ROMANIAN JOURNAL OF INFORMATION SCIENCE AND TECHNOLOGY
Volume 24, Number 1, 2021, 99-116

Machine Learning Techniques for Improving the Performance Metrics of Functional Verification

Mihai-Corneliu CRISTESCU

Politehnica University of Bucharest

Email: mihai.cristescu2303@stud.etti.upb.ro

Abstract. In this paper, it is explored how *machine learning techniques* help improve the *performance metrics of functional verification*. The article outlines a landscape of the current practice, underlines some of the most prominent solutions, compares different approaches, and locates other possible synergy points. In the end, a personal vision is expressed for future research directions.

Key-words: Artificial Intelligence, Machine Learning, Data Mining, Support Vector Machine, Evolutionary Programming, Functional Verification, Formal Verification, Intelligent Verification, Automated Code Generation, Automated Requirements Extraction, Bug-oriented Verification, Automated Coverage Feedback, Automated Bug Detection, Intelligent Proving Theory.

1. Introduction

During the development process of VLSI (Very Large-Scale Integration) systems, the phase of functional verification continues to be the pre-silicon bottleneck that drags the end-product time to market [1].

Two of the most widely used ASIC (Application-Specific Integrated Circuit) verification paradigms are simulation-based verification and formal verification. The former strategy requires generating several input vectors for the DUV (Design Under Verification), reason for which the verification closure can be typically reached with a high degree of human effort. The latter strategy is less tedious to apply on simple circuits, but it becomes computationally impractical with usual, complex systems [2].

To reach the verification closure more quickly, during the last decade, several automation frameworks that use ML (Machine Learning) algorithms have been proposed across the research community. These techniques are generally identified under the field of IV (Intelligent Verification). Depending on the nature of the "signal" or the "feedback" to the learning system, there are different ML approaches that can be involved [3]:

- Supervised learning, where the system learns a general rule;
- Unsupervised learning, where the system discovers hidden patterns in the data;
- Reinforcement learning, where the system tries its best to accomplish a mission.

2. The Typical Framework Structure

For increasing different performance metrics, supervised learning methods are the most widely adopted in many fast-growing domains, like medical diagnosis, legal assistance, and financial prediction [4]. The typical supervised ML flow that can enhance an EDA (Electronic Design Automation) tool is presented in Figure 1.

After pre-formatting (i.e. Data Normalization) the database structures (i.e. Coverage DB, Trace DB) extracted from the VE (Verification Environment), the obtained training set is input to the supervised learning algorithm, during the training phase. This way, the framework creates a model that captures a target behavior expected to improve at least one performance metric, like reducing the simulation runtime. Afterward, this model can be validated using a test set. The next phase of the process is the inference phase, where the obtained model generates responses for new, previously unseen data patterns. This way, the process of reaching a target functional verification objective is improved or even optimized.

One of the main concerns of this framework is a large training time. Concretely, having a high data volume, the training phase would require way too many days to complete, which is computationally infeasible. Using server farms is also impractical when the training examples are computed based on very large simulation traces that need to be downloaded from these data centers. Also, it is time-expensive to store a full dataset on disks (i.e. use a cache functionality) after each algorithm iteration.

Thus, for building reliable learning models for EDA tools, the framework must be provided with enough training examples, and the learning processes must be developed in an "on-site" fashion (i.e. learn and refine the model using proprietary customer data) [5].

Considering all the mentioned observations, the article [5] points out the following IV trends that are meant to deliver faster functional verification closure:

- Automatic code generation (i.e. specification and constraint mining), where the IV framework recommends assertions and constraints to engineers, and thus reduces the VE implementation time.
- Feedback-based coverage-directed test generation, where the stimuli are automatically adjusted by some coverage model feedback.
- Automated failure root-cause debug, where the framework points out more detailed information about the identified functional bugs.
- Speed-up formal verification processes, where the theorem proving task can be enhanced with an automatic solver mechanism.

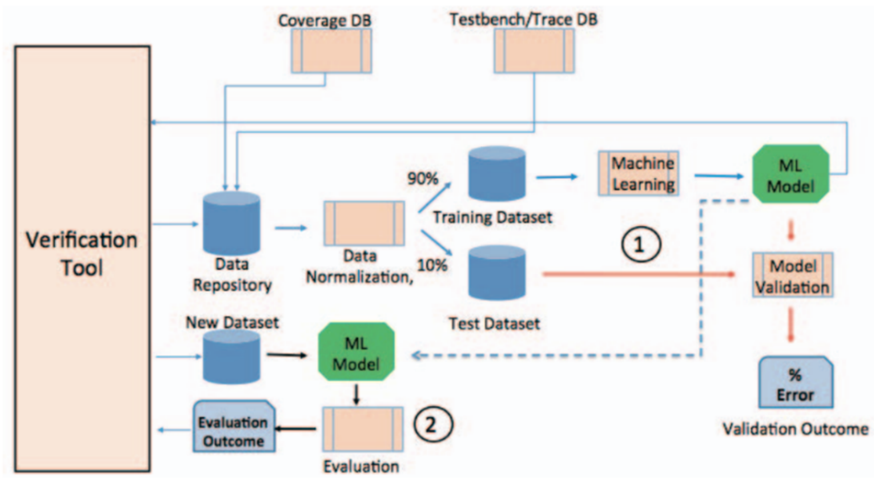


Fig. 1. Supervised Machine Learning Flow for Intelligent Verification [5]

These strategies will be detailed in the following sections. Precisely, the first three strategies target the simulation-based verification paradigm, while the latter one targets the formal verification paradigm.

3. Intelligent Verification Strategies

3.1. Automated Code Generation using Intelligent Requirements Extraction

A possibility for shortening the time frame towards functional verification closure is adding ML techniques in one of the early stages of the ASIC development process. Specifically, when interpreting the system's requirement specification. Today's most widely used approach is to manually extract the functional specification from the requirement document, which can be tedious and prone to unintentional human errors. Because of these weak points, there is a considerable research effort in trying to automate this step [6].

