Automating Arduino Programming: From Hardware Setups to Sample Source Code Generation

Imam Nur Bani Yusuf, Diyanah Binte Abdul Jamal, Lingxiao Jiang
School of Computing and Information Systems
Singapore Management University, Singapore
{imamy.2020, diyanahj.2020, lxjiang}@smu.edu.sg

***Abstract*—An embedded system is a system consisting of software code, controller hardware, and I/O (Input/Output) hardware that performs a specific task. Developing an embedded system presents several challenges. First, the development often involves configuring hardware that requires domain-specific knowledge. Second, the library for the hardware may have API usage patterns that must be followed. To overcome such challenges, we propose a framework called ArduinoProg towards the automatic generation of Arduino applications. ArduinoProg takes a natural language query as input and outputs the configuration and API usage pattern for the hardware described in the query. Motivated by our findings on the characteristics of real-world queries posted in the official Arduino forum, we formulate ArduinoProg as three components, i.e., Library Retriever, Configuration Classifier, and Pattern Generator. First, Library Retriever preprocesses the input query and retrieves a set of relevant libraries using either lexical matching or vector-based similarity. Second, given Library Retriever's output, Configuration Classifier infers the hardware configuration by classifying the method definitions found in the library's implementation files into a hardware configuration class. Third, Pattern Generator also takes Library Retriever's output as input and leverages a sequence-to-sequence model to generate the API usage pattern. Having instantiated each component of ArduinoProg with various machine learning models, we have evaluated ArduinoProg on real-world queries. Library Retriever achieves a Precision@K range of 44.0%-97.1%; Configuration Classifier achieves an Area under the Receiver Operating Characteristics curve (AUC) of 0.79-0.95; Pattern Generator yields a Normalized Discounted Cumulative Gain (NDCG)@K of 0.45-0.73. Such results indicate that ArduinoProg can generate practical and useful hardware configurations and API usage patterns to guide developers in writing Arduino code.**

***Index Terms*—arduino, embedded system, deep learning, information retrieval, code generation, api recommendation**

I. Introduction

An embedded system is a system consisting of software code, controller hardware, and I/O (Input/Output) hardware that performs a specific task. The software contains the logic to control the behavior of the system, the controller hardware executes the code, and the I/O hardware (e.g., sensor, actuator, display) is used to interact with the environment. Embedded systems have been applied to various use cases, ranging from high-critical applications such as rocket controllers to low-critical applications such as home automation using IoTs.

In this work, we are interested in the activity of developing an embedded system using a specific controller family called Arduino. For simplicity, we refer to an embedded system that leverages Arduino as the controller hardware as an *Arduino application*. We target Arduino programming because Arduino is open source, the user base is large [1] and the documentation [2] is well-maintained.

Developing Arduino applications poses several challenges. Writing an Arduino code often involves configuring certain hardware, both in the code and physical space. To make the system work properly, the configuration in the physical space must match the code. This challenge is illustrated in Figure 1. Although the developer has the option to connect the servo to any interface between 0-13 in the physical space, the developer must connect the servo to interface number 10 because the servo is initialized using interface number 10 in the code when invoking myServo.attach(pin=10). Choosing the correct interface number often requires hardware-specific knowledge that the developer may not be familiar with [3]. Additionally, each hardware may have its own set of API usage patterns. It is unlikely for a developer to remember all such patterns, given a large number of hardware libraries available.

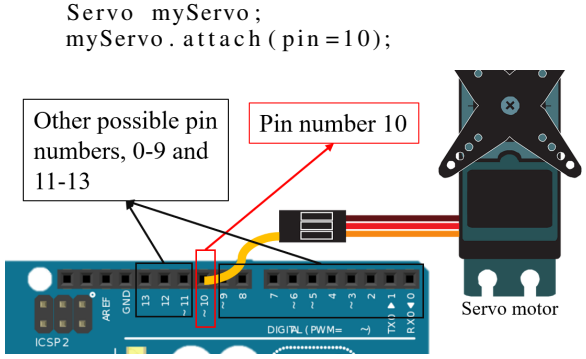

Fig. 1. An example of how the setup in the physical space and code should match. The servo should be connected to interface number 10 because the servo motor is initialized using interface number 10 in the code.

