification problems, in which 0/1 indicates the model (un)satisfies the specification. Zhu et al. [278] evaluate several ML algorithms (i.e., boosted tree, random forest, decision tree, and logistic regression) to predict whether an automaton satisfies a specification. While receiving comparable prediction accuracy to the classical LTL model checking tool, NuXMV [47], these algorithms exhibit limited generalization across different LTL formula lengths and automaton sizes. To achieve better generalization, GNN is adapted to encode the automaton with the target LTL formula expressed as node embeddings [185]. This graph classification method is capable of predicting results up to 17× faster than the tool LTL3BA [19], especially when dealing with very large LTL formulas.

Finding counter-examples. Model checking tools should be geared towards efficient error detection [64]. Several studies improve the efficiency of finding counter-examples by using Q-learning to explicitly guide the exploration of paths in automaton that would violate properties of interest. For instance, Araragi and Mo Cho [17] give rewards to explorations leading to cycles or infinite paths between the premise and the response that invalidate the response property, which states that if a premise event occurs, a response event will be true thereafter. Behjati et al. [23] then extend the Q-learning algorithm to accommodate more LTLs. To enhance the search efficiency of Q-learning based approaches, neural Monte Carlo tree search (MCTS) is employed [269], which is capable to find counter-examples in less runtime. While Q-learning is effective in identifying counter-examples, it may face challenges due to high sensitivity in reward functions [239]. Hence, crafting meaningful rewards is crucial in these problems, especially in situations when it is challenging to build a clear relationship between the optimization goal and rewards.

Refining abstraction based on counter-examples. A counter-example can be spurious, if the last state of a path in the abstracted system model unites both deadend states (i.e. states without concrete transitions to failure states) and bad states (i.e., states with transitions to failure states) in

the actual system. This is because some system variables that are useful to prove the properties and distinguish these states are not considered in the system abstraction, suggesting a refinement in the abstracted system model by making these variables visible. To separate these states, Clarke et al. [60, 62] employ decision trees to classify whether system variables from the failure state should be visible.

2.5 Theorem Proving

In theorem proving, systems are described by appropriate mathematical logics, and important mathematical properties of the system are proved by theorem provers. Theorem provers check whether a conjecture, the target mathematical statement to be inferred, can be derived from a set of statements (e.g., axiom, hypothesis) and output the proof for the conjecture or a trace thereof. This is similar to computer algebra since both of them are used for symbolic computation, but with the advantages of more flexibility in logic expressiveness, clearer expression, and more rigor.

Theorem provers are essentially based on mathematical logics, and there is always a trade-off between automation and expressive power of logics [36]. For example, first-order logic (FOL) has high-level automation and is usually used to construct automated theorem provers (ATPs) [186]; higher-order logic (HOL) is more expressive and has the ability to prove complex problems, but requires more human guidance, which is often used to build interactive theorem provers (ITPs), e.g., Isabelle [30, 189], MIZAR [116], and Coq [26]. These theorem provers can be effectively applied for hardware design verification with light modifications. For example, Kami [55] is an extended library embedded in Coq, enabling expressive and modular reasoning for hardware designs, which has the capability to verify fairly realistic processors within Coq.

ML-based techniques have been successfully employed to aid theorem-proving systems in the following aspects: ❶ **the selection of facts (also know as premises)** that are most relevant for proving a new conjecture, which can be formulated as ML-based classification tasks [11, 18, 91, 143, 148, 151]; ❷ **automatic heuristic configuration and selection in ATPs**, where ML can help to model the relationship between input problems and the chosen heuristics [36, 37, 98, 160]; and ❸ **autoformalization** [263, 268].

Premise selection. The premise selection task entails a conjecture and a set of facts/premises, with the objective of strategically choosing a subset of premises to be forwarded to an ATP. Alama et al. [11] employ naive Bayes and kernel-based learning methods to estimate which premises are likely to be useful for constructing a proof. To improve the scalability, Kaliszyk and Urban [151] adopt the sparse implementation of a multi-class naive Bayes classifier to rank available facts, where the top-ranked facts are selected to realize the proof. Irving et al. [143] simplify the subset selection of premises to computing pairwise relevance: two CNN/RNN sequence models are developed to compute the embeddings for a conjecture and a premise, respectively, which are then concatenated to produce the probability that this premise is useful for proving the conjecture.

Recent approaches convert logic statements into graphs and leverage graph representations to assist theorem provers. Aygun et al. [18] build a FOL prover by replacing all the clause-scoring heuristics of the state-of-the-art E prover [218, 219] with a Transformer-based clause scoring function. The Transformer-based model takes in clauses represented in graphs with spectral features and outputs the probability of a clause appearing in the proof, which is trained using hindsight experience replay in an incremental learning setting. The proposed approach can compete with the E prover in its best configuration. To further improve the transferability across domains using different vocabularies, NIAGRA [91] generates name invariant graph neural representations of clauses that provide more meaningful embeddings of non-logical symbols, such as predicates, functions, and constants. It also adopts the ensemble method that leverages different configurations

of the underlying ATP to learn proof guidance models. Evaluation shows that NIAGRA outperforms the method proposed by Aygun et al. [18] by 10%.

Heuristic configuration adaptation and selection. Heuristic configuration refers to a set of parameters in theorem provers that affect their preference on clause weighting and selection schemes, term orderings, and sets of inference and reduction rules used. Tuning such parameters is time-consuming and can be automated by different ML-based techniques. Fuchs [98] makes an early attempt to learn search-guiding heuristics from past experiences, where gradient-based methods are used to adapt coefficients in the heuristics to ensure application for a wider range of target problems. Kühlwein and Urban [160] propose an automatic framework to tune parameters inside ATPs. Given the parameter setting of an ATP and a target problem, a Gaussian-kernel-based method is adopted to predict its runtime, which enables more efficient adaptation of parameter settings for different problems. In addition to setting parameters in heuristics, Bridge et al. [36, 37] emphasize on selecting useful heuristics for proof search. Based on features of the conjecture to be proved and the associated axioms/facts, SVM and Gaussian-kernel-based methods are capable to select a beneficial set of heuristics. The developed ML-assisted system is also able to decline proof attempts, significantly reducing the time spent in proof with only a moderate reduction in the number of provable theorems.

ML-based techniques can greatly improve the scalability of premise selection and configuring heuristics, but a key consideration is the quality of these ML models often heavily depends on the available training data. Given the large search space of premises or configurations, it is highly probable that the collected training data implicitly incorporate the preference and domain knowledge from human experts. On the positive side, ML models can quickly imitate the behavior of human experts, while on the negative side, the learned solutions may be suboptimal and bounded by human perception. Possible solutions to address this concern involve combining techniques such as active learning, RL, or generative approaches with existing methods, which can generate new data points on-the-fly and potentially mitigate the limitations of relying solely on human-derived data.

Autoformalization. Autoformalization refers to the automatic translation from NL mathematics to formal specifications, theorems, and proofs. MetaGen [263] is a neural theorem generator, which synthesizes new theorems and their proofs expressed in the formalism in Metamath [179]. Two scenarios are considered: if both human-written theorem statements and their proofs are available, MetaGen is trained by imitation learning to provide human-like theorems; if only theorems are available, MetaGen uses the REINFORCE algorithm to find synthetic theorems similar to human-written theorems, where a discriminator, such as the one in generative adversarial networks (GAN) [114], is trained to measure the similarity. Wu et al. [268] observe that LLMs, such as OpenAI Codex [51, 193] and PaLM [57], can correctly translate a certain portion (25.3%) of mathematical competition problems in English to formal code used by the interactive proof assistant Isabelle [189]. The performance of a neural theorem prover, LISA [147], can be improved by training with these autoformalized theorems, reaching a new state-of-the-art proof rate from 29.6% to 35.2% on the MiniF2F [277] theorem proving benchmark. Though LLMs may not be able to generate perfect specifications or correct statements, they can produce slightly incorrect formal specifications and properties quickly, making it easier for hardware designers to iterate them efficiently.

2.6 Invariant Generation

Invariants, by capturing properties that remain unchanged during program execution, provide a foundation for reasoning about programs' behavior. Once appropriate inductive invariants are identified, the focus of program verification shifts to checking the validity of verification conditions

derived from finite loop-free paths extracted based on the invariants. This reduction in complexity is a key benefit of leveraging invariants in the verification process.

ML-based invariant generation excels in uncovering the implicit and intricate logic embedded within programs and target properties, with a moderate level of generalization to handle similar constraints and program structures. ICE-learning [103] (which stands for learning using implications, counter-examples, and examples) consists of two components in its framework: a teacher, which is a program verifier fully aware of the program and the property to be verified, and a learner, which is entirely agnostic of the program and property, aiming to generate invariants through interactions with the teacher. During each iteration, the teacher evaluates whether the current invariant is sufficient to prove a certain property and provides (counter-)examples accordingly; the learner refines the constructed invariant based on samples received from the teacher. In this abstraction, the learner can leverage any ML algorithm effective in generating Boolean functions. One example is to tailor a decision tree by developing new measures to determine the best attributes to branch current examples and implications [104]; the well-trained model will output a universal representation of Boolean functions composed of atomic formulas that are either inequalities bounding the numerical attributes by constants or Boolean attributes. To enhance the generalization capability across structurally similar programs, Code2Inv [228] utilizes a GNN to acquire neural representations of program structures. It frames the loop invariant generation as a deep RL problem: the learning agent, structured as an LSTM, accepts neural program representations as inputs and employs the policy gradient algorithm to execute a sequence of actions for generating a loop invariant. Remarkably, as invariants in hardware designs can be learned through the use of examples and counter-examples [236, 275] and hardware designs are naturally represented in graph formats, techniques developed in ICE-learning [103] (e.g., leveraging examples of properties) and Code2Inv [228] (e.g., generating invariants via neural program representations) can be applied in the hardware domain with insignificant modifications.

The decoupling of a comprehensive program verifier and an agnostic learner for invariant generation brings more flexibility, allowing the learner to apply various ML algorithms. Since this framework exhibits similarities to the general RL formulation, it can be seamlessly integrated with different RL algorithms as well as imitation learning to accelerate the learning process. Additional efforts are expected on multiple fronts, including designing new metrics for assessing the quality of generated invariants to effectively guide the ML-based learner, improving the capability of the teacher (i.e., the environment in the RL setting) to offer more representative (counter-)examples, and developing strategies for generalization, either across various invariant generation methods or diverse programs.

2.7 Cost Estimation

Estimating the complexity of formal methods in advance provides the possibility to applying more appropriate verification methods for target hardware designs and better formal results with the minimum efforts. The cost estimation and planning of formal verification can be conducted manually by systematically analyzing attributes of DUTs, property specifications, design constraints/assumptions, which requires sufficient domain expertise and long analysis time. By contrast, ML-based fast and accurate modeling improves the efficiency of automatic cost estimation.

Elmandouh and Wassal [84] employ multiple linear regression models, such as ridge regression and lasso regression, to predict the total CPU time of formal verification runs, based on features extracted from DUT attributes, design properties, formal netlists, and formal run initialization. Later, they give more attention on formal engine orchestration, the methodology to select the most appropriate formal engine for a specific verification problem [85]. The proposed framework includes a group of multi-class classifiers, each of which is trained to predict the capabilities of

one formal engine/algorithm to solve a verification problem in the form of categorical labels (i.e., easy, medium, hard, very hard, or not able to verify). These classifiers take in features distilled from RTL designs (written in VHDL, Verilog or SystemVerilog) and design properties (described in PSL, OVL, or SVA); several ML models are explored, among which the decision tree performs the best, reaching up to 59% of the maximum achievable time improvement. Twigg et al. [247] employ a multi-layer perceptron (MLP) to predict the time and memory usage of Synopsys Formality, a tool used for equivalence checking [241], by using features extracted from Verilog designs.

While cost estimation of formal verification is important, as a majority of failures stem from exceeding memory limits or time constraints, the process of fully automating this estimation still has a considerable journey ahead. First, data scarcity poses a significant obstacle. Each run of verification can consume a substantial amount of time, making it difficult to collect enough data to well train an ML model. Second, different formal methods may require different sets of features, and their behavior can vary largely on different platforms, presenting difficulties in developing generalizable ML models across diverse verification scenarios. There await innovative solutions and further advancements in data collection strategies, feature engineering techniques, and the development of robust ML algorithms tailored to the intricacies of formal verification processes.

2.8 Roadmap of ML in Formal Verification

To leverage current ML practices to improve and enhance formal verification tools, we often follow the pipeline for efficient integration: ❶ identify the formal verification problem, e.g., SAT solver, theorem proving, etc., and formulate it into an ML task, e.g., classification, regression, etc.; ❷ collect information or data from formal verification for the ML task, e.g., converting CNFs or Boolean networks to graphs for GNNs [15, 163, 221, 266, 272], extracting features of variables [108, 118] for NLP models; ❸ figure out the best ML model, as well as parameters, for the formal verification problem.

However, formal verification and ML have opposite mathematical foundations and opposite use in real-life problems: formal verification usually relies on determinate mathematics and aims at ensuring correctness; ML often relies on probabilistic models and consists of learning patterns from training data. As a result, ML techniques cannot “replace" the conventional formal verification techniques or adequately “automatically solve" formal verification problems at present. Instead, ML techniques can "help" or "guide" the formal verification solving process. ML techniques have shown their great potential in improving/accelerating the formal verification process compared to human design. Specifically, ML techniques improve the runtime/efficiency of formal verification by providing a shortcut from data (input variables) to answer (if valuable in formal verification) to help formal verification experts to filter meaningless cases. Additionally, ML techniques can provide higher quality than manual designs by learning from the training data, which is confirmed especially in assertion generation tasks (Section 2.2).

While there have been exciting progresses, existing approaches in leveraging ML for formal verification have critical challenges to address. ❶ **Functional-aware ML.** For example, there are great needs in enforcing strong logic relations in ML features and the training process. Unfortunately, ML approaches in fundamental formal verification solving problems such as SAT solving only include soft logical constraints in the training and embedding, which greatly limits the generalizability in scaling to practical scaled SAT problems. Exploring hard logical constraints in developing domain-specific ML approaches for such problems is believed to generate broad impacts, e.g., developing computer algebra-based gradient descent mechanisms for training. ❷ **Heuristics mining and orchestration.** In addition to directly using ML to “solve” formal verification problems, there are substantial opportunities in leveraging the strength of decision-making to boost the solving capabilities of formal verification solvers and heuristics. For example, in SAT

solving, the major progress of runtime improvements for modern SAT solvers comes from the introduction of novel heuristics in the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [1]. There is an increasing number of such heuristics on either cause/variable elimination or DPLL solving, each of which maintains its own strength in different SAT instances. Similar cases happen to other mathematical solving procedures such as solving Gröbner Basis [58, 59, 169, 172, 212] for arithmetic circuits verification problems. Leveraging the power of adaptive decision making in orchestrating existing heuristics and mining novel heuristics in formal solvers will be of great interest. ❸ **Parallelism in formal verification.** One of the greatest challenges in practical formal verification applications is the runtime cost. While most of the formal verification instances are NP-hard in nature, exploring high-performance computing capabilities offered by modern DL frameworks could significantly improve the scalability. Recently, we have seen lightweight neural network models or convex/non-convex formulation based approaches that solve combinatorial optimization problems with extensive parallelization on heterogeneous platforms. While formal verification solvers start leveraging GPU acceleration [73, 197], it is believed that exploring the tensor-level computation kernels of high-performance DL frameworks can further improve runtime performance of formal verification solvers. ❹ **Integratability.** While ML approaches are demonstrated to be effective in improving the verification process in various aspects, landing such approaches in end-to-end system integration is a non-trivial task. For example, the majority of the formal verification solvers and industrial verification toolchains are built with high-performance implementations in C/C++ and are mostly optimized for CPU computations. However, ML frameworks mostly involve multiple-level intermediate representations and require a GPU environment to efficiently perform the computations. Moreover, due to the nature of randomness in ML algorithms, lacking determinism could be a concern to land ML-based verification approaches in real-world products. One interesting and challenging direction will be exploring ML deployment in terms of system engineering challenges, e.g., considering the efficiency of heterogeneous computing platforms, programmability, etc.

