Received 11 July 2022, accepted 30 July 2022, date of publication 4 August 2022, date of current version 10 August 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3196347

SURVEY

Code Generation Using Machine Learning: A Systematic Review

ENRIQUE DEHAERNE[1,2], (Graduate Student Member, IEEE), BAPPADITYA DEY[2], (Member, IEEE), SANDIP HALDER[2], STEFAN DE GENDT[1,3], (Senior Member, IEEE), AND WANNES MEERT[1], (Member, IEEE)
[1]Department of Computer Science, KU Leuven, 3001 Leuven, Belgium
[2]Interuniversity Microelectronics Centre (IMEC), 3001 Leuven, Belgium
[3]Department of Chemistry, KU Leuven, 3001 Leuven, Belgium

Corresponding author: Enrique Dehaerne (enrique.dehaerne@student.kuleuven.be)

ABSTRACT Recently, machine learning (ML) methods have been used to create powerful language models for a broad range of natural language processing tasks. An important subset of this field is that of generating code of programming languages for automatic software development. This review provides a broad and detailed overview of studies for code generation using ML. We selected 37 publications indexed in arXiv and IEEE Xplore databases that train ML models on programming language data to generate code. The three paradigms of code generation we identified in these studies are description-to-code, code-to-description, and code-to-code. The most popular applications that work in these paradigms were found to be code generation from natural language descriptions, documentation generation, and automatic program repair, respectively. The most frequently used ML models in these studies include recurrent neural networks, transformers, and convolutional neural networks. Other neural network architectures, as well as non-neural techniques, were also observed. In this review, we have summarized the applications, models, datasets, results, limitations, and future work of 37 publications. Additionally, we include discussions on topics general to the literature reviewed. This includes comparing different model types, comparing tokenizers, the volume and quality of data used, and methods for evaluating synthesized code. Furthermore, we provide three suggestions for future work for code generation using ML.

INDEX TERMS Automatic programming, computer languages, data collection, machine learning, natural language processing, neural networks, recurrent neural networks, software debugging, software maintenance, text mining.

I. INTRODUCTION

Software development is a complex and time-consuming process. It consists of two main phases: analysis and coding [1]. In the analysis phase, the requirements and architecture of the software system are formalized. In the coding phase, source code is written and tested to meet the requirements set in the first phase. Usually, maintenance of the system is included as an additional phase in the software development cycle where previous steps can be adapted to reflect changes in the needs of the system user. Figure 1 shows a flowchart for a simple software development model. In this review, we focus on the coding phase which works directly with source code.

Modern society relies on complex software applications. These applications can consist of millions of lines written in many programming languages (PLs) by many teams of developers. Even small software projects will often leverage large libraries that are expected to be easy to use and trusted to be efficient and safe. PLs are difficult to read and understand quickly so the developers must also document their programs to make them more maintainable. Mistakes made during the coding phase lead to software bugs that can cost time and

The associate editor coordinating the review of this manuscript and approving it for publication was Gustavo Olague.

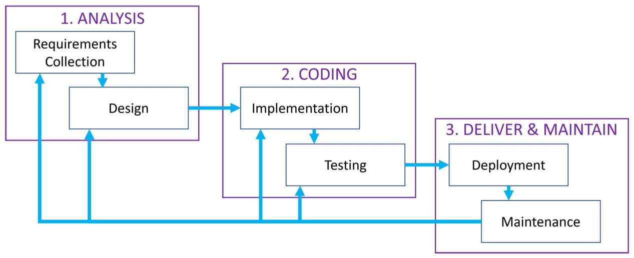


FIGURE 1. An example model for software development with three phases, each consisting of multiple steps. Software development models used in practice vary in the number of steps and their ordering compared to the model depicted in this flowchart.

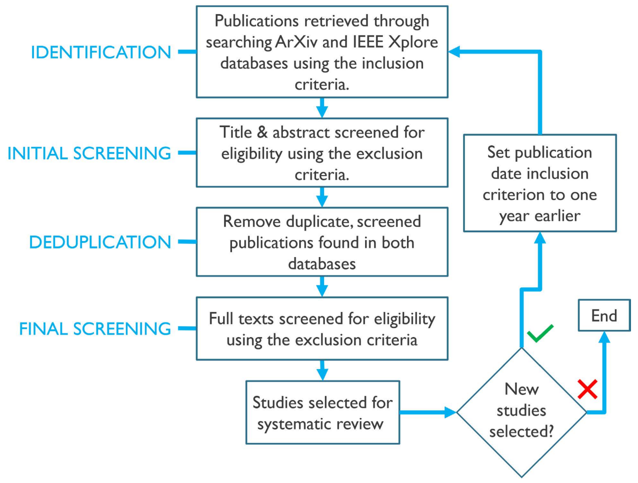

IDENTIFICATION
Publications retrieved through searching ArXiv and IEEE Xplore databases using the inclusion criteria.
INITIAL SCREENING
Title & abstract screened for eligibility using the exclusion criteria.
DEDUPLICATION
Remove duplicate, screened publications found in both databases
FINAL SCREENING
Full texts screened for eligibility using the exclusion criteria
Studies selected for systematic review
New studies selected?
Set publication date inclusion criterion to one year earlier
End

FIGURE 2. Flow diagram which depicts the sequence of steps taken to search and select studies to be reviewed.

money for the software creators and users. In the worst-case scenario, software bugs can jeopardize the safety of human beings.

As a result, many software development tools and technologies have been created to help developers write better software. A popular technology used by software developers is a "linter" which flags syntactic errors in code. Auto-formatters will add or remove whitespace and "newline" characters to code to improve readability. Statement auto-complete tools can suggest tokens that programmers might write next to improve their productivity. While these traditional tools can be useful for programmers, most of them can't help a developer with complex tasks such as writing understandable code documentation or implementing algorithms.

More recently, machine learning (ML) has opened up the possibility to automate difficult programming-related tasks. In particular, advances in neural network (NN) architectures, such as recurrent neural networks (RNNs) and transformers [2], have been used to advance the state-of-the-art (SOTA) for many difficult automated software engineering tasks. These tasks include code generation from code documentation [3], [4], documentation generation from code [5], [6], and cross-PL translation [7]. These technologies, among others, have even led to commercial products such as Tabnine [8] and Github [9]'s Copilot [10].

To provide a broad and detailed introduction to this field, this systematic review summarizes and discusses publications that use ML to generate code. More specifically, publications retrieved from searches on arXiv [11] and IEEE Xplore [12] databases that propose models that synthesize code from non-code inputs (description-to-code), generate code documentation given code inputs (code-to-description), or modify existing code from one form to another (code-to-code) were reviewed. We categorize each publication by its ML model and the relevant sub-domain. The intention of this review is to provide a broad but detailed overview of ML techniques applied to the domain of automatic software generation. A summary of each publication and general discussions are provided. Topics discussed in this review include the application categories, popular ML models, tokenization strategies, the quantity and quality of data, and metrics used to evaluate synthesized code. Additionally, three directions for future work are suggested before concluding this systematic review.

II. METHODOLOGY

This review followed the Preferred Reporting Elements for Systematic Reviews and Meta-analyses (PRISMA) guidelines [13] where appropriate for the scope of this review (see Section III for more details). The arXiv [11] and IEEE Xplore [12] databases were searched to identify potential publications for review. Inclusion criteria refer to filters applied to the search functions of each database. The search applied two general inclusion criteria and one inclusion criterion specific for each of the databases searched. Four exclusion criteria were then applied to the studies retrieved from this search to remove studies that are not appropriate for this review. Figure 2 provides a graphical overview of the search and selection methodology explained in more detail in the remainder of this section.

The first inclusion criterion applied to both databases was the search terms used. The search phrase "code generation using machine learning" was applied to all fields (title, abstract, full-text, etc.) using each database's search engine to identify possible publications for review. The second general inclusion criterion was the publication date. To ensure a variety in the different ML models studied by the retrieved publications, the first publication date range used was chosen to be 2016, one year before the introduction of the transformer architecture [2]. This was done because transformers have become very popular recently and we wanted to make sure that studies using other ML models were also retrieved by our search. Therefore the first searches were limited to studies published between 2016 and 2021. Additional searches were applied on earlier years iteratively by decrementing the last year searched, keeping all other inclusion criteria the same, until no new studies are selected for the review. An additional filter was added to the search of each database using filtering options provided by the search function of each database, respectively. The search on the IEEE Xplore database was limited to conference and journal publications. For the search on the arXiv database, the search was limited to publications on the subject of computer science (including cross-listings).

Four exclusion criteria were defined to filter publications returned by search queries using the inclusion criteria that are not applicable to the review. These exclusion criteria are as follows: (i) publications that do not propose new ML techniques or models for code-to-description, description-to-code, or code-to-code applications; (ii) survey, benchmark, and vision papers; (iii) publications where an ML model predicts numerical parameters for non-ML code generation engines; and (iv) publications not written in English. These criteria were applied to the title and abstract of publications first. Titles and abstracts that do not provide sufficient information to justify removal from consideration of the review are not filtered at this stage. Next, duplicates between the remaining publications from both databases were removed. Finally, the full texts of the remaining publications were screened based on the exclusion criteria. The publications not filtered in this final stage are added to the set of selected publications. All searches were performed in April 2022.

III. RESULTS

For the initial query using the publication date range of 2016-2021, the query on the arXiv [11] and IEEE Xplore [12] databases returned 613 and 274 publications, respectively. After applying the inclusion and exclusion criteria to the titles and abstracts of the 887 identified publications, 811 publications were excluded from consideration for the review. One duplicate study was found among the remaining 76 studies between the two databases and was removed. The final selection of publications was obtained by applying the inclusion and exclusion criteria to the full text of the 75 remaining publications. The final selection consisted of 37 publications, 28 indexed in the arXiv database [11] and 9 indexed in the IEEE database [12].

An overview of these search results for every step in the selection methodology is shown in Figure 3. Figure 3 also shows the results of the second search which was limited to studies published in 2015, one year earlier than the publication date range of the first search. Since no new studies were selected from this second search, no additional searches were conducted. All searches were performed in April 2022.

A direct comparison of the results of each publication is not possible as they use different model types, are trained and evaluated on different datasets, and use different methods for evaluation. Instead, a summary of each publication is provided in Table 1. The table consists of the following columns: (i) the application studied, (ii) the ML model used, (iii) the datasets used by the study, (iv) the results of the study and discussion, and (v) limitations and/or future work.

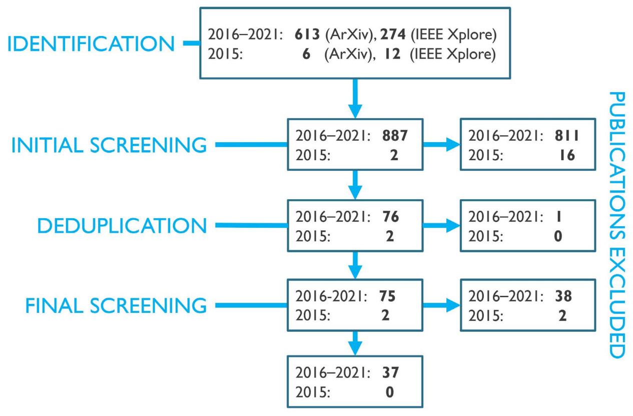


FIGURE 3. Flow diagram which shows the number of publications at each step of the search and filtering process.

General aspects of the applications, ML model types, tokenizers, datasets, and evaluation methods of these publications are discussed in more detail in the next section.

Common abbreviations used in Table 1 and tables in Section IV are as follows (in alphabetical order): **API** (Application Programming Interface), **APR** (Automatic Program Repair), **AST** (Abstract Syntax Tree), **CNN** (Convolutional Neural Network), **DSL** (Domain Specific Language), **ENN** (Essence Neural Network), **GRU** (Gated Recurrent Unit), **GUI** (Graphical User Interface), **kNN** (k Nearest Neighbors), **LSTM** (Long Short-Term Memory network), **MLP** (Multi-Layer Perceptron), **NL** (Natural language), **NN** (Neural Network), **NMT** (Neural Machine Translation), **PBE** (Programming-By-Example), **PL** (Programming Language), **RF** (Random Forests), **RL** (Reinforcement Learning), **RNN** (Recurrent Neural Network), **SM** (Statistical Model), **SOTA** (State-Of-The-Art).

IV. DISCUSSION

This review examines 37 studies that propose ML models to generate code from non-code descriptions, generate code documentation, or modify code. This section discusses common challenges these studies faced as well as key findings from the selected studies as a whole. First, the different application categories are introduced and explained with the help of examples. Next, the popular model types are introduced and compared. Subsequently, different tokenization strategies for code generation are compared. The quantity and quality of readily-available data for the different applications are discussed in their respective sub-sections. A section where the different metrics for measuring the quality of synthesized source code are compared is also included. Finally, three suggestions for future work are listed.

A. APPLICATION CATEGORIES

The studies reviewed in Table 1 can be categorized into three paradigms: description-to-code, code-to-description, and code-to-code. The studies can further be categorized into a number of application categories as shown in Table 2.

TABLE 1. Summary of all selected publications. The rows are ordered chronologically by publication date.

Study	Application	ML Model	Dataset	Results & Discussion	Limitations & Future Work
[14]	Code formatting tailored to a representative corpus	The proposed model, CodeBuff, is a kNN model used to compare a given code context with code contexts extracted from a training corpus. The amount of preceding whitespace suggested for the given code context is then determined by the whitespace preceding the nearest neighbors.	The corpora used for training and evaluation include manually formatted ANTLR [15] grammars, two open-source Java [16] projects, and an open-source SQL project. Additionally, a custom corpus written in Quorum [17] was used to evaluate the model's ability to generalize to an unfamiliar language and grammar.	For each corpus consisting of several documents, models were trained and evaluated using leave-one-out cross-validation. The proposed method achieved median leave-one-out error rates between 0.01 and 0.19. The more consistently formatted corpora yielded better error rates. Changing the grammar used to parse the corpora barely changed the error rates suggesting that CodeBuff is grammar invariant. A study on the size of the corpus needed to achieve good error rates suggested a minimum corpus size of 10 files. Finally, CodeBuff was evaluated on a corpus of an unfamiliar language and grammar, Quorum, without being trained on it. The achieved error rate was 0.04 indicating that CodeBuff generalizes well to unseen corpora.	CodeBuff is relatively memory intensive since all code context features are stored in memory for testing purposes. A 1.75GB RAM footprint was observed for the largest corpora evaluated.
[18]	Generate code from a given label that can include API calls to use, data types, and keywords	This study presents, BAYOU, a model that uses a Gaussian encoder-decoder RNN to generate code given API calls to use, data types, and keywords. The model learns code abstractions called *sketches*, tree-structured data that capture key elements of program syntax and ignores low-level elements such as variable names.	A dataset consisting of 1500 Android apps from the Android Drawer [19] online repository.	BAYOU outperformed or tied the SOTA model at the time, GSNN [20], in terms of five, custom, code-specific, metrics that aim to measure program equivalence. An ablation study suggested that making sketches from programs explicit leads to better performanpercentage ce in these metrics.	Traditional program synthesis uses instance-specific semantic constraints while BAYOU only handles syntactic constraints. Combining syntactic and semantic constraints is left as future work.
[21]	Documentation generation	This study trains an attention-based encoder-decoder LSTM model that takes Java [16] method ASTs as input and outputs comments in NL. The model, DeepCom, uses structure-based traversal (SBT) to process ASTs in a way that improves training.	Trained and evaluated on a dataset of Java method and comment pairs extracted from 9 714 Java repositories on Github [9].	DeepCom achieved a BLEU-4 [22] score of 38.17%. It outperformed Code-NN [23] and traditional Seq2Seq baselines which achieved BLEU-4 scores of 25.30% and 35.50%, respectively. When pre-order traversal is used instead of the proposed SBT, DeepCom's score drops to 36.01%. This suggests that the structural information of an AST is important for code translation tasks since SBT better captures structural information.	The AST traversal method used by DeepCom does not generalize to all PLs. Creating a similar traversal method for other PLs is proposed as future work.
[24]	Generating GUI code from screenshots	This study uses a model composed of a combination of different NN architectures to translate a screenshot to GUI code. First, a CNN extracts high-level visual features from the screenshot. These features are then given to two LSTMs, a block LSTM determines the number of blocks to generate in the program and a token LSTM generates program tokens for each block.	The model is trained and evaluated on the dataset from pix2code [25] as well as a more challenging custom dataset created by the authors of the study. Both datasets contain code-screenshot pairs of Android, IOS, and Web applications. The code in these datasets is translated to a DSL which is what the model generates.	The proposed model achieves a token error rate between 26.79-11.5% for different subsets of the dataset. The baselines, pix2code [25], a CNN encoder with an LSTM decoder, and a hierarchical LSTM network, achieve worse error rates for every subset of the dataset compared to the proposed model, respectively. Beam search sizes of 3 and 5 performed best for IOS and Android/Web respectively.	The proposed model ignores the values of textual elements. Text values are assigned randomly in the generated code.
[26]	Translating CUDA [27] code to OpenCL [28] code	This study trains a model to translate CUDA source code to OpenCL source code.	Data was generated by extracting CUDA API usages from CUDA samples and manually writing corresponding OpenCL API usages. From these manually written pairs, general API usage patterns are extracted and used to automatically find new samples.	NA	This publication does not present any experimental results or discussion.
[29]	Generating HTML from images of hand-drawn mock-ups	This study proposes an algorithm for generating HTML code from hand-drawn images. The algorithm has the following steps; (i) object detection using image processing techniques such as contour detection, (ii) object cropping, (iii) object recognition using a CNN model, and (iv) model output converted to HTML code through an HTML builder script.	The dataset used was a collection of images from Sketch2Code [30].	The CNN model achieved a validation accuracy of 73% after 200 epochs of training.	The study provides no baseline models to compare the proposed model's results to. The HTML builder script is not a learned algorithm and is instead a rule-based script.