The goal of our work is to automatically infer hardware configurations based on a natural language description of the desired hardware, and to generate API usage patterns that advise developers on the correct usage of the hardware with the specified configuration. We leverage natural language as the input specification because it is easy for developers to provide. To design the framework, we curate and study real-world user questions posted on the official Arduino forum [4]. Our finding shows that it is possible to infer the hardware

configuration based on the method definitions found in the library's implementation files and there are common API usage patterns for guiding developers.

Based on real-world user questions and needs, we formulate our approach by proposing a framework called ArduinoProg. ArduinoProg is composed of three components, i.e., Library Retriever, Configuration Classifier, Pattern Generator. ArduinoProg solves the problem of inferring hardware configurations and generating API usage patterns from a query by dividing the problem into three main steps. First, Library Retriever takes a query as input, preprocesses the query, and retrieves a set of relevant libraries using either lexical matching or vector-based similarity. We improve the performance of Library Retriever by extracting important keywords using NLP techniques before feeding the query to the retrieval model. Given Library Retriever's output, Configuration Classifier infers the hardware configuration by classifying the method definitions found in the library's implementation files into a hardware configuration class. Furthermore, Pattern Generator takes Library Retriever's output as input and leverages a sequence-to-sequence model [5], [6] to generate the API usage pattern.

Leveraging various machine learning models to instantiate each component of ArduinoProg, we have evaluated ArduinoProg on real-world queries. The performance of Library Retriever ranges from 44.0% to 97.1% in terms of Precision@K. The Configuration Classifier can achieve an AUC of 0.79-0.95. Pattern Generator can yield NDCG@K scores of 0.45-0.73. Such results indicate that ArduinoProg can generate practical and useful hardware configurations and API usage patterns to guide developers in writing Arduino code.

The novelty of our work lies in the following aspects. First, our proposed framework is new because it can generate both hardware configurations and API usage patterns for hardware libraries. In the software engineering domain, existing works can only generate API usage patterns given natural language descriptions [7]–[10]. Generating only API usage patterns is not sufficient because Arduino applications require developers to configure the hardware in addition to writing the code, and API usage patterns alone may not convey how to configure hardware properly. Second, our framework can infer hardware configuration from *only* software artifacts without requiring additional third-party hardware, while prior works rely on such hardware [11]–[14]. Third, we evaluate our framework using various machine learning models and demonstrate promising results.

In summary, our main contributions are as follows:

- We propose ArduinoProg, a novel framework to infer hardware configurations and generate API usage patterns for the hardware described in a natural language query. The implementation of ArduinoProg is available at https://github.com/imamnurby/ArduProg.
- We evaluate ArduinoProg on real-world user queries by instantiating each component of ArduinoProg using various machine learning models.

The rest of the paper is structured as follows. Section II covers our study on real-world Arduino codes and users' programming queries. Section III explains the design and internal details of ArduinoProg. Section IV describes our evaluation settings. Section V presents the empirical results. Section VI discusses the qualitative analysis of some results. Section VIII concludes our paper.

II. Study on Real-World Questions and Codes

We formulate our approach by conducting a preliminary empirical study on real-world user questions posted on the official Arduino forum [4]. In this forum, users can ask questions on hardware or software topics (e.g., motors and actuators, displays, sensors). We explain the data collection process and the results of our study as follows.

A. Collecting Real-World Questions and Codes

We use "read sensor", "interface sensor", and "use sensor" as the queries to search the relevant forum posts. We define the query using our domain-specific knowledge. We search the questions using the forum search engine. The forum search engine relies on lexical matching between a query and a forum post and employs some preprocessing steps, such as stopwords removal, frequent-words removal, and word stemming. We collect forum posts whose titles reflect "how-to" questions, mention a specific hardware name, or mention a hardware functionality. Some examples of the relevant post titles are "How can I use a water level sensor to turn on and off a DC motor?", "Help with LM34 Temperature Sensor", "Calculating altitude above ground level with BMP280". We leverage the question title as the first proxy to collect relevant questions because a question title often summarizes and identifies the key topic of the questions [15]–[17]. We get 67 forum posts from this step. We further narrow down the forum posts by only considering the posts that: 1) have been marked as resolved, and 2) contain code snippets or mention specific hardware library names either in the question body or answer. We get 35 forum posts in the end.