Therefore, we can say employing ML in formal verification is promising but needs more efforts. We believe ML for formal verification can be further improved when more data is collected and more ML models are developed. ML for formal verification can get rid of human designs and abstractions, and may find a new aspect for problem-solving in formal verification, providing new understandings/interpretations in the formal verification domain.

3 ORACLE-BASED VERIFICATION

Oracle-based verification focuses on generating tests and achieving sufficient coverage [40]. The primary goal is to reveal failures by executing tests and comparing output results with the oracle provided by a golden reference model. Since the design under test (DUT) is exercised by input tests or stimulus, oracle-based verification is also acknowledged as simulation-based or dynamic verification. In general, simulation-based verification consists of three aspects: coverage measurement, test/stimulus generation, and response checking [205]. Specifically, coverage goals define the scope of a verification problem, and coverage measurement monitors the verification progress with respect to different types of coverage, such as code and functional coverage; test/stimulus generation intends to fully exercise a DUT, i.e., taking a DUT into all the possible behaviors, by constructing tests/stimulus required by the coverage goals; response checking is responsible for demonstrating how behaviors of the DUT conform with its specifications or the golden reference, which triggers the troubleshooting once the discrepancy is detected.

As mentioned by the Moore's law [171, 182], the principle powering the integrated-circuit revolution since the 1960s, the transistor density doubles every 18 months. Even if we are approaching the end of the Moore's law, the electronic industry continues to move to larger, more complex,

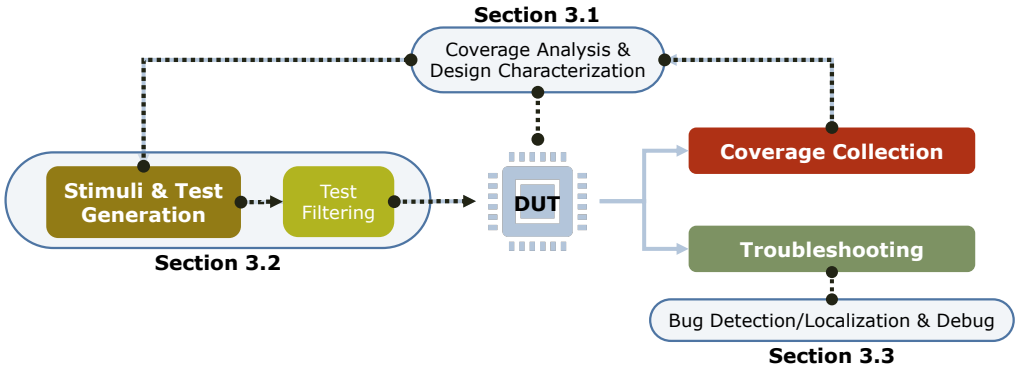

Fig. 4. An overview of the oracle-based functional verification. We identify the major ML-based components that could be inserted into the verification flow and how they change the classical verification flow.

and heterogeneous designs. Thus, simulation-based verification, which has been the major yet time-expensive technique for hardware verification, is encountering new challenges. The increasing design size is merely one aspect of the scalability challenge. It is the movement towards system-on-a-chip (SoC) integration and heterogeneous systems that drives simulation-based verification into a corner: nearly 70% of ASIC or FPGA designs contain multiple embedded processors, and around one-fifth to one-fourth of the designs incorporate RISC-V processors or AI processors/accelerators [93]. First, many of the techniques that work well for IP-block-level or sub-system-level verification (e.g., constrained-random and functional coverage) do not scale well to the entire SoC integration or system-level validation [93, 96]. Second, along with the increasing design complexity, simulation-based verification is producing so many data that it has become a big data problem [261, 262], begging for a new level of intelligence in verification, including but not limited to smarter test generation, coverage collection/analysis, and debug.

Reckoning on the growth in design and SoC complexity, system heterogeneity, and the amount of verification data, any endeavor enhancing the aforementioned three principal verification aspects (i.e., coverage measurement, test generation, and troubleshooting), either independently or integrally, will greatly promote the productivity and efficacy of simulation-based verification and hardware development. Figure 5 illustrates the breakdown of the time that verification engineers spend on different verification steps, also indicating that debug, test generation, and running simulation dominate the verification time. These all boost opportunities and prospects for incorporating ML-based techniques in verification.

In oracle-based verification, ML-based techniques usually serve as three roles. ❶ **The predictive models for fast and accurate coverage predictions** (Section 3.1). ML has been widely adopted for predictive models [86], since many supervised learning algorithms can infer a general law from observations of particular instances. Such a law is generalizable from labeled examples to new instances under certain constraints. These ML-based predictive models, typically mining the relationships among design signals or from tests to coverage metrics, support eliminating the time-consuming simulation time of DUTs. ❷ **The optimizer for test generation or selection** (Section 3.2). It has always been challenging to find the most stressful and comprehensive tests to stimulate DUTs, so that the coverage closure can be reached within time limits. Several ML-based optimization techniques, such as gradient-based methods [34, 235] or maximum a posteriori estimations [80], have the potentials to advance test generation. ❸ **The troubleshooting assistant** (Section 3.3). Debug is the bottleneck in verification. Clustering and classification algorithms [181]

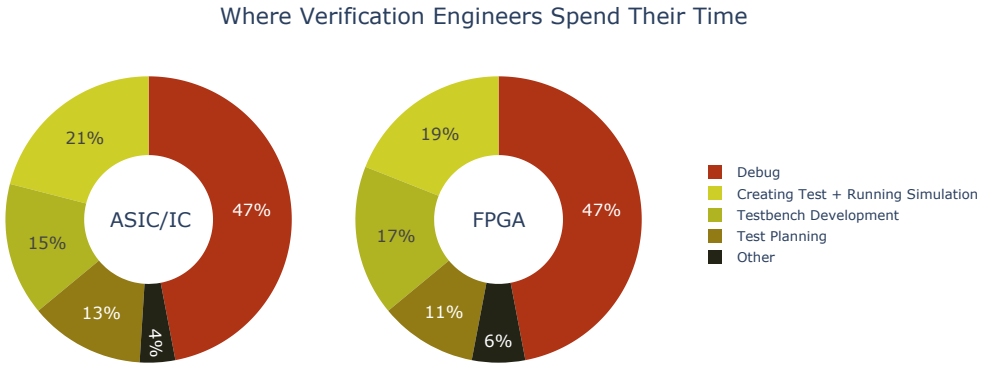

Fig. 5. Breakdown of the average time spent on different verification steps by verification engineers in 2022, which is redrawn based on the data from the 2022 Wilson Research Group Functional Verification Study [94].

are flourishing in reasoning out root causes of failures and pinpointing the suspicious design signals/areas.

3.1 Coverage Prediction and Design Characterization

A satisfactory coverage level is necessary for verification sign-off. Commonly used coverage metrics include fault coverage, code coverage, and functional coverage [244]: fault coverage indicates the ability to handle faults in a DUT; code coverage measures the execution status of the actual code, which can be recognized as a quantitative measure; functional coverage attempts to measure whether the functionalities described in specifications and the implementation of the DUT have been adequately exercised, which can be considered as a qualitative measure.

Many studies adopt ML models to accelerate the verification cycle by fast and accurate coverage predictions, instead of resorting to the time-consuming simulation process. These learned ML models, which are surrogate models approximating the relation between tests or test templates and coverage metrics, can be either merely employed as predictive models, combined with conventional heuristic-based search algorithms to speed up the evaluation phase of test generation, or interpreted for rule learning of testing knowledge. We review the studies regarding ❶ **how ML-based surrogate models are embraced in fault** [157, 170, 238], **code** [99, 112, 146, 233], **and functional coverage analysis** [2, 82, 112, 161] in Section 3.1.1, and ❷ **ML-based design characterization** [77, 139, 215] **and rule learning** [53, 132, 133, 153] in Section 3.1.2.

3.1.1 Coverage Prediction. Fault coverage often refers to the percentage of detected faults in all potential faults. Test point insertion (TPI) is a commonly adopted approach to improving fault coverage, which modifies DUTs by inserting extra control points or observation points, but it may degrade the performance of a design in terms of area, power, and timing. The optimal TPI aims to achieve high fault coverage with as little performance degradation as possible, which is NP-complete [156]. Thus, ML techniques are employed to enhance the TPI process by providing accurate predictions on the testability. Given circuit netlists and node features (e.g., controllability, observability, logic depth from the furthest primary input), the impact on fault coverage after inserting a candidate test point can be accurately estimated by an ANN [238]. To better characterize the graph structure of circuits, Ma et al. [170] use graph convolutional networks (GCNs) to predict whether a node is easy or hard to observe, based on which new observation points are inserted.

Table 2. Coverage prediction and design characterization with ML models. S and U stand for supervised and unsupervised learning, respectively.

Problem	FWK	ML Model	Target HW
Testability prediction	S	ANN [238]	Industrial circuits
	S	GCN [170]	Four industrial designs
	S	SiFS [157]	Combinational circuits
Code coverage prediction	S	Linear regression, decision tree, random forest [146]	An IP module
Toggle pair coverage analysis	U	K-means clustering [233]	Not mentioned
FSM coverage prediction	S	Random forest [112]	Quad-core cache (four private L1 caches and a shared L2 cache) [112]
	S	MLP, random forest [99]	
Functional coverage analysis or prediction	U	K-means clustering [82]	Industrial designs [82]
	S	LSTM, random forest, SVM, DNN [112]	Quad-core cache [112]
	S	Linear regression, SVR/C, KNN [161]	MESI cache coherence intersection controller [78]
	S	ANN, DNN, SVR, decision tree [2]	CORDIC core [79]
	S	MLP [76]	GPU units [76]
Cross-design coverage prediction	U + S	K-means clustering and DNNs [215]	Nvidia designs [215]
Design characterization	U	K-means clustering [139]	Queue management block and serial line interface [139]
	S	Random forest [77]	UART custom protocol transceiver [111]
Rule learning	U	ILP [133]	Five-stage pipelined superscalar DLX processor [201]
	S	Decision tree [153]	IBM Power7 Processor [152]
	U + S	CN2-SD [53]	Load-store unit (LSU) in a processor
	S	CART [132]	Control unit in a commercial dual-core SoC

Compared with classical ML models, such as logistic regression, random forest, SVM, and ANN, the classification accuracy is improved by at least 7.5%; compared with commercial testability analysis tools, this GCN-aided TPI flow achieved similar fault coverage with an 11% reduction in observation points and a 6% reduction in test pattern count. To better explain decisions made by the ML models, Krishnamurthy et al. [157] introduce an intrinsically interpretable and rule-based ML technique, namely Sentences in Feature Subsets (SiFS) [158], to classify whether a point is hard to test, reaching up to 95% accuracy with extra bonus of model interpretability.

Code coverage often refers to the percentage of the source code covered by tests, and typical code coverage analysis includes line, branch, condition, toggle, and finite state machine (FSM) coverage. Aiming to save manual efforts spent for general code coverage analysis, several ML techniques (i.e., linear regression, decision tree, and random forest) can be applied to predict per-test coverage percentage and simulation runtime based on input test parameters [146]. (1) Toggle coverage often refers to describing design activity in terms of changes in signal values, which monitors value changes on registers and nets and measures whether the value has changed from either 1 to 0 or 0 to 1. Stan Sokorac [233] introduces a new coverage metric, toggle pair coverage, which extracts all possible pairs of signals and measures their toggles, reaching a sweet point between the vanilla toggle coverage and the full state space regarding the number of coverage points; then K-means clustering is leveraged to group tests that cover similar areas of toggle pair coverage; after the clustering, genetic algorithm is applied individually within each group to generate tests that would exercise the rarely reached parts of DUTs as frequently as possible. (2) FSM coverage typically refers to how many states and transitions in an FSM have been visited during simulation. To recognize the correlation between instructions and FSM states, ML models, such as random forest [112] and MLP [99], can take the current state of the DUT and instructions to be executed as inputs to predict the next state. These learned models either help construct an augmented FSM upon which the shortest path algorithm (e.g., the Dijkstra [9] algorithm) is applied to direct generate sequences of

instructions as tests, or examine state transitions of current tests to prune out the ones unhelpful to improve transition coverage.

Functional coverage often refers to how many required design functionalities have been examined. These functionalities are converted to coverage points awaiting to be activated by the tests. Since different coverage points may share similarities, clustering algorithms are naturally suitable to group similar coverage points to provide guidance for new test generation. Mandouh et al. [82] take a two-step K-means clustering to prioritize coverage points with low coverage: the first step groups coverage points based on their functional similarities, and the second step assembles the identified clusters based on their coverage ratios. In terms of predicting functional coverage based on input tests, from the micro-architectural level, Gorgi et al. [112] notice that instruction-based predictions provide finer-grained feedback compared with test-template-based predictions, offering better guidance on generating test patterns towards coverage holes: specifically, the instruction-based predictor (LSTM or random forest) takes a sequence of instructions to forecast whether the interested events happen; the test-template-based predictor (DNN, random forest, or SVM) classifies whether an interested event is covered, uncovered, or uncertain. From the RTL, targeting a multi-processor cache controller, Kulkarni [161] studies various types of linear regression, SVR, SVC, and K-nearest neighbors to predict the FIFO queue depth based on test constraints; targeting arithmetic units, a case study [2] of the CORDIC core [79] performs comparative analysis among multiple ML models (MLP, DNN, SVR, and decision trees) and indicates that decision trees have the highest accuracy; targeting channel/datapath stalling to uncover corner-case timing critical bugs in GPU units, Dhodhi *et al.* [76] use an MLP to model the relation between stall parameters and the coverage metric, which is integrated with GA to efficiently tune the parameters for better coverage.

Among these ML-based coverage predictions, the more complex the hardware designs are, the larger ML models are employed. Though simple ML models are easy to train with moderate amount of data, their representation power is limited due to the number of parameters in the models. Currently, most of the discussed studies focus on single functional units instead of entire designs, and thus they tend to adopt conventional ML models. One issue that may arise is these ML models usually rely heavily on design-specific features, which is nearly impossible to migrate well-trained models to new hardware designs. Recent efforts tend to make analogy between ML models and target problems, such as using LSTM to handle sequences of instructions [112] and exploiting GNNs to represent circuit designs [170], so that domain-specific knowledge or contextual information can be integrated into ML models to enable better transferability.

3.1.2 Design Characterization and Model Interpretation. Several studies put more emphasis on design characterization with ML models. Ikram and Ellis [139] use K-means clustering to group modules based on code coverage metrics, and once there are changes in the DUT, the clustering information helps to find all the probably affected modules, providing guidance to test suite generation to cover the influence of the new changes. To reduce the mammoth simulation overhead of coverage collection, Roy et al. [215] use K-means clustering to select representative modules from the DUT, whose coverage metrics are collected and used to train a DNN that can predict the module-level coverage metrics (i.e. the reachability conditions in a module by a test during simulation) for the rest of modules given the same set of tests; experiment results indicate that collecting coverage for 3% of the modules enables predictions with less than 5% error averaged across various modules. Dinu et al. [77] use ML models to analyze the correlations among DUT signals, where a well-trained model can be recognized as a reference model to the DUT whose predictions help reduce the running time of DUT simulation; specifically, a wide range of ML techniques are evaluated to predict whether the data transmission by an UART [111] custom protocol transceiver is erroneous, among which the random forest shows the superior performance.