TABLE 1. *(Continued.)* **Summary of all selected publications. The rows are ordered chronologically by publication date.**

[31]	APR	The proposed model is an encoder-decoder model with a combination of LSTM and GRU cells. The proposed model expects bug-fix pairs and edit actions as inputs for training.	This study mines over 787k method-level bug-fix pairs (BFPs) from "thousands" of GitHub [9] repositories. GumTree [32] is used to identify the list of edit actions performed between BFPs. This custom dataset was used for training and evaluation. A publicly-available dataset, CodRep [33], was also used for evaluation.	The proposed method generated corresponding fixed code given buggy code for 9% and 3% of small- and medium-sized BFPs, respectively, when using beam search with a beam size of 1. When using a beam size of 50, the corresponding fixed code was generated for 50% and 29% of small and medium BFPs. using a beam size of 1 resulted in generated code that is syntactically correct for 99% and 98% of small and medium BFPs. Increasing the beam size resulted in lower percentages of generated code that is syntactically correct.	This study expresses concerns about poor data quality due to the highly automated nature of the mining procedure for BFPs from GitHub repositories. It also does not compare results with SOTA APR models.
[34]	Documentation generation	This study proposes a Multi-way Tree-LSTM architecture that extends Tree-LSTMs by being able to handle an arbitrary number of ordered children. A Multi-way Tree-LSTM is used to encode a program's AST. The encoding is then decoded to NL using an LSTM decoder with attention.	The model is trained and evaluated on the same dataset from [21], a dataset of Java [16] methods, and comment pairs extracted from 9714 Java repositories on Github [9].	In terms of BLEU [22], CIDEr [35], METEOR [36], and ROUGE [37] metrics, the proposed model outperformed conventional Tree-LSTMs which in turn outperformed all other baselines including DeepCom [21].	The transformer [2] baseline was not designed for source code summarization and is therefore not an accurate representation of the potential of the transformer architecture for the task.
[38]	APR	This study proposes ENCORE, a novel NMT architecture using convolutional layers and attention as well as an ensembling method based on hyper-parameter selection where each model in the ensemble specializes in fixing certain types of bugs.	ENCORE is trained on GitHub [9] repositories written in one of four languages, Java [16], Python [39], C++ [40], or JavaScript.	ENCORE performs on par with SOTA APR techniques and solves 6 and 10 bugs in the Defects4J [41] and QuixBugs [42] datasets, respectively, that have not been fixed before. The NMT approach and the large ensemble of models allow ENCORE to generalize over PLs, this is something many other APR techniques cannot do.	The study assumes perfect fault localization. Fault localization is left as future work.
[43]	APR	This study approaches APR as an NMT problem by using an encoder-decoder LSTM architecture with attention.	The dataset used for training and evaluation consists of Java code from five open-source projects.	Pattern-based patch suggestions from the training set were used as a baseline for evaluation. The proposed model achieves the same F1 scores (0.50, 0.29, and 0.83) for three of the repositories evaluated on. The proposed model outperforms the baseline for the other two repositories achieving improvements in terms of F1 scores of 0.68 and 0.01. Furthermore, this study performs a qualitative human study to measure the helpfulness of the model. The human study suggests that fixes generated from queries with no *unknown* tokens are helpful for both correct and incorrect fixes since the (attempted) fix is understandable. Queries with *unknown* tokens generated unhelpful and even confusing fixes.	The two main problems of this study's NMT approach are the out-of-vocabulary problem, the need to use *unknown* tokens for infrequently occurring tokens, and the coverage problem, no guarantee that all query tokens are translated which can lead to incomplete statements. Additional limitations of the proposed system include the inability to generate patches that need additions or deletions of statements, the inability to generate patches that consist of multiple statements, and the lack of a bug-localization step.
[44]	Documentation generation	This study trains an encoder-decoder LSTM on code-comment pairs to generate documentation written in Japanese from source code inputs at inference time. Two important data processing techniques used in the proposed system are (i) using term frequency-inverse document frequency (TF-IDF) to find and increase the generation probability of important terms in the source code and (ii) replacing variable names with a known token and replacing the known token post-generation.	The training dataset consisted of "C language examples" and the evaluation data consisted of 53 problem statements and solutions written in C [45] used in university lectures for first-year students of "information studies".	A human study was performed to evaluate the proposed model. Students were asked to evaluate the quality of generated comments for a given snippet of code by assigning a score between 1-6 (bad-good) for the generated comment. The encoder-decoder model augmented with the TF-IDF keyword highlighter achieved an average score of 3.99. When replacing variables during pre-processing and reintroducing them post-generation, the average score dropped to 3.52. The encoder-decoder model by itself achieved an average score of 3.68.	The study does not provide an evaluation of the proposed models using an objective metric and relies solely on the human study. In certain instances, such as variable definition, the obfuscation of variable name information did improve the scores of the model's output so the improvement of this processing step is suggested for future work.
[46]	Code generation from NL descriptions	This study uses a phrase-based statistical machine translation model to generate method bodies from method descriptions written in NL.	The model was trained and evaluated on a dataset of 1000 GitHub [9] projects that use APIs from popular Java libraries.	When given queries consisting of an NL version of a method name, the model achieves an F1 score of 73.06%.	This study generates query data by performing transformations to the source code directly. Therefore, the results are most likely optimistic compared to what the model would score on organic queries written by humans. The study does not provide a thorough discussion of the results of the experiment.

TABLE 1. *(Continued.)* **Summary of all selected publications. The rows are ordered chronologically by publication date.**

[47]	Code generation from NL descriptions	This study trains and evaluates two neural models for the proposed dataset. The two models are an LSTM encoder-decoder with attention and a transformer [2] model.	This study presents JuICe: a large-scale distantly supervised dataset for open-domain context-based code generation. The training set consists of 1.5M target code and context pairs extracted from 659K publicly available Jupyter notebooks [48]. The test set consists of 3725 pairs from 13 905 nbgrader [49] programming exercises assignments and solutions written in Jupyter notebooks.	LSTM baselines outperform corresponding transformer models both in terms of BLEU [22] and *exact match* scores. Both baseline types benefit from additional code context.	NA
[50]	Code generation from NL descriptions	The proposed model consists of an NL encoder and an AST decoder, both using LSTMs. The decoder is trained on code idioms, frequently occurring and meaningful ASTs of code, with a custom training objective.	This study uses Hearthstone [51] and Spider [52], two code semantic parsing datasets.	The baseline model consists solely of a similar decoder trained using a cross-entropy objective. For Spider and Hearthstone, in terms of *exact match* and BLEU [22] score respectively, the proposed model outperforms the baseline decoder for most configurations (number of idioms trained on and idiom scoring function). The baseline ties with the best configuration of the proposed model in terms of exact match for the Hearthstone development set.	The Spider test dataset is unreleased so the study only evaluates the development split for Spider. Out of 80 mined idioms, 51 appeared in the inferred ASTs which suggests that idiom mining could be improved to filter out unhelpful abstract syntax sub-trees.
[53]	Generating tests for Android GUI applications	This study proposes an RL algorithm that uses formal specifications from a developer and observed actions on an Android application to generate a test.	A dataset of GUI-level test specifications for the Chess Walk and Notes applications from F-Droid [54] were obtained from F-Droid bug reports, novel bugs found, app-agnostic test oracles [55], and manually created specifications.	Different but equivalent linear temporal logic (LTL) formulas to depict GUI actions for the RL learner affected both the satisfiability of a given specification as well as the speed of test generation. The proposed algorithm was able to generate a satisfactory test for all nine formal specifications tested for two-thirds of LTL formulae. For these LTL formulae, the proposed algorithm was faster than baselines, outperforming baselines. The baselines, a random test explorer, Monkey [56], QBEa [57], generated a satisfactory test for six, three, and four out of nine specifications, respectively.	The sensitivity of the proposed algorithm to the LTL formula depiction of GUI actions is a problem that must be addressed before these LTL formulae can be generated automatically. The proposed method cannot specify GUI functions that depend on timing. The support of metric temporal logic, an extension of LTL which uses time-constrained temporal operators [58], is left as future work by the authors.
[59]	PBE	This study compares 14 different NNs that construct probability matrices for functions to be applied to input data. This probability matrix is used to enhance a depth-first search over 32 possible programs written in a custom DSL. Each NN has 3-5 layers using sigmoid or LeakyRelu activation functions and Adam [60] or Nadam [61] optimizers.	This study uses a custom dataset of input-output examples with corresponding programs written in the custom DSL accompanied by at least five test cases to validate generated programs.	For 500 programs of length 3, the best model, using LeakyRelu with Nadam optimization, was able to complete each program in 1718 steps on average. The average proposed model took fewer steps than the same search strategy without the use of NNs.	This study uses a small, custom DSL to keep the search space small. The search space and consequently the inference time of the proposed method would be intractable for general-purpose languages such as Java [16] or Python [39].
[62]	Code refactoring	This study proposes an RF classifier that identifies valid source code transformations.	A dataset of Google Code Jam submissions composed of 27 300 C++ source code files, consisting of 273 topics each with 10 parallel files, from Google Code Jam . From this dataset, approximately 152 000 paraphrases were generated. Of these paraphrases, 11% yield valid code transformations.	Given source code input, the RF classifier was able to correctly classify a snippet of code as being a valid transformation of another snippet of code for 83.7% of transformation pairs tested.	The main focus of this work was on a formal methodology for creating paraphrases given many semantically equivalent source code versions. As a result, there is a minimal discussion on how to improve the proposed RF classifier and no comparisons to baselines.
[63]	Pre-process source code to reduce the search space for PBE	The proposed pre-processing model includes a criticality predictor and a time-savings predictor. The criticality predictor is a feed-forward NN that takes a set of constraints as input and predicts if a terminal	The training set for the criticality predictor consists of 124 928 programs generated by CVC4 [64] and corresponding	A SyGuS solver, CVC4, augmented with the proposed model outperformed half of the given benchmarks with a reduction in total synthesis time for all benchmarks. CVC4 using only the criticality predictor and not the time-savings predictor	The two predictors of the model simply vote to combine the predictions on terminals. This largely ignores interactions between the two metrics. The

TABLE 1. *(Continued.)* **Summary of all selected publications. The rows are ordered chronologically by publication date.**

		belongs to the set of terminals for the constraints. The time-savings predictor uses an SM to calculate the expected time saved if a terminal is removed.	input-output pairs generated by random inputs and calculating the outputs on the programs. The training set for the time-saving predictor consists of 10 human-generated syntax-guided synthesis (SyGuS) problems and 20 automatically generated SyGuS problems. The evaluation dataset consists of public competition benchmarks from the SyGuS competition [65].	performed worse than just the CVC4 solver, increasing the running time on 53 out of the 62 benchmarks that did not timeout.	time-savings predictor uses an average expected value over all samples in the dataset. This predictor might be able to be improved by using a NN instead and this is left as future work.
[66]	Documentation generation	The proposed model architecture consists of two components. The first is a CNN model that predicts keywords from the source code. The second phase is a CNN-based encoder and LSTM decoder. The inputs for the second phase are source code and the predicted keywords from the first phase via an attention mechanism.	The study creates its own dataset consisting of 114 099 code-annotation pairs of Java methods collected from Github [9] repositories. Additionally, a dataset collected by [23] is used which contains 66 015 C# programming questions and related code pairs.	On average, the proposed model outperforms two baselines, the proposed model without keyword prediction and an encoder-decoder model from [23], over all datasets as measured by BLEU [22] score. The relative gains of the proposed model are largest for BLEU-4, the most rigorous version of BLEU.	NA
[67]	Generation of assert statements for unit tests	An LSTM encoder-decoder model that takes unit tests and the method to be tested as input and outputs an assert statement.	A custom dataset consisting of 188 154 raw JUnit [68] test-assert pairs (TAPs) mined from 9 275 open-source projects.	On raw TAPs, the model achieves *exact match for 17.66%, 23.33%, and 27.01% of total predictions* for beam sizes of 1, 5, and 50, respectively. When abstracting uncommon tokens out of the input, the perfect prediction percentage improves to 31.42%, 49.69%, and 65.31%.	No comparative evaluation with a baseline model is provided.
[69]	Documentation generation	The proposed model takes as input the source code token sequence, the AST, and previously generated documentation tokens. The source code tokens and preceding documentation tokens are first embedded using recurrent layers while the AST is embedded using a Convolutional GNN. These embeddings are then input to GRUs and attention mechanisms to obtain the probability distribution for the next documentation token.	This study uses a dataset from [70] which contains 2.1M Java method comment pairs mined from open-source repositories.	The proposed model outperformed all graph-based and non-graph-based baselines tested. It achieved 19.93 in terms of BLEU [22] and 56.08 in terms of ROUGE [37]. It outperforms the nearest graph-based baseline by 4.6% and 0.06%, respectively. It outperforms the non-graph baseline by 5.7% and 12.72%, respectively.	Ensembling the proposed model with other models that outperform the proposed model for specific types of source code is left as future work.
[71]	Code editing and APR	The proposed model consists of two stages, (i) an LSTM encoder-decoder model that predicts the output syntax tree structure from an input source code syntax tree and (ii) another LSTM encoder-decoder model to generate tokens given the input source code as well as the output from the first encoder-decoder.	This study uses three datasets, (i) Code-Change-Data, a custom dataset consisting of pre- and post-patch Java file pairs, (ii) a subset of Pull-Request-Data [72], and (iii) a subset of Defects4j [41]	The proposed model outperforms all other baselines when predicting the correct edit within the top 2 and 5 predictions for Code-Change-Data and the top 5 predictions for Pull-Request-Data. Despite not being designed for bug-fixing, the proposed model can generate bug-fix patches for 15 bugs and partial patches for 10 bugs in Defects4J.	This study has difficulties with the out-of-vocabulary problem (handling *unknown* tokens, see Section IV-C). This study focuses on small code changes of a single-line or single token. Ensemble learning is proposed as future work to address hyper-parameter tuning as well as context-dependent bug-fix problems.
[73]	PBE	Top-down search strategy using a bidirectional LSTM model for processing global leaf representations before score calculation for guiding search. The main contribution of this study is the addition of data types as additional input features.	A custom dataset consisting of generated code in a subset of the Haskell language [74].	The proposed model outperforms baseline neural search techniques in terms of node prediction accuracy by a substantial margin. The proposed method outperforms uniform random search in most cases. Uniform random search almost matches the proposed method for programs of 3 nodes and outperforms the proposed model when evaluated on 100 sample programs of 1 node.	This study uses a limited dataset. It used a simple LSTM encoder rather than a more complex variant which is more computationally intensive but most likely would produce better results. The evaluation was restricted to programs of 3 nodes or less.
[75]	Translating Swift code to Java code and vice versa	Transformer model which takes as input source code and outputs source code. The preprocessing methods used include code formatting, line separation, and line selection.	A custom dataset of manually written, equivalent Swift and Java code snippet pairs.	The proposed model achieves token accuracy ranging between 55%-94% and BLEU scores between 19-94 for different types of code tasks. The model performs best on a certain translation task when trained only on similar training data. It was found that using a variety of different training data requires more training data and time to achieve similar results.	The study uses a dataset of very narrow types of code snippets. The model performs translations line-by-line which means that information from other lines is not used for each translation operation.