B. Processing Arduino Codes

We extract a code snippet from each post. For forum posts that direct users to third-party websites, such as GitHub or tutorial websites, we directly extract the code snippets from the source websites. Next, we extract the relevant lines that correspond to the hardware from each code snippet and analyze these lines to gain insight into how Arduino developers write codes for hardware. We manually identify a set of hardware $H = (hw_1, hw_2, \ldots, hw_n)$ involved in the code. For each $hw \in H$, we extract the object declaration obj followed by the statements that involve method calls of each obj. If a method call returns a value that is assigned to another variable, we also include the statements that involve the assigned variable. Including such statements allows us to capture the possible data dependencies and interactions between hardware objects in the code. Additionally, we include statements that involve analogRead, digitalRead, and pinMode method invocations or invocations from a Wire object. Such invocations are used to

interact with hardware. To enhance our understanding of the code, we extract the readme, header (.h), and implementation (.cpp) files of the libraries that correspond to the hardware involved in the code. Such files can enhance our understanding because they may contain hardware and methods documentation.

C. Analysis Results

Insight#1: A piece of codes may contain multiple hardware with some interactions, but the interactions are often simplistic. Our analysis reveals that 17 out of 35 studied codes involve multiple hardware. We scrutinize these 17 codes and find that such interactions do not occur when configuring hardware interfaces. The interactions only involve simple if-else logic between the value obtained from input devices (e.g., sensor, keypad) to control actuators, memory devices, or display devices, such as moving an actuator if the sensor reading is more than a certain threshold. The first insight implies that we can independently infer hardware configurations and generate API usage patterns although the query contains multiple hardware.

Insight#2: Hardware configurations can be classified into explicit and implicit configurations. In the explicit configuration, the user manually specifies the physical connection between the hardware and controller in the code. This configuration is set by either: 1) instantiating the hardware object with an interface number, or 2) invoking the initialization method (a method that is always called at the beginning after the hardware object is instantiated). For example, "dht11 sensor(10, DHT11)" specifies that a DHT11 sensor is interfaced through pin number 10. In contrast, the implicit configuration does not require the user to specify the interface number because the hardware configuration is already embedded in the internal implementation of the library. For instance, "Adafruit_SHT31 sht31 = Adafruit_SHT31()" is an example of the implicit configuration. Our analysis shows 63 hardware components are found in the 35 codes that we analyzed, with 26 of them falling under the implicit configuration category and 37 under the explicit configuration category.

Insight#3: Implicit configurations can be classified further into several configuration classes. We scrutinized the readme, header, and implementation files of the hardware libraries that fall under the implicit configuration category. Our study reveals that we can categorize the implicit configuration based on the communication protocol between the hardware and controller. Specifically, we identified three configuration classes: address-based (I2C), serial (SPI), and asynchronous (UART) communications. Address-based communication uses SDA/SCL interfaces, serial communication uses MISO/MOSI/CLK/CS interfaces, and asynchronous communication uses RX/TX interfaces. Out of the 26 implicit configurations we analyzed, 21 are address-based, 4 are serial, and 1 is asynchronous.

Insight#4: Method definitions found in the library's implementation files can be used to infer the hardware's configuration class. We contrast the readme, header, and implementation files of the hardware libraries that belong to different configuration categories. Our comparison shows that hardware libraries from the same configuration category often have similar tokens in the constructors and method definitions. For example, in Figure 2, Both DHT12 and ADS1X15 belong to the address-based category and utilize the wire objects.