Instead of simply constructing surrogate models that associate test cases and their simulation traces with certain events, a series of studies focuses on rule learning from test data and interpreting the learned surrogate models. The learned rules explain the reasons why the target coverage points are hit by the tests, and thus can help update the constraints in new test templates to activate desired coverage points. From the micro-architectural level, Hsueh et al. [133] adopt inductive logic programming (ILP) [164] to discover the relationships between input test sequences and their related coverage using the first-order logic representation. Katz et al. [153] build a decision tree to approximate the relations from instructions and processor states (i.e., tests and simulation traces) to micro-architectural behaviors (e.g., more or fewer internal operations, long or short execution time in clock cycles). The rules derived from the learned decision tree are then embedded in the stimuli generator to control the micro-architectural behaviors of newly generated instructions so that interesting events in the design can be reached. Taking a similar set of input features, Chen et al. [53] apply the CN2-SD [165] algorithm that stands in the middle of supervised and unsupervised learning to find rules that can guide the generated tests to hit certain components of an assertion. They pay attention to the load store unit in a dual-threaded low-power 64-bit power-architecture-based processor core, and compared with the best test templates refined by human experts, the test template refined by the learned rules can effectively generate new tests that activate an assertion either with low coverage or not yet covered before. From the RTL, Hsieh et al. [132] leverage text mining to automatically select proper RTL signals from design documents, upon which the classification and regression trees (CARTs) are used to learn the rules that can hit target assertion coverage points with respect to the selected RTL signals.

One criticism of using ML as surrogate models is the lack of transparency and interpretability, since verification aims to ensure guaranteed correctness or understand the reasons for potential incorrectness. Rule learning is a great attempt to improve the explainability but may suffer from scalability issue when handling large designs. Thus, more advanced explainable ML techniques are highly expected in production-ready ML for verification. In addition to explainable ML, another possible solution is integrating uncertainty quantification [3] to provide the confidence of different predictions, which can also calibrate ML models during runtime.

3.2 Stimuli and Test Generation

Stimuli and test generation aim to produce tests that are (1) effective in terms of meeting constraints specified by users, DUTs, or verification environments and (2) random but expected to trigger particular or hard-to-hit events. There are different forms of tests or stimuli, such as bit vectors applied to input ports of DUTs or assembly programs (i.e., sequences of instructions) directly loaded into the program memory. Conventionally, this generation process is often guided by manually devised test templates or directives based on the testing knowledge from experts, which is labor-intensive, error-prone, and hard to maintain, especially when DUTs are getting increasingly complicated and large-scale. Striving to release manual efforts and improve the quality of generated tests, the exploitation of ML provides the possibility of automatic testing knowledge extraction and efficient test generation.

ML-based techniques typically involve two roles in the stimuli and test generation [262]. ❶ The first role **leverages the collected coverage metrics to enable a smarter and more powerful search for newly generated tests**. This involves applying some ML-based optimization approaches to generating better sets of tests or directly imitating the test generation flow using ML models, which is discussed in the Section 3.2.1. ❷ The second role intends to **improve simulation efficiency by filtering out unimportant or redundant tests**, which is discussed in the Section 3.2.2. Even though there exist reviews regarding data mining in test generation [261] and ML-based

Table 3. ML as the optimizer for test generation. S, U, and R stand for supervised, unsupervised, and reinforcement learning, respectively.

Problem	FWK	ML Model	Target HW
ML as surrogate models, with MAP estimations for test directive generation	S	Dynamic Bayesian network [90]	NorthStar processor [33], the storage control element (SCE) of an IBM zSeries processor [90]
	S	Bayesian network [35]	SCE of an IBM zSeries processor [35]
	S	Bayesian network [20]	Instruction fetch unit (IFU) of IBM z10 processor [138]
	S	Bayesian network [89]	PowerPC processor [178]
ML as surrogate models, with gradient-based test generation	S	Deep residual NN [100, 101]	NorthStar processor [33]
	S	GNN [253]	TPU [149], IBEX v1 and v2 [208]
Reverse engineering of DUTs to directly generate tests	S	Linear regression and ANN [14]	Variable width comparator [14]
	S	ANN [252]	Variable width comparator [252]
	S	Linear regression and ANN [4]	Variable width comparator and multiplier [4]
	S	ANN [71, 72]	32-bit address bus [71, 72]
	S	MLP, DNN, SVR, and decision tree [144]	Variable width squarer [144]
	R	DQN [137]	Cache in the RISCV-Ariane [117]
	R	DQN [190]	LZW compression encoder [190]
	R	Soft actor-critic [225]	Run length encoding compressor and AXI controller [225]
	R	Policy gradient [54]	Peripheral circuits of DRAM [54]
	R	Transformer + actor-critic [259]	Memory management units [259]
Imitating instruction generation and execution flow	S	Markov model [257]	DLX [201] and Alpha [245] processors
	S	Hopfield network [88]	Codasip processors [168]

test generation prior to 2012 [142], we mostly concentrate on more recent studies and limit our scope to discuss how ML-based techniques enhance test generation.

3.2.1 Constrained-Random and Coverage-Directed Test Generation. Stimuli and test generators usually have built-in constraint solvers [234], which take test templates or directives to constrain the randomly generated tests or stimulus to better target particular events or coverage points, referred to as constrained-random test generation. The goal is to find proper sets of constraints that increase the probability of hitting desired events and produce sufficiently stressful tests with a wide variety. Given a large number of tests, the corresponding coverage metrics are collected as feedback to further improve test templates/constraints or directly produce tests with a higher likelihood of activating interested scenarios, referred to as coverage-directed test generation [25].

Aiming to accelerate coverage closure, which is the major quality criterion of the functional verification process, constrained-random and coverage-directed test generation are often seamlessly integrated. However, even if all the required coverage metrics are reached, critical corner cases may still be under-exercised whereas straightforward scenarios are over-activated. Thus, the primary objectives of efficacious test generation include not only hitting uncovered events but also improving the coverage rate of events that are not exercised enough. In response to the requirement of comprehensively examining every target coverage event, ML-based techniques are promising as ❶ **more powerful search approaches to better test templates or tests** [20, 35, 89, 90, 100, 101, 253] and ❷ **smarter test generation methods by reverse engineering DUTs** [4, 14, 71, 72, 137, 144, 225, 252] **or mimicking the instruction generation flow** [88, 257], which is recapitulated in Table 3.

ML as a surrogate model for powerful test search. ML techniques can be applied to learn surrogate models that approximate the relationships between tests and coverage metrics of interest, followed by maximum a posteriori (MAP) estimations or gradient-based search algorithms to generate better tests to activate hard-to-hit or uncovered cover points. ❶ **Bayesian networks as surrogate models with MAP estimations for test generation.** A collection of studies uses Bayesian networks to represent the complex relationships among test directives (i.e., sets of

constraints) and coverage points [20, 35, 89, 90], offering an efficient modeling scheme as well as the capability of encoding essential domain knowledge. In a Bayesian network, input nodes correspond to test directives; output nodes relate to coverage points; hidden nodes represent expert domain knowledge capturing implicit correlations between input/output nodes and help reduce compute complexity by dimensionality reduction; each edge represents the conditional probability for the sink node given the source node. The construction of Bayesian networks involves three stages: (1) sampling data from simulation results and feature extraction of relevant directives, (2) structure learning that defines the structure of the Bayesian network by domain knowledge from the DUT and the simulation environment, and (3) parameter learning that finds the conditional probability distribution functions within the network. After the Bayesian network is well-trained, a statistical inference of test directives can be made by either selecting the MAP estimations or obtaining the most probable explanation (MPE), given the requested coverage tasks. An initial attempt [90] employs a two-slice dynamic Bayesian network [106] to describe the relationships between the directives and the coverage points. Several enhancements are brought forward regarding the training efficiency [35], feature selection and structure learning [20] of Bayesian networks: Braun et al. [35] smartly sample the simulation data related to low log-likelihood coverage events for better training efficiency; Baras et al. [20] improve the feature selection (which finds directives with high impacts on the coverage points) and the structure learning by greedy algorithms and genetic algorithms, respectively. Fine et al. [89] employ Bayesian networks to model the effects of the DUT's initial states on test generation success. Specifically, they learn the relationship between the machine state register (MSR) and exceptions, which reduces the number of exceptions in generated tests and accelerates coverage closure. ❷ **Neural network variants as surrogate models with gradient-based search for test generation.** Gal et al. [100, 101] adopt deep residual neural networks to capture the relations between test templates and the expected probabilities of achieving coverage events with these templates. The trained DNN model proposes new test templates by maximizing the hitting probability of hard-to-hit events through gradient-based methods; if the new test template does not help with reducing hard-to-hit events, heuristic-based optimization methods (i.e., simulated annealing, genetic algorithm, particle swarm optimization, and implicit filtering) are applied to explore new test templates. To better utilize the graph structure inside RTL designs, Vasudevan et al. [253] exploit IPA-GNN [27] to characterize RTL semantics and computation flows, which predicts the probability of covering a cover point (specifically, in branch coverage) with current test parameters; new tests targeting uncovered points are generated by maximizing the predicted probability of covering desired points with respect to test parameters through gradient-based search. This GNN-based test generator is complementary to random generators to guarantee high coverage.

Undoubtedly, leveraging ML-based surrogate models can accelerate test generation compared to conventional approaches, as they are capable of exploring a wider design space and generating tests covering rare or complex scenarios due to their stochastic nature. One key distinction between using Bayesian networks and neural network variants lies in the assumption of whether the learned surrogate models should be differentiable or not. The adoption of Bayesian networks releases the requirement of being differentiable but often requires more domain expertise in developing models; while neural network variants offer more automatic model training, they are more sensitive to biases in the training data and the assumption of differentiability may not always hold true. Another notable issue is that trivial coverage points are much easier to collect than hard-to-hit ones, which may cause trained ML models to overfit to regular coverage points. Possible solutions include employing efficient sampling strategies and data augmentation methods to improve the likelihood of hitting hard-to-cover events.

Reverse engineering based test generation. To directly generate proper input stimuli that would activate the desired coverage points, several studies utilize ML models to reverse engineer the inputs and outputs of DUTs, and update constraints according to the predicted test stimuli. ❶ **Supervised learning approaches.** Linear regression and MLPs can take the output signals of DUTs as inputs to predict the test stimuli that could generate the desired outputs, which currently work well for small circuit designs, such as comparators [4, 14, 252], multipliers [4], and a 32-bit address bus [71, 72]. Ismail et al. [144] conduct a comparison among multiple ML modes (MLP, DNN, SVR, and decision trees) targeting a variable width squarer, in which decision trees perform the best in terms of fast training time and prediction accuracy. With the help of these ML models, the number of simulation cycles is significantly eliminated since unnecessary tests either activating previously hit coverage points or out-of-interested groups are weeded out. ❷ **RL approaches.** By observing current simulation results of DUTs, RL-based approaches [54, 137, 190, 225] are capable of proactively exploring test parameters that guide test stimuli generation towards desired coverage points. Hughes et al. [137] take advantage of the deep Q-network (DQN) [239] to explore input knobs such as test parameters and DUT configurations. Specifically, they target cache designs: the state is the current simulation results; the action space involves all possible combinations of input knobs; the optimization goal is to maximize the FIFO occupancy or the victim buffer occupancy so that the generated test parameters can extensively stress DUTs and hit corner cases. Ohana [190] also exploits DQN to generate tests for a Lempel–Ziv–Welch (LZW) compression encoder: the state is a representation of the last 17 input symbols received by the encoder; the action drives a specific input symbol; the reward function returns high values when more input symbols are written or matched in the content-addressable memory (CAM) of the LZW encoder, aiming to accelerate the CAM write functional coverage closure. With the goal of facilitating RL applied for test generation, VeRLPy [225] is an open-source library, which employs the soft actor-critic algorithm [124] to figure out test parameters by maximizing accumulated rewards that represent the coverage of interested functional events. One limitation is that VeRLPy formulates test parameter exploration as a one-step single-state MDP, and a more general multi-step MDP formulation awaits future efforts. Choi et al. [54] focus on peripheral circuit blocks of DRAM and apply the policy gradient method to directly generate input stimuli (i.e., sequences of legal DRAM commands as test vectors). The state is a sequence of past 100 actions; the action space includes all possible DRAM commands; the reward relates to the coverage score that measures state transition coverage of the target circuitry. The generated test vectors are then curtailed by Bayesian optimization to remove redundant actions without compromising the coverage score, whose length is only 7% compared to human-crafted vectors. Wang et al. [259] deal with the challenging corner case verification of FIFO full conditions in memory management units (MMUs). They devise a two-stage framework: the first step adopts the Transformer model [255] to identify appropriate constrained sub-ranges that can trigger higher numbers of PUSH counts, and the second step uses the actor-critic method to select a sequence of constraints based on the sub-range set provided in the first step. This approach increases the hit rate of the corner case by up to 380× compared to traditional constrained random generation.

These reverse-engineering-based techniques offer a relatively straightforward approach to generating input tests. However, they are often tailored to specific hardware designs by design-specific features and signals. This implies that a new ML model must be trained for each new design, which can outweigh the benefits derived from using ML.

ML models imitating instruction flows. A series of studies use ML models to imitate instruction generation and execution flows for micro-processor verification, where test constraints are inherently embedded in the learned ML models to guide desired test generation. One early attempt constructs a Markov model as a proxy for the FSM of a micro-processor [257], where each node represents one type of instructions and each edge indicates the transition probability

Table 4. ML techniques for test selection. S stands for supervised learning.

Problem	FWK	ML Model	Target HW
Detecting novel tests	S	One-class SVM [123]	Execution unit in OpenSPARC T1 [192]
	S	One-class SVM [120]	Godson-2 [136]
	S	One-class SVM [49]	Plasma/MIPS CPU core [206]
	S	One-class SVM [52]	Dual-thread low-power power architecture-based processor core [52]
Classifying tests for selection	S	Two-class SVM [214]	StreamProc module of a Bluetooth adapter [32]
	S	MLP [260]	Dual-core RISC processor [260]
	S	Ensemble model [200]	Four IP designs and one open-source design (i.e. Ibex [216])
Selecting valid test vectors	S	MLP [105]	Small combinational circuits represented in NAND gates [105]

between two nodes. The tests, represented as sequences of instructions or assembly programs, correspond to paths in the graph-structured Markov model; transition probabilities controlling test program generation are adjusted based on feedback from activity monitors that assess signals in the DUT for particular activities of interest. Fajcik et al. [88] use the Hopfield network [130] to mimic the instruction set architecture (ISA) of Codasip processors [168], where each neuron represents one type of instructions and its appearing probability. The proposed model updates probability constraints for instructions based on feedback from functional coverage metrics and alters the constraints in the stimuli generator accordingly.

Recent efforts explore the application of LLMs for direct test generation. Chip-Chat [31] examines the capabilities of several conversational LLMs to produce Verilog, among which ChatGPT-4 [194] performs the best. By careful prompt engineering, ChatGPT-4 can design an 8-bit accumulator-based processor, but cannot write non-trivial test programs.

3.2.2 Test Selection and Filtering. In addition to improving test generation, test selection is another approach that accelerates the verification process by filtering out redundant or unimportant tests from the testbench. The goal is to figure out representative and important tests from a large pool of generated tests, so that a high coverage can be reached with a significant reduction in the number of simulations. ML-based techniques are majorly employed for ❶ **novel test detection** [49, 52, 120, 123], ❷ **test classification towards interested coverage events** [200, 214, 260], and ❸ **valid test selection** [105], as encapsulated in Table 4.