TABLE 1. ***(Continued.)*** **Summary of all selected publications. The rows are ordered chronologically by publication date.**

[76]	Code generation from NL descriptions	Transformer model augmented with (i) a pseudo-relevance feedback mechanism that retrieves relevant source code using the input as a query and emphasizes common tokens in the source code as possible output tokens and (ii) a copy generation mechanism that emphasizes tokens from the inputs as possible outputs.	This study uses three datasets, (i) Django [77] consisting of annotated lines of Django (web framework) code, Hearthstone [51] consisting of names, descriptions, and statistics of cards from a card game and corresponding code, and (iii) CoNaLa [78] consisting of question-answer pairs.	The proposed model achieves BLEU [22] scores of 82.3, 74.5, ad 22.3 for the Django, Hearthstone, and CoNaLa datasets, respectively. The proposed model outperforms all retrieval methods and all SOTA generative methods tested.	NA
[79]	Code generation from NL descriptions	The proposed model is the GPT-2 transformer model [80] fine-tuned on the datasets used in the study.	The dataset used in this study is a Python [39] only subset of the CodeSearchNet [81] dataset which consists of comment-code pairs from open-source repositories in many different PLs.	With the best hyper-parameters for each model, the proposed model achieves a BLEU [22] score of 0.22 while the baseline, Char-RNN [82], achieves a BLEU score of 0.117 on the training dataset. An ablation study showed that byte-pair encoding performed better than character encoding.	A clear, quantitative comparison between the proposed model and the baselines is not shown in the study.
[83]	Documentation generation	The proposed model, API2Com, is a transformer [2] model that combines API documentation with source code and AST to generate comments	This study uses a Java [16] only subset of the CodeSearchNet [81] dataset which consists of comment-code pairs from open-source repositories in a variety of PLs.	API2Com outperforms all non-transformer baselines in terms of BLEU 1-4 [22] and METEOR [36] scores but not ROUGE-L [37] score. The "TransformerBased" [84] baseline outperforms all baselines as well as the proposed model in terms of all metrics used. An ablative study suggested that giving API documentation data improves performance while giving AST decreases performance. Using a GRU architecture instead of a transformer architecture was shown to consistently and significantly decrease performance.	Beam search with a beam size greater than one was deemed to be too computationally expensive for the study.
[85]	APR	The proposed model is a combined model consisting of the context-aware neural translation (CoNuT) [86] architecture with a GPT-2 Transformer [80] as the language model between CoNuT's encoders and decoders.	The training dataset used in the CoNuT study is used for this study as well. It consists of bug patch data extracted from open-source repositories searched on keywords in commit messages. Two evaluation datasets were used, Defects4J [41] and QuixBugs [42].	Compared to baselines, the proposed model fixes the most number of bugs, 57 and 26 for Defects4J and QuixBugs, respectively. It fixes 12 bugs in Defects4J that have not been fixed before by previous methods. It fixes 1 bug for QuixBugs that has not been fixed before by previous methods. An ablation study suggested that the GPT language model, code-aware beam search strategy, and subword-tokenization components of the proposed model all positively impact its performance.	NA
[4]	Code generation from NL descriptions & docume-nation generation from code	Large decoder-only Transformer [2] model comparable to GPT-3 [87].	The training dataset consists of a large collection of publicly available code from GitHub [9]. A custom evaluation dataset, HumanEval, which consists of 164 hand-written programming problems each with a function signature, docstring, body, and on average 7.7 unit tests to test functional correctness.	The proposed model solves 28.8% of problems in the HumanEval test set while baselines GPT-3 and GPT-J [88] solve 0% and 11.4%, respectively, for a single generated solution. When generating 100 solutions and picking the highest mean log probability or solution that passes all unit tests, the proposed model solves 44.5% and 77.5% of problems, respectively. For document generation, generated solutions were hand-graded and the document generation variant of the proposed model successfully generated documentation for 20.3% and 46.5% of problems using pass@k (a metric estimating the fraction of solutions generated that solve a given problem) for k=1 and k=10, respectively.	The proposed model is not sample efficient to train, it was trained on hundreds of millions of lines of code. The proposed model struggles most with longer and/or higher-level docstrings.
[89]	Pseudo-code generation for snippets of code	The proposed model is a transformer model augmented with a CNN for extracting and encoding code features. The model takes source code as inputs and outputs NL pseudo-code using beam search.	The study uses two datasets, Django [77] and SPoC [90] consisting of C++ [40] code-pseudocode pairs.	The proposed model outperforms all baselines tested in terms of BLEU-4 [22], METEOR [36], ROUGE-L [37], and CIDEr [35] metrics. An ablation study suggested that the code feature extractor CNN can improve the performance of the model.	No cross-validation is used for the evaluation due to the high training computational cost.
[91]	PBE & code generation from NL descriptions	The proposed model consists of three main components, (i) a latent predictor that predicts a distribution of latent codes conditioned on program specification, (ii) a latent	For PBE, a synthetic dataset generated by sampling programs from a string transformation DSL [92] was used. For NL	For PBE, string transformation programs were generated. The proposed method performed 4, 6, and 6 percentage points better than the best variation of the RobustFill model studied [94] in terms of	The specifications in the Python dataset are noisy and the code is often not executable due to missing dependencies.

TABLE 1. *(Continued.)* **Summary of all selected publications. The rows are ordered chronologically by publication date.**

		program decoder that provides a distribution over possible program outputs conditioned on specifications, and (iii) a program encoder only used for training. All three components are implemented as transformers.	description-driven generation, a dataset of 111k code-docstring pairs [93] written in Python [39] was used.	accuracy for beam search sizes of 1, 10, and 100, respectively. An ablative study suggested that using two-level discrete latent codes was important. Without it, the performance of the model was similar to the best baseline. For NL description-driven generation, the proposed method had a BLEU [22] score larger than that of the next best model, RobustFill [94], by 1.9, 3.1, and 4.1 for beam sizes of 1, 10, and 100, respectively.	
[95]	Documentation generation	Three transformer models with different encoding fusion methods are studied. The three different fusion methods are (i) the source code and its AST are encoded with their own encoder and the code and AST vectors together form the output vector, (ii) a shared encoder that takes as input either source code or AST, and (iii) a single encoder that takes as input both source code and AST	A dataset containing over 460k java method-comment pairs [96] sourced from GitHub [9].	The proposed model outperforms the next best model by 3.28%-5.07% in terms of BLEU 1 and 4 [22], respectively. In a human evaluation, the proposed model still outperforms all baselines. The single encoder fusion method performs best out of all the fusion methods studied including not using an AST.	NA
[97]	APR	The proposed model, Beep, takes as input the sub-token (token splits based on camel case naming or underscores) and the AST of a code snippet as well as potential bug-fix operations. Using the methods described in code2seq [5], these inputs are encoded and given to a fully-connected layer followed by an LSTM encoder-decoder model, followed by a pointer network. The final output is a path prediction that characterizes the location of a fault and the appropriate operator to fix it. For patch generation, Beep substitutes faulty tokens using heuristics.	The model is trained on patches from CoCoNut [86]. For evaluation of path localization, bugs from the ManySStuBs4J [98] and Defects4J [41] datasets were used. For patch generation evaluation, bugs from Defects4J [41], Bears [99], QuixBugs [42], and Bugs.jar [100] were used.	Beep is compared to two baselines, an SM that uses probabilities of faulty code elements [101] and an RF classifier that predicts whether a given token is faulty. Beep outperforms these baselines on buggy lines and methods for Recall@Top-1,5,10,20 as well as Mean First Rank metrics. When required to also predict the operation needed to fix a fault in a method, the performance of Beep lowers but still performs significantly better than the baselines. For patch generation, Beep achieves a perfect correctness ratio for all bugs it corrects and fixes 2, 2, and 4 bugs not fixed before for Defects4J, Bears, and Bugs.jar, respectively.	The localization capabilities of Beep are restricted to the token level and assume perfect line/method level localization. In this study, Beep only created short AST/operation paths so the effect of long paths in larger code contexts is unknown. Extending Beep to other programming languages and collecting datasets with more diverse bugs are listed as future work.
[102]	Code completion from function signatures	This study proposes conditional distributional policy gradients (CDPG) to condition finetuning of language models towards satisfying a set of constraints. CDPG approximates energy-based models which represent control objectives. CDPG is used to finetune GPT-Neo [103], a decoder-only transformer, to generate Python [39] code conditioned on a constraint. The two constraints experimented with are compilability and compliance of PEP8 [104] Python style guide. The input data consists of function signatures and the model predicts the corresponding function body.	The data used in this study was extracted from Python150 [105], a dataset of 150k files collected from Python projects on Github [9]. 5000 function signatures were used for training and another 5000 for testing. No function bodies were used.	Using compilability as the conditioning constraint, CDPG finetuning produced compilable functions for 65% of samples, an increase from 40% before finetuning. Using PEP8 errors, the average PEP8 error count decreased and compilability increased. Two RL baselines, Reinforce [106] and Ziegler [107], outperformed CDPG on compilability but produced heavily degenerated code measured predominately by expected Kullback-Leibler (KL) divergence from the pretrained model's output as well as an optimal output that has minimal KL divergence to the original pretrained model. CDPG performed similarly in almost all metrics to its unconditional DPG abalation.	Applying CDPG to other tasks as well as using different control objectives are listed as future work.
[108]	PBE	This study proposes a method that generates Python [39] code from ENNs [109]. ENNs map inputs to corresponding output labels using symbolic learning approaches. A condensed representation of a trained ENN in the form of code is produced as follows: neurons with similar connection patterns are organized and grouped, these groups are interpreted as logical functions, and these functions can then be translated to a high-level programming language.	Collections of input-output pairs for various computational problems were generated for this study. The problems studied include cellular automata [110], finding the maximum absolute value of a list of inputs, maximum Boolean satisfiability, and classifying shape orientations.	The proposed method is able to generate code that that perfectly describes cellular automaton update rules. This was accomplished using a very limited subset of all available data and scaled to large cellular automata. It outperformed a "simple deep learning" baseline as well as an RF baseline, both in terms of the number of problems solved as well as data efficiency. For maximum Boolean satisfiability, the code generated outperformed a 3/4 approximation algorithm as well as a greedy algorithm in terms of the fraction of clauses satisfied. 100% accuracy is achieved for the maximum absolute value problem and shape orientation prediction problems.	The current method is limited to producing Boolean logic without recursion or re-writable variables. It is assumed that an extension of the proposed method to overcome these limitations is possible due to the general principles of ENNs. Combining natural language processing with ENNs to allow for code generation from natural language descriptions is suggested as future work.

TABLE 1. *(Continued.)* **Summary of all selected publications. The rows are ordered chronologically by publication date.**

[111]	Function body generation conditioned on static analysis specifications of the code context.	This study proposes neurosymbolic attribute grammars (NSG) which use a static analyzer to create code context specifications (variable types, documentation strings, etc), a context-free grammar to specify generation rules, and a neural model to predict which rules to expand. The neural model is similar to the one proposed by [18] and uses LSTMs.	More than 1.5M pairs of method bodies and corresponding class contexts were created from the Android Drawer [19] Java repository for training and evaluation.	Three Transformer [2] baselines (GPT-Neo [103], CodeGPT [112], and Codex [4]) and two grammar-based baselines (GNN2NAG [113] and CNG, the latter of which is identical to NSG but is not trained on attributes) were used to compare results to the proposed NSG model. NSG is able to generate code that passes all 13 static checks for 86.41% of samples while the best baseline, Codex, achieves this for 67.3% of samples. NSG outperforms all baselines for all four fidelity checks (set of API calls, sequences of API calls, sequences of program paths, and AST exact match) that compare generated code to reference code.	The proposed method requires designing an appropriate context-free grammar extended with attributes as well as static analysis rules. Substituting the LSTMs used in the proposed NSG model with a Transformer model is suggested as future work.
[114]	Data visualization code generation from hand-drawn sketches	This study trains and evaluates encoder-decoder models to generate data visualization code in a DSL from images of hand-drawn sketches that represent the desired data visualization. A CNN-RNN and CNN-Transformer model were used for experiments.	A dataset of sketch-code pairs was created by (i) automatically generating DSL code, (ii) executing the code to render visualizations, and (iii) applying style transfer techniques. Style transfer mechanisms used include two traditional visualization tools as well as a deep-learning style transfer model, Photo-Sketching [115]. Additionally, 100 hand-drawn visualizations with corresponding DSL code were manually created for evaluation purposes.	The proposed RNN and Transformer models were evaluated using classification accuracy (output tokens compared to reference tokens) as well as two novel metrics, structural accuracy (e.g., syntax errors and incorrect visualization types in the output code) and decoration accuracy (correctness of visualization specifications). On the automatically generated test data, both RNN and Transformer models performed similarly, only deviating by a few percentage points for each metric. Approximately 97% of the predictions were valid DSL expressions that generated correct visualizations. On the hand-drawn test set, the Transformer model achieves 95% structural accuracy while the RNN model only achieves 66% structural accuracy.	The dataset used a very limited vocabulary of DSL tokens. Expanding the capabilities of these models to produce additional types of visualizations would require creating additional DSL tokens and training the model on new data that uses these tokens.