In the article [7], the authors are presenting a framework that intelligently extracts semantic information from an SRS (Software Requirements Specification). In the first step, semantic frames are defined by extracting and grouping similar verbs identified in the document. In a second step, manual sentence annotation is performed, so that important features can be outlined. Afterward, the feature set can be extracted from each semantic frame for generating the training set. In the next step, a custom function is used to correct the parsing results (generated with the Stanford Parser [8]) so that there is no domain ontology conflict. In the last step, the obtained decision tree is used to build semantic role labels that are convenient in different implementation stages of the software solution.

The authors of the aforementioned paper applied this learning process and analyzed 18 requirement specifications that comply with the IEEE-STD-830 standard [9]. After that, 212 sentences were selected from a total of 2635 sentences, and thus chose 20 most frequently used verbs to define 9 semantic frames. Manual annotation was required for some sentences. Only a preliminary experiment was deployed that indicated a recall of above 90% for identifying the semantic roles. As a result, for extracting the data model from an SRS, the document should not use restrictive or controlled natural language, but rather consider domain-specific semantic role labeling to extract semantic information from the SRS in free-text documents.

A paper that also explores semantic information extraction is presented in reference [10], but in this approach, the objective is to generate SVAs (System Verilog Assertion) by learning from either the specification documentation or from the requirements specification. An NLP (Natural Language Processing) algorithm called GLA (Grammatical Inference Algorithm) is proposed for learning a target document written in a human-readable language. Another key difference is that an intermediate specification model is used as a bridge between the natural language requirements and the executable, high-level design implementation. For this use case, it can be observed that supervised methods are more accurate than unsupervised ones because the former ones have some knowledge of how valid grammar structures look like (i.e. it is trained on specification sentence samples), but good grammar is not available in many cases. Because of this constraint, the literature mainly focuses on unsupervised algorithms that use positive learning examples. Practically, the proposed algorithm explores the search space of the grammars and selects the optimal one using the MDL (Minimum Description Length) strategy. The solution is pointed by the final selected grammar (i.e. the most compact grammar) that captures the generalized structure of the sentences in the learning set and maps them to the SVAs patterned after those found in the learning set.

In the experiment of the article [10], 88 sentences are extracted from the AMBA APB protocol. These sentences are preprocessed to remove pronoun references. Out of these, 17 sentences are used for creating the learning set (i.e. selected the most representative sentences, in terms of structure and word content) together with their manually implemented SVAs, while the rest of 71 sentences are used as the cross-validation set. After inference, the target 71 new SVAs are automatically generated with a semantically correct value.

A similar solution is outlined in reference [5]. The proposed objective is to automate the troubleshooting task using expert systems. Concretely, the engineer states an assertion violation, and the NLP system interactively gains more knowledge from the user.

One major lowlight of the method outlined in the article [7] is the manual effort required for annotating the selected sentences during the second step. An apparent strong point of this framework is its flexibility and good performance results when applied on a large variety of documents that comply with the standardized recommendations [9]. However, the majority of today's development teams do not precisely follow these recommendations, and consequently add more custom rules. This fact reveals an even weaker point for this framework.

One lowlight of the framework pointed by the article [10] is the manual task of choosing the learning set. This task can become very tedious if the learning algorithm requires a large training set. The intermediate specification model ensures a better translation quality when extracting meaning, which is a strong point. However, this architectural decision heavily affects the runtime.

The major strong point of the framework described in the article [5] is that the obtained learning model can suggest constraint updates, which is a more high-level VE implementation automation compared to SVA generation. This is a less straightforward approach, but a more flexible solution that can adjust the VE's parameters. Unfortunately, the strategy implementation is quite complex and heavily dependent on functional verification domain knowledge.

Because of the weak points mentioned in the above paragraphs, the approach of 'automatic code generation with intelligent requirements extraction' should benefit from more research effort in attempting to automate sentence annotation and training examples selection.

3.2. Automated, Feedback-based, Coverage-directed Test Generation

Another possibility for reaching functional verification closure more quickly is by reducing the overall simulation time of the project's regression (i.e. reduce the runtime of the entire test set). Without dropping any significant scenario, the most intuitive optimization would be to reduce the number of redundant stimuli that are generated across a regression.

Today, the majority of the CDV-based (Coverage-Driven Verification) environments have an inefficient scenario generation approach because the stimuli redundancy rate increases together with the coverage levels. This translates in a very low rate of new scenario generation towards the end of the regression, which inefficiently extends the runtime. This issue is depicted in Figure 2.

Because of the lowlight of typical CDV solutions, there is a great research interest to reduce the regression runtime. Theoretically, by using another IV strategy, the VE could adjust the stimuli generator so that redundant scenarios are avoided. Specifically, the stimuli generator could be adjusted using some feedback information received from the coverage model. This relatively new strategy is known as CDTG (Coverage-Directed Test Generation) that aims at efficiently capturing the scenarios required for verification closure. This remarkable highlight makes this approach an enhancement of the typical CDV strategy.

The research outlined the following general ML techniques for enabling performant CDTG solutions:

- Evolutionary Programming
- Bayesian Network
- Markov Chain
- Data Mining
- Inductive Logic Programming

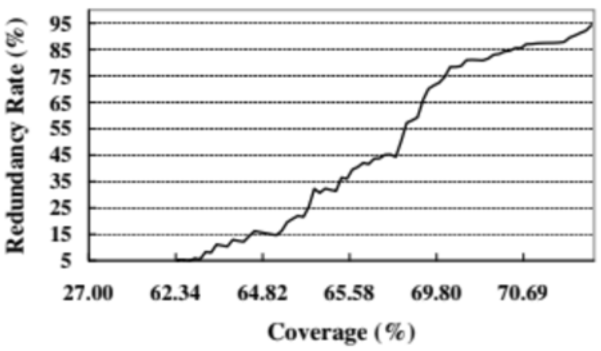


Fig. 2. Increasing the Stimuli Redundancy Rate in Typical CDV Implementations [11]

In the following paragraphs, one can observe a comparison between these ML approaches in the context of CDTG.