Striving for estimating the similarity among different tests, many SVM variants are explored to detect novel tests or classify tests based on the interested similarity metrics. ❶ One-class SVM, which uses one hypersphere to encompass data instances in the projected high-dimension space and identifies data points lying outside the hypersphere as outliers, is widely applied to detect novel tests. Guzey et al. [123] incorporate domain knowledge into the kernel function that projects input test parameters into the similarity metric space, and employ a one-class SVM to search for outliers as the selected tests. To update the one-class SVM dynamically based on runtime coverage reports, an online learning approach is proposed to modify the trained model with new tests while retaining memory of historical tests [120]. Moreover, there are some innovations in kernel functions of the one-class SVM: Chang et al. [49] develop a graph-based kernel to measure the similarity of a pair of assembly programs, which converts assembly programs into directed graphs and measures the similarity based on the graph edit distance between a pair of graphs; Chen et al. [52] employ a coverage-based kernel, which measures the similarity of assembly programs by estimating coverage results before simulation, and dynamically update the training set by accumulating newly simulated tests so that the novelty of a test is always evaluated based on its probability to activate the currently uncovered points. ❷ Two-class SVM, which is also referred to as the conventional SVM that separates data points in the high-dimension space by a hyperplane,

is typically used to classify tests based on their similarity. Romero et al. [214] employ multiple two-class SVM classifiers to compose a multi-class classifier. In this approach, each test is labeled with the most activated coverage event. Test selection is achieved by selecting the tests associated with the desired class (i.e., the coverage event of interest) and discarding the rest.

More recent studies apply MLPs for test selection, leveraging their compatibility with fixed-size test vectors for RTL verification. Wang et al. [260] group assertions according to their functional and structural dependencies, and then train an MLP to classify input test vectors based on their potential target assertion groups. Test vectors that trigger assertion groups with high priorities are selected and passed to the DUT, while irrelevant test vectors are pruned. Gaur et al. [105] eliminate invalid input test vectors that either violate design constraints or may cause unexpected output signals; they employ a six-layer MLP to predict the transition probability (i.e., the probability of a node switching from 0 to 1) of outputs given input signal transition probabilities, which measures the output randomization to help identify whether an input test is randomizable; subsequently, these random input test vectors are filtered to prevent randomization-induced failures. Parthasarathy et al. [200] select a subset of tests based on their predicted failure likelihood given an incremental design modification. They build an ensemble model (including adaptive boosting, gradient boosted machines, logistic regression, Naive Bayes, MLP, and random forest) to analyze information from code changes and history behaviors of tests, which provides a binary classification of whether a test will pass or fail along with its failure likelihood.

In general, test selection tasks typically exhibit a higher tolerance for false positives: it is more acceptable to mispredict a small number of trivial tests as important ones. Although running these false positives may incur some overhead, mistakenly excluding novel tests can cause more additional efforts, since it is usually generating tests targeting either hard-to-hit or not fully exercised events that dominates the test generation process.

3.3 Bug Detection/Localization and Debug

Debug usually takes the majority of time in the functional verification cycle, even for experienced engineers or experts, since it is extremely time-consuming and challenging to detect, localize, and analyze bugs from a sea of data produced from design simulation, as shown in Figure 5. Once mismatches in the execution results between the DUT and the golden reference are detected, verification engineers should precisely identify the root causes of observed failures and swiftly narrow down the critical design signals and areas that warrant further examination. However, this process is non-trivial: given test failures, it is difficult to distinguish whether they stem from the same root cause or multiple different issues, since a single bug in the DUT or verification environment can result in numerous test failures and different bugs can also lead to similar failures; in cases involving multiple root causes, there is a strong demand of grouping failures with the common root cause and pinpointing the exact root problem. To promote debug efficiency, many endeavors exploiting ML-based techniques have been made to automate different steps in debug, such as **bug detection with version control systems** [115, 119], **root cause recognition with failure information** [207, 246], **simulation trace analysis** [83, 177], **bug localization with version control systems** [174] **or bug signatures** [176, 209, 258], and **generating repairs for bugs** [7], as summarized in Table 5.

Version control system are a category of software tools designed to track modifications made to code as projects are being developed. Subversion (SVN) [16] and Git [110] are two widely used version control systems. Besides recording code changes, version control systems also contain basic information about projects, such as source code history, commit logs, and engineer assignments. Making use of the features extracted from version control systems (e.g., author information, code complexity, and commits information), Guo et al. [119] predict whether a committed design has

Table 5. Bug detection or localization with ML techniques. FWK, S, and U stand for framework, supervised learning, and unsupervised learning, respectively.

Problem	FWK	ML Model	Target HW
Predicting whether a commit has bugs using version control systems	S S	Naive Bayes, SVM, MLP, decision tree, random forest [119] XGBoost [115]	Four OpenCores designs [195] ASIC IP [115]
Bug localization using version control systems	U + S	Affinity propagation clustering and SVM [174]	Eight OpenCores designs [195] and one in-house real-life industrial design
Grouping test failures based on root causes	U U, S	Hierarchical clustering [207] DBSCAN, random forest [246]	Three OpenCores designs [195] IPs related to image processing
Simulation trace diagnosis	U U	K-means clustering [83] X-means clustering [177]	Five OpenCores designs [195] and UART [111] Ethernet MAC IP core [195]
Bug localization using bug signatures	S S S	Random forest [176] XGBoost, random forest [209] Decision trees [258]	SimpleScalar PISA ISA by FabScalar [56] 32-bit MIPS processor [206] Two-threaded x86 processor [258]
Generating repairs for bugs	S	LLM [7]	CWE [68], OpenTitan SoC [67], and the Hack@DAC 2021 SoC [125]

bugs as well as the number of bugs in the new version using decision trees, SVM, MLP, etc. Likewise, Graber et al. [115] examine several ML models with 51 manually designed features (e.g., developer and debugger information) to predict whether a commit has bugs, among which XGBoost reaches the highest accuracy. To further locate bugs in hardware designs, Maksimovic et al. [174] leverage version control systems for bug localization by a three-step method: first, affinity propagation clustering is used to find possibly erroneous code areas; next, SVM is utilized to analyze commit logs, predicting the likelihood of one commit being buggy; finally, by combining the results from both clustering and SVM in a weighted scheme, a ranked revision list is produced to direct the debug process.

Failure information collected after design simulation/execution not only reveals hints to root causes but also implicates potential debug strategies. To prevent overburdening with the vast amount of failure information, clustering and classification algorithms are employed to identify root causes of failures, so that crucial issues requiring debug are prioritized. Poulos et al. [207] introduce a metric of measuring the similarities among failures regarding the likelihood of sharing the same root cause, upon which hierarchical clustering algorithms effectively group failures originating from the same root errors. These non-overlapping groups are then assigned to appropriate engineers for debug. Truong et al. [246] extensively compare different clustering and classification algorithms to cluster and classify test failures based on root causes, and emphasize that the top performant algorithms in clustering and classification are the density-based spatial clustering of applications with noise (DBSCAN) [87] and the random forest, respectively.

Simulation traces of DUTs depict the changes in design signal values during execution. By repeatedly examining and analyzing the trace dump from a huge number of design signals, verification engineers can identify erroneous signals and design areas. However, manual screening is not scalable with the increasing complexity of hardware designs, which has become the bottleneck in the debug process. Thus, any attempt that automates the diagnosis of simulation traces will greatly improve the debug efficiency. To alleviate the tedious efforts on inspecting overwhelming signal traces, K-means clustering can help to group trace segments with high similarities and identify unique ones that may exhibit rare design behaviors [83], which provides indicators of anomalous behaviors or areas worthy of careful examination. Later, a more advanced version of K-means clustering, X-means clustering [203], is leveraged to group similar trace segments from passed tests, with each time window having a separate clustering model [177]. Once the models are well-trained, the buggy trace segment will be identified as an outlier failing to be assigned to any previously

identified clusters. After the buggy trace window is detected, its design signals are compared with the signals from its closest neighbor selected by a one-nearest-neighbor classifier, so that the bug localization module can report the design module name, the list of signals, and the trace window number that contains bugs.

A bug signature provides contextual information explaining the cause or effect of a bug [237]. BugMD [176] detects bugs by comparing the architectural states (e.g., register values, memory contents, program counter values) of the DUT against a golden reference model, and records mismatches (e.g., the difference and the Hamming distance between the wrong and correct values) starting from the first manifestation of a bug as symptoms to form a bug signature; with a bug signature, BugMD adopts a random forest (comprising 64 trees) as the classifier to pinpoint the hardware unit responsible for the bug. BugMD is evaluated with synthetic bugs and identifies the location for a single bug in a single test run, reaching an accuracy of 70%. A comprehensive assessment of different ML techniques in BugMD, such as XGBoost, random forest, and MLP, indicates that XGBoost and random forest generally show higher accuracy [209]. Wahba et al. [258] analyze both single- and multi-defect signatures extracted from past failing simulations, and leverage decision trees to learn a set of rules that differentiate failures caused by defects in RTL code or the verification environment, achieving 90% and 95% RTL bug capturing rates for single- and multi-defect signatures, respectively.

To generate repairs for hardware security bugs, Ahmad et al. [7] investigate the potential of using LLMs, such as OpenAI Codex [51, 193] and CodeGen [188]. They build a corpus of benchmark designs, consisting of ten hardware security bugs from Common Weakness Enumerations (CWE) descriptions on the MITRE website [68], OpenTitan SoC [67], and the Hack@DAC 2021 SoC [125]. Given a hardware design, a static analysis tool detects certain weaknesses within the RTL. For each bug, human experts develop different instructions to assist LLMs in generating repairs, such as comments before and after the buggy code. By combining the code before the bug, buggy code with comments, and instructions, prompts are formed and fed to LLMs. The repairs output by LLMs will be evaluated in terms of functionality and security. Experimental results show that an ensemble of LLMs can repair all ten bugs in the proposed benchmark and outperform the hardware bug repair tool Cirfix [8]. However, at the current stage, some assistance based on the designers' expertise is still required to identify the location and nature of the bug.

In bug detection and localization, one observation is that many studies rely on analyzing information from version control systems and bug signatures to detect the presence and location of bugs. However, a significant challenge arises: while data from version control systems are easily interpretable by humans, they may not be as clear to ML models. Additionally, the data often contains considerable redundancy. Therefore, the careful selection of features becomes highly important [48].

In analyzing failure/simulation information, a notable trend is the inclination of many studies to utilize unsupervised learning techniques. This aligns with the motivation behind employing ML, i.e., automatically uncovering patterns within a vast sea of failure information or simulation traces. One more factor contributing to the prevalence of unsupervised learning is the substantial effort required to provide high-quality labels for failure information. However, when employing unsupervised learning, the twist is the requirement for stronger assumptions on data distributions to choose suitable algorithms and ensure performance [107]. This still necessitates considerable expertise in both debug and ML. Hence, efficiently cleaning the collected failure data to feed into unsupervised learning models is crucial in this context.

3.4 Roadmap of ML in Oracle-based Verification

Building upon current achievements in ML-assisted simulation-based verification, there are several dimensions worth further investigation, including generalization, interpretation, and scalability, which have the potential to propel ML applied in simulation-based verification forward into a bright future.

3.4.1 Generalization. Generalization capability is a fundamental requirement for ML-based predictive models, optimizers, and debug assistants to be well adopted in simulation-based verification. Currently, most studies reviewed in Section 3 focus on design-specific approaches, indicating data collection and model training are invoked for every new DUT. Though the initialization process for each new DUT is tolerable, ML models that can generalize across a family of similar DUTs remain highly desirable. Such models are capable to keep the knowledge learned from past experiences and significantly reduce the number of data samples required for new DUTs, releasing the burden on data collection and model retraining. Potential cures for DUT-level generalization include meta-learning [131, 187] and transfer learning [279]. The incorporation of out-of-distribution methods [270] will benefit model robustness toward outliers of both DUTs and tests targeting certain coverage points.

3.4.2 Interpretation. Promising as ML for verification is, the lack of interpretability regarding the behaviors and decisions of ML models hinders their broader adoption. Fortunately, several studies have shifted the focus from high-performant models to explainable models, such as rule learning to explain the reasons why coverage points are hit by tests [53, 132, 133, 153] and rule-based ML techniques for test point analysis [157]. Since interpretation and explanation are important to identify and expose potential problems during training, encode expert knowledge and intuitions into models, and ensure the fidelity of predictions, interpretable ML [46, 109] is expected to offer more confidence, reliability, and security for the decisions made in simulation-based verification.

3.4.3 Scalability. The scalability of ML-based methods in simulation-based verification is transforming from extra credit to an obligation, especially when considering the skyrocketing complexity of electronic designs. This challenge is caused by the synergy of technology scaling, design size increase, SoC and heterogeneous system integration, adoption of emerging technologies, etc. Even though many studies discussed in Section 3 conspicuously improve verification efficiency by reducing DUT simulation time, generating better tests to accelerate coverage closure, filtering unimportant tests, and automatic debugging, most DUTs are single circuit units or IP blocks. Notably, many of the techniques effective for IP-block-level or sub-system-level verification, no matter conventional or ML-based, often struggle to scale to the entire SoC integration or system-level validation, heralding that more attention should be given to how to exploit ML-based techniques for large-scale and system-level verification. This poses challenges to how to formulate system-level verification as ML problems and how to develop ML algorithms that can handle multi-level abstraction and multi-granularity optimization. Potential solutions are hierarchical ML algorithms. For example, hierarchical RL [162] has flexible goal specifications and can learn goal-directed behaviors in complex environments with sparse feedback, enabling more flexible and effective multi-level design and control, which is helpful for test generation and selection; hierarchical clustering algorithms [210] are capable to cluster samples either bottom-up or top-down, naturally matching multi-level abstraction in EDA flows, which is helpful for diagnosing root causes from failure information.

3.4.4 Data Collection. Data are the mainstay of ML. It is the sufficiency, variety, and representativeness of data that majorly determine the behavior and capability of ML models. In contrast to some

EDA tasks suffering data scarcity (e.g., placement and routing in physical synthesis), hardware verification naturally comes with an enormous amount of data. However, these data are mostly designed for human perception instead of ML, which could be imperfect in terms of implicit labeling, high redundancy, and inevitable noise. Meanwhile, real hardware designs are often proprietary, which limits most academic research to open-source and relatively small-scale hardware designs. Thus, we shed light on how to handle imperfect data through different supervision and data augmentation, and provide pointers to open-source datasets and benchmarks for hardware designs, verification libraries, and EDA tools.

Imperfect Data. In the case of lacking perfect labels, potential alternatives are to adopt hybrid supervision, such as self-supervised learning [128], semi-supervised learning [250], or combining supervised with unsupervised techniques [12]; transfer learning [279] can be a workaround since fewer data samples are required to transfer pre-trained models to target domains; active learning [211] seeks to maximize the performance of a model with the fewest annotated samples possible. In the case of data augmentation, generative methods can generate synthetic data [10] that are artificial but realistic, i.e., indistinguishable from real hardware designs; by applying different constraints during data generation, the redundancy in training data can be reduced. In the case of noisy data, different data cleaning approaches [122, 140] are beneficial to improve model capability.

Dataset and Benchmark. From the hardware design standpoint, OpenCores [195] is a prominent platform for open-source digital IP cores (e.g., central processing cores, memory controllers, peripherals), aiming to slash hardware development costs. It is noted that many verification-related studies from academia choose designs from OpenCores as their DUTs. From the verification library standpoint, Open Verification Library [95] is a vendor- and language-independent assertion library applicable across multiple verification processes, which provides common assertions that have been compiled and optimized by experts and can be easily tailored according to users' needs. From the general EDA standpoint, OpenROAD [150, 196] provides open-source EDA infrastructures, including an industry-strength database and timing analysis, Tcl and Python script interfaces, and design-rule clean layout generation, which greatly fosters the transparency and reproductivity in academic research. The openness of datasets and benchmarks matters the innovations in hardware verification. The low-barrier access to industry-strength data is expected to be a thrust in more general adoption of ML in verification.