DHT12 (Temperature Sensor)

```
int DHT12::_readSensor(){
  …
  _wire->beginTransmission(DHT12_ADDRESS);
  _wire->write(0);
  int rv = _wire->endTransmission();
  if (rv < 0) return rv;
  …
  int bytes = _wire->requestFrom(DHT12_ADDRESS, length);
  …}
```

ADS1X15 (Analog to Digital Comparator)

```
uint16_t ADS1X15::_readRegister(uint8_t address, uint8_t reg){
  …
  _wire->beginTransmission(address);
  _wire->write(reg);
  _wire->endTransmission();
  int rv = _wire->requestFrom((int) address, (int) 2);
  if (rv == 2)
  …}
```

Fig. 2. Fragments of two method definitions from two different hardware that belong to the same configuration class. Both method definitions involve similar tokens such as "wire", "beginTransmission", and "rv".

As we mention earlier, ArduinoProg is composed of three components, i.e., Library Retriever, Configuration Classifier, Pattern Generator. Insight#1 implies that we can treat multiple hardware in the query individually and also generate its API usage patterns independently. Insight#1 motivates the design of Library Retriever and Pattern Generator. Insight#2, Insight#3, and Insight#4 allow us to frame the problem of inferring hardware configurations as a classification problem. The goal is to classify whether a library belongs to either explicit, address-based, serial, or asynchronous categories based on the library's implementation files. Insight#2, Insight#3, and Insight#4 motivate the design of Configuration Classifier.

III. ARDUINOPROG

This section explains the architecture of ArduinoProg and how each component in the architecture works. Figure 3 shows the high-level overview of ArduinoProg.

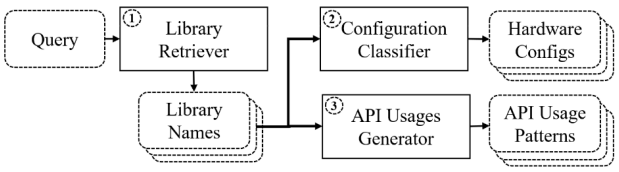

Fig. 3. The general overview of ArduinoProg. ArduinoProg consists of three modules: Library Retriever, Configuration Classifier, and Pattern Generator.

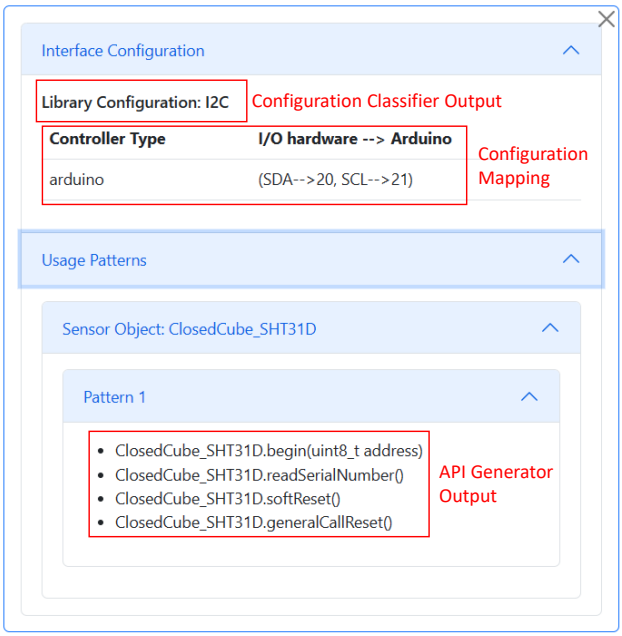

Fig. 4. Input and output examples of ArduinoProg.

The input and output of ArduinoProg is shown in Figure 4. Library Retriever takes the natural language query as input and recommends a set of library names. Then, Configuration Classifier receives the library names as input and predicts the configuration class of each recommended library. The configuration class is then mapped to the interface configuration using a predefined dictionary, based on Insight#3 in the preliminary study described in Section II. Similarly, given the library names as input, Pattern Generator generates the API usage pattern for each recommended library.

The working mode of ArduinoProg consists of training and inference phases. In the training phase, ArduinoProg trains a set of models to perform the required tasks, i.e., retrieving relevant libraries, predicting hardware configurations, and generating API usage patterns. In the inference phase, ArduinoProg is ready to interact with a developer. ArduinoProg receives a query that describes the intended library and recommends a set of libraries with the hardware configurations and API usage patterns. We explain each module and the workflow in each working mode as follows.