TABLE 2. **Application categorization of the selected publications. The applications categories are grouped together by three paradigms: description-to-code, code-to-description, and code-to-code. These paradigms describe the nature of the inputs and outputs of its applications. All studies within a row are ordered chronologically by publication date.**

Paradigm	Application category	Studies
Description-to-code	From NL descriptions	[46], [47], [50], [76], [79], [4], [91]
	PBE	[59], [63], [73], [91], [108],
	From images	[24], [29], [114]
	From other structured data inputs	[18], [53], [111],
Code-to-description	Documentation generation	[21], [34], [44], [66], [69], [83], [4], [89], [95]
Code-to-code	APR	[31], [38], [43], [71], [85], [97]
	Cross-PL translation	[26], [75]
	Refactoring	[14], [62]
	Code completion	[67], [102],

Figure 4 shows the percentage of selected studies belonging to each paradigm. This section explains the three paradigms and introduces the most popular application categories of the selected publications.

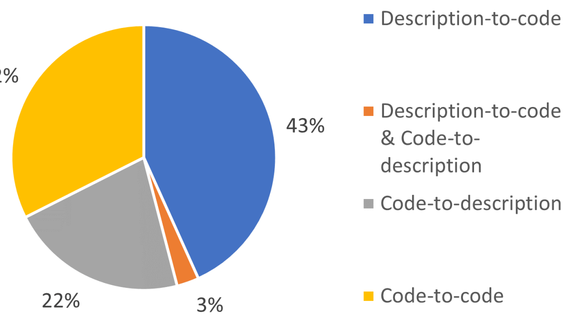
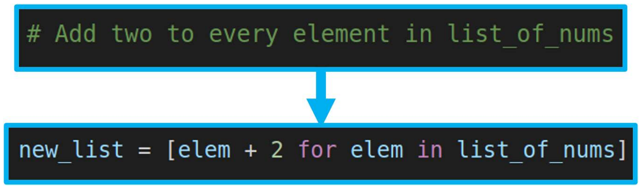


FIGURE 4. **Percentage of selected publications that study each of the three application paradigms: description-to-code, code-to-description, and code-to-code.**

Description-to-code applications involve generating code conditioned on model inputs that are not code. This was the most popular paradigm, applicable to 46% of all selected studies. The descriptions can come in various forms. The most popular description type is natural language (NL) documentation. These descriptions are often obtained from code comments written before a code snippet. An example NL description with an associated code implementation is shown in Figure 5.

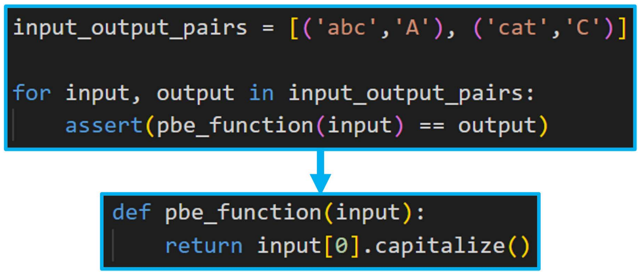
Programming-by-example (PBE) is the second most popular application category for the description-to-code paradigm. For PBE, the functionality of the desired program to be



```
# Add two to every element in list_of_nums
```

```
new_list = [elem + 2 for elem in list_of_nums]
```

FIGURE 5. An input-output pair example for code generation from NL. The input consists of an NL description, depicted here as a code comment, which describes the functionality of the desired code output.



```
input_output_pairs = [('abc','A'), ('cat','C')]

for input, output in input_output_pairs:
    assert(pbe_function(input) == output)
```

```
def pbe_function(input):
    return input[0].capitalize()
```

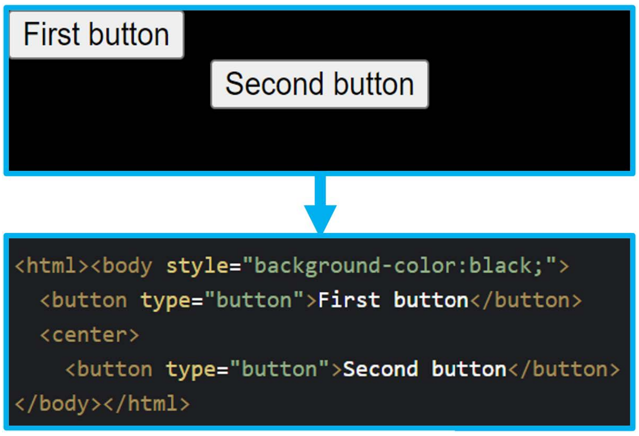
FIGURE 6. An example of a data-program pair for programming-by-example (PBE) with a verification code snippet. PBE generates a program that satisfies the functionality described by input-output data. This data is depicted here as a list of tuples (top rectangle) consisting of an input and then an output. Under this data is a for loop which verifies that the function to be generated should return the corresponding output for each input. The example program (bottom rectangle) returns the desired output for each input in the list "inputs."

generated is described by pairs of program input and output examples. Figure 6 shows an example of a list of possible program inputs and a list of corresponding outputs as well as a program that satisfies the given input-output pairs.

Another important description type is images. For [24] and [29] the images are screenshots of graphical user interfaces (GUIs) and for [114] the images are sketches of data visualizations. The desired output is a program that can synthesize the given image. Figure 7 shows a possible image-program pair for a simple GUI implemented in the HTML language.

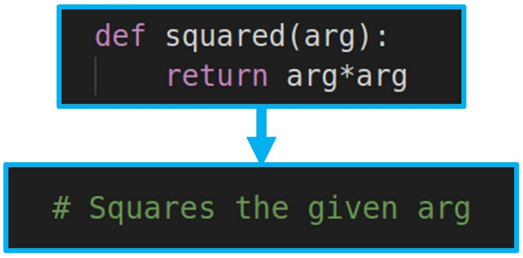
Code-to-description studies in this review all belong to a single application category, documentation generation. With 25% of all selected studies being code-to-description studies, the paradigm is the least popular of the three paradigms while documentation generation is the single most popular application category. Sometimes called source code summarization, the objective of this task is to generate an NL description of the code, usually in the form of a comment. An example of documentation generation is shown in Figure 8 as well as Figure 7 if the input and output data were swapped.

Code-to-code applications generate code conditioned on other code. The most popular application category of this paradigm is automatic program repair (APR). The input for APR is faulty, or buggy, code for which the model should generate similar code that does not have the bug. Figure 9 shows a buggy line of Python [39] code that has a syntactic error and the fixed line of code as the output.



```
<html><body style="background-color:black;">
  <button type="button">First button</button>
  <center>
    <button type="button">Second button</button>
</body></html>
```

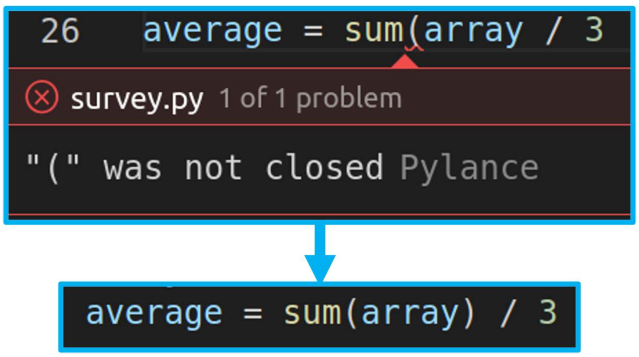
FIGURE 7. An example of an image-code pair for code generation from images. The image depicted here is a simple GUI with a black background and two buttons. The desired code to be generated, shown in the bottom rectangle, is HTML code that synthesizes this GUI.



```
def squared(arg):
    return arg*arg
```

```
# Squares the given arg
```

FIGURE 8. An input-output pair example for documentation generation from code. The input consists of code, in this case a Python [39] function. The desired output is NL which describes the semantics of the given code. The problem statement for documentation generation is the same as the problem statement of code generation from NL with the inputs and outputs reversed.



```
average = sum(array / 3
```

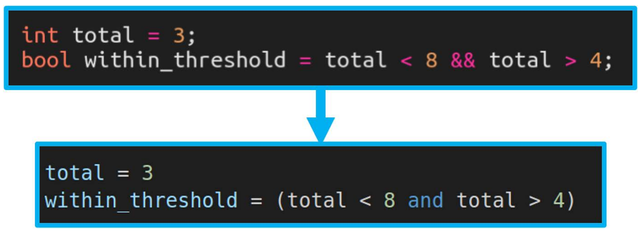
survey.py 1 of 1 problem

"(" was not closed Pylance

```
average = sum(array) / 3
```

FIGURE 9. An example of an input-output pair for automatic program repair (APR). The input example here contains a syntactic error that has been recognized by the Pylance [116] linting tool. The output example is the same code as the input with an additional closing bracket which fixes the input's error.

Cross-PL translation involves translating code written in one programming language (PL) to code written in another PL while preserving as many features of the original code as possible. An example where C++ [40] code is translated to Python that preserves the same functionality is shown in Figure 10. Refactoring is similar to cross-PL translation



```
int total = 3;
bool within_threshold = total < 8 && total > 4;
```

```
total = 3
within_threshold = (total < 8 and total > 4)
```

FIGURE 10. A cross-PL translation pair example. The first code snippet is written in C++ [40]. A functionally equivalent Python [39] translation of the code snippet is shown in the second code snippet.

TABLE 3. Overview of the types of ML models used by selected studies. Types are partitioned into sub-types where appropriate. All studies within a row are ordered chronologically by publication date. If a study proposes multiple models or proposes a model that combines different model-types, they are all listed in this table with the exception of MLPs which are not listed if connected to another model type (e.g. RNN, Transformer, CNN, etc).

ML model type	Sub-type	Studies
RNN	LSTM	[21], [24], [34], [43], [44], [47], [50], [66], [67], [73], [97], [111]
	GRU	[31], [69]
	Other	[18], [114]
Transformer [2]		[47], [75], [76], [79], [83], [85], [4], [89], [91], [95], [102], [114]
CNN		[24], [29], [38], [66], [69], [89], [114]
Other neural	MLP	[59], [63]
	Graph NN	[69]
	Neural trees	[71]
	ENN [109]	[108]
	Unspecified	[26]
Non-neural	SM	[46], [63]
	RL	[53]
	RF	[62]
	kNN	[14]

in that features of the input code should be preserved but different because the input and output code are written in the same PL. Refactoring aims to transform the input code to a form that is better understandable for humans. Reference [14] does this by adding or removing whitespaces or new-line characters to input code while [62] attempts to paraphrase code statements so that the transformed code is more concise than the input. The final application category, code completion, involves predicting subsequent code statements only from prior code.

B. MACHINE LEARNING MODEL TYPES

The review shows that a wide variety of ML methods can be used for different code generation tasks. Table 3 shows which of the selected studies use certain model types. Figure 11 shows the number of times a ML method is used for the different application categories introduced in Section IV-A. The popular ML methods used by the selected studies are introduced and compared in this section.

Recurrent neural networks (RNN) are a class of NN often used with sequential data such as NL. RNNs use

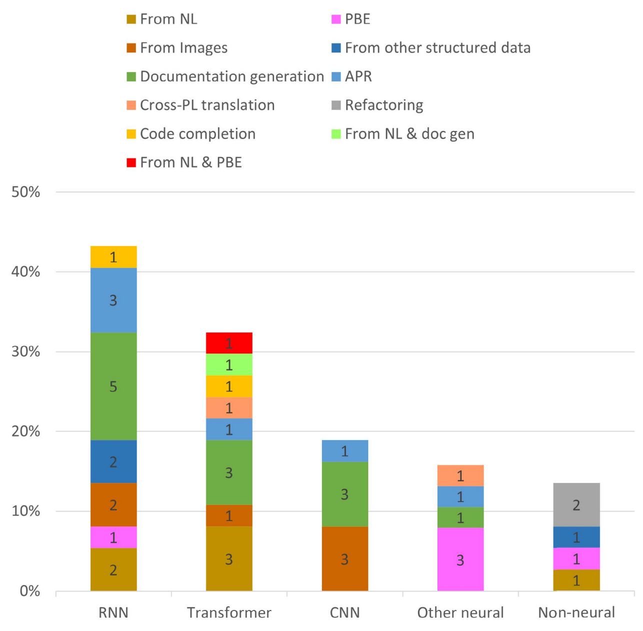


FIGURE 11. The number of selected studies that used a certain model type. Each bar is categorized by application. The vertical axis gives the percentage of the number of studies that use a certain model type out of 37, the total number of selected studies.

previous outputs and states of the network as supplementary information to the current input. Diagrams and equations of three types of RNNs, a basic RNN, LSTM, and GRU, are shown in figures 12, 13, and 14, respectively. The hidden state, h_t, allows the model to use previous data in a sequence alongside the current input. Basic RNNs do not capture long-term dependencies well. LSTMs address this issue by passing along its cell state, c_t [117]. GRUs combine the hidden state and cell state into one state, simplifying the network. Attention mechanisms, first proposed in [118], are often used by RNNs to encode information in variable-length vectors which reduces information loss for large inputs.

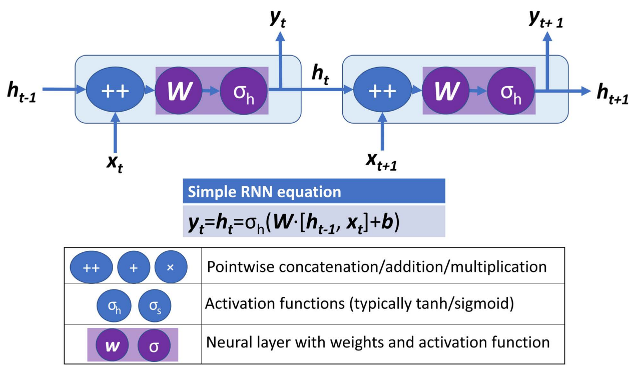
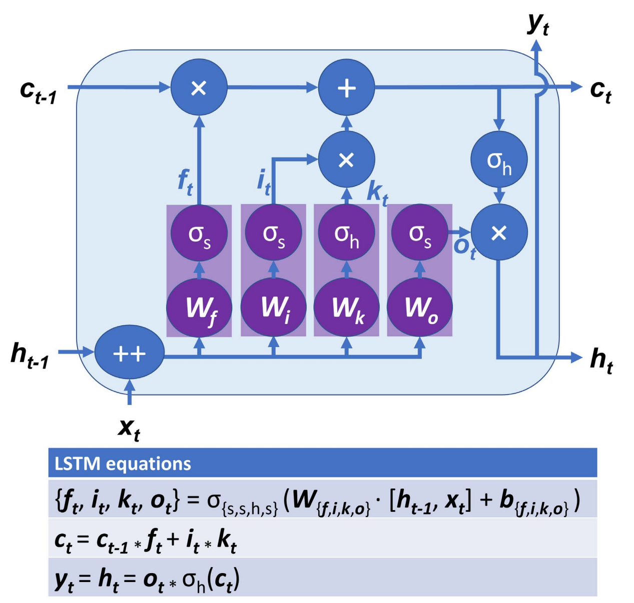

h_{t-1}
x_t
y_t
h_t
x_{t+1}
y_{t+1}
h_{t+1}
Simple RNN equation
$y_t=h_t=\sigma_h(W\cdot[h_{t-1}, x_t]+b)$
Pointwise concatenation/addition/multiplication
Activation functions (typically tanh/sigmoid)
Neural layer with weights and activation function

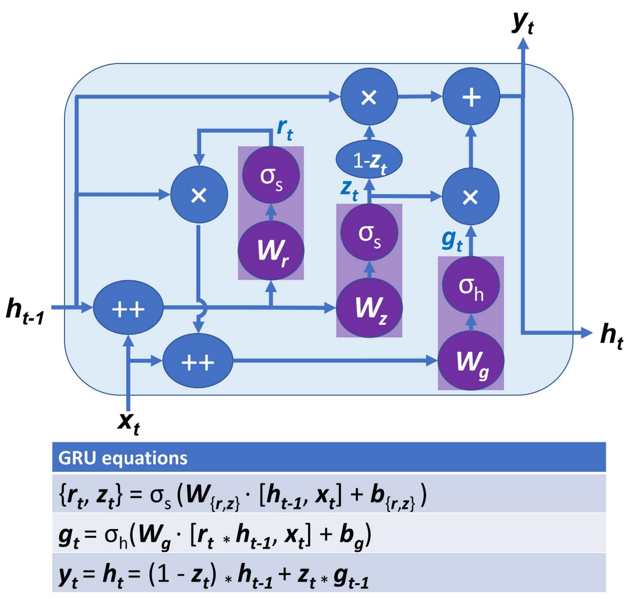
FIGURE 12. A simple recurrent neural network (RNN) rolled across time for two inputs of sequence (t and t + 1).