4 INDUSTRIAL ADOPTION

4.1 From Academic to Industrial

As we tackle increasing design complexity and more challenging technology nodes, industry players are looking ever more seriously at minimizing simulation cycles, reducing manual verification effort, and accelerating the verification engines on which they rely. Motivated by various proposals of ML for verification from academia, the industry has been expressing immense interest in integrating ML algorithms into product verification efforts to achieve the aforementioned goals. In the last few years, both chip design and design automation companies have formed working groups to identify areas of the verification flow that would most immediately benefit from an ML approach and implemented prototypes to experiment with various proposals. A typical industry approach to ML for verification focuses on oracle-based verification and involves breaking the verification flow into different steps, including test generation, test selection, simulation, failure analysis, bug detection, and bug fix, many of which are covered in the previous sections. Engineers then look for ML opportunities within each step and propose relevant ML techniques based on the characteristics of the specific problem in that step. While a few proposals have been incorporated into actual

products, most remain under development pending stronger results. This fact underlies that with opportunities come challenges of practically applying ML in a production work flow.

Test generation sits at the forefront of the verification flow. The quality of generated test cases directly affects key coverage metrics used for design sign-off. Today, test generation remains a largely manual process because of the semantic gap between design specification and design entry. Verification engineers need to manually parse through design specifications and translate these specifications into either constraints for randomized tests or directed test cases. While there exists heuristic-level automation to expand human-designed test templates into many tests, the initial step of translating design specifications into these templates remains time-consuming and error-prone. In addition, human-designed tests and automatically expanded tests either are repetitive or lack the focus needed to activate corner case bugs and hard-to-hit combinations. We expect supervised learning and RL approaches that model the relationship between the signals of the DUT and configurations of test templates to be very helpful in being more selective during test generation [137]. However, it is necessary to demonstrate that these techniques scale to large circuits. More importantly, though, we hope that recent advances in natural language processing can be applied similarly to design specification language and hardware description language so that we can automatically extract design intents from design specifications and generate tests in a manner similar to that of a natural language translation task. This would alleviate the most laborious part of the test generation process and also create good-quality test templates in the first place.

Test selection is very important because simulation time is similarly determined by how many tests need to be run to reach a certain coverage goal. We are interested in techniques that compute the similarity between tests so that similar stimuli don't need to be repeatedly stressed and that more simulation cycles can be allocated to novel or corner test cases. Novelty detection based on similarity metrics can be applied to more intelligently filter out unnecessary tests [120, 123]. On top of that, we should focus on failing tests as opposed to repeatedly stressing passing tests. However, we don't know ahead of time whether a test will pass or fail without running it. That's why ARM researcher has initiated the effort of using ML models to predict the likelihood of failure of new stimuli and to run tests on those that are more likely to fail [226, 227]. The proposed method is a combination of supervised classification and unsupervised outlier detection. Similar functionality has been proposed by Synopsys [200] and also available with Cadence Verisium [45].

Because coverage is a key metric for sign-off, industry tends to be most interested in test selection techniques that optimize for coverage. These techniques typically leverage RL or blackbox optimization such as Bayesian optimization and consist of ML models that predict coverage and a feedback mechanism to update test templates/constraints based on predicted or measured coverage. In these cases, ML coverage prediction model is used in conjunction with actual coverage collection from simulation to strike a balance between efficiency and accuracy. While coverage is the top-level metric at sign-off, it may not be the most effective metrics for optimizing test selection. In addition to considering coverage in test selection, we should consider other metrics that more effectively stress the DUT. Examples of common ways to stress a design include maximizing the occupancy of FIFO, creating stalls and backpressure, and forcing resource contention [76].

Once failures are observed from the test cases, we need to analyze these failures and detect the bug related to the failure. Typically there are many more failures occurring simultaneously than what the limited number of engineers can handle. As a result, it is necessary to triage the many failures by categorizing them into different bins. We can then analyze a representative failure from each bin. In addition, we need to relate failures to bugs and find the expert responsible for resolving each bug. This involves studying source code revisions or looking at important waveform signals to pinpoint the root cause of a test failure. To facilitate this process, for example, Cadence

Verisium AutoTriage builds ML models to classify failures that have common root causes. In addition, Cadence provides Verisium SemanticDiff for classifying and ranking the importance of revisions and Verisium WaveMiner for predicting the signals and the times most relevant to the failure [45].

4.2 Commercial ML-driven Verification Platforms

Cadence has integrated ML into their commercial products such as Verisium [45], Xcelium Logic Simulation [44], and Jasper RTL Apps [43]. Verisium is an ML-driven verification platform to optimize verification workloads, boost coverage, and accelerate root cause analysis, which employs ML models to rank code revisions most disruptive to system behaviors, classify failures with same root causes, and identify signals/times most likely to expose failures. Xcelium is a comprehensive logic simulation tool equipped with an Xcelium ML accelerator for boosting user regression throughput. The Xcelium ML is trained on coverage random seed data collected during user regressions, which predicts the influence of these conditions on the target coverage goal and generates a much faster regression set that can reach the same coverage. It enables up to 5× faster verification closure on randomized regressions [42]. Jasper is a formal verification tool incorporating ML to improve verification productivity. Inside Jasper, ML is used to select and tune the hyperparameters of verification proof solvers to speed up proofs; ML has also been used to speed up successive runs of regression testing. The observed speedup is 2× to 5×.

Synopsys has been integrating ML into their VC Formal [242], VC SpyGlass RTL Static Signoff Platform [243], and VCS functional verification solution [240]. In VC Formal, ML is used for speeding up the convergence of formal proofs for subsequent runs; in VC SpyGlass, ML is used for reducing the number of false clock domain crossing (CDC) violations, and thus to enable faster identification of root cause; in VCS functional verification, ML is integrated into intelligent coverage optimization (ICO), which analyzes the relations between the user inputs and the tests executed in simulation and improves the test quality through diversification in the constraint space. A case study adopting ICO in early verification stages of cache designs achieves faster testbench stabilization by discovering and fixing hard-to-hit testbench bugs due to under- and over-constraints [248].

Siemens [229] has been exploring ML techniques in the Siemens EDA Solido Characterization Suite [230] to accelerate the production quality library characterization and verification of standard cells, memory, and custom blocks; ML techniques are also used in Solido Variation Designer [231] to accelerate simulation. Unfortunately, we do not find more details due to commercial confidentiality.

TestGuru [256] from VerifAI, which uses LLM, RL, and other algorithms to analyze code, is capable to generate tests, write code, explain code, finds bugs, and fixes bugs on Python and Verilog, achieving one order magnitude speedup compared to human developers.

5 CONCLUSION

There have been many victories in hardware verification coming from the exploitation of ML-based techniques. In formal verification, ❶ GNN-based approaches greatly improve the performance of SAT solvers; ❷ NLP-based methods automate the assertion generation across human languages, verification languages, and hardware description languages; ❸ different classifiers provide potentials to always make the smartest selections in equivalence checking, model checking, theorem proving, and saving verification costs. In simulation-based verification, ML-based techniques act as ❶ predictive models for fast and accurate coverage analysis, ❷ optimizer for smart test generation or selection, and ❸ automatic troubleshooting assistants. To fully unveil potentials and possibilities of ML applied for hardware verification, we provide our visions on the road ahead and expect the broader, generic, and efficacious ML applications be the enabler for more scalable, more intelligent, and more productive hardware design verification.

REFERENCES

[1] Accessed: 2023. The International SAT Competition. http://www.satcompetition.org/.

[2] Mohamed A Abd El Ghany and Khaled A Ismail. 2021. Speed Up Functional Coverage Closure of CORDIC Designs Using Machine Learning Models. In *2021 International Conference on Microelectronics (ICM)*. IEEE, 91–95.

[3] Moloud Abdar, Farhad Pourpanah, Sadiq Hussain, Dana Rezazadegan, Li Liu, Mohammad Ghavamzadeh, Paul Fieguth, Xiaochun Cao, Abbas Khosravi, U Rajendra Acharya, et al. 2021. A review of uncertainty quantification in deep learning: Techniques, applications and challenges. *Information fusion* 76 (2021), 243–297.

[4] Mostafa AboelMaged, Maggie Mashaly, and Mohamed A Abd El Ghany. 2021. Online Constraints Update Using Machine Learning for Accelerating Hardware Verification. In *2021 3rd Novel Intelligent and Leading Emerging Sciences Conference (NILES)*. IEEE, 113–116.

[5] Fnu Aditi and Michael S Hsiao. 2022. Hybrid Rule-based and Machine Learning System for Assertion Generation from Natural Language Specifications. In *2022 IEEE 31st Asian Test Symposium (ATS)*. IEEE, 126–131.

[6] Anthony Agnesina, Sung Kyu Lim, Etienne Lepercq, and Jose Escobedo Del Cid. 2020. Improving FPGA-based logic emulation systems through machine learning. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 25, 5 (2020), 1–20.

[7] Baleegh Ahmad, Shailja Thakur, Benjamin Tan, Ramesh Karri, and Hammond Pearce. 2023. Fixing Hardware Security Bugs with Large Language Models. *arXiv preprint arXiv:2302.01215* (2023).

[8] Hammad Ahmad, Yu Huang, and Westley Weimer. 2022. Cirfix: automatically repairing defects in hardware design code. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 990–1003.

[9] Ravindra K Ahuja, Kurt Mehlhorn, James Orlin, and Robert E Tarjan. 1990. Faster algorithms for the shortest path problem. *Journal of the ACM (JACM)* 37, 2 (1990), 213–223.

[10] Ahmed Alaa, Boris Van Breugel, Evgeny S Saveliev, and Mihaela van der Schaar. 2022. How faithful is your synthetic data? sample-level metrics for evaluating and auditing generative models. In *International Conference on Machine Learning*. PMLR, 290–306.

[11] Jesse Alama, Tom Heskes, Daniel Kühlwein, Evgeni Tsivtsivadze, and Josef Urban. 2014. Premise selection for mathematics by corpus analysis and kernel methods. *Journal of automated reasoning* 52, 2 (2014), 191–213.

[12] Mohamad Alawieh, Fa Wang, and Xin Li. 2017. Efficient hierarchical performance modeling for integrated circuits via bayesian co-learning. In *Proceedings of the 54th Annual Design Automation Conference 2017*. ACM, 9.

[13] Medhat Ashraf Alhaddad, Seif Eldin Mohamed Hussein, Abanoub Ghadban Helmy, Nagy Raouf Nagy, Muhammad Ziad Muhammad Ghazy, and Ahmed H Yousef. 2021. Utilization of Machine Learning In RTL-GL Signals Correlation. In *2021 8th International Conference on Signal Processing and Integrated Networks (SPIN)*. IEEE, 732–737.

[14] Sarath Mohan Ambalakkat and Eldon G Nelson. 2019. Simulation Runtime Optimization of Constrained Random Verification using Machine Learning Algorithms. In *Proceedings of the Design and Verification Conference and Exhibition, US (DVCon)*.

[15] Saeed Amizadeh, Sergiy Matusevych, and Markus Weimer. 2018. Learning to solve circuit-SAT: An unsupervised differentiable approach. In *International Conference on Learning Representations*.

[16] Apache. Accessed: June-2022. Apache Subversion. https://subversion.apache.org/.

[17] Tadashi Araragi and Seung Mo Cho. 2007. Checking liveness properties of concurrent systems by reinforcement learning. In *Model Checking and Artificial Intelligence: 4th Workshop, MoChArt IV, Riva del Garda, Italy, August 29, 2006, Revised Selected and Invited Papers 4*. Springer, 84–94.

[18] Eser Aygün, Ankit Anand, Laurent Orseau, Xavier Glorot, Stephen M Mcaleer, Vlad Firoiu, Lei M Zhang, Doina Precup, and Shibl Mourad. 2022. Proving Theorems using Incremental Learning and Hindsight Experience Replay. In *International Conference on Machine Learning*. PMLR, 1198–1210.

[19] Tomáš Babiak, Mojmír Křetínskỳ, Vojtěch Řehák, and Jan Strejček. 2012. LTL to Büchi automata translation: Fast and more deterministic. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 95–109.

[20] Dorit Baras, Shai Fine, Laurent Fournier, Dan Geiger, and Avi Ziv. 2011. Automatic boosting of cross-product coverage using Bayesian networks. *International Journal on Software Tools for Technology Transfer* 13, 3 (2011), 247–261.

[21] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. 1997. *The Coq proof assistant reference manual: Version 6.1*. Ph. D. Dissertation. Inria.

[22] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. Cvc4. In *International Conference on Computer Aided Verification*. Springer, 171–177.

[23] Razieh Behjati, Marjan Sirjani, and Majid Nili Ahmadabadi. 2010. Bounded rational search for on-the-fly model checking of LTL properties. In *Fundamentals of Software Engineering: Third IPM International Conference, FSEN 2009, Kish Island, Iran, April 15-17, 2009, Revised Selected Papers 3*. Springer, 292–307.

[24] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. 2021. Machine learning for combinatorial optimization: a methodological tour d'horizon. *European Journal of Operational Research* 290, 2 (2021), 405–421.
[25] Mike Benjamin, Daniel Geist, Alan Hartman, Gerard Mas, Ralph Smeets, and Yaron Wolfsthal. 1999. A study in coverage-driven test generation. In *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*. 970–975.
[26] Yves Bertot and Pierre Castéran. 2013. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media.
[27] David Bieber, Charles Sutton, Hugo Larochelle, and Daniel Tarlow. 2020. Learning to execute programs with instruction pointer attention graph neural networks. *Advances in Neural Information Processing Systems* 33 (2020), 8626–8637.
[28] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Masahiro Fujita, and Yunshan Zhu. 1999. Symbolic model checking using SAT procedures instead of BDDs. In *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*. 317–320.
[29] Armin Biere, Marijn Heule, and Hans van Maaren. 2009. *Handbook of satisfiability*. Vol. 185. IOS press.
[30] Jasmin Christian Blanchette, David Greenaway, Cezary Kaliszyk, Daniel Kühlwein, and Josef Urban. 2016. A learning-based fact selector for Isabelle/HOL. *Journal of Automated Reasoning* 57, 3 (2016), 219–244.
[31] Jason Blocklove, Siddharth Garg, Ramesh Karri, and Hammond Pearce. 2023. Chip-Chat: Challenges and Opportunities in Conversational Hardware Design. *arXiv preprint arXiv:2305.13243* (2023).
[32] Bluetooth. Accessed: June-2022. Specification of the Bluetooth System. http://www.tscm.com/BluetoothSpec.pdf.
[33] John M Borkenhagen, Richard J Eickemeyer, Ronald N Kalla, and Steven R Kunkel. 2000. A multithreaded PowerPC processor for commercial servers. *IBM Journal of Research and Development* 44, 6 (2000), 885–898.
[34] Léon Bottou, Frank E. Curtis, and Jorge Nocedal. 2018. Optimization Methods for Large-Scale Machine Learning. *SIAM Rev.* 60, 2 (2018), 223–311. https://doi.org/10.1137/16M1080173
[35] Markus Braun, Shai Fine, and Avi Ziv. 2004. Enhancing the efficiency of Bayesian network based coverage directed test generation. In *Proceedings. Ninth IEEE International High-Level Design Validation and Test Workshop (IEEE Cat. No. 04EX940)*. IEEE, 75–80.
[36] James P Bridge. 2010. *Machine learning and automated theorem proving*. Technical Report. University of Cambridge, Computer Laboratory.
[37] James P Bridge, Sean B Holden, and Lawrence C Paulson. 2014. Machine learning for first-order theorem proving. *Journal of automated reasoning* 53, 2 (2014), 141–172.
[38] Randal E Bryant. 1986. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on* 100, 8 (1986), 677–691.
[39] Yirng-An Chen Randal E Bryant. 1995. Verification of arithmetic circuits with binary moment diagrams. In *32nd Design Automation Conference*. IEEE, 535–541.
[40] Thibaut Bultiaux, Stephane Guenot, Serge Hustin, Alexandre Blampey, Joseph Bulone, and Matthieu Moy. 2005. *Functional Verification*. Springer US, 153–206. https://doi.org/10.1007/0-387-26233-4_5
[41] Benedikt Bünz and Matthew Lamm. 2017. Graph neural networks and boolean satisfiability. *arXiv preprint arXiv:1702.03592* (2017).
[42] Cadence. Accessed: June-2023. Cadence Delivers Machine Learning-Optimized Xcelium Logic Simulation with up to 5X Faster Regressions. https://www.cadence.com/en_US/home/company/newsroom/press-releases/pr/2020/cadence-delivers-machine-learning-optimized-xcelium-logic-simula.html.
[43] Cadence. Accessed: March-2022. Cadence Jasper RTL Apps. https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform.html.
[44] Cadence. Accessed: March-2022. Cadence Xcelium Logic Simulation. https://www.cadence.com/en_US/home/tools/system-design-and-verification/simulation-and-testbench-verification/xcelium-simulator.html.
[45] Cadence. Accessed: October-2022. Verisium AI-Driven Verification Platform. https://www.cadence.com/en_US/home/tools/system-design-and-verification/ai-driven-verification.html.
[46] Diogo V Carvalho et al. 2019. Machine learning interpretability: A survey on methods and metrics. *Electronics* (2019).
[47] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. 2014. The nuXmv symbolic model checker. In *Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings 26*. Springer, 334–342.
[48] Girish Chandrashekar and Ferat Sahin. 2014. A survey on feature selection methods. *Computers & Electrical Engineering* 40, 1 (2014), 16–28.
[49] Po-Hsien Chang, Dragoljub Drmanac, and Li-C Wang. 2010. Online selection of effective functional test programs based on novelty detection. In *2010 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 762–769.