3.2.1. Evolutionary Programming

The theory points out that there are two major EP (Evolutionary Programming) techniques, namely GA (Genetic Algorithms) and GP (Genetic Programming). Across the research community, these approaches are regarded as the best ones for CDTG, because such studies indicated the best time reduction towards functional coverage closure [13].

A GA has the disadvantage of requiring manual "trial and error" iterations when setting up a suitable learning model for the target application [13].

In the article [14], a reinforcement learning approach that uses a GA is proposed. The data is represented using a ternary alphabet, and the binary reward scheme is 1000/0 for tests that manage or do not manage to successfully toggle a signal. The authors considered using some third-party utilities, namely "Fire Drill" as the test generator, and "Fire Path" as the DUV.

GP is a technique that requires minimal engineering expertise, in the sense that the underlying framework requires minimal domain knowledge [13].

The investigation revealed a research paper [22] that targets post-silicon fault diagnosis in which powerful GA strategies are involved. Concretely, the solution is based on two methods:

- Attribute reduction, in the context of rough set theory using GA;
- Value reduction techniques.

A good decision behind this implementation decision is that GAs ensure global optimization and have a high degree of parallelism which proved to be key requirements for successful CDTG solutions. Thus, the improved GA features an amendment operator (for accelerating convergence and improving accuracy) and is applied for attribute reduction to obtain a minimal decision table (that contains the rules for power transformer fault diagnosis). For simplicity, swap mutation and one-point crossover are used instead of higher-order crossover methods. The paper outlines two examples in which the framework is used, and the results point to faster convergence compared to similar techniques applied to the same case studies.

3.2.2. Bayesian Network

The BN (Bayesian Network) is one of the probabilistic methods that outlined interesting results for IV applications. In the context of CDTG, a BN can be used to cluster the covered tasks (i.e. covered bins) based on their test similarities, and thus, the model can indicate test directives that are most probable to cover a hole [13].

Many framework improvements have been proposed across the research community. One enhancement is to increase the probability of covering corner-cases by using a "waterfalling algorithm" [13]. Another option is to improve the quality of the generated test directives by employing an "adaptation algorithm" [13]. For reducing stimuli generation failures (i.e. reducing redundant stimuli), some authors considered relating the test sequences with the DUV's initial state. The proposed framework uses a BN for learning the legal state register values and a BN for modeling each instruction of the ISA (Instruction Set Architecture). This way, the time needed for reaching coverage closure was significantly reduced. Also, the set-up time was reduced compared to other similar solutions [13]. Moreover, this presents a way to automate the network's generation [13], but the associated disadvantage is that the framework set-up requires a consistent amount of domain knowledge.

In article [12], the authors propose a framework that can reduce the stimuli generation time using a learning engine based on HLDDs (High-Level Decision Diagram) and EFSMs (Extended Finite State Machine). The HLDD identifies the untestable areas that are useful when generating sequences. The EFSM traverses the DUV's testable areas so that the simulator can hit novel RTL (Register-Transfer Level) transitions [12].

3.2.3. Markov Chain

Another probabilistic method that proved to be efficient in CDTG tasks is the Markov Chain. This approach is useful when there is an interest in maximizing the activity of some DUV signals [13]. However, a major lowlight of this strategy is the requirement of using a template description language [13] for generating the underlying learning model.

3.2.4. Data Mining

DM (Data Mining) is another probabilistic method with some achievements for CDTG. This technique has the advantage of not requiring a tedious amount of domain knowledge, but the disadvantage is that this method performs very badly when presented with new, previously unseen examples [13].

3.2.5. Inductive Logic Programming

Another interesting probabilistic ML approach that enables CDTG is ILP (Inductive Logic Programming) which uses clustering techniques on captured/hit coverage bins for generating rules between the sequences and their associated coverage results [15]. Consequently, the patterns learned from the dataset can be used as test generation constraints [13].

The general highlights of the algorithm are:

- The user does not require ML expertise when setting up the infrastructure [15];
- The algorithm can direct stimuli for covering functional coverage holes (i.e. not only code coverage holes) [13]. In the article [12], the framework based on HLDD and EFSM is indicating improvements in terms of statement coverage and fault coverage.

The extracted machine knowledge is easily understandable by the user, which makes it human-readable [13].

The majority of ILP and GA solutions intended for CDTG are based on unsupervised clustering methods. More precisely, several proposed solutions use 1-class SVM (Support Vector Machine) classifiers because their underlying algorithms allow reaching the coverage goals more quickly. There is one proposed framework that performs on-the-fly stimuli filtering so that the classifier is incrementally updated using coverage results after each run. In this case, the chosen kernel is the RBF (Radial Basis Function) [11]. Another similar solution uses online test selection (required before the simulation) so that the tests are modeled as execution graphs. For measuring similarity, the GED (Graph Edit Distance) metric is used [16].

In the article [17], the authors present a framework that filters out tests by using novelty detection. The coverage results are estimated by monitoring some DUV signals (for each test).

The gain is that the target coverage is obtained by running 100 novel tests instead of running the entire set of 2590 tests, as in the typical CDV approach. Specifically, this paper uses test content optimization, which is a supervised learning algorithm applied to a test-set to generate novel tests.

In contrast to the aforementioned articles that present 1-class SVM solutions, there is an article that proposes a multi-class SVM classifier (i.e. uses a structure with several binary-class classifiers). Its learning phase is based on the SMO (Sequential Minimal Optimization) strategy. The inference phase maps each covered bin to the most representative test-case that generated that scenario.