RNNs are the most popular NN type in the review, used in 43% of the selected studies, 80% of which use LSTMs specifically. RNNs are used for every application category from



LSTM equations
$\{f_t, i_t, k_t, o_t\} = \sigma_{\{s,s,h,s\}}(W_{\{f,i,k,o\}} \cdot [h_{t-1}, x_t] + b_{\{f,i,k,o\}})$
$c_t = c_{t-1} * f_t + i_t * k_t$
$y_t = h_t = o_t * \sigma_h(c_t)$

FIGURE 13. A long short-term memory (LSTM) network. network. Compared to simple RNNs such as the one shown in figure 12, LSTMs add a cell state (c_t) which allows for better handling of long-term dependencies in sequential data.

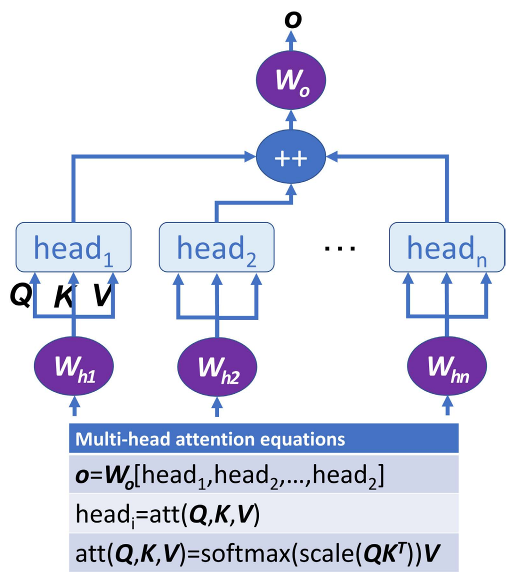


GRU equations
$\{r_t, z_t\} = \sigma_s(W_{\{r,z\}} \cdot [h_{t-1}, x_t] + b_{\{r,z\}})$
$g_t = \sigma_h(W_g \cdot [r_t * h_{t-1}, x_t] + b_g)$
$y_t = h_t = (1 - z_t) * h_{t-1} + z_t * g_{t-1}$

FIGURE 14. A gated recurrent unit (GRU) network. GRUs aim to handle long-term dependencies like LSTMs but combine the hidden state and cell state into one state to simplify the network.

Section IV-A except for refactoring and cross-PL translation. They are most used for documentation generation studies. Five out of nine documentation generation studies use RNNs.

RNN decoders are used in combination with CNN encoders in four studies, [24], [66], [69], [114].References [24], [114] use CNNs as encoders for image inputs while [66], [69] use CNNs for encoding text.



Multi-head attention equations
$o = W_o[\text{head}_1, \text{head}_2, \ldots, \text{head}_2]$
$\text{head}_i = \text{att}(Q, K, V)$
$\text{att}(Q, K, V) = \text{softmax}(\text{scale}(QK^T))V$

FIGURE 15. Graphic depiction of multi-head attention, the main building block of the transformer architecture [2]. Multi-head attention allows for multiple levels of attention between tokens to be learned.

Transformers [2] rely solely on attention mechanisms to capture dependencies between tokens in sequential data. While complex RNNs incorporate attention mechanisms to enhance dependency information, only using self-attention allows for greater parallelization. Figure 15 shows multi-head attention, the main building block of the original transformer architecture [2].

The transformer architecture is studied in 12 out of the 37 selected publications, as shown in Figure 16, making it the second most popular model type overall. Figure 16 also shows that transformers are the most popular model type if only the last year is considered. Similar to RNNs, transformers are used for a wide variety of tasks. Transformers are used

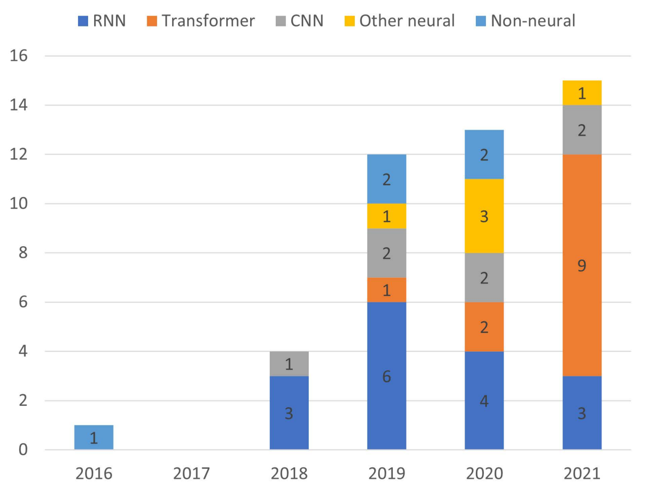
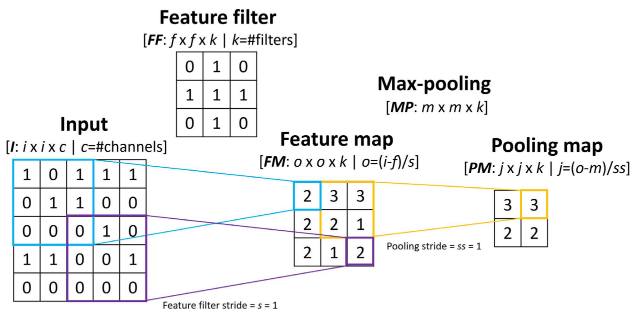


FIGURE 16. The number of selected studies that use a certain model type by year of publication. Each bar is categorized by application.

for five out of seven code generation from NL description studies, more than any other model type. The only tasks that do not have a selected study that uses a transformer are code generation from other structured data and refactoring.

As RNNs and transformers work well with sequential data, they are often compared in experiments. Of the 12 studies that use transformers in their proposed models, 9 of them [47], [76], [79], [83], [85], [89], [91], [95], [102] compare their results with RNN baselines. The transformer model used in [47] underperformed a comparative RNN model studied. The other eight publications propose transformer-based models that outperformed RNN-based baselines of each respective study. Reference [111] proposed a method with an LSTM component that outperformed transformer baselines but it was suggested that replacing the LSTM component with a transformer could improve the proposed method.

Convolutional neural networks (CNN) use convolution layers that sweep an input using a feature filter to aggregate information about the input. Convolutional operators work well on grid-like data such as images. Pooling is frequently used in CNNs to reduce the number of parameters of the model. Figure 17 shows an example of the convolution operation as well as the pooling operation.


Feature filter
[FF: f x f x k | k=#filters]
Max-pooling
[MP: m x m x k]
Input
[I: i x i x c | c=#channels]
Feature map
[FM: o x o x k | o=(i-f)/s]
Pooling map
[PM: j x j x k | j=(o-m)/ss]
Pooling stride = ss = 1
Feature filter stride = s = 1

Operation	Equation
Convolution	$FM_{u,v}=I_{u:u+f,v:v+f} \cdot FF$
Max-pooling	$PM_{u,v}=\max(FM_{u:u+m,v:v+m})$
Average-pooling	$PM_{u,v}=\text{mean}(FM_{u:u+m,v:v+m})$

FIGURE 17. Convolution and pooling operation examples, equations, and dimensions. Note that no zero-padding is used in this example and simple parameters are used (e.g., strides of size 1). For more detailed information on these, we refer to [119].

All three publications on code generation from images used CNNs. Reference [29] used a CNN to classify objects in GUI screenshots. References [24], [114] used CNNs to extract features from images in combination with RNN or transformer decoders. CNNs are also used as encoders in other encoder-decoder models to create embeddings of code inputs [38], [66], [89]. References [38], [66] use CNNs in RNN encoder-decoder architectures while [89] use CNNs to augment a transformer architecture. Reference [69] uses a convolutional graph NN on AST inputs due to their ability to encode spatial information well.

ML augmented search for PBE is a method of generating programs by building an AST node-by-node where the next node is chosen by an ML model until the program satisfies all input-output examples. Explicit functional specifications such as input-output examples are important so that the model knows when to stop searching. The advantage of this method, compared to other code-generation techniques, is that a generated program is guaranteed to compile and behave as specified by the input-output examples given. Three of the five PBE studies [59], [63], [73] use ML augmented search. These studies use DSLs specifically designed to reduce the search space of all possible programs. General PLs like Java [16] or Python [39] have large program spaces that make these search techniques infeasible. For general PLs, RNNs or transformers are commonly utilized as they can more efficiently build sequences one token at a time by using decoding strategies such as beam search [120]. These decoding strategies usually do not provide any syntactic or functional guarantees for the generated program in contrast to PBE with AST search.

C. TOKENIZERS AND THE OUT-OF-VOCABULARY PROBLEM

Tokenization is a preprocessing step where an input string is partitioned into chunks. These chunks or "tokens" are mapped to numbers that ML models can process. The outputs of the models can be mapped back to tokens which form a part of the model output. Models with tokenizers recognize a finite set of tokens which are called the vocabulary of the model. Whenever a chunk of the input string does not have a matching token in the vocabulary, a special <*unknown*> token must be used. This results in a loss of information which is referred to as the out-of-vocabulary (OOV) problem.

Table 4 categorizes the selected studies of the review into three main tokenizer types: word-based, character-based, and subword-based. Figure 19 shows examples of the three main types of tokenizers: word-based, character-based, and subword-based tokenizers. Word-based tokenizers split words on whitespace characters with special rules for punctuation. In the context of code, parsers are frequently used to handle "code punctuation" such as brackets. Word tokens capture a complete unit of meaning from the input string but require a large vocabulary. Character-based tokenizers split the input string on every character. This simplifies the vocabulary, but each token usually holds little meaningful information. Subword-based tokenization provides a compromise between character and word-based tokenization. The vocabulary consists of all base characters as well as frequently occurring sequences of characters. Sub-word tokenizers are the most popular tokenizer type as shown in Figure 18. Figure 19 shows examples of each tokenizer type.

A problem that tokenizing source code faces is the fact that the number of unique "words" in code is generally much larger than in NL. This is mostly due to identifiers for functions or variables which are multiple words concatenated together using some naming or casing convention (e.g., a function that prints "hello world" to the console can be

TABLE 4. Overview of tokenizer types used by the selected studies. For non-string data, tokenizers are not used.

ML model type	Tokenizer type			Non-string data
	Character-based	Word-based	Subword-based	
RNN		[21], [24], [31], [43], [44], [114]	[47], [66], [67], [69], [89], [97]	[18], [34], [50], [73], [111]
Transformer [2]		[75], [76], [91], [114]	[79], [83], [85], [4], [95], [102]	
CNN			[38]	[29]
Other neural	[63]	[59]		[71], [108]
Non-neural	[63]	[62], [26]		[14], [46], [53]

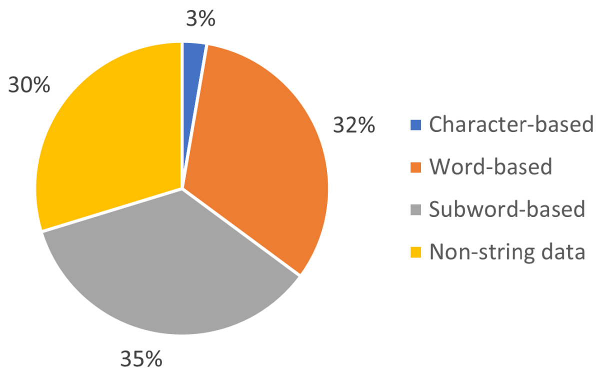


FIGURE 18. Percentages of selected studies that used different tokenizer types. Non-string data refers to studies that did not use tokenizers to preprocess data for ML models.

named "helloWorld" or "hello_world"). Therefore, tokenizers should be designed with the OOV problem in mind. Using character-level tokenization is a possible solution but has limits for statements with more than 15 characters [121]. Only one of the selected studies used character-based tokenization. Word-based tokenization split on whitespace characters is popular for processing NL but is susceptible to the OOV problem. References [38], [66], [83], [97] used subword-based tokenization by separating words on capital letters, underscores, and numbers as well. The byte-pair-encoding (BPE) algorithm [122], which builds a vocabulary of frequently occurring subwords in a training corpus, was used by [4], [47], [79], [85], [95], [102].

Custom tokenization processes can be used to keep vocabulary sizes small while encapsulating useful information in each token. Token copying is one such process observed in the selected publications. Reference [76] used positioned <*unknown*> tokens to copy tokens not in the model vocabulary from the input to the output string. Similarly, [75] keeps out of vocabulary tokens and a position encoding in a lookup table to replace <*unknown*> tokens during the decoding of the output. Reference [50] used copying mechanisms for AST tokens based on probabilities from the training data.

The second type of tokenization enhancement process observed in the selected literature is token abstraction by using multiple <*unknown*> tokens that distinguish different types of tokens. Reference [21] noticed that <*unknown*> tokens are terminal nodes in the AST and uses the node type as a replacement token rather than a generic <*unknown*> token. Similarly, [34], [67] abstracted OOV identifiers for methods and variables to the separate <*unknown*> tokens. Reference [31] used an identifier abstraction mechanism where all instances of an identifier were tokenized to a numbered identifier token.

D. VOLUME OF AVAILABLE DATA

The volume of data needed to train and evaluate code generation models is critical to the performance of the model. The studies in this review use open-source repository data, manually created data, and/or automatically generated data. Table 5 shows which data source types were used by each selected study. Percentages of the number of studies that use data from different combinations of these sources are shown in Figure 20.

Data from **open-source repositories** are used by 62% of selected publications. Open-source repositories provide large volumes and varied data in many PLs. Reference [4] shows an example of this in their proposed model which was trained on millions of lines of code to be able to generate a wide variety of multi-line Python [39] functions. These lines of code were collected from Github [9]. Eighteen of the selected publications [4], [14], [21], [31], [34], [38], [43], [46], [47], [66], [67], [71], [79], [85], [91], [95], [97], [102] used data sourced from git repositories. This data often includes source code, documentation, as well as repository change information. This last type of data is especially useful for APR studies as these changes are occasionally pre- and post-bug-fix pairs. All APR studies used open-source repository data. Reference [76] used data sourced from StackOverflow [123], a forum where users ask and answer programming-related questions, which is especially useful for obtaining NL-code pairs.

A lack of available data led five studies [26], [44], [53], [59], [114] to **manually create data**. For example, [75] manually wrote translations in a target programming language given code from a source programming language. Manually created datasets made publicly available from other studies were used by six [29], [50], [62], [63], [76] of the selected studies. Data generation performed by humans is time-intensive which is why it is avoided where possible.