[50] Abhishek Chauhan. 2022. Automatic Translation of Natural Language to SystemVerilog Assertions. In *Proceedings of the Design and Verification Conference and Exhibition, US (DVCon)*.

[51] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[52] Wen Chen, Nik Sumikawa, Li-C Wang, Jayanta Bhadra, Xiushan Feng, and Magdy S Abadir. 2012. Novel test detection to improve simulation efficiency—A commercial experiment. In *2012 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 101–108.

[53] Wen Chen, Li-Chung Wang, Jay Bhadra, and Magdy Abadir. 2013. Simulation knowledge extraction and reuse in constrained random processor verification. In *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[54] Hyojin Choi, In Huh, Seungju Kim, Jeonghoon Ko, Changwook Jeong, Hyeonsik Son, Kiwon Kwon, Joonwan Chai, Younsik Park, Jaehoon Jeong, et al. 2021. Application of Deep Reinforcement Learning to Dynamic Verification of DRAM Designs. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 523–528.

[55] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. 2017. Kami: a platform for high-level parametric hardware specification and its modular verification. *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 1–30.

[56] Niket K Choudhary, Salil V Wadhavkar, Tanmay A Shah, Hiran Mayukh, Jayneel Gandhi, Brandon H Dwiel, Sandeep Navada, Hashem H Najaf-abadi, and Eric Rotenberg. 2011. FabScalar: Composing synthesizable RTL designs of arbitrary cores within a canonical superscalar template. In *38th Annual International Symposium on Computer Architecture (ISCA)*.

[57] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2022. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311* (2022).

[58] Maciej Ciesielski, Tiankai Su, Atif Yasin, and Cunxi Yu. 2019. Understanding algebraic rewriting for arithmetic circuit verification: a bit-flow model. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 6 (2019), 1346–1357.

[59] Maciej Ciesielski, Cunxi Yu, Walter Brown, Duo Liu, and André Rossi. 2015. Verification of gate-level arithmetic circuits by function extraction. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[60] Edmund Clarke, Anubhav Gupta, James Kukula, and Ofer Strichman. 2002. SAT based abstraction-refinement using ILP and machine learning techniques. In *Computer Aided Verification: 14th International Conference, CAV 2002 Copenhagen, Denmark, July 27–31, 2002 Proceedings 14*. Springer, 265–279.

[61] Edmund M Clarke. 1997. Model checking. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer, 54–56.

[62] Edmund M Clarke, Anubhav Gupta, and Ofer Strichman. 2004. SAT-based counterexample-guided abstraction refinement. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 23, 7 (2004), 1113–1123.

[63] Edmund M Clarke, William Klieber, Miloš Novácek, and Paolo Zuliani. 2012. Model Checking and the State Explosion Problem. *Tools for Practical Software Verification* (2012), 1.

[64] Edmund M Clarke and Jeannette M Wing. 1996. Formal methods: State of the art and future directions. *ACM Computing Surveys (CSUR)* 28, 4 (1996), 626–643.

[65] Jason Cong and Majid Sarrafzadeh. 2000. Incremental physical design. In *Proceedings of the 2000 international symposium on Physical design*. 84–92.

[66] Stephen A Cook. 1971. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*. 151–158.

[67] The MITRE Corporation. 2019. Hardware | OpenTitan Documentation. https://cwe.mitre.org/data/definitions/1194.html.

[68] The MITRE Corporation. 2023. CWE-1194: Hardware Design (4.11). https://cwe.mitre.org/data/definitions/1194.html.

[69] David Cox, John Little, and Donal OShea. 2013. *Ideals, varieties, and algorithms: an introduction to computational algebraic geometry and commutative algebra*. Springer Science & Business Media.

[70] Mihai-Corneliu Cristescu. 2021. Machine Learning Techniques for Improving the Performance Metrics of Functional Verification. *SCIENCE AND TECHNOLOGY* 24, 1 (2021), 99–116.

[71] Mihai-Corneliu Cristescu and Cristian Bob. 2021. Flexible Framework for Stimuli Redundancy Reduction in Functional Verification Using Artificial Neural Networks. In *2021 International Symposium on Signals, Circuits and Systems (ISSCS)*. IEEE, 1–4.

[72] Mihai-Corneliu Cristescu and Daniel Ciupitu. 2021. Stimuli Redundancy Reduction for Nonlinear Functional Verification Coverage Models Using Artificial Neural Networks. In *2021 International Semiconductor Conference (CAS)*. IEEE, 217–220.

[73] Alessandro Dal Palù, Agostino Dovier, Andrea Formisano, and Enrico Pontelli. 2015. Cud@ sat: Sat solving on gpus. *Journal of Experimental & Theoretical Artificial Intelligence* 27, 3 (2015), 293–316.
[74] Martin Davis, George Logemann, and Donald Loveland. 1962. A machine program for theorem-proving. *Commun. ACM* 5, 7 (1962), 394–397.
[75] Leonardo De Moura and Nikolaj Bjørner. 2011. Satisfiability modulo theories: introduction and applications. *Commun. ACM* 54, 9 (2011), 69–77.
[76] Siddhanth Dhodhi, Debarshi Chatterjee, Eric Hill, and Saad Godil. 2021. Deep Stalling using a Coverage Driven Genetic Algorithm Framework. In *2021 IEEE 39th VLSI Test Symposium (VTS)*. IEEE, 1–4.
[77] Alexandru Dinu, Gabriel Mihail Danciu, and Ștefan Gheorghe. 2021. Level up in verification: learning from functional snapshots. In *2021 16th International Conference on Engineering of Modern Electric Systems (EMES)*. IEEE, 1–4.
[78] Dale Drinkard. Accessed: July-2022. Configurable cordic core in verilog. https://opencores.org/projects/mesi_isc.
[79] Dale Drinkard. Accessed: May-2022. Configurable cordic core in verilog. https://opencores.org/projects/verilog_cordic_core.
[80] Ke-Lin Du and Madisetti NS Swamy. 2013. *Neural networks and statistical learning*. Springer Science & Business Media.
[81] Haonan Duan, Pashootan Vaezipoor, Max B Paulus, Yangjun Ruan, and Chris Maddison. 2022. Augment with Care: Contrastive Learning for Combinatorial Problems. In *International Conference on Machine Learning*. PMLR, 5627–5642.
[82] Eman El Mandouh, Ashraf Salem, Mennatallah Amer, and Amr G Wassal. 2018. Cross-product functional coverage analysis using machine learning clustering techniques. In *2018 13th International Conference on Design & Technology of Integrated Systems In Nanoscale Era (DTIS)*. IEEE, 1–2.
[83] Eman El Mandouh and Amr G Wassal. 2016. Accelerating the debugging of fv traces using k-means clustering techniques. In *2016 11th International Design & Test Symposium (IDT)*. IEEE, 278–283.
[84] Eman Elmandouh and Amr G Wassal. 2016. Estimation of formal verification cost using regression machine learning. In *2016 IEEE International High Level Design Validation and Test Workshop (HLDVT)*. IEEE, 121–127.
[85] Eman M Elmandouh and Amr G Wassal. 2018. Guiding formal verification orchestration using machine learning methods. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 23, 5 (2018), 1–33.
[86] Frank Emmert-Streib, Zhen Yang, Han Feng, Shailesh Tripathi, and Matthias Dehmer. 2020. An introductory review of deep learning for prediction models with big data. *Frontiers in Artificial Intelligence* 3 (2020), 4.
[87] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*. 226–231.
[88] Martin Fajcik, Pavel Smrz, and Marcela Zachariasova. 2017. Automation of processor verification using recurrent neural networks. In *2017 18th International Workshop on Microprocessor and SOC Test and Verification (MTV)*. IEEE, 15–20.
[89] Shai Fine, Ari Freund, Itai Jaeger, Yishay Mansour, Yehuda Naveh, and Avi Ziv. 2006. Harnessing machine learning to improve the success rate of stimuli generation. *IEEE Trans. Comput.* 55, 11 (2006), 1344–1355.
[90] Shai Fine and Avi Ziv. 2003. Coverage directed test generation for functional verification using bayesian networks. In *Proceedings of the 40th annual Design Automation Conference*. 286–291.
[91] Achille Fokoue, Ibrahim Abdelaziz, Maxwell Crouse, Shajith Ikbal, Akihiro Kishimoto, Guilherme Lima, Ndivhuwo Makondo, and Radu Marinescu. 2023. An Ensemble Approach for Automated Theorem Proving Based on Efficient Name Invariant Graph Neural Representations. *arXiv preprint arXiv:2305.08676* (2023).
[92] Harry Foster. 2008. Assertion-based verification: Industry myths to realities (invited tutorial). In *International Conference on Computer Aided Verification*. Springer, 5–10.
[93] Harry Foster. Accessed: March-2022. The 2020 Wilson Research Group Functional Verification Study. https://blogs.sw.siemens.com/verificationhorizons/2020/10/27/prologue-the-2020-wilson-research-group-functional-verification-study/.
[94] Harry Foster. Accessed: March-2024. The 2022 Wilson Research Group Functional Verification Study. https://blogs.sw.siemens.com/verificationhorizons/2022/10/10/prologue-the-2022-wilson-research-group-functional-verification-study/.
[95] Harry Foster, Kenneth Larsen, and Mike Turpin. 2006. Introduction to the new accellera open verification library. In *DVCon'06: Proceedings of the Design and Verification Conference and exhibition*. Citeseer.
[96] Harry D Foster. 2015. Trends in functional verification: A 2014 industry study. In *Proceedings of the 52nd Annual Design Automation Conference*. 1–6.
[97] Harry D Foster, Adam C Krolnik, and David J Lacey. 2004. *Assertion-based design*. Springer Science & Business Media.
[98] Matthias Fuchs. 1998. A feature-based learning method for theorem proving. In *AAAI/IAAI*. 457–462.
[99] Muhammad Gad, Mostafa Aboelmaged, Maggie Mashaly, and Mohamed A Abd el Ghany. 2021. Efficient Sequence Generation for Hardware Verification Using Machine Learning. In *2021 28th IEEE International Conference on Electronics,*

Circuits, and Systems (ICECS). IEEE, 1–5.
[100] Raviv Gal, Eldad Haber, Brian Irwin, Marwa Mouallem, Bilal Saleh, and Avi Ziv. 2021. Using Deep Neural Networks And Derivative Free Optimization To Accelerate Coverage Closure. In *2021 ACM/IEEE 3rd Workshop on Machine Learning for CAD (MLCAD)*. IEEE, 1–6.
[101] Raviv Gal, Eldad Haber, and Avi Ziv. 2020. Using dnns and smart sampling for coverage closure acceleration. In *Proceedings of the 2020 ACM/IEEE Workshop on Machine Learning for CAD*. 15–20.
[102] Honghao Gao, Baobin Dai, Huaikou Miao, Xiaoxian Yang, Ramon J Duran Barroso, and Hussain Walayat. 2023. A novel gapg approach to automatic property generation for formal verification: The gan perspective. *ACM Transactions on Multimedia Computing, Communications and Applications* 19, 1 (2023), 1–22.
[103] Pranav Garg, Christof Löding, Parthasarathy Madhusudan, and Daniel Neider. 2014. ICE: A robust framework for learning invariants. In *Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings 26*. Springer, 69–87.
[104] Pranav Garg, Daniel Neider, Parthasarathy Madhusudan, and Dan Roth. 2016. Learning invariants using decision trees and implication counterexamples. *ACM Sigplan Notices* 51, 1 (2016), 499–512.
[105] Priyanshi Gaur, Sidhartha Sankar Rout, and Sujay Deb. 2019. Efficient Hardware Verification Using Machine Learning Approach. In *2019 IEEE International Symposium on Smart Electronic Systems (iSES)(Formerly iNiS)*. IEEE, 168–171.
[106] Zoubin Ghahramani. 1997. Learning dynamic Bayesian networks. *International School on Neural Networks, Initiated by IIASS and EMFCSC* (1997), 168–197.
[107] Zoubin Ghahramani. 2004. Unsupervised Learning. *Advanced Lectures on Machine Learning* (2004), 72.
[108] Shalini Ghosh, Daniel Elenius, Wenchao Li, Patrick Lincoln, Natarajan Shankar, and Wilfried Steiner. 2016. ARSENAL: automatic requirements specification extraction from natural language. In *NASA Formal Methods Symposium*. Springer, 41–46.
[109] Leilani H Gilpin et al. 2018. Explaining explanations: An overview of interpretability of machine learning. In *Proc. DSAA*.
[110] Git. Accessed: June-2022. Git. http://git-scm.com.
[111] Tim Goddard. Accessed: May-2022. Documented Verilog UART. https://opencores.org/projects/osdvu.
[112] Saumil Gogri, Jiang Hu, Aakash Tyagi, Mike Quinn, Swati Ramachandran, Fazia Batool, and Amrutha Jagadeesh. 2020. Machine Learning-Guided Stimulus Generation for Functional Verification. In *Proceedings of the Design and Verification Conference and Exhibition (DVCon)*.
[113] Eugene Goldberg and Yakov Novikov. 2003. On complexity of equivalence checking. *Cadence Berkeley Labs, Univ. California, Berkeley, CA, USA, Tech. Rep. CDNL-TR-2003-08026* (2003).
[114] Ian J Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative adversarial nets. In *Proceedings of the 27th International Conference on Neural Information Processing Systems-Volume 2*. 2672–2680.
[115] Christian Graber, Daniel Hansson, and Adam Tornhill. 2019. Predicting Bad Commits: Finding bugs by learning their socio-organizational patterns. In *Proceedings of the Design and Verification Conference and Exhibition, US (DVCon)*.
[116] Adam Grabowski, Artur Kornilowicz, and Adam Naumowicz. 2010. Mizar in a nutshell. *Journal of Formalized Reasoning* 3, 2 (2010), 153–245.
[117] OpenHW Group. Accessed: June-2022. Ariane RISC-V CPU. https://github.com/openhwgroup/cva6.
[118] Nikita Gulliya, Abhishek Bora, Nitin Chaudhary, and Amanjyot Kaur. 2019. Using Machine Learning in Register Automation and Verification. In *Proceedings of the Design and Verification Conference and Exhibition, US (DVCon)*.
[119] Qi Guo, Tianshi Chen, Yunji Chen, Rui Wang, Huanhuan Chen, Weiwu Hu, and Guoliang Chen. 2014. Pre-silicon bug forecast. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 33, 3 (2014), 451–463.
[120] Qi Guo, Tianshi Chen, Haihua Shen, Yunji Chen, and Weiwu Hu. 2010. On-the-fly reduction of stimuli for functional verification. In *2010 19th IEEE Asian Test Symposium*. IEEE, 448–454.
[121] Wenxuan Guo, Junchi Yan, Hui-Ling Zhen, Xijun Li, Mingxuan Yuan, and Yaohui Jin. 2022. Machine Learning Methods in Solving the Boolean Satisfiability Problem. *arXiv preprint arXiv:2203.04755* (2022).
[122] Shivani Gupta and Atul Gupta. 2019. Dealing with noise problem in machine learning data-sets: A systematic review. *Procedia Computer Science* 161 (2019), 466–474.
[123] Onur Guzey, Li-C Wang, Jeremy Levitt, and Harry Foster. 2008. Functional test selection based on unsupervised support vector analysis. In *2008 45th ACM/IEEE Design Automation Conference*. IEEE, 262–267.
[124] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. 2018. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*. PMLR, 1861–1870.
[125] HACK@EVENT. 2021. HACK@DAC 2021. https://hackatevent.org/hackdac21/.
[126] Mohamed Hanafy, Hazem Said, and Ayman M Wahba. 2015. Complete properties extraction from simulation traces for assertions auto-generation. In *2015 IEEE 24th North Atlantic Test Workshop*. IEEE, 1–6.