One of the most notable solutions is described in reference [19]. It presents a supervised SVM with a ternary alphabet for enhanced data representation when clustering tests. Practically, the structure of this framework is engineered for performing novelty detection. During the learning phase, the engine is trained using a test subset. These tests are modeled as bit-vectors that represent the entire sequence of values for some sampled DUV signals throughout the test runtime. During the inference phase, the learning engine acts as a stimuli filter for redundant scenarios. This way, novelty detection is achieved, and these outliers are ranked and run in the order of their expected coverage novelty.

A general observation for CDTG strategies that use SVM is that the associated frameworks feature low dependencies on the training set size. However, the downside is that these implementations do not enable a considerable reduction of the execution time compared to the GA-enabled solutions. [18].

The literature also points to a CDTG approach in which pairs of tests and the associated simulation trace are analyzed for extracting verification knowledge. In this thesis, the objective is to learn some key DUV properties like instruction execution time or the number of internal operations [20].

Compared to the article [18] where the covered bins are being clustered, in the article [21], the SVM is used to cluster the correlated uncovered cross bins. This way, in the latter paper, the functional test filtering component proves to have a higher quality compared to the former approach. Practically, in the first phase, K-means with Jaccard kernel is used on these crosses. In the second phase, K-means with Euclidean kernel is used on previously generated clusters. Thus, the framework automatically identifies the clusters that have low coverage ratios and issue them first for coverage analysis. As a result, the coverage-hole analyzer will have little data to work with, but of higher quality [21].

Based on the analysis, it can be observed that the advantage of ILP compared to GA is that the former can model more complex dependencies, which makes it a more general-use approach. However, the disadvantage of the former method is that it requires a significant amount of domain/background knowledge in terms of functional verification [13]. Moreover, the major strong points of the latter method are the higher performance results of these CDTG implementations.

The research points out that GAs are one of the most promising techniques for enabling CDTG solutions that have a valuable impact on reducing the time towards reaching functional coverage closure.

3.3. Automated Bug Detection with Intelligent Analysis

When debugging a faulty DUV, the main objective is to reduce the investigation time. This can be achieved by using the so-called bug-oriented verification methodology which efficiently regulates the spent resources across all the DUV's submodules.

An early study of this approach started in 2006 with an article [23]. The paper compares the impacts of different ML techniques on improving fault coverage for an ATPG-like (Automatic Test Pattern Generator) framework for post-silicon validation. Eventually, it was observed that this approach can be accustomed to pre-silicon verification as well.

It is notable that the proposed SVM solution has a very good accuracy but is very slow in comparison with the ONN-based (Ordered Nearest Neighbor) algorithm. The latter approach converts the data matrix into an OBDD (Ordered Binary Decision Diagram) which is less computationally intensive. Similarly, an OIR (Ordered Input Reduction) solution is also presented, featuring a good accuracy and a low runtime. These two solutions also outperformed a basic NN (Nearest Neighbor) algorithm. This outcome points that 'ordering data' is a key performance improvement step in Boolean learning.

An important objective at that time was to efficiently compute the inverse matrix of the learned model. Therefore, ARM (Association Rule Mining) was applied to the input data matrix to improve the input ordering even more.

With the last decade's advances in computation speed, SVM solutions are becoming a preferred strategy for automating bug detection tasks in functional verification.

The literature points out the following main directions:

- Analyze the past design revisions;
- Analyze the DUV's simulation traces.

In the following paragraphs, one can acknowledge the functional details of these two approaches as well as a comparison between them.

3.3.1. Analyze Past Design Revisions

One interesting strategy is to estimate the current DUV bug probability distribution by analyzing past revisions of the project. A first such effort is outlined in reference [24] which proposes a bug forecast framework that can indicate the functional bug distribution across the DUV's submodules. The learning engine correlates the design characteristics with the "hidden" bug information.

During the training phase, historical data from previous design derivatives (i.e. the reference revision) is used. The training set is obtained using these preprocessing steps:

- Extract the bug descriptions that model the update history;
- Collect module characteristics such as code snapshots that model either the code complexity or the engineers' experiences.

For successfully performing these steps, a GA selects the most suitable design characteristics (i.e. characteristics that are most closely related to any bug occurrence) out of any noisy design characteristics. Thus, feature selection/filtering is achieved, and a robust training set is created. A supervised method can be used for training the model.

During the inference phase, on the current design revision (i.e. when using description data associated with the current system implementation), the model predicts functional bugs information for each DUV subcomponent.

As supervised learning algorithms, the authors considered comparing an ANN (Artificial Neural Network), an NBC (Naïve Bayes Classifier), an M5P (M5 Model Tree), and an SVM. For

5 different projects (i.e. 5 different use cases), the SVM-based solutions outperformed the rest of the methods.

When preprocessing the training examples before the training phase, it was observed that PCA-constructed (Principal Component Analysis) bug models do not provide better performance results. Fortunately, GA-filtered properties generally performed much better than PCA by selecting more appropriate characteristics for constructing accurate bug models.

The already presented solutions focused on intra-revision predictions, which means that the models were created without using correlation data between different reference revisions. Contrary, other conducted experiments focused on evaluating cross-revision data, (i.e. correlate reference revision bug models) for predicting current revision bugs.

For improved resource allocation, the framework should also predict the bug's severity during the classification phase. However, this requires having a clear severity classification of the recorded reference revision bugs. In this sense, one can use a severity formalism as in some third-party platforms, like BugZilla.

For being able to provide confidence intervals for the prediction results, multiple bug models are required (each made on a different reference revision). Thus, verification closure can be stated when the number of detected bugs exceeds the number of predicted bugs with high confidence (e.g. 95% confidence).

A similar case study is presented in the article [25]. The paper proposes a framework in which debugging is eased by using ML algorithms to rank historical data of revision systems.

Precisely, revisions are classified and then ranked based on their estimation of causing a failure (identified in a regression) and then are sent to their owners/engineers for debug.