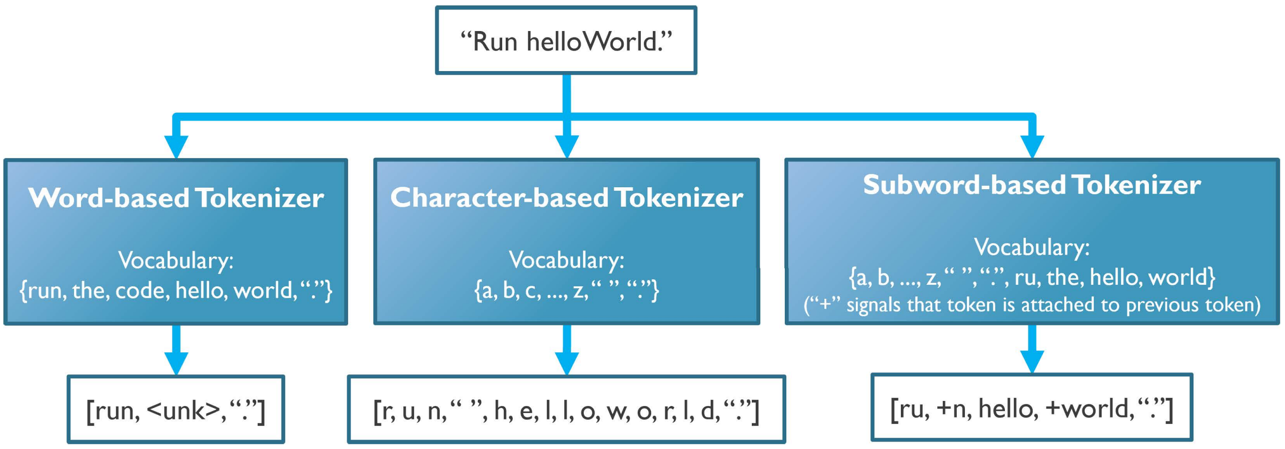

"Run helloWorld."
Word-based Tokenizer
Vocabulary:
{run, the, code, hello, world, "."}
Character-based Tokenizer
Vocabulary:
{a, b, c, ..., z, " ", "."}
Subword-based Tokenizer
Vocabulary:
{a, b, ..., z, " ", ".", ru, the, hello, world}
("+" signals that token is attached to previous token)
[run, <unk>, "."]
[r, u, n, " ", h, e, l, l, o, w, o, r, l, d, "."]
[ru, +n, hello, +world, "."]

FIGURE 19. Example word-, character-, and subword-based tokenizers with limited vocabularies. Each tokenizer processes the input string differently. The output elements are called tokens. These tokens can then be translated to numbers using a lookup table to create a valid input for a neural network. The <*unknown*> token is denoted as <unk>.

TABLE 5. Overview of data source types of the datasets used by the selected studies. Manually created datasets have humans-in-the-loop during data generation while automatic generation implies data created in a programmatic manner.

Paradigm	Application category	Data source type		
		Open-source repositories	Manually created	Automatically generated
Description-to-code	From NL descriptions	[46], [47], [79], [4], [91], [76]	[50], [76], [89]	
	PBE		[63]	[59], [63], [73], [91], [108]
	From images		[29], [114]	[24], [114]
	From other structured data inputs	[18] [111]	[53]	
Code-to-description	Documentation generation	[21], [34], [66], [69], [83], [4], [95]	[44], [4]	
Code-to-code	APR	[31], [38], [43], [71], [85], [97]		
	Cross-PL translation		[26]	[26], [75]
	Refactoring	[14]	[14], [62]	
	Code completion	[67], [102]		

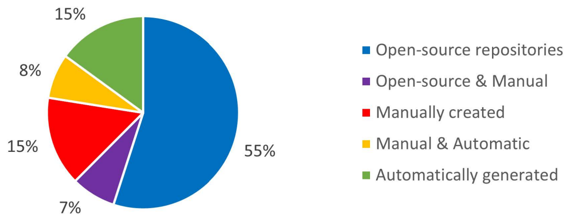


FIGURE 20. Percentages of data-source types used by the selected studies.

If there is not enough data readily available, **automatic data generation** methods can also be considered. Machine-generated data is used in all of the works reviewed in the domain of PBE since any randomly generated input-output pairs for numeric calculations or string transformations are usually acceptable to derive a program from. References [26], [114] generate automatic data in addition to creating data manually to achieve a balance between quality and quantity of data.

The need for a partition of the dataset for evaluation purposes reduces the amount of data that can be used for training. This problem is usually resolved by using cross-validation. Cross-validation involves training many models and is often computationally expensive. This is especially the case for large language models. More discussion on this topic is provided in Section IV-G.

E. QUALITY OF AVAILABLE DATA

As mentioned in the previous section, researchers leverage automatic mining from open-source repositories to obtain large volumes of data. Even after preprocessing and filtering, the quality of the automatically collected data can be unreliable. Automatically mined source code often has dependencies that can be difficult to obtain automatically, making the source code non-executable. Reference [91] found this to be problematic for obtaining input-output pairs in the domain of PBE. Executable source code is also important for functional evaluation of other applications. References [4], [114] manually curated test datasets to ensure better quality testing data. An alternative to evaluation by comparing synthesized code with code snippet is human evaluation. This is done

TABLE 6. Overview of evaluation methods used by the selected studies. Token match metrics used in the review which are popular for NLP applications include BLEU [22], CIDEr [35], ROUGE [37], and METEOR [36]. Examples of other token match metrics are "exact match" and "token accuracy". Dynamic analysis analyzes runtime behavior of code while static analysis does not require code to be executed.

Paradigm	Application category	Evaluation methods						
		Token match		Dynamic analysis		Static analysis		Other
		From NLP	Other	Functional correctness	Time-to-completion	Syntactic correctness	Human evaluation	
Description-to-code	From NL descriptions	[47], [50], [76], [79], [91]	[46], [47], [50]	[4]				
	PBE		[73]	[91], [108]	[59], [63]			
	From images		[24], [114]					[29]
	From other structured data inputs		[18], [111]	[53]	[53]	[111]		
Code-to-description	Documentation generation	[21], [34], [66], [69], [83], [89], [95]					[44], [4], [95]	
Code-to-code	APR	[97]	[31], [38], [43], [71], [85]			[31]	[43]	
	Cross-PL translation	[75]	[75]					[26]
	Refactoring		[14]					[62]
	Code completion	[102]	[67], [102]			[102]		[102]

by [4], [43], [44], [95]. Manually generating or evaluating data is time-intensive which is why automatic data mining or generation and automatic evaluation are more common.

F. EVALUATING GENERATED CODE

Evaluating the quality of synthesized code is done either by comparing it to a "ground-truth" code statement, analyzing it statically, or analyzing it at runtime. Table 6 shows which evaluation methods each of the selected studies used.

The selected studies that evaluate synthesized code by comparing it to ground-truth code statements do so at the token level. Token comparisons are either performed by algorithms from NLP literature such as BLEU [22],CIDEr [35], ROUGE [37], and METEOR [36] or other metrics such as "exact match" and "token accuracy". Figure 21 shows that token match is used by 76% of the selected studies. Token match evaluation is popular because the same data used for training can be used for evaluation, it is automatic, and does not require the code to be executable. Using token match metrics from NLP for documentation generation is not a problem as these metrics show they correlate with human judgments. However, reference [4] shows that this is not the case for synthesized code. Reference [124] argues that BLEU and *exact match* do not properly capture code semantics and instead propose a code-specific metric, CodeBLEU. CodeBLEU uses a weighted sum of BLEU, BLEU weighted on code keywords, syntactic similarity by comparing ASTs, and data-flow similarity [124]. CodeBLEU was not used by any of the works reviewed. References [18], [111] used custom, code-specific token match metrics to measure program equivalence to better measure code semantics.

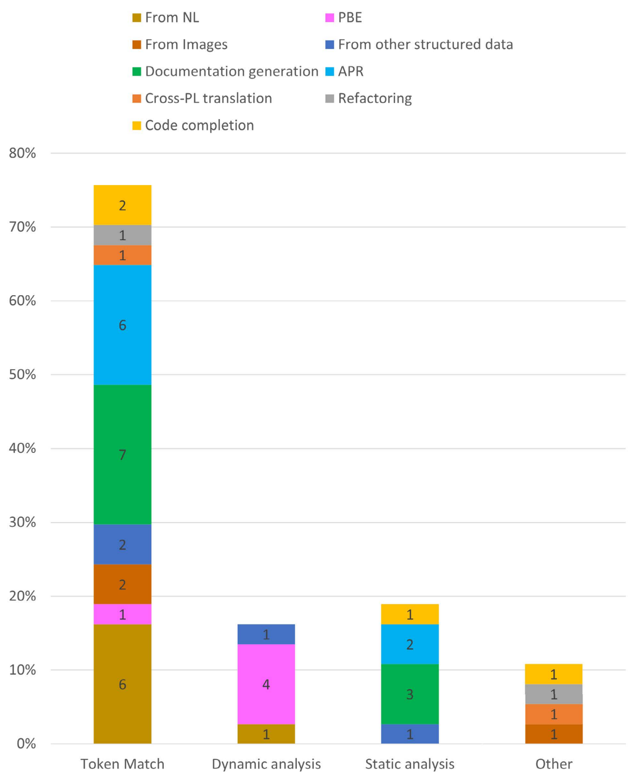


FIGURE 21. The number of selected studies that used certain evaluation methods. The vertical axis gives the percentage of the number of studies that used a certain evaluation method out of 37, the total number of selected studies.

Dynamic analysis involves evaluating the functional correctness and/or the time-to-completion of executable code at runtime. Functional correctness requires certain types of data such as input-output examples (e.g. PBE), unit tests [4], or formal specifications [53]. This is a code-appropriate metric and allows for different code implementations that are functionally equivalent to obtain the same score. This is unlike token match evaluation which will give a better score to an implementation that is most similar to reference code. Furthermore, reference [4] argues that functional correctness correlates well with what humans would consider to be quality code. Dynamic analysis often requires the code to be syntactically correct which is not always guaranteed by ML-based code generation methods as discussed at the end of Section IV-B.

Static analysis is more accessible than dynamic analysis since the code does not need to be executable and no ground-truth references are needed. However, only using syntactic correctness as a metric to validate models can lead to degeneration where synthesized code does not exhibit any desirable functional or semantic properties [111]. Human evaluation is a holistic evaluation method but is time-consuming and requires programmers with knowledge of the PL the code is written.

G. FUTURE WORK

In this section, we list three suggestions for future work that would contribute to the field of code generation using ML.

Improving language model efficiency is our first suggestion for future work. Models such as transformers [2] are good for general code generation tasks but are extremely data-hungry [4], [95]. Training and evaluating these models are therefore computationally expensive. Similarly, improving the energy efficiency of language models would also lower the barrier of entry for research. High energy consumption leads to high monetary costs. References [43], [71], [75], [83] mention computation costs as restrictive to their research.

Ensemble learning is our second suggestion for future work. Some models excel in specific contexts while performing poorly in general. References [69], [71] are examples of this as they found that different models performed better on certain bug types in the context of APR. Reference [75] gives an example of how training and evaluating a model for three specific types of cross-PL translations required half as much data to achieve similar performance compared to a model trained and evaluated on four types of translations. Studying ensembles that combine the strengths of different models to improve performance over a variety of cases is a promising direction for future work.

New ways of using Abstract Syntax Trees (AST) representations of source code is our final suggestion for future work. Multiple studies discussed in this review, such as [21], [95], use AST representations of code for their models. Further exploitation of this data structure, which is characteristic of PLs, is recommended for future research. Generalized ASTs over multiple programming languages could lead to greater transfer learning capabilities and models that generalize to multiple languages. Code-specific decoding methods for sequential output models remain unexplored to the best of our knowledge. A decoding method that exploits AST or other syntax information could lead to more efficient and syntactically correct synthesized code.

V. THREATS TO VALIDITY

This section first discusses threats to the validity of the search criteria used by this review. Afterward, the threats to the validity of the models proposed by the selected studies are discussed.

One search phrase was used across two databases due to a large number of publications returned. While we believe the search phrase accurately and precisely defines the types of works we aimed to survey, we recognize that it is sensitive to variations in terminology and/or missing keywords. For example, [6] presented an influential code language model, CodeBERT, but never mentions ''machine learning'' even though ML techniques were used by the study. This led to it not being returned by our search. Other examples of influential models that were not retrieved by our searches are PyMT5 [3], Code2Seq [5], and TransCoder [7]. This survey should first and foremost be used as an introduction to the various applications, ML models, tokenizers, data, and evaluation methods used in the various sub-domains of code generation. We encourage readers who want to read more about a topic covered in this review to perform citation searches on the selected publications, or the influential publications mentioned above, to find additional relevant literature.

To investigate whether our chosen search phrase retrieves a disproportionate amount of description-to-code publications, a small experiment was conducted. Table 7 compares retrieval statistics for the original query for publication dates between 2016 and 2021 with a similar query which also includes two similar search phrases. These additional search phrases, ''code modification using machine learning'' and ''code summarization using machine learning'', are more specific for the code-to-code and code-to-description paradigms, respectively. This extended search query retrieved roughly 8% more publications than the query with only the original search query. Extrapolated to the number of selected studies, this increase leads to 40 selected studies. We consider this to be a relatively small number of additional studies and therefore conclude that our search phrase adequately covers all three paradigms discussed in this review.

As mentioned in Section IV-F, many of the metrics used to evaluate generated code are not code-specific and rely on comparisons with a ground truth code statement. Comparing generated code to a ground-truth program is limiting as the space of valid output programs for a given input is generally large. Furthermore, many token match metrics such as BLEU are sensitive to the tokenizer used [125]. This means that results from different studies using these metrics should only be compared if a similar tokenizer is used to tokenize the output before calculating the token match metric. Functional correctness is the best metric to avoid this problem but requires extra data in the form of test programs.

TABLE 7. Statistics for database publication searches using different search phrases and a publication date range between 2016 and 2021. The search phrase "code generation using machine learning" is the one applied to identify publications for this review. Adding two additional search phrases, "code modification using machine learning" and "code summarization using machine learning", using the *or* logical operator retrieves only slightly more publications.

Statistic	Search phrases applied as inclusion criteria	
	"code generation using machine learning"	"code generation using machine learning" or "code modification using machine learning" or "code summarization using machine learning"
Number of publications retrieved	887 (613=ArXiv, 274=IEEE Xplore)	957 (670=ArXiv, 287=IEEE Xplore)
Number of publications retrieved as % of the number of publications retrieved from original search query	100%	108%
Number of selected publications	37	40 (extrapolated)

VI. CONCLUSION

This systematic review selected 37 works published in the last six years in the arXiv [11] and IEEE Xplore [12] databases that proposed ML models for code generation, documentation generation, and code modification. Each publication's application, model, datasets, results, limitations, and proposed future work were summarized. Then, the general findings of these 37 studies were discussed.

The discussion started by introducing the various application categories of the selected studies. The most popular applications of the selected publications include code generation from NL descriptions, documentation generation, APR, and PBE. The popular model types used by these studies such as RNNs, transformers [2], CNNs, and ML augmented search, were introduced and compared in the context of different applications. RNNs and transformer models were used mostly for code generation given natural language (NL) description as well as documentation generation. In general, transformer models outperformed RNN models when the two were compared by an evaluative study. CNNs are used for image data but also to augment other models such as transformers.

Different tokenization strategies used by these publications are listed. How some of these strategies handle the out-of-vocabulary problem is also discussed. Effective tokenization processes used by the reviewed publications are subword-tokenization, copy-mechanisms, and/or multiple <*unknown*> tokens to capture a particular subset of possible tokens such as variable identifiers.

Limitations in the quantity and quality of data for code generation models were highlighted in respective sections of the discussion. Language models require large datasets, especially when the model is expected to be able to generate code in many different types of contexts. Automatic mining of online sources such as Github [9] is often needed to obtain large enough volumes of data. The quality of automatically mined source code varies greatly. This led two studies [4], [114] to manually create their own test datasets. Automatically generating data is a fast alternative for obtaining data but is only appropriate for certain contexts such as PBE.