A. Library Retriever

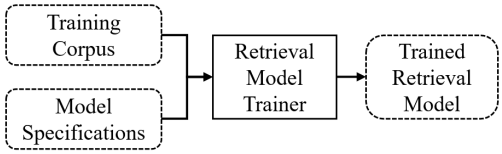

Fig. 5. Library Retriever leverages Retrieval Model Trainer module to train a retrieval model. The dashed boxes indicate input and output.

1) Training: Figure 5 demonstrates the workflow of Library Retriever in the training phase. The inputs of this phase are model specifications and a training corpus, while the output is a trained retrieval model.

Retrieval Model Trainer receives model specifications as input to initialize the retrieval model. The model specification contains information about the model type and the corresponding hyperparameters. There are two supported model types, i.e., BM25 and deep learning-based models.

For the deep learning-based models, each training instance is represented as a triplet of *(anchor, positive sample, negative sample)*. Anchor refers to a library name, the positive sample is a library relevant to the anchor, and the negative sample is a library irrelevant to the anchor. To train the model, a tokenizer tokenizes each element in the triplet and the model converts each tokenized element to a vector representation. We leverage Triplet Loss [18] as the objective function, aiming to minimize the distance between the anchor and positive sample while maximizing the distance between the anchor and negative sample in the vector space. The Triplet Loss equation is shown in equation 1, where f is a function that converts its input into a vector and α is a hyperparameter.

$$\mathcal{L}_{triplet} = \max(||f(anchor) - f(sample^{+})|| - ||f(anchor) - f(sample^{-})|| + \alpha, 0) \quad (1)$$

We perform the tokenization at the subword where the smallest unit can be a character. Subword-level tokenization allows the model to leverage the similarity between similar library names. For example, in Table I, three different libraries (i.e., "BMP280_DEV", "Adafruit_BMP280_Library", and "DFRobot_BMP3XX") contain the same subword token, i.e., BMP. Intuitively, the same subword token can help the model to learn that those libraries are similar.

TABLE I
AN EXAMPLE OF TOKENIZATION IN THE SUBWORD LEVEL.

Library Name	Tokenization Results
BMP280_DEV	BMP, 280, DEV
Adafruit_BMP280_Library	Adafruit, BMP, 280, Library
DFRobot_BMP3XX	DF, Robot, BMP, 3XX

For BM25, the training corpus consists of a list of library names. A tokenizer tokenizes each library name at the subword level, resulting in a list of subword tokens. Retrieval Model Trainer then counts the term frequency, inverse document frequency, and length of the resulting subword tokens for each library name.

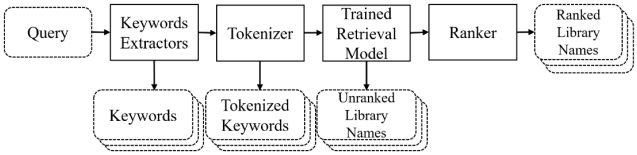

Fig. 6. Library Retriever involves four modules in the offline phase: Keyword Extractors, Tokenizer, Retrieval Model, and Ranker. The dashed boxes indicate input and output.

2) Inference: Figure 6 illustrates the workflow of Library Retriever in the inference phase. Retrieval Model receives a

query as input and converts the query to its vector representation. Then, Retrieval Model computes the similarity of the query vector with the library name vectors. The output of the inference phase is the top-K library names with the highest similarity scores. The detailed steps are as follows.

First, Keyword Extractors takes a query Q as input and outputs a set of keywords $K = (k_1, k_2, \ldots, k_m)$, where $k_i \subseteq Q$ and $m \leq n$. A keyword refers to the tokens in the query that are most likely to appear in a library name. The reason to extract the keywords instead of using the query directly is that unimportant tokens (tokens other than the keywords) can lower the similarity score between the query and the target library name. To exemplify, consider an ideal retrieval condition where a query exactly matches a library name. Hence, the similarity score is maximum in such a condition. Conversely, the presence of unimportant tokens lowers the similarity score because the query does not match the library name.