In the first phase, the source code updates are evaluated so that suspects are identified by using an SAT-based (Boolean Satisfiability Problem) automated debugger. Clustering is used to represent the perceived number of actual errors in the RTL. Afterward, each suspect is assigned a likelihood of being the cause of each regression failure by using a graphic representation as in Figure 3. Two versions of the same source code file are represented on the axes, where the values represent the line indexes of these source code files. The code which overlaps the most has the highest probability of pointing out the RTL bug, and thus is given a higher-ranking order.

In the second phase, the commit logs are ranked depending on the revision type (i.e. either the revision fixes an RTL bug, or it is just a usual development revision. The latter type of revisions has a much higher probability of inserting an RTL bug, the reason for which they receive a higher-ranking order.

In the third phase, the revisions are ranked once again based on their likelihood of uncovering an RLT-bug.

The used clustering algorithm is AP (Affinity Propagation) instead of K-means because the former one maximizes the similarities between the points. This also uses the Euclidean distance as the similarity metric. The smaller the distance between the AP's exemplar and another point, the more the associated revision will receive a higher ranking.

The revision classification is performed using a binary SVM classifier, where the classes are "bug fix" and "not a bug fix". The SVM is trained using examples of "bug-fix revisions" and "implementation revisions". The author chooses 'M' keywords in 'N' commit logs to construct the features of the training set. If the number of features is much higher than the number of training examples, then the SVM becomes very inefficient. Therefore dimensionality/feature/keyword reduction is required so that N $<$ M/2. After this step, the revisions are ranked for each cluster to maximize the chances of detecting a functional bug.

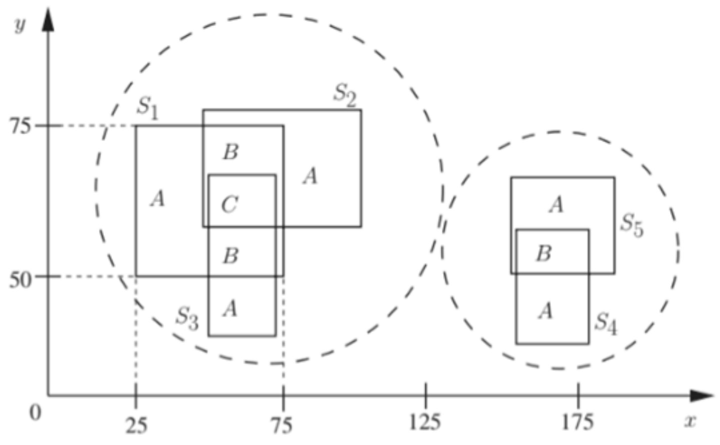


Fig. 3. Suspect impact distribution over different code revisions [25]

If there is a small number of revisions, and the differences between the revisions are very big, then the learning algorithm might become very inefficient due to underfitting.

There are several lowlights associated with these automatic bug detection techniques. First, the strategy is applicable only when a reference revision is available. Secondly, since the revisions are classified based on comments, there are many cases in which the commit messages are absent or contain misleading comments. This is expected to heavily affect the performance of the learning system. Thirdly, because today there is a high competition to deliver better products with even smaller time to market, the development teams consider adding several RTL features in just a couple of design revisions. These significant upgrades imply many repository updates that heavily affect the training phase. Another disadvantage is the following: when identifying a problematic revision, and the framework manages to come up with a fix for it, then that fixed revision needs to be updated with the latest feature implementations that followed afterward, during the development cycle. Typically, these updates are not performed on the main project revision, the reason for which a separate revision that contains these updates must be maintained within that project.

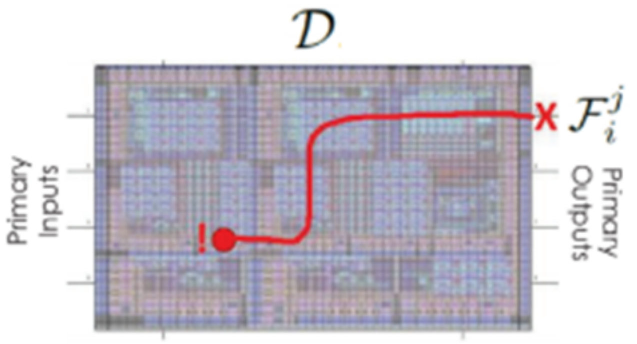
3.3.2. Analyze Simulation Traces

When reporting regression results, typical EDA tools perform error message grouping which is intended to regulate the investigation resources. Technically, this feature outlines the type of checks that fail, but it cannot always indicate the number of RTL bugs because there are situations in which an RTL error triggers different checks, from a run to another, and thus be represented in two or more groups. Other cases are those in which different RTL bugs trigger the same check, and thus, be represented by the same group.

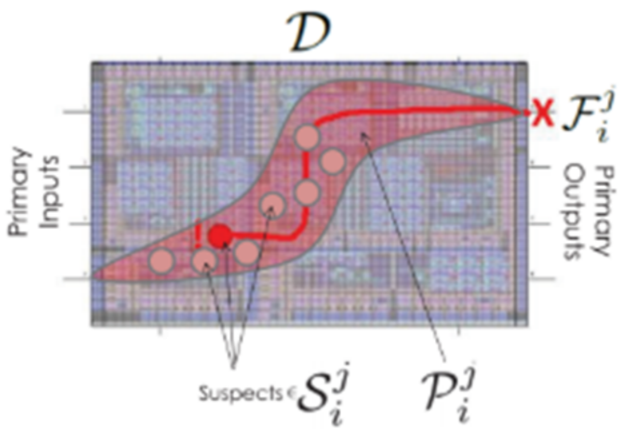
Because of these issues, in many situations, engineers end up investigating the same RTL issue which is very time inefficient. One solution to identify the number of RTL bugs (i.e. root causes) is analyzing the simulation traces.

Article [26] proposes a framework that can indicate a set of suspects (i.e. potentially buggy locations in the RTL) by exploiting error signatures from the error simulation traces. This is achieved by using SAT-based debuggers and clustering algorithms.