The question of how to measure the quality of code is also discussed in a corresponding section. Automatic evaluation by comparing tokens of generated code to tokens from ground-truth references was conducted by 76% of the selected studies. However, most of these token match algorithms are not appropriate for evaluating code. Functional correctness is a viable alternative but requires certain types of data and requires the generated code to be executable. Static analysis doesn't require the code to be executable but can lead to degeneration when only syntax is considered or is time-consuming if performed by humans.

Finally, three promising directions for future work were suggested: (i) improving the efficiency of language models, (ii) ensemble learning for specialized models, and (iii) more research on the possibilities for exploiting abstract syntax tree representations of source code.

REFERENCES

[1] W. W. Royce, "Managing the development of large software systems: Concepts and techniques," in *Proc. 9th Int. Conf. Softw. Eng.*, 1987, pp. 328–338.
[2] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2017, *arXiv:1706.03762*.
[3] C. B. Clement, D. Drain, J. Timcheck, A. Svyatkovskiy, and N. Sundaresan, "PyMT5: Multi-mode translation of natural language and Python code with transformers," 2020, *arXiv:2010.03150*.
[4] M. Chen *et al.*, "Evaluating large language models trained on code," 2021.
[5] U. Alon, S. Brody, O. Levy, and E. Yahav, "Code2seq: Generating sequences from structured representations of code," 2018, *arXiv:1808.01400*.
[6] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," 2020, *arXiv:2002.08155*.
[7] M.-A. Lachaux, B. Roziere, L. Chanussot, and G. Lample, "Unsupervised translation of programming languages," 2020, *arXiv:2006.03511*.
[8] Tabnine. *Code Faster With AI Code Completions*. Accessed: Mar. 6, 2022. [Online]. Available: https://www.tabnine.com/
[9] Github. *Github: Where the World Builds Software*. Accessed: Mar. 6, 2022. [Online]. Available: https://github.com/
[10] Github. *Github Copilot—Your AI Pair Programmer*. Accessed: Mar. 6, 2022. [Online]. Available: https://copilot.github.com/
[11] Cornell University. *E-Print Scholarly Articles Archive*. Accessed: Mar. 6, 2022. [Online]. Available: https://arxiv.org/
[12] IEEE. *IEEE Xplore Digital Library*. Accessed: Mar. 6, 2022. [Online]. Available: https://www.ieee.org/publications/subscriptions/products/mdl/ieeexplore-access.html
[13] (2011). *Finding What Works in Health Care: Standards for Systematic Reviews*, J. Eden, L. Levit, A. Berg, and S. Morton, Eds. Washington, DC, USA: The National Academies Press. Institute of Medicine. [Online]. Available: https://www.nap.edu/catalog/13059/finding-what-works-in-health-care-standards-for-systematic-reviews
[14] T. Parr and J. Vinju, "Technical report: Towards a universal code formatter through machine learning," 2016, *arXiv:1606.08866*. [Online]. Available: https://arxiv.org/abs/1606.08866, doi: 10.48550/arxiv.1606.08866.
[15] T. Parr. *ANTLR (Another Tool for Language Recognition)*. Accessed: Mar. 6, 2022. [Online]. Available: https://www.antlr.org/

[16] *Oracle*. Java. Accessed: Mar. 6, 2022. [Online]. Available: https://www.java.com/en/
[17] Quorum. *The Quorum Programming Language*. Accessed: Mar. 6, 2022. [Online]. Available: https://quorumlanguage.com/
[18] V. Murali, L. Qi, S. Chaudhuri, and C. Jermaine, "Neural sketch learning for conditional program generation," 2017, *arXiv:1703.05698*. [Online]. Available: https://arxiv.org/abs/1703.05698, doi: 10.48550/arxiv.1703.05698.
[19] AndroidDrawer. *Androiddrawer, Andriod App Repository*. Facebook Page. Accessed: Jun. 12, 2022. [Online]. Available: https:/www.facebook.com/android.drawer/about/?ref=page_internal
[20] K. Sohn, X. Yan, and H. Lee, "Learning structured output representation using deep conditional generative models," in *Proc. 28th Int. Conf. Neural Inf. Process. Syst.*, Cambridge, MA, USA, vol. 2, 2015, pp. 3483–3491.
[21] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proc. 26th Conf. Program Comprehension*, New York, NY, USA, May 2018, pp. 200–210, doi: 10.1145/3196321.3196334.
[22] K. Papineni, S. Roukos, T. Ward, and W. J. Zhu, "BLEU: A method for automatic evaluation of machine translation," in *Proc. 40th Annu. Meeting Assoc. Comput. Linguistics*, Philadelphia, PA, USA, Jul. 2002, pp. 311–318. [Online]. Available: https://aclanthology.org/P02-1040
[23] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proc. 54th Annu. Meeting Assoc. Comput. Linguistics*, 2016.
[24] Z. Zhu, Z. Xue, and Z. Yuan, "Automatic graphics program generation using attention-based hierarchical decoder," 2018.
[25] T. Beltramelli, "Pix2code: Generating code from a graphical user interface screenshot," 2017, *arXiv:1705.07962*.
[26] Y. Kim and H. Kim, "Translating CUDA to OpenCL for hardware generation using neural machine translation," in *Proc. IEEE/ACM Int. Symp. Code Gener. Optim. (CGO)*, Feb. 2019, pp. 285–286.
[27] P. Vingelmann and F. H. Fitzek. (2020). *Cuda, Release: 10.2.89*. [Online]. Available: https://developer.nvidia.com/cuda-toolkit
[28] NVIDIA Developer. *Opencl*. Accessed: Mar. 6, 2022. [Online]. Available: https://developer.nvidia.com/opencl
[29] B. Asiroglu, B. R. Mete, E. Yildiz, Y. Nalcakan, A. Sezen, M. Dagtekin, and T. Ensari, "Automatic HTML code generation from mock-up images using machine learning techniques," in *Proc. Sci. Meeting Elect.-Electron. Biomed. Eng. Comput. Sci. (EBBT)*, Apr. 2019, pp. 1–4.
[30] V. Jain, P. Agrawal, S. Banga, R. Kapoor, and S. Gulyani, "Sketch2Code: Transformation of sketches to UI in real-time using deep neural network," 2019, *arXiv:1910.08930*.
[31] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," 2018, *arXiv:1812.08693*. [Online]. Available: https://arxiv.org/abs/1812.08693, doi: 10.48550/arxiv.1812.08693.
[32] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proc. 29th ACM/IEEE Int. Conf. Automated Softw. Eng.*, New York, NY, USA, Sep. 2014, pp. 313–324, doi: 10.1145/2642937.2642982.
[33] Z. Chen and M. Monperrus, "The CodRep machine learning on source code competition," 2018, *arXiv:1807.03200*.
[34] Y. Shido, Y. Kobayashi, A. Yamamoto, A. Miyamoto, and T. Matsumura, "Automatic source code summarization with extended tree-LSTM," 2019, *arXiv:1906.08094*. [Online]. Available: https://arxiv.org/abs/1906.08094, doi: 10.48550/arxiv.1906.08094.
[35] R. Vedantam, C. L. Zitnick, and D. Parikh, "CIDEr: Consensus-based image description evaluation," 2014, *arXiv:1411.5726*.
[36] S. Banerjee and A. Lavie, "METEOR: An automatic metric for mt evaluation with improved correlation with human judgments," in *Proc. ACL Workshop Intrinsic Extrinsic Eval. Meas. Mach. Transl. Summarization*, Jun. 2005, pp. 65–72. [Online]. Available: https://aclanthology.org/W05-0909
[37] C.-Y. Lin, "ROUGE: A package for automatic evaluation of summaries," in *Proc. Workshop ACL Text Summarization Branches Out*, Jul. 2004, pp. 74–81. [Online]. Available: https://aclanthology.org/W04-1013
[38] T. Lutellier, L. Pang, V. H. Pham, M. Wei, and L. Tan, "ENCORE: Ensemble learning using convolution neural machine translation for automatic program repair," 2019, *arXiv:1906.08691*. [Online]. Available: https://arxiv.org/abs/1906.08691, doi: 10.48550/arxiv.1906.08691.
[39] Python Software Foundation. *Welcome to Python*. Accessed: Mar. 6, 2022. [Online]. Available: https://www.python.org/
[40] Standard C++ Foundation. *Standard C++*. Accessed: Mar. 6, 2022. [Online]. Available: https://isocpp.org/
[41] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs," in *Proc. Int. Symp. Softw. Test. Anal. (ISSTA)*, New York, NY, USA, 2014, pp. 437–440, doi: 10.1145/2610384.2628055.
[42] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama, "Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge," in *Proc. Companion ACM SIGPLAN Int. Conf. Syst., Program., Lang., Appl., Softw. Hum.*, New York, NY, USA, 2017, pp. 55–56, doi: 10.1145/3135932.3135941.
[43] H. Hata, E. Shihab, and G. Neubig, "Learning to generate corrective patches using neural machine translation," 2018, *arXiv:1812.07170*. [Online]. Available: https://arxiv.org/abs/1812.07170, doi: 10.48550/arxiv.1812.07170.
[44] A. Takahashi, H. Shiina, and N. Kobayashi, "Automatic generation of program comments based on problem statements for computational thinking," in *Proc. 8th Int. Congr. Adv. Appl. Informat. (IIAI-AAI)*, Jul. 2019, pp. 629–634.
[45] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, 2nd ed. Upper Saddle River, NJ, USA: Prentice-Hall, 1988.
[46] H. Phan, "Self Learning from large scale code corpus to infer structure of method invocations," 2019, *arXiv:1909.03147*. [Online]. Available: https://arxiv.org/abs/1909.03147, doi: 10.48550/arxiv.1909.03147.
[47] R. Agashe, S. Iyer, and L. Zettlemoyer, "JuICe: A large scale distantly supervised dataset for open domain context-based code generation," in *Proc. Conf. Empirical Methods Natural Lang. Process. 9th Int. Joint Conf. Natural Lang. Process. (EMNLP-IJCNLP)*, 2019.
[48] Jupyter Team. *Jupyter Notebok Documentation*. Accessed: Mar. 6, 2022. [Online]. Available: https://jupyter-notebook.readthedocs.io/en/stable/notebook.html
[49] Jupyter Team. *Nbgrader Documentation*. Accessed: Mar. 6, 2022. [Online]. Available: https://nbgrader.readthedocs.io/en/stable/
[50] R. Shin, M. Allamanis, M. Brockschmidt, and O. Polozov, "Program synthesis and semantic parsing with learned code idioms," 2019, *arXiv:1906.10816*. [Online]. Available: https://arxiv.org/abs/1906.10816, doi: 10.48550/arxiv.1906.10816.
[51] W. Ling, P. Blunsom, E. Grefenstette, K. M. Hermann, T. Kočiský, F. Wang, and A. Senior, "Latent predictor networks for code generation," in *Proc. 54th Annu. Meeting Assoc. Comput. Linguistics*, 2016, pp. 1–11.
[52] T. Yu, R. Zhang, K. Yang, M. Yasunaga, D. Wang, Z. Li, J. Ma, I. Li, Q. Yao, S. Roman, Z. Zhang, and D. Radev, "Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task," in *Proc. Conf. Empirical Methods Natural Lang. Process.*, 2019, pp. 1–11.
[53] Y. Koroglu and A. Sen, "Reinforcement learning-driven test generation for Android GUI applications using formal specifications," 2019, *arXiv:1911.05403*. [Online]. Available: https://arxiv.org/abs/1911.05403, doi: 10.48550/arxiv.1911.05403.
[54] C. Gultniek. (2010). *F-Droid Benchmarks*. [Online]. Available: https://f-droid.org/
[55] R. N. Zaeem, M. R. Prasad, and S. Khurshid, "Automated generation of oracles for testing user-interaction features of mobile apps," in *Proc. IEEE 7th Int. Conf. Softw. Test., Verification Validation*, Mar. 2014, pp. 183–192.
[56] Android Developers. *Android UI/Application Exerciser Monkey*. Accessed: Jun. 12, 2022. [Online]. Available: https://developer.android.com/studio/test/monkey
[57] Y. Koroglu, A. Sen, O. Muslu, Y. Mete, C. Ulker, T. Tanriverdi, and Y. Donmez, "QBE: QLearning-based exploration of Android applications," in *Proc. IEEE 11th Int. Conf. Softw. Test., Verification Validation (ICST)*, Apr. 2018, pp. 105–115.
[58] U. Hustadt. *Metric Temporal Logic: Tools and Experiments*. Accessed: Mar. 16, 2022. [Online]. Available: https://cgi.csc.liv.ac.uk/ ullrich/MTL/
[59] S. Shim, P. Patil, R. R. Yadav, A. Shinde, and V. Devale, "DeeperCoder: Code generation using machine learning," in *Proc. 10th Annu. Comput. Commun. Workshop Conf. (CCWC)*, Jan. 2020, pp. 0194–0199.
[60] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014, *arXiv:1412.6980*. [Online]. Available: https://arxiv.org/abs/1412.6980, doi: 10.48550/arxiv.1412.6980.
[61] T. Dozat, "Incorporating Nesterov momentum into Adam," Comput. Sci., Stanford Univ., Stanford, CA, USA, Tech. Rep. 54, 2016. Accessed: Jul. 7, 2022. [Online]. Available: https://cs229.stanford.edu/proj2015/054_report.pdf
[62] A. J. Stein, L. Kapllani, S. Mancoridis, and R. Greenstadt, "Exploring paraphrasing techniques on formal language for generating semantics preserving source code transformations," in *Proc. IEEE 14th Int. Conf. Semantic Comput. (ICSC)*, Feb. 2020, pp. 242–248.