[127] Christopher B Harris and Ian G Harris. 2016. Glast: Learning formal grammars to translate natural language specifications into hardware assertions. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 966–971.
[128] Dan Hendrycks, Mantas Mazeika, Saurav Kadavath, and Dawn Song. 2019. Using self-supervised learning can improve model robustness and uncertainty. *Advances in neural information processing systems* 32 (2019).
[129] Matthew Honnibal, Ines Montani, Sofie Van Landeghem, and Adriane Boyd. 2020. spaCy: Industrial-strength Natural Language Processing in Python. (2020).
[130] John J Hopfield. 1982. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences* 79, 8 (1982), 2554–2558.
[131] Timothy Hospedales, Antreas Antoniou, Paul Micaelli, and Amos Storkey. 2021. Meta-learning in neural networks: A survey. *IEEE transactions on pattern analysis and machine intelligence* 44, 9 (2021), 5149–5169.
[132] Kuo-Kai Hsieh, Sebastian Siatkowski, Li-C Wang, Wen Chen, and Jayanta Bhadra. 2017. Feature extraction from design documents to enable rule learning for improving assertion coverage. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 51–56.
[133] Hsiou-Wen Hsueh and Kerstin Eder. 2006. Test directive generation for functional coverage closure using inductive logic programming. In *2006 IEEE International High Level Design Validation and Test Workshop*. IEEE, 11–18.
[134] Jian Hu, Yongyang Hu, Qi Lv, Wentao Wang, Guanwu Wang, Guilin Chen, Kang Wang, Yun Kang, and Haitao Yang. 2021. A path-based equivalence checking method between system level and RTL descriptions using machine learning. *Journal of Circuits, Systems and Computers* 30, 04 (2021), 2150074.
[135] Jian Hu, Tun Li, and Sikun Li. 2016. Equivalence checking between SLM and RTL using machine learning techniques. In *2016 17th International Symposium on Quality Electronic Design (ISQED)*. IEEE, 129–134.
[136] Wei-Wu Hu, Fu-Xin Zhang, and Zu-Song Li. 2005. Microarchitecture of the Godson-2 processor. *Journal of Computer Science and Technology* 20, 2 (2005), 243–249.
[137] William Hughes, Sandeep Srinivasan, Rohit Suvarna, and Maithilee Kulkarni. 2019. Optimizing Design Verification using Machine Learning: Doing better than Random. *arXiv preprint arXiv:1909.13168* (2019).
[138] IBM. Accessed: June-2022. IBM z10. http://en.wikipedia.org/wiki/IBM_z10.
[139] Shahid Ikram and Jim Ellis. 2017. Dynamic regression suite generation using coverage-based clustering. In *Proceedings of the Design and Verification Conference and Exhibition, US (DVCon)*.
[140] Ihab F Ilyas and Xu Chu. 2019. *Data cleaning*. Morgan & Claypool.
[141] Global Market Insights. 2023. Hardware-Assisted Verification Market - By Platform (Hardware Emulation, FPGA Prototyping), By Application (Automotive, Consumer Electronics, Industrial, Aerospace & Defense, Medical, Telecom) & Global Forecast, 2023-2032. https://www.gminsights.com/toc/detail/hardware-assisted-verification-market.
[142] Charalambos Ioannides and Kerstin I Eder. 2012. Coverage-directed test generation automated by machine learning–a review. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 17, 1 (2012), 1–21.
[143] Geoffrey Irving, Christian Szegedy, Alexander A Alemi, Niklas Eén, François Chollet, and Josef Urban. 2016. Deepmath-deep sequence models for premise selection. *Advances in neural information processing systems* 29 (2016).
[144] Khaled A Ismail and Mohamed A Abd El Ghany. 2021. High Performance Machine Learning Models for Functional Verification of Hardware Designs. In *2021 3rd Novel Intelligent and Leading Emerging Sciences Conference (NILES)*. IEEE, 15–18.
[145] Khaled A Ismail and Mohamed A Ghany. 2021. Survey on Machine Learning Algorithms Enhancing the Functional Verification Process. *Electronics* 10, 21 (2021), 2688.
[146] V. Jayasree. 2021. Machine Learning for Coverage Analysis in Design Verification. In *Proceedings of the Design and Verification Conference and Exhibition, Europe (DVCon)*.
[147] Albert Qiaochu Jiang, Wenda Li, Jesse Michael Han, and Yuhuai Wu. 2021. LISA: Language models of ISAbelle proofs. In *6th Conference on Artificial Intelligence and Theorem Proving*. 378–392.
[148] Yaqing Jiang, Petros Papapanagiotou, and Jacques Fleuriot. 2018. Machine learning for inductive theorem proving. In *International Conference on Artificial Intelligence and Symbolic Computation*. Springer, 87–103.
[149] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*. 1–12.
[150] Andrew B Kahng. 2022. Leveling Up: A Trajectory of OpenROAD, TILOS and Beyond. In *Proceedings of the 2022 International Symposium on Physical Design*. 73–79.
[151] Cezary Kaliszyk and Josef Urban. 2014. Learning-assisted automated reasoning with Flyspeck. *Journal of Automated Reasoning* 53, 2 (2014), 173–213.
[152] Ron Kalla, Balaram Sinharoy, William J Starke, and Michael Floyd. 2010. Power7: IBM's next-generation server processor. *IEEE micro* 30, 2 (2010), 7–15.

[153] Yoav Katz, Michal Rimon, Avi Ziv, and Gai Shaked. 2011. Learning microarchitectural behaviors to improve stimuli generation quality. In *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 848–853.
[154] Christoph Kern and Mark R Greenstreet. 1999. Formal verification in hardware design: a survey. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 4, 2 (1999), 123–193.
[155] Nikita Kitaev and Dan Klein. 2018. Constituency Parsing with a Self-Attentive Encoder. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2676–2686.
[156] Balakrishnan Krishnamurthy. 1987. A dynamic programming approach to the test point insertion problem. In *24th ACM/IEEE Design Automation Conference*. IEEE, 695–705.
[157] Prashanth Krishnamurthy, Animesh Basak Chowdhury, Benjamin Tan, Farshad Khorrami, and Ramesh Karri. 2020. Explaining and Interpreting Machine Learning CAD Decisions: An IC Testing Case Study. In *2020 ACM/IEEE 2nd Workshop on Machine Learning for CAD (MLCAD)*. IEEE, 129–134.
[158] Prashanth Krishnamurthy, Alireza Sarmadi, and Farshad Khorrami. 2021. Explainable classification by learning human-readable sentences in feature subsets. *Information Sciences* 564 (2021), 202–219.
[159] Andreas Kuehlmann and Florian Krohm. 1997. Equivalence checking using cuts and heaps. In *Proceedings of the 34th annual Design Automation Conference*. 263–268.
[160] Daniel Kühlwein and Josef Urban. 2015. MaLeS: A framework for automatic tuning of automated theorem provers. *Journal of Automated Reasoning* 55, 2 (2015), 91–116.
[161] Maithilee Rajendra Kulkarni et al. 2019. *Improving coverage of simulation-based design verification using Machine Learning techniques*. Ph. D. Dissertation.
[162] Tejas D Kulkarni, Karthik R Narasimhan, Ardavan Saeedi, and Joshua B Tenenbaum. 2016. Hierarchical deep reinforcement learning: integrating temporal abstraction and intrinsic motivation. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*. 3682–3690.
[163] Vitaly Kurin, Saad Godil, Shimon Whiteson, and Bryan Catanzaro. 2020. Can Q-learning with graph networks learn a generalizable branching heuristic for a SAT solver? *Advances in Neural Information Processing Systems* 33 (2020), 9608–9621.
[164] Nada Lavrac and Saso Dzeroski. 1994. Inductive Logic Programming.. In *WLP*. Springer, 146–160.
[165] Nada Lavrac, Branko Kavsek, Peter Flach, and Ljupco Todorovski. 2004. Subgroup discovery with CN2-SD. *J. Mach. Learn. Res.* 5, 2 (2004), 153–188.
[166] Lingyi Liu, Chen-Hsuan Lin, and Shobha Vasudevan. 2012. Word level feature discovery to enhance quality of assertion mining. In *2012 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 210–217.
[167] Donald W Loveland. 2016. *Automated theorem proving: A logical basis*. Elsevier.
[168] Codasip Ltd. Accessed: May-2022. Codix Cobalt Processor Specification. https://riscv.org/wp-content/uploads/2016/07/Tue1430_RISC-V_Codasip-SecureRF_2016-07-12.pdf.
[169] Jinpeng Lv, Priyank Kalla, and Florian Enescu. 2013. Efficient Gröbner basis reductions for formal verification of Galois field arithmetic circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32, 9 (2013), 1409–1420.
[170] Yuzhe Ma, Haoxing Ren, Brucek Khailany, Harbinder Sikka, Lijuan Luo, Karthikeyan Natarajan, and Bei Yu. 2019. High performance graph convolutional networks with applications in testability analysis. In *Proceedings of the 56th Annual Design Automation Conference 2019*. 1–6.
[171] Chris A Mack. 2011. Fifty years of Moore's law. *IEEE Transactions on semiconductor manufacturing* 24, 2 (2011), 202–207.
[172] Alireza Mahzoon, Daniel Große, and Rolf Drechsler. 2018. PolyCleaner: clean your polynomials before backward rewriting to verify million-gate multipliers. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–8.
[173] Alireza Mahzoon, Daniel Große, and Rolf Drechsler. 2019. RevSCA: Using reverse engineering to bring light into backward rewriting for big and dirty multipliers. In *Proceedings of the 56th Annual Design Automation Conference 2019*. 1–6.
[174] Djordje Maksimovic, Andreas Veneris, and Zissis Poulos. 2015. Clustering-based revision debug in regression verification. In *2015 33rd IEEE International Conference on Computer Design (ICCD)*. IEEE, 32–37.
[175] Sharad Malik and Lintao Zhang. 2009. Boolean satisfiability from theoretical hardness to practical success. *Commun. ACM* 52, 8 (2009), 76–82.
[176] Biruk Mammo, Milind Furia, Valeria Bertacco, Scott Mahlke, and Daya S Khudia. 2016. BugMD: Automatic mismatch diagnosis for bug triaging. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–7.
[177] Eman El Mandouh, Laila Maher, Moutaz Ahmed, Yasmin ElSharnoby, and Amr G. Wassal. 2018. Guiding Functional Verification Regression Analysis Using Machine Learning and Big Data Methods. In *Proceedings of the Design and Verification Conference and Exhibition, Europe (DVCon)*.

[178] Cathy May, Ed Silha, Rick Simpson, Hank Warren, and CORPORATE International Business Machines, Inc. 1994. *The PowerPC Architecture: A specification for a new family of RISC processors*. Morgan Kaufmann Publishers Inc.
[179] Norman Megill and David A Wheeler. 2019. *Metamath: a computer language for mathematical proofs*. Lulu. com.
[180] Ninareh Mehrabi, Fred Morstatter, Nripsuta Saxena, Kristina Lerman, and Aram Galstyan. 2021. A survey on bias and fairness in machine learning. *ACM computing surveys (CSUR)* 54, 6 (2021), 1–35.
[181] Boris Mirkin. 1996. *Mathematical classification and clustering*. Vol. 11. Springer Science & Business Media.
[182] Gordon E Moore. 1998. Cramming more components onto integrated circuits. *Proc. IEEE* 86, 1 (1998), 82–85.
[183] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. 2001. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th annual Design Automation Conference*. 530–535.
[184] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.
[185] Prasita Mukherjee and Tiark Rompf. 2021. A GNN Based Approach to LTL Model Checking. *arXiv preprint arXiv:2110.14824* (2021).
[186] M Saqib Nawaz, Moin Malik, Yi Li, Meng Sun, and M Lali. 2019. A survey on theorem provers in formal methods. *arXiv preprint arXiv:1912.03028* (2019).
[187] Alex Nichol, Joshua Achiam, and John Schulman. 2018. On first-order meta-learning algorithms. *arXiv preprint arXiv:1803.02999* (2018).
[188] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* (2022).
[189] Tobias Nipkow, Markus Wenzel, and Lawrence C Paulson. 2002. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer.
[190] Eric Obana. 2023. Closing Functional Coverage With Deep Reinforcement Learning: A Compression Encoder Example. In *Proceedings of the Design and Verification Conference and Exhibition, US (DVCon)*.
[191] Andreas Olofsson. 2017. Intelligent Design of Electronic Assets (IDEA) & Posh Open Source Hardware (POSH). (2017). https://www.darpa.mil/attachments/eri_design_proposers_day.pdf.
[192] Online. Accessed: March-2024. OpenSPARC. https://www.oracle.com/servers/technologies/opensparc.html.
[193] OpenAI. 2021. OpenAI CodeX. https://openai.com/blog/openai-codex.
[194] OpenAI. 2023. GPT-4. https://openai.com/research/gpt-4.
[195] OpenCores. Accessed: May-2022. OpenCores Benchmarks. https://opencores.org/.
[196] OpenROAD. Accessed: Sept-2022. *The OpenROAD Project*. https://theopenroadproject.org/
[197] Muhammad Osama, Anton Wijs, and Armin Biere. 2021. SAT solving with GPU accelerated inprocessing. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 133–151.
[198] Emils Ozolins, Karlis Freivalds, Andis Draguns, Eliza Gaile, Ronalds Zakovskis, and Sergejs Kozlovics. 2021. Goal-Aware Neural SAT Solver. *arXiv preprint arXiv:2106.07162* (2021).
[199] Ganapathy Parthasarathy, Saurav Nanda, Parivesh Choudhary, and Pawan Patil. 2021. SpecToSVA: Circuit Specification Document to SystemVerilog Assertion Translation. In *Second Document Intelligence Workshop at KDD*.
[200] Ganapathy Parthasarathy, Aabid Rushdi, Parivesh Choudhary, Saurav Nanda, Malan Evans, Hansika Gunasekara, and Sridhar Rajakumar. 2022. RTL Regression Test Selection using Machine Learning. In *2022 27th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 281–287.
[201] David A. Patterson and John L. Hennessy. 2016. *Computer Organization and Design: The Hardware Software Interface ARM Edition* (1st ed.). Morgan Kaufmann Publishers Inc.
[202] Hammond Pearce, Benjamin Tan, and Ramesh Karri. 2020. Dave: Deriving automatically verilog from english. In *Proceedings of the 2020 ACM/IEEE Workshop on Machine Learning for CAD*. 27–32.
[203] Dan Pelleg and Andrew W Moore. 2000. X-means: Extending K-means with Efficient Estimation of the Number of Clusters. In *Proceedings of the Seventeenth International Conference on Machine Learning*. 727–734.
[204] Georgios Petasis, Georgios Paliouras, Vangelis Karkaletsis, Constantine Halatsis, and Constantine D Spyropoulos. 2004. e-GRIDS: Computationally Efficient Gramatical Inference from Positive Examples. *Grammars* 7 (2004), 69–110.
[205] Andrew Piziali. 2007. *Functional verification coverage measurement and analysis*. Springer Science & Business Media.
[206] Plasma. Accessed: June-2022. Plasma/MIPS CPU. http://www.opencores.com/project,plasma.
[207] Zissis Poulos, Yu-Shen Yang, and Andreas Veneris. 2013. A failure triage engine based on error trace signature extraction. In *2013 IEEE 19th International On-Line Testing Symposium (IOLTS)*. IEEE, 73–78.
[208] 71 IBEX project contributors. 2016. IBEX RTL source. https://github.com/lowRISC/ibex.
[209] Sanjay Rajashekar. 2020. *A study on Machine Learning-based Hardware Bug Localization*. Ph. D. Dissertation.
[210] Chandan K Reddy and Bhanukiran Vinzamuri. 2018. A survey of partitional and hierarchical clustering algorithms. In *Data clustering*. Chapman and Hall/CRC, 87–110.