In the first phase, the debuggers are fed with erroneous traces (i.e. counter examples), and thus, a set of possible suspects, as well as their error propagation paths, are returned. The effect is depicted in Figure 4.



(a) Failure and actual error with propagation path



(b) Error propagation paths returned by the debugger

Fig. 4. Single failure and the result of root cause analysis [26]

An error proximity metric is used for dividing two target paths is 'W' time windows. This way, the framework estimates the convergence/similarity of the errors.

For performing error clustering, an initial guess concerning the number of distinct errors/-clusters must be made. Specifically, the authors indicate a heuristic that estimates the mutuality of different suspect signatures. These suspect signatures point to which RTL bugs trigger the same error message. Also, the framework uses the Ward method for merging clusters if the current number of clusters is too high.

As a result, for each cluster, the engineers will have the error propagation paths and the suspect frequencies (i.e. for how many different failures is a suspect responsible for).

It was observed that the failure classification accuracy increases together with the number of time windows per error path. Compared to typical automated methods present in the industry, this solution features a 43% reduction of the time cost.

Another article that presents simulation trace investigations is referenced in [27]. The paper presents a framework that clusters segments of simulation traces that can point useful scenario corner-cases.

The first step is to generate a trace dump file by simulating the design (i.e. run a full test set regression). Next, the trace file is processed for identifying interesting scenarios/events, and then it is segmented into chunks. Next, a TF-IDF (Term Frequency - Inverse Document Frequency) segments encoding is performed. This step enables a better segment projection on the learning space. Then the output is fed to an unsupervised K-means++ clustering algorithm that uses the normalized Euclidean distance. The used similarity function is the cosine distance similarity measure. Finally, each segment is assigned to a cluster so that suitable verification engineers can investigate.

For encoding the trace, simulation events are identified in the trace, and using a sliding window of a given width, the trace is divided into trace chunks/segments that will need to be projected into the learning space before being fed to the clustering algorithm.

For identifying 'K', the most suitable number of centroids, empirical assignments and analysis should be performed. By iterating through several combinations, a curve like the one depicted in Figure 5 can be obtained.

The heterogeneity is high if clusters are tight, and it is low if the clusters have some scatter trace segments (i.e. some large distances between points). The most suitable number of clusters is the abscess of the curve's elbow.

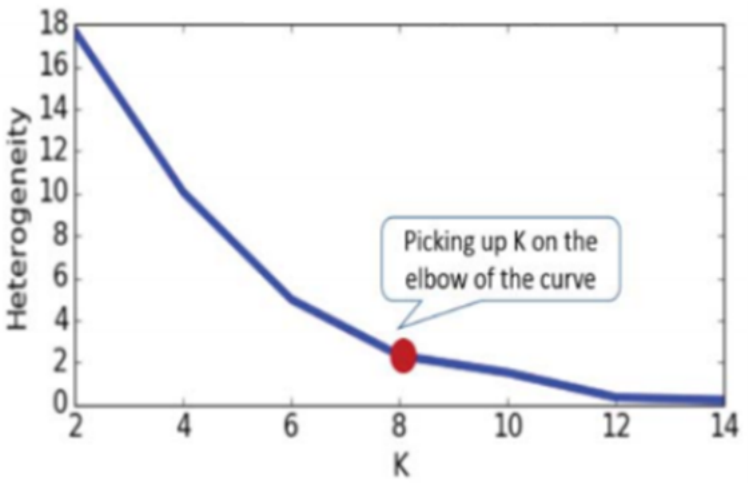


Fig. 5. Identifying K, the most suitable number of centroids [27]

3.4. Intelligent Proving in Formal Verification

At first glance, ML algorithms seem incompatible with formal verification principles, as the former one uses probabilistic models, whereas the latter one uses a discrete exact description of the block's functionality.

Despite these different mathematical models, the literature points to some investigations that outline interesting synergies/themes between ML and formal verification.

Article [28] reviews such themes. The first one is used in SAT Solving which uses some techniques that are derived from the DPLL (Davis–Putnam–Logemann–Loveland) algorithm.

One representative problem/instance in SAT Solving is predicting/optimizing runtime of an instance by switching from an algorithm that becomes inefficient to an algorithm that is more efficient for the current computation task. This involves having suitable computation restart strategies. Thus, the given process is seen as a reinforcement learning problem where the restart strategies are modeled as possible actions to reinforce the agent. The end-user has the difficult task of setting the right parameter values for the chosen strategies. Another researched topic for reducing the execution time in the context of using reinforcement learning is selecting the branching variable.

Therefore, ML methods can be used in FOL (First-Order Logic) TP (Theorem Proving) for selecting only the mathematical facts that improve the performance metrics. This automation strategy is generally called ATP (Automated Theorem Proving). Also, regression tasks are used for evaluating the runtime of different parameter values across different proof heuristics. Instead of using random predefined parameters, it is proved being better when using the values computed by ML feature selection.

Classification and clustering themes can be used as a recommender system for pointing the next proof step to the user.

Reinforcement learning may be used for identifying counterexamples (i.e. model errors) by searching for property invalidation instead of proving model correctness.

For MC (Model Checking), reinforcement learning of frequent itemset mining tasks can be used for finding counterexamples more quickly or for reducing the number of “false positives”.

For SA (Static Analysis), some research has been performed in classifying different data structures. One such use case is suitable for identifying which repository code updates contain bugs or not. Another use case is identifying the actionable alerts for reducing the negative impact of false alarms on the developers’ productivity.

The paper concludes with identifying the general trend, the most challenging being formal verification approaches that have an even higher degree of learning automation.