The steps for Keyword Extractors to identify keywords in a query are as follows. Keyword Extractors tokenizes the query Q into n tokens in the word level. Then, each token is labeled with the corresponding part-of-speech tag, and Keyword Extractors parses the tagged query to generate the dependency tree. Subsequently, Keyword Extractors extracts noun phrases from the parsed query. A noun phrase consists of a root noun and its modifiers. In each noun phrase, frequent words and PoS tags are removed. Next, Keyword Extractors identifies the token corresponding to a sensor name in each noun phrase. Keyword Extractors determines a sensor name by checking whether the token contains both alphabetical and numeric characters. We use this heuristic since most library names are alphanumeric, e.g., BMP280, DHT11, SHT31. If the noun phrase does not contain a sensor name, the root noun of the phrase is used as the keyword.

Tokenizer tokenizes each keyword $k_i \in K$ and yields a set of tokenized keywords $W = (w_1, w_2, \ldots, w_m)$ where $w_i \in W$ corresponds to the subword tokenization result of each $k_i \in K$. Next, Retrieval Model converts each keyword $w_i \in W$ to a vector and computes the similarity of this vector with all the library name vectors, resulting in a list of K library names for each keyword. Then, the top-K library names from each keyword are gathered, resulting in an unordered list of K x m library names. Here, K is the number of considered results (i.e., top-K results), while m is the number of keywords. Then, the results of K library names from each keyword are gathered, resulting in an unordered list of K x m library names. K is the number of the considered results (i.e., top-K results), while m is the number of keywords. Finally, Ranker outputs a ranked list of K library names.

B. Configuration Classifier

1) Training: Figure 7 (top side) shows the workflow of Configuration Classifier in the training phase. The input of Configuration Classifier is a library name and the output is a trained encoder and a trained classifier that will be used in the inference phase.

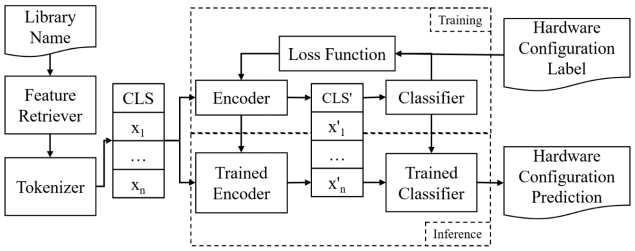

Fig. 7. The workflow of Configuration Classifier in the training and inference phases.

Feature Retriever takes a library name as input and then converts it to a library feature. To get the library feature, Feature Retriever extracts the method definitions found in the library's implementation files. Next, Tokenizer receives the library feature as input, then tokenizes it at the subword level and appends a CLS token prefix. The result is a tokenized feature $F = (CLS, x_1, \ldots, x_n)$, where $x_i \in F$ indicates a subword token and n indicates the length of the tokenized feature. Encoder receives the tokenized feature F and converts each $x \in F$ to a vector, resulting in a list vectorized feature $F' = (CLS', x'_1, \ldots, x'_n)$. Next, the CLS' token is fed to Classifier to predict the configuration class. The predicted configuration and the corresponding label are passed to the loss function to compute the loss. We compute the loss using Binary Cross Entropy (BCE) to update the Encoder and Classifier weights. The BCE equation is shown in Equation 2, where y_n indicates the label, x_n indicates the predicted hardware configuration class, and σ indicates the sigmoid function.

$$\mathcal{L}_{Binary-CE} = -y_n \log(\sigma(x_n)) - (1 - y_n) \log(1 - \sigma(x_n)) \quad (2)$$

2) Inference: Configuration Classifier is ready to predict the hardware configuration of the given input library name in the inference phase. Figure 7 (bottom side) illustrates the workflow of Configuration Classifier for the inference. The workflow in the inference phase is similar to the workflow in training phase. The difference is that there is no optimization using the loss function.

C. Pattern Generator

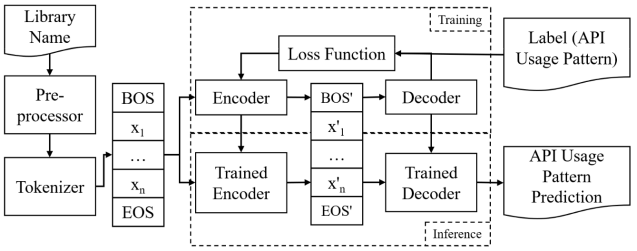

Fig. 8. The workflow of Pattern Generator in the training and inference phases.