[211] Pengzhen Ren, Yun Xiao, Xiaojun Chang, Po-Yao Huang, Zhihui Li, Brij B Gupta, Xiaojiang Chen, and Xin Wang. 2021. A survey of deep active learning. *ACM computing surveys (CSUR)* 54, 9 (2021), 1–40.

[212] Daniela Ritirc, Armin Biere, and Manuel Kauers. 2017. Column-wise verification of multipliers using computer algebra. In *2017 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 23–30.

[213] Lauro Rizzatti. 2021. Hardware Emulation Embraces Machine Learning. https://www.eeweb.com/hardware-emulation-embraces-machine-learning/.

[214] Edgar Romero, Raul Acosta, Marius Strum, and Wang Jiang Chau. 2009. Support vector machine coverage driven verification for communication cores. In *2009 17th IFIP International Conference on Very Large Scale Integration (VLSI-SoC)*. IEEE, 147–152.

[215] Rajarshi Roy, Chinmay Duvedi, Saad Godil, and Mark Williams. 2018. Deep predictive coverage collection. In *Proceedings of the Design and Verification Conference and Exhibition, US (DVCon)*.

[216] Pasquale Davide Schiavone, Francesco Conti, Davide Rossi, Michael Gautschi, Antonio Pullini, Eric Flamand, and Luca Benini. 2017. Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for Internet-of-Things applications. In *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*. IEEE, 1–8.

[217] Frank Schirrmeister, Pete Hardee, Larry Melling, Amit Dua, and Moshik Rubin. Accessed: July-2022. *Next Generation Verification for the Era of AI/ML and 5G*. https://dvcon-proceedings.org/document/next-generation-verification-for-the-era-of-ai-ml-and-5g/ Design and Verification Conference and Exhibition, US (DVCon), 2020.

[218] Stephan Schulz. 2002. E–a brainiac theorem prover. *Ai Communications* 15, 2-3 (2002), 111–126.

[219] Stephan Schulz, Simon Cruanes, and Petar Vukmirović. 2019. Faster, higher, stronger: E 2.3. In *Automated Deduction–CADE 27: 27th International Conference on Automated Deduction, Natal, Brazil, August 27–30, 2019, Proceedings 27*. Springer, 495–507.

[220] Erik Seligman, Tom Schubert, and MV Achutha Kiran Kumar. 2015. *Formal verification: an essential toolkit for modern VLSI design*. Morgan Kaufmann.

[221] Daniel Selsam and Nikolaj Bjørner. 2019. Guiding high-performance SAT solvers with unsat-core predictions. In *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 336–353.

[222] Daniel Selsam, Matthew Lamm, B Benedikt, Percy Liang, Leonardo de Moura, David L Dill, et al. 2018. Learning a SAT Solver from Single-Bit Supervision. In *International Conference on Learning Representations*.

[223] David Sheridan, Lingyi Liu, Hyungsul Kim, and Shobha Vasudevan. 2014. A coverage guided mining approach for automatic generation of succinct assertions. In *2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems*. IEEE, 68–73.

[224] Feng Shi, Chonghan Lee, Mohammad Khairul Bashar, Nikhil Shukla, Song-Chun Zhu, and Vijaykrishnan Narayanan. 2021. Transformer-based Machine Learning for Fast SAT Solvers and Logic Synthesis. *arXiv preprint arXiv:2107.07116* (2021).

[225] Aebel Joe Shibu and Pratyush Kumar. 2021. VeRLPy: Python Library for Verification of Digital Designs with Reinforcement Learning. In *The First International Conference on AI-ML-Systems*. 1–7.

[226] Hongsup Shin. 2019. Case study: Real-world machine learning application for hardware failure detection. https://www.semanticscholar.org/paper/Case-study%3A-Real-world-machine-learning-application-Shin/34d6b71fc28975843f019a1222e593c589f8733b.

[227] Hongsup Shin. Accessed: March-2022. Efficient Bug Discovery with Machine Learning for Hardware Verification. https://community.arm.com/arm-research/b/articles/posts/efficient-bug-discovery-with-machine-learning-for-hardware-verification/.

[228] Xujie Si, Aaditya Naik, Hanjun Dai, Mayur Naik, and Le Song. 2020. Code2inv: A deep learning framework for program verification. In *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part II 32*. Springer, 151–164.

[229] Siemens. Accessed: March-2022. Machine Learning to Accelerate Electronic Design. https://webinars.sw.siemens.com/machine-learning-to-accelerate/.

[230] Siemens. Accessed: March-2022. Solido Characterization Suite. https://resources.sw.siemens.com/en-US/white-paper-addressing-library-characterization-and-verification-challenges-using-ml.

[231] Siemens. Accessed: March-2022. Solido Variation Designer. https://eda.sw.siemens.com/en-US/ic/solido/variation-designer/.

[232] Suraj Singireddy, Rickard Ewetz, and Sumit Jha. 2022. Deep Learning Toolkit-Driven Equivalence Checking of Flow-Based Computing Systems. In *2022 IEEE 4th International Conference on Artificial Intelligence Circuits and Systems (AICAS)*. IEEE, 50–53.

[233] Stan Sokorac. 2017. Optimizing random test constraints using machine learning algorithms. In *Proceedings of the Design and Verification Conference and Exhibition, US (DVCon)*.

[234] Chris Spear. 2008. *SystemVerilog for verification: a guide to learning the testbench language features*. Springer Science & Business Media.
[235] Suvrit Sra, Sebastian Nowozin, and Stephen J Wright. 2012. *Optimization for machine learning*. Mit Press.
[236] Jeffrey X Su, David L Dill, and Clark W Barrett. 1996. Automatic generation of invariants in processor verification. In *Formal Methods in Computer-Aided Design: First International Conference, FMCAD'96 Palo Alto, CA, USA, November 6–8, 1996 Proceedings 1*. Springer, 377–388.
[237] Chengnian Sun and Siau-Cheng Khoo. 2013. Mining succinct predicated bug signatures. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 576–586.
[238] Yang Sun and Spencer Millican. 2019. Test point insertion using artificial neural networks. In *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 253–258.
[239] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction*. MIT press.
[240] Synopsys. Accessed: July-2023. The Synopsys VCS functional verification. https://www.synopsys.com/verification/simulation/vcs.html.
[241] Synopsys. Accessed: June-2023. Formality Equivalence Checking. https://www.synopsys.com/implementation-and-signoff/signoff/formality-equivalence-checking.html.
[242] Synopsys. Accessed: March-2022. VC Formal. https://www.synopsys.com/verification/static-and-formal-verification/vc-formal.html.
[243] Synopsys. Accessed: March-2022. VC SpyGlas RTL Static Signoff Platform. https://news.synopsys.com/2020-02-27-Synopsys-Announces-Next-Generation-VC-SpyGlass-RTL-Static-Signoff-Platform.
[244] Serdar Tasiran and Kurt Keutzer. 2001. Coverage metrics for functional validation of hardware designs. *IEEE Design & Test of Computers* 18, 4 (2001), 36–45.
[245] Scott Taylor, Michael Quinn, Darren Brown, Nathan Dohm, Scot Hildebrandt, James Huggins, and Carl Ramey. 1998. Functional verification of a multiple-issue, out-of-order, superscalar Alpha processor—the DEC Alpha 21264 microprocessor. In *Proceedings of the 35th Annual Design Automation Conference*. 638–643.
[246] Andy Truong, Daniel Hellström, Harry Duque, and Lars Viklund. 2018. Clustering and Classification of UVM Test Failures Using Machine Learning Techniques. In *Proceedings of the Design and Verification Conference and Exhibition, Europe (DVCon)*.
[247] Jason Twigg, Erik Torkelson, and Nazanin Mansouri. 2021. Predicting Formal Verification Resource Needs (Computation Time and Memory) through Machine Learning. *Journal of Student Research* 10, 4 (2021).
[248] Srikanth Vadanaparthi, Pooja Ganesh, Dharmesh Mahay, and Malay Ganai. 2023. Accelerating Functional Verification Through Stabilization of Testbench Using AI/ML. In *Proceedings of the Design and Verification Conference and Exhibition, US (DVCon)*.
[249] CAJ Van Eijk. 2000. Sequential equivalence checking based on structural similarities. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 19, 7 (2000), 814–819.
[250] Jesper E Van Engelen and Holger H Hoos. 2020. A survey on semi-supervised learning. *Machine Learning* 109, 2 (2020), 373–440.
[251] Rajesh Koti Mourya Vangara, Bhaskar Kakani, and Sandeep Vuddanti. 2021. An Analytical Study on Machine Learning Approaches for Simulation-Based Verification. In *2021 IEEE International Conference on Intelligent Systems, Smart and Green Technologies (ICISSGT)*. IEEE, 197–201.
[252] B Samhita Varambally and Naman Sehgal. 2020. Optimising Design Verification Using Machine Learning: An Open Source Solution. *arXiv preprint arXiv:2012.02453* (2020).
[253] Shobha Vasudevan, Wenjie Joe Jiang, David Bieber, Rishabh Singh, C Richard Ho, Charles Sutton, et al. 2021. Learning Semantic Representations to Verify Hardware Designs. *Advances in Neural Information Processing Systems* 34 (2021).
[254] Shobha Vasudevan, David Sheridan, Sanjay Patel, David Tcheng, Bill Tuohy, and Daniel Johnson. 2010. Goldmine: Automatic assertion generation using data mining and static analysis. In *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*. IEEE, 626–629.
[255] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).
[256] VerifAI. Accessed: July-2023. TestGuru generates Tests, Writes Code, Explains Code And Runs Tests. https://testguru.ai/.
[257] Ilya Wagner, Valeria Bertacco, and Todd Austin. 2007. Microprocessor verification via feedback-adjusted Markov models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26, 6 (2007), 1126–1138.
[258] Ahmed Wahba, Justin Hohnerlein, and Farhan Rahman. 2019. Expediting Design Bug Discovery in Regressions of x86 Processors Using Machine Learning. In *2019 20th International Workshop on Microprocessor/SoC Test, Security and Verification (MTV)*. IEEE, 1–6.
[259] Chung-An Wang, Chiao-Hua Tseng, Chia-Cheng Tsai, Tung-Yu Lee, Yen-Her Chen, Chien-Hsin Yeh, Chia-Shun Yeh, and Chin-Tang Lai. 2022. Two-stage framework for corner case stimuli generation Using Transformer and

Reinforcement Learning. In *Proceedings of the Design and Verification Conference and Exhibition, US (DVCon)*.

[260] Fanchao Wang, Hanbin Zhu, Pranjay Popli, Yao Xiao, Paul Bodgan, and Shahin Nazarian. 2018. Accelerating coverage directed test generation for functional verification: A neural network-based framework. In *Proceedings of the 2018 on Great Lakes Symposium on VLSI*. 207–212.

[261] Li-C Wang. 2015. Data mining in functional test content optimization. In *The 20th Asia and South Pacific Design Automation Conference*. IEEE, 308–315.

[262] Li-C Wang. 2016. Experience of data analytics in EDA and test—principles, promises, and challenges. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 36, 6 (2016), 885–898.

[263] Mingzhe Wang and Jia Deng. 2020. Learning to prove theorems by learning to generate theorems. *Advances in Neural Information Processing Systems* 33 (2020), 18146–18157.

[264] Robert Wille, Daniel Große, Lisa Teuber, Gerhard W Dueck, and Rolf Drechsler. 2008. RevLib: An online resource for reversible functions and reversible circuits. In *38th International Symposium on Multiple Valued Logic (ismvl 2008)*. IEEE, 220–225.

[265] Haoze Wu. 2017. Improving sat-solving with machine learning. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. 787–788.

[266] Nan Wu, Yingjie Li, Cong Hao, Steve Dai, Cunxi Yu, and Yuan Xie. 2023. Gamora: Graph learning based symbolic reasoning for large-scale boolean networks. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[267] Nan Wu and Yuan Xie. 2022. A survey of machine learning for computer architecture and systems. *ACM Computing Surveys (CSUR)* 55, 3 (2022), 1–39.

[268] Yuhuai Wu, Albert Qiaochu Jiang, Wenda Li, Markus Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. 2022. Autoformalization with large language models. *Advances in Neural Information Processing Systems* 35 (2022), 32353–32368.

[269] Ruiyang Xu and Karl Lieberherr. 2022. On-the-Fly Model Checking with Neural MCTS. In *NASA Formal Methods: 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24–27, 2022, Proceedings*. Springer, 557–575.

[270] Jingkang Yang, Kaiyang Zhou, Yixuan Li, and Ziwei Liu. 2021. Generalized out-of-distribution detection: A survey. *arXiv preprint arXiv:2110.11334* (2021).

[271] Saeyang Yang. 1991. *Logic synthesis and optimization benchmarks user guide: version 3.0*. Citeseer.

[272] Emre Yolcu and Barnabás Póczos. 2019. Learning local search heuristics for boolean satisfiability. *Advances in Neural Information Processing Systems* 32 (2019).

[273] Cunxi Yu, Walter Brown, Duo Liu, André Rossi, and Maciej Ciesielski. 2016. Formal verification of arithmetic circuits by function extraction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35, 12 (2016), 2131–2142.

[274] Lantao Yu, Weinan Zhang, Jun Wang, and Yong Yu. 2017. Seqgan: Sequence generative adversarial nets with policy gradient. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 31.

[275] Hongce Zhang, Weikun Yang, Grigory Fedyukovich, Aarti Gupta, and Sharad Malik. 2020. Synthesizing environment invariants for modular hardware verification. In *Verification, Model Checking, and Abstract Interpretation: 21st International Conference, VMCAI 2020, New Orleans, LA, USA, January 16–21, 2020, Proceedings 21*. Springer, 202–225.

[276] Wenjie Zhang, Zeyu Sun, Qihao Zhu, Ge Li, Shaowei Cai, Yingfei Xiong, and Lu Zhang. 2020. NLocalSAT: Boosting local search with solution prediction. *arXiv preprint arXiv:2001.09398* (2020).

[277] Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. 2021. miniF2F: a cross-system benchmark for formal Olympiad-level mathematics. In *International Conference on Learning Representations*.

[278] Weijun Zhu, Huanmei Wu, and Miaolei Deng. 2019. LTL model checking based on binary classification of machine learning. *IEEE access* 7 (2019), 135703–135719.

[279] Fuzhen Zhuang, Zhiyuan Qi, Keyu Duan, Dongbo Xi, Yongchun Zhu, Hengshu Zhu, Hui Xiong, and Qing He. 2020. A comprehensive survey on transfer learning. *Proc. IEEE* 109, 1 (2020), 43–76.