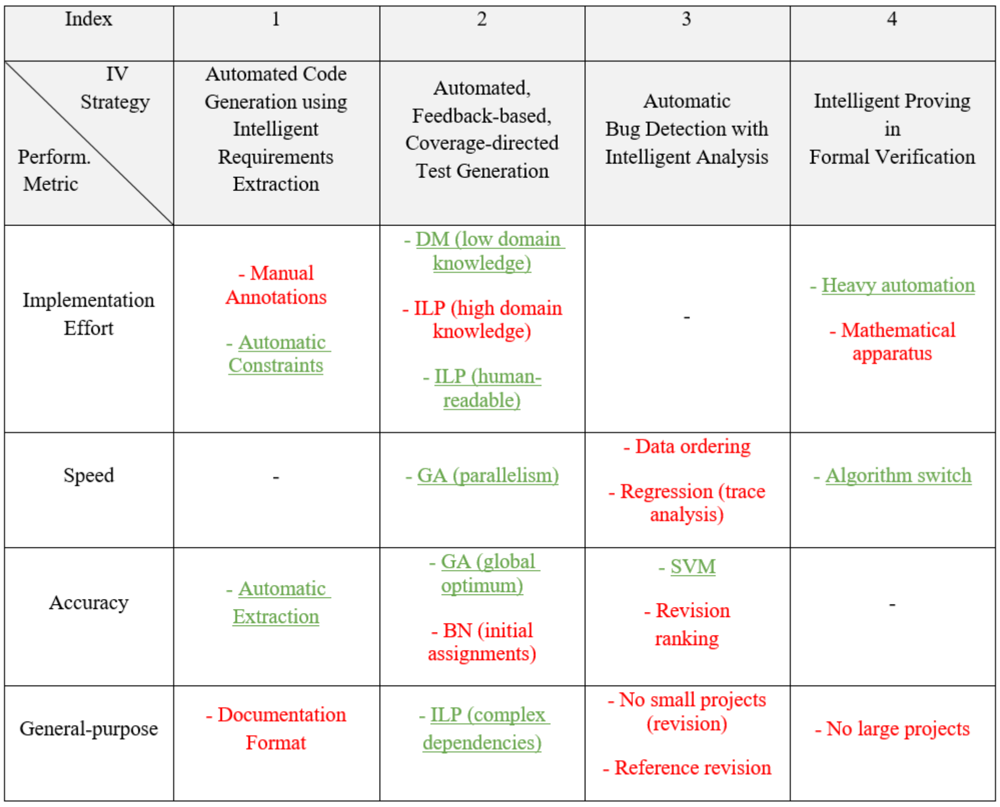
Another interesting article is [29] where the frequent itemset mining problem targets identifying associations between items in a dataset.

The paper presents the example of a grocery store in which such associations can be performed to make informed decisions such as item placement and sale expectations. The paper uses the BMF (Bird-Meertens Formalism) notation in defining the solution of the frequent itemset mining problem. A use case for this mining problem can be found in formal verification as well, and this paper uses a third-party theorem prover called Coq. This engine transforms an inefficient program/specification into an efficient one. This can be achieved with some program transformation techniques (i.e. BMF) that are used for deriving an initial program (i.e. a specification) into a more efficient (optimized) program by eliminating intermediate data structures that are passed between recursive calls (i.e. called fusion transformation). However, the original program semantics are preserved (i.e. the programs are functionally equivalent).

In a similar article pointed by the reference [30], the author performs a sequence of five transformations using a “pen-on-paper” proof, but in this paper, the author is using Coq as a specialized proof assistant. The paper demonstrates the usefulness of Coq in proving the correctness of a practical data mining problem. This way, an automation framework can deliver a robust formal verification ML model.

4. Comparison between the IV Strategies

Based on the details provided in the previous sections, the comparison between the IV strategies is outlined in Figure 6:



Index	1	2	3	4
IV Strategy / Perform. Metric	Automated Code Generation using Intelligent Requirements Extraction	Automated, Feedback-based, Coverage-directed Test Generation	Automatic Bug Detection with Intelligent Analysis	Intelligent Proving in Formal Verification
Implementation Effort	- Manual Annotations - Automatic Constraints	- DM (low domain knowledge) - ILP (high domain knowledge) - ILP (human-readable)	-	- Heavy automation - Mathematical apparatus
Speed	-	- GA (parallelism)	- Data ordering - Regression (trace analysis)	- Algorithm switch
Accuracy	- Automatic Extraction	- GA (global optimum) - BN (initial assignments)	- SVM - Revision ranking	-
General-purpose	- Documentation Format	- ILP (complex dependencies)	- No small projects (revision) - Reference revision	- No large projects

Fig. 6. Comparison of different Intelligent Verification Strategies

For each approach, the highlights are underlined with 'green' while the lowlights are marked in 'red'.

Regarding the implementation effort, it is observed that the second strategy has the most interesting advantages. Practically, DM solutions do not require a tedious amount of domain knowledge. However, ILP requires a considerable amount of domain knowledge, but this approach features human-understandable machine knowledge. For the first strategy, the strong point is the automatic adjustments of VE parameters and constraints, while the weak point is marked by the manual sentence annotation requirement for some solutions. The fourth strategy outlines a heavy automation feature when the project is not very complex, but the complexity of the underlying mathematical apparatus requires a considerable amount of implementation effort.

In terms of solution speed, (i.e. execution time), it can be noticed that GAs are relatively fast due to their parallel processing structure. At the same time, formal methods can benefit from an algorithm switch mechanism that optimizes the execution time. Despite these advantages, the third strategy is relatively slow because data ordering steps are necessary for enabling better accuracy and, for trace analysis solutions, the prerequisite to run a full regression with coverage closure requires a lot of execution time.

Concerning the accuracy of the model, the GAs have the advantage of ensuring global optimization, but the similar BN-based solutions require specific initial state assignment for significant accuracy improvement. The first strategy reduces the probability of unintentional human error due to the automatic requirements extraction. For the fourth category, the lowlight is that the bug models heavily depend on additional steps like historical data ranking, which affects the execution time. Fortunately, many such SVM-based solutions deliver powerful learning models that provide relatively better accuracy.