[63] K. Morton, W. Hallahan, E. Shum, R. Piskac, and M. Santolucito, "Grammar filtering for syntax-guided synthesis," 2020, *arXiv:2002.02884*. [Online]. Available: https://arxiv.org/abs/2002.02884, doi: 10.48550/arxiv.2002.02884.
[64] C. W. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli, "Cvc4," in *Proc. CAV*, 2011, pp. 171–177.
[65] R. Alur, D. Fisman, R. Singh, and A. Solar-Lezama, "SyGuS-comp 2017: Results and analysis," *Electron. Proc. Theor. Comput. Sci.*, vol. 260, pp. 97–115, Nov. 2017, doi: 10.4204/EPTCS.260.9.
[66] Y. Choi, S. Kim, and J.-H. Lee, "Source code summarization using attention-based keyword memory networks," in *Proc. IEEE Int. Conf. Big Data Smart Comput. (BigComp)*, Feb. 2020, pp. 564–570.
[67] C. Watson, M. Tufano, K. Moran, G. Bavota, and D. Poshyvanyk, "On learning meaningful assert statements for unit test cases," in *Proc. ACM/IEEE 42nd Int. Conf. Softw. Eng.*, Jun. 2020, pp. 1398–1409.
[68] JUnit Team. *Developer-Side Testing on the Java Virtual Machine*. Accessed: Mar. 6, 2022. [Online]. Available: https://junit.org/junit5/
[69] A. LeClair, S. Haque, L. Wu, and C. Mcmillan, "Improved code summarization via a graph neural network," in *Proc. 28th Int. Conf. Program Comprehension*, Jul. 2020, pp. 184–195.
[70] A. LeClair and C. Mcmillan, "Recommendations for datasets for source code summarization," in *Proc. Conf. North*, 2019, pp. 1–7.
[71] S. Chakraborty, Y. Ding, M. Allamanis, and B. Ray, "CODIT: Code editing with tree-based neural models," *IEEE Trans. Softw. Eng.*, vol. 48, no. 4, pp. 1385–1399, Apr. 2022, doi: 10.1109/TSE.2020.3020502.
[72] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk, "On learning meaningful code changes via neural machine translation," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng. (ICSE)*, May 2019, pp. 25–36.
[73] K. Grouwstra, "Type-driven neural programming by example," 2020, *arXiv:2008.12613*. [Online]. Available: https://arxiv.org/abs/2008.12613, doi: 10.48550/arxiv.2008.12613.
[74] Haskell. *Haskell Language*. Accessed: Mar. 6, 2022. [Online]. Available: https://www.haskell.org/
[75] M. H. Hassan, O. A. Mahmoud, O. I. Mohammed, A. Y. Baraka, A. T. Mahmoud, and A. H. Yousef, "Neural machine based mobile applications code translation," in *Proc. 2nd Novel Intell. Lead. Emerg. Sci. Conf. (NILES)*, Oct. 2020, pp. 302–307.
[76] C. Gemmell, F. Rossetto, and J. Dalton, "Relevance transformer: Generating concise code snippets with relevance feedback," *Proc. 43rd Int. ACM SIGIR Conf. Res. Develop. Inf. Retr.*, Jul. 2020, pp. 2005–2008, doi: 10.1145/3397271.3401215.
[77] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura, "Learning to generate pseudo-code from source code using statistical machine translation," in *Proc. 30th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2015, pp. 574–584.
[78] P. Yin, B. Deng, E. Chen, B. Vasilescu, and G. Neubig, "Learning to mine aligned code and natural language pairs from stack overflow," in *Proc. 15th Int. Conf. Mining Softw. Repositories*, May 2018, pp. 476–486.
[79] L. Perez, L. Ottens, and S. Viswanathan, "Automatic code generation using pre-trained language models," 2021, *arXiv:2102.10535*. [Online]. Available: https://arxiv.org/abs/2102.10535, doi: 10.48550/arxiv.2102.10535.
[80] T. B. Brown *et al.*, "Language models are few-shot learners," 2020, *arXiv:2005.14165*. [Online]. Available: https://arxiv.org/abs/2005.14165, doi: 10.48550/arxiv.2005.14165.
[81] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "CodeSearchNet challenge: Evaluating the state of semantic code search," 2019, *arXiv:1909.09436*. [Online]. Available: https://arxiv.org/abs/1909.09436, doi: doi.org/10.48550/arxiv.1909.09436.
[82] Y. Kim, Y. Jernite, D. Sontag, and A. M. Rush, "Character-aware neural language models," 2015, *arXiv:1508.06615*.
[83] R. Shahbazi, R. Sharma, and F. H. Fard, "API2Com: On the improvement of automatically generated code comments using API documentations," in *Proc. IEEE/ACM 29th Int. Conf. Program Comprehension (ICPC)*, May 2021, pp. 411–421.
[84] W. Wang, Y. Zhang, Z. Zeng, and G. Xu, "TranS^3: A transformer-based framework for unifying code summarization and code search," 2020, *arXiv:2003.03238*. [Online]. Available: https://arxiv.org/abs/2003.03238, doi: 10.48550/arxiv.2003.03238.
[85] N. Jiang, T. Lutellier, and L. Tan, "CURE: Code-aware neural machine translation for automatic program repair," in *Proc. IEEE/ACM 43rd Int. Conf. Softw. Eng. (ICSE)*, May 2021, pp. 1161–1173, doi: 10.1109/ICSE43902.2021.00107.
[86] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "CoCoNuT: Combining context-aware neural translation models using ensemble for program repair," in *Proc. 29th ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, Jul. 2020, pp. 101–114.
[87] T. B. Brown *et al.*, "Language models are few-shot learners," 2020, *arXiv:2005.14165*. [Online]. Available: https://arxiv.org/abs/2005.14165, doi: 10.48550/arxiv.2005.14165.
[88] B. Wang and A. Komatsuzaki. (May 2021). *GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model*. [Online]. Available: https://github.com/kingoflolz/mesh-transformer-jax
[89] G. Yang, Y. Zhou, X. Chen, and C. Yu, "Fine-grained pseudo-code generation method via code feature extraction and transformer," in *Proc. 28th Asia–Pacific Softw. Eng. Conf. (APSEC)*, Dec. 2021, pp. 213–222.
[90] S. Kulal, P. Pasupat, K. Chandra, M. Lee, O. Padon, A. Aiken, and P. Liang, "SPoC: Search-based pseudocode to code," 2019, *arXiv:1906.04908*.
[91] J. Hong, D. Dohan, R. Singh, C. Sutton, and M. Zaheer, "Latent programmer: Discrete latent codes for program synthesis," 2020, *arXiv:2012.00377*. [Online]. Available: https://arxiv.org/abs/2012.00377, doi: 10.48550/arxiv.2012.00377.
[92] E. Parisotto, A.-R. Mohamed, R. Singh, L. Li, D. Zhou, and P. Kohli, "Neuro-symbolic program synthesis," 2016, *arXiv:1611.01855*. [Online]. Available: https://arxiv.org/abs/1611.01855, doi: 10.48550/arxiv.1611.01855.
[93] Y. Wan, Z. Zhao, M. Yang, G. Xu, H. Ying, J. Wu, and P. S. Yu, "Improving automatic source code summarization via deep reinforcement learning," in *Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng.*, New York, NY, USA, Sep. 2018, pp. 397–407, doi: 10.1145/3238147.3238206.
[94] J. Devlin, J. Uesato, S. Bhupatiraju, R. Singh, A. Mohamed, and P. Kohli, "Robustfill: Neural program learning under noisy I/O," in *Proc. 34th Int. Conf. Mach. Learn. (ICML)*, Mar. 2017, pp. 990–998. [Online]. Available: https://www.microsoft.com/en-us/research/publication/robustfill-neural-program-learning-noisy-io/
[95] G. Yang, X. Chen, J. Cao, S. Xu, Z. Cui, C. Yu, and K. Liu, "ComFormer: Code comment generation via transformer and fusion method-based hybrid code representation," in *Proc. 8th Int. Conf. Dependable Syst. Their Appl. (DSA)*, Aug. 2021, pp. 30–41.
[96] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation with hybrid lexical and syntactical information," *Empirical Softw. Eng.*, vol. 25, no. 3, pp. 2179–2217, May 2020.
[97] S. Wang, K. Liu, B. Lin, L. Li, J. Klein, X. Mao, and T. F. Bissyandé, "Beep: Fine-grained fix localization by learning to predict buggy code elements," 2021, *arXiv:2111.07739*.
[98] R.-M. Karampatsis and C. Sutton, "How often do single-statement bugs occur? The ManySStuBs4J dataset," 2019, *arXiv:1905.13334*.
[99] F. Madeiral, S. Urli, M. Maia, and M. Monperrus, "BEARS: An extensible Java bug benchmark for automatic program repair studies," in *Proc. IEEE 26th Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*, Feb. 2019, pp. 468–478.
[100] R. Saha, Y. Lyu, W. Lam, H. Yoshida, and M. Prasad, "Bugs.jar: A large-scale, diverse dataset of real-world Java bugs," in *Proc. IEEE/ACM 15th Int. Conf. Mining Softw. Repositories (MSR)*, May 2018, pp. 10–13.
[101] K. Liu, D. Kim, A. Koyuncu, L. Li, T. F. Bissyande, and Y. Le Traon, "A closer look at real-world patches," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Sep. 2018, pp. 275–286.
[102] T. Korbak, H. Elsahar, G. Kruszewski, and M. Dymetman, "Controlling conditional language models without catastrophic forgetting," 2021, *arXiv:2112.00791*.
[103] S. Black, L. Gao, P. Wang, C. Leahy, and S. R. Biderman, "GPT-Neo: Large scale autoregressive language modeling with mesh-tensorflow," 2021.
[104] G. van Rossum, B. Warsaw, and N. Coghlan. (2001). *Style Guide for Python Code*. [Online]. Available: https://peps.python.org/pep-0008/
[105] P. Bielik, V. Raychev, and M. Vechev, "PHOG: Probabilistic model for code," in *Proc. The 33rd Int. Conf. Mach. Learn.*, vol. 48, M. F. Balcan and K. Q. Weinberger, Eds. New York, New York, USA, Jun. 2016, pp. 2933–2942. [Online]. Available: https://proceedings.mlr.press/v48/bielik16.html
[106] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Mach. Learn.*, vol. 8, nos. 3–4, pp. 229–256, May 1992, doi: 10.1007/BF00992696.
[107] D. M. Ziegler, N. Stiennon, J. Wu, T. B. Brown, A. Radford, D. Amodei, P. Christiano, and G. Irving, "Fine-tuning language models from human preferences," 2019, *arXiv:1909.08593*.

[108] P. J. Blazek, K. Venkatesh, and M. M. Lin, "Deep distilling: Automated code generation using explainable deep learning," 2021, *arXiv:2111.08275*.
[109] P. J. Blazek and M. M. Lin, "Explainable neural networks that simulate reasoning," *Nature Comput. Sci.*, vol. 1, no. 9, pp. 607–618, Sep. 2021, doi: 10.1038/s43588-021-00132-w.
[110] W. Stephen, "Statistical mechanics of cellular automata," *Rev. Mod. Phys.*, vol. 55, pp. 601–644, Jul. 1983.
[111] R. Mukherjee, Y. Wen, D. Chaudhari, T. W. Reps, S. Chaudhuri, and C. Jermaine, "Neural program generation modulo static analysis," 2021, *arXiv:2111.01633*.
[112] S. Lu *et al.*, "CodeXGLUE: A machine learning benchmark dataset for code understanding and generation," 2021, *arXiv:2102.04664*.
[113] M. Brockschmidt, M. Allamanis, A. L. Gaunt, and O. Polozov, "Generative code modeling with graphs," 2018, *arXiv:1805.08490*.
[114] Z. Teng, Q. Fu, J. White, and D. C. Schmidt, "Sketch2Vis: Generating data visualizations from hand-drawn sketches with deep learning," in *Proc. 20th IEEE Int. Conf. Mach. Learn. Appl. (ICMLA)*, Dec. 2021, pp. 853–858.
[115] M. Li, Z. Lin, R. Mech, E. Yumer, and D. Ramanan, "Photo-sketching: Inferring contour drawings from images," 2019, *arXiv:1901.00542*.
[116] S. Ostrowski. (2020). *Announcing Pylance: Fast, Feature-Rich Language Support for Python in Visual studio Code*. Accessed: Jun. 7, 2022. [Online]. Available: https://devblogs.microsoft.com/python/announcing-pylance-fast-feature-rich-language-support-for-python-in-visual-studio-code/
[117] C. Colah. (Aug. 2015). *Understanding LSTM networks*. [Online]. Available: https://colah.github.io/posts/2015-08-Understanding-LSTMs/
[118] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," 2014, *arXiv:1409.0473*. [Online]. Available: https://arxiv.org/abs/1409.0473, doi: 10.48550/arxiv.1409.0473.
[119] A. Amidi and S. Amidi. *CS 230—Convolutional Neural Networks Cheatsheet*. Accessed: Mar. 6, 2022. [Online]. Available: https://stanford.edu/ shervine/teaching/cs-230/cheatsheet-convolutional-neural-networks
[120] P. Platen. (2020). *How to Generate Text: Using Different Decoding Methods for Language Generation With Transformers*. Accessed: Jun. 8, 2022. [Online]. Available: https://huggingface.co/blog/how-to-generate
[121] R. Gupta, S. Pal, A. Kanade, and S. Shevade. (2017). *Fixing Common C Language Errors by Deep Learning*. [Online]. Available: http://www.iisc-seal.net/deepfix
[122] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," in *Proc. ACL*, Aug. 2016, pp. 1715–1725. [Online]. Available: https://aclanthology.org/P16-1162
[123] StackExchance. (2022). *Stackoverflow | Where Developers, Learn, Share, & Build Carreers*. [Online]. Available: https://stackoverflow.com/
[124] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, "CodeBLEU: A method for automatic evaluation of code synthesis," 2020, *arXiv:2009.10297*. [Online]. Available: https://arxiv.org/abs/2009.10297, doi: 10.48550/arxiv.2009.10297.
[125] M. Post, "A call for clarity in reporting BLEU scores," in *Proc. 3rd Conf. Mach. Transl., Res. Papers*. Brussels, Belgium, Oct. 2018, pp. 186–191. [Online]. Available: https://aclanthology.org/W18-6319

ENRIQUE DEHAERNE (Graduate Student Member, IEEE) received the Bachelor of Engineering degree from KU Leuven, Belgium, in 2020, where he is currently pursuing the Master of Engineering degree in computer science.

He was a Research Engineering Intern at Nokia Bell Labs, Antwerp, in Summer 2021. He is currently collaborating with the Advanced Patterning Center, Interuniversity Microelectronics Centre (IMEC), to complete his master thesis. Among other projects, the IMEC Advanced Patterning Laboratory researches computer vision and data analysis applications for use in the semiconductor industry. His research interests include applications of machine learning, computer vision, natural language processing, and robotics.

BAPPADITYA DEY (Member, IEEE) received the B.Sc. and M.Sc. degrees (Hons.) in physics and electronic science from the University of Calcutta, Kolkata, India, in 2006 and 2008, respectively, the second M.Tech. degree in computer science and engineering from MAKAUT (formerly known as WBUT), Kolkata, in 2010, and the third M.Sc. and Ph.D. degrees in computer engineering from the Center for Advanced Computer Studies (CACS), University of Louisiana at Lafayette, USA, in 2017 and 2022, respectively. He joined IMEC, Belgium, in 2018, and worked in various roles since then. He is currently a Research and Development Engineer at the Advanced Patterning Center, IMEC. Till now, he has authored/coauthored 24 publications and has presented at several international conferences. His research interests include VLSI, microelectronics, reconfigurable hardware, machine learning, computer vision, artificial intelligence, and semiconductor process optimization. He is a member of SPIE.

SANDIP HALDER received the Ph.D. degree in metallurgy and materials science from RWTH Aachen, in 2006. He joined IMEC, in 2007, as a Research Scientist at the Advanced Materials and Process Department and was responsible for leading the metrology and inspection path-finding activities for the 3D SIC program. In 2013, he moved to the Advanced Patterning Center, IMEC, where he has worked as a Research Scientist and then as the Team Leader for metrology and inspection. Since 2020, he has been the Litho Group Lead within the same department. He has published more than 90 papers and has ten published patents.

STEFAN DE GENDT (Senior Member, IEEE) received the Doctor of Science degree from the University of Antwerp, in 1995. He subsequently was recruited by IMEC, Leuven, Belgium, the world's largest independent research institute in nanoelectronics and technology. He is currently a Full Professor (part-time) at KU Leuven and the Scientific Director of IMEC. Together with his respective teams, he has (co)authored more than 500 peer-reviewed journal publications. His research interests over his 25-year career at IMEC include metrology, semiconductor cleaning and passivation, high-k and metal gate unit process research, and post-CMOS nanotechnology (including nanowires, carbon nanotubes, graphene, and related 2D materials).

WANNES MEERT (Member, IEEE) received the Master of Electrotechnical Engineering degree in microelectronics, the Master of Artificial Intelligence degree, and the Ph.D. degree in computer science from KU Leuven, in 2005, 2006, and 2011, respectively. He is currently an IOF Fellow and a Research Manager at the DTAI Section, Department of CS, KU Leuven. His work is focused on applying machine learning, artificial intelligence, and anomaly detection technology to industrial application domains.