1) Training: Figure 8 (top side) shows the workflow of Pattern Generator in the training phase. Pattern Generator receives a library name as input and outputs Trained Encoder and Trained Decoder.

Given a library name, Preprocessor appends the library name with the class names, resulting in a set of processed inputs $P = (p_1, p_2, \ldots, p_n)$ where each $p_i \in P$ is $libname\ SEP\ classname_i$ and n is the number of class names in a library that represent a hardware object. The intuition behind appending the library name with a class name is to give the encoder more information about which API usage patterns to generate because a library may support multiple hardware with different functionalities. For instance, the library Adafruit_ICM20X supports two different hardware, i.e., ICM20649 and ICM20948. ICM20649 can read linear acceleration and orientation values, while ICM20948 can also read the magnetic field value besides acceleration and orientation values. These two hardware have different API usage patterns.

Tokenizer takes each $p_i \in P$ as input, then appends two special tokens, i.e., BOS to indicate the beginning of a sequence and EOS to indicate the end sequence. Then Tokenizer tokenizes the processed input at the subword level. The tokenization result is $BOS, l_1, l_2, \ldots, l_a, SEP, c_1, c_2, \ldots, c_b, EOS$ where a indicates the length of the tokenized library name and b indicates the length of the tokenized class name. A special token SEP is used to differentiate whether a token belongs to the library or class names. Afterward, Encoder converts each subword token in the tokenized result to a vector with a dimension d, resulting in a context vector $BOS', l'_1, l'_2, \ldots, l'_a, SEP', c'_1, c'_2, \ldots, c'_b, EOS'$. Decoder takes the context vector as input to generate the API usage pattern. The generated API usage pattern and the ground truth are passed to the loss function to compute the loss and subsequently update the weights of Encoder and Decoder. We leverage Cross-Entropy loss with softmax activation function to train Encoder and Decoder.

2) Inference: Pattern Generator is ready to generate API usage patterns given a library name as input. Figure 8 (bottom side) shows the workflow of Pattern Generator for the inference. The workflow in the inference phase is similar to the workflow in the training phase. The difference is that there is no optimization using the loss function.

IV. Experimental Settings

This section explains the experimental setting of ArduinoProg. The experimental setting includes the training and benchmark data collection, evaluated models, and evaluation metrics. We explain the experimental setting of each component in ArduinoProg as follows.

A. Library Retriever

1) Training: The training data consists of triplets, each consisting of an *anchor*, a *positive sample*, and a *negative sample*. The anchor is a library name. The positive sample is a library that is relevant to the anchor, i.e., it interfaces with the same type of hardware. Conversely, the negative sample is a library that is irrelevant to the anchor. To construct training triplets, we crawl 4,309 library names from the Arduino library reference [2], belonging to 7 different categories. The categories are communication, data storage, device control, display, sensors, signal input/output, and timing. For each library name, we extract the hardware name and remove the frequent tokens. Then, we sample 50 positive and negative samples. In the end, the total number of training triplets is 234,577.

2) Evaluation: The evaluation data consists of pairs, each consisting of a *query* and a set of *ground truth libraries*. We used 35 real-world question titles from our preliminary study in Section II as the queries. We only use the question title as the query because a question title summarizes and identifies the key idea of the question, while the question body often contains irrelevant information [15]–[17]. We observe that posts in the forum often contain incomplete ground truths, where the solution of a post only mentions the library from a specific hardware manufacturer although the question title does not specify the manufacturer's name. To ensure the completeness of the ground truth libraries, we collect them as follows. First, we manually extract the hardware mentioned in the query. Next, we use string matching to identify the libraries from the Arduino library reference [2] that support the same hardware. Specifically, we match the hardware name in the query against the library name, description, and readme file of each library in [2]. Then, we manually inspect the results to remove any noises. In total, we have 35 pairs of a query with the ground truth libraries.