Lastly, concerning the general use of the discussed strategies, the second one underlines a remarkable advantage: The ILP methods can even model the systems in which complex dependencies are observed in the coverage implementation. Despite this, the other categories outline only negative impacts: The first method is very restrictive because the specification document must comply with the standardized recommendations for NLP. The third category is not suitable for small projects, because these usually expose a smaller number of functional bugs, and the effort of bug resource allocation is no longer feasible. Also, the bug models heavily depend on the quality of the reference revisions, like comments, incremental updates, coding styles, and the chosen hardware description language. The fourth strategy is not feasible for complex projects, because the required execution time of formal methods is known to be extremely large in these cases.

5. Conclusions

The field of IV pinpoints many research opportunities for improving different performance metrics of functional verification. Because the VEs are becoming more and more complex, and the required human effort increases significantly, today's most sought-after optimization in pre-silicon engineering is to achieve verification closure as fast as possible. More precisely, to reduce the coverage closure time. Thus, the performance metric which appears to bring the most important positive impact in this area is the total time spent on verification tasks, which implies both VE implementation time, as well as the strategy's execution time.

After analyzing the proposed techniques, from a personal point of view, there is a considerable success potential in extending the research efforts around the "Coverage-Directed Test Generation" (CDTG) strategy. One investigation direction would be to combine the strong points of both GA and ILP algorithms within a hybrid-like framework. The GA is suitable for improving the quality of the training set, while ILP could be involved in modeling complex coverage models. Also, an SVM engine could assist this set-up with high-quality classification tasks.

References

[1] FINE S., et. al., *Coverage Directed Test Generation for Functional Verification using Bayesian Networks*, IEEE Proceedings 2003, DAC 2003.

[2] WIEMANN A., *Standardized Functional Verification*, Springer 2008.

[3] *Lifting the Fog on Intelligent Verification, SCDSource*, May 2008, Available at: `https://en.wikipedia.org/wiki/Intelligent_verification`

[4] ELLA A., et. al., *Advances in Intelligent Systems and Computing*, Springer 2020, AMLTA 2019.

[5] PANDEY M., *Machine Learning and Systems for Building the Next Generation of EDA tools*, IEEE 2018, ASP-DAC 2018.

[6] GHOSH S., *ARSENAL: Automatic Requirements Specification Extraction from Natural Language*, Springer 2016, NASA Formal Methods 2016.

[7] WANG Y., *Semantic Information Extraction for Software Requirements using Semantic Role Labeling*, IEEE 2016, PIC 2015.

[8] *The Stanford Natural Language Processing Group, Stanford Parser*, August 2020, Available at: `http://nlp.stanford.edu:8080/parser/`

[9] *IEEE Recommended Practice for Software Requirements Specifications*, IEEE Standard 830-1998.

[10] HARRIS C., et. al., *GLAsT: Learning Formal Grammars to Translate Natural Language Specifications into Hardware Assertions*, IEEE 2016, DATE 2016.

[11] GUO Q., et. al., *On-the-Fly Reduction of Stimuli for Functional Verification*, IEEE 2011, ATS 2010.

[12] GUGLIELMO G. D., et. al., *On the validation of embedded systems through functional ATPG*, IEEE 2008, PRIME 2008.

[13] IOANNIDES C., et. al., *Coverage-Directed Test Generation Automated by Machine Learning - A Review*, ACM 2012.

[14] IOANNIDES C., et. al., *XCS cannot learn all Boolean functions*, GECCO 2011.

[15] HSUEH, H. W., et al., *Test Directive Generation for Functional Coverage Closure Using Inductive Logic Programming*, IEEE 2007, High-Level Design Validation and Test Workshop 2006.

[16] CHANG P. H., et. al., *Online selection of effective functional test programs based on novelty detection*, IEEE 2010, ICCAD 2010.

[17] WANG L. C., et. al., *Data mining in functional test content optimization*, IEEE 2015, ASP-DAC 2015.

[18] ROMERO E., et. al., *Support vector machine coverage driven verification for communication cores*, IEEE 2009, VLSI-SoC 2009.

[19] GUZEY O., et. al., *Functional test selection based on unsupervised support vector analysis*, IEEE 2008, DAC 2008.

[20] KATZ Y., et. al., *Learning microarchitectural behaviors to improve stimuli generation quality*, IEEE 2011, DAC 2011.

[21] MANDOUH E. E., et. al., *Cross-product functional coverage analysis using machine learning clustering techniques*, IEEE 2018, DTIS 2018.

[22] ZHU J., et. al., *Application of rough set and genetic algorithm to transformer fault diagnosis*, in: IEEE 2011, ACI 2011.

[23] WEN C., et. al., *Simulation-based functional test justification using a Boolean data miner*, IEEE 2007, ICCD 2006.

[24] GUO Q., et. al., *Pre-Silicon Bug Forecast*, IEEE 2014, IEEE TCADICS 2014.

[25] MAKSIMOVIC D., et. al., *Clustering-based Revision Debug in Regression Verification*, IEEE 2015, ICCD 2015.

[26] POULOS Z., et. al., *A failure triage engine based on error trace signature extraction*, in: IEEE 2013, IOLTS 2013.

[27] MANDOUH E. E., et. al., *Accelerating the debugging of FV traces using K-means clustering techniques*, IEEE 2017, IDT 2016.

[28] AMRANI M., et. al., *ML + FV = LOVE? A Survey on the Application of Machine Learning to Formal Verification*, Cornell University, Jun 2018.

[29] LOULERGUE F., et. al., *Verified Programs for Frequent Itemset Mining*, IEEE 2018, SCALCOM 2018.

[30] HU Z., et. al., *Practical Aspects of Declarative Languages (PADL)*, "Calculating a New Data Mining Algorithm for Market Basket Analysis", Springer, 2000, pp. 169-184.