3) Models: We instantiate Library Retriever with the following models, BM25 [19], BERT [20], DistillBERT [21], RoBERTa [22], and CodeBERT [7]. We chose BM25 because it is widely used for tackling lexical-based search problems. We select BERT, RoBERTa, and CodeBERT to compare the effects of different pretraining corpora. BERT and RoBERTa are pretrained using human language corpora, while CodeBERT is pretrained using code corpora. We chose DistillBERT to evaluate the performance on a smaller deep learning model.

4) Metrics: We leverage Precision@K [23] to measure each model's performance. Precision@K measures the proportion of relevant libraries in the result list. Precision@K ranges from 0 to 1. Higher Precision@K is better; it means that the result list contains more relevant results. Equation 3 shows the formula to compute Precision@K.

$$Prec@K = \frac{|\{ground\ truths\} \cap \{retrieved\ libraries@K\}|}{|\{retrieved\ libraries@K|\}} \tag{3}$$

B. Configuration Classifier

1) Training: The training data for the classifier is a pair of a *library feature* and a *configuration class*. Here, the configuration class corresponds to the ground truth. To obtain the library feature, we crawl 475 libraries from the Arduino library reference [2] that belong to the sensors and signal input/output categories. Then, we extract the cpp files from these libraries and parse each file using tree-sitter [24] to obtain the abstract syntax tree. Subsequently, we traverse the abstract syntax tree and extract the method definitions. For the configuration class, we manually label each library

TABLE II
THE DISTRIBUTION OF THE CONFIGURATION CLASSES IN OUR DATASETS.

Dataset	Split	#UART	#SPI	#I2C	#Explicit
475-libraries	Train	8	20	75	49
475-libraries	Validation	13	31	89	49
475-libraries	Testing	11	40	90	51
real-queries	Testing	5	23	156	118

with address-based (I2C), serial (SPI), asynchronous (UART), or Explicit. We infer the label of each library by manually inspecting the readme file and hardware datasheet, i.e., the official specification document provided by the manufacturer. Next, we create training, validation, and test sets from the 475 crawled libraries. We use stratified sampling to keep the class distribution across different sets equal. The number of pairs is 150 for the training set, 155 for the validation set, and 170 for the testing set.

2) Evaluation: The evaluation data is a pair of a *library feature* and a *configuration class*. We evaluate the classifier using the testing set obtained from splitting the initial 475 libraries. Additionally, we also evaluate the classifier using 35 real-world queries from our preliminary study in Section II. We refer to this setting as "real-queries". In the second scenario, we feed the queries into the retrieval model, which then generates the top-10 library recommendations. We perform the same steps for each recommended library to obtain the library feature and configuration class. This evaluation scenario is more challenging because the recommendations may belong to categories that differ from the training data, such as display and device control. The total number of testing data in the second scenario is 290. The number is smaller than 350 (35 queries x 10 recommendations) because we exclude cases where our heuristics fail to extract method definitions when traversing an abstract syntax tree. The distribution of the classes in the training and testing sets is shown in Table II.

3) Models: We leverage CodeBERT [7] as the encoder to produce the vector representation of the library feature because CodeBERT has been pretrained using corpora containing method definitions. For the classifier, we use the Dense layer, Random Forest [25] (RF), and Support Vector Machine [26] (SVM).

4) Metrics: We use AUC (Area Under the Curve) to measure how well the model can correctly classify a library into the correct communication protocol classes. We choose AUC because it does not depend on the classifier threshold. The AUC value ranges between 0 and 1, and a higher AUC value indicates better class differentiation by the model than a lower AUC value.

C. Pattern Generator

1) Training: The training data for the API usage pattern generation is a pair of an *input feature* and an *API usage patterns*. The input feature refers to the library name followed by the filename used in the #include directive. We create the training data by collecting 106,777 client codes from GitHub. First, we check the #include directives to identify the libraries used in the code, then extract the object declarations $O = (o_1, o_2, \ldots, o_n)$. We only extract the object that corresponds to the libraries listed in the official Arduino library reference. Second, we extract all API usage patterns that belong to each $o \in O$ inside the setup and loop functions. We focus on the setup and loop functions because the execution of Arduino code always begins with the setup function followed by the loop function. If the setup or loop functions invoke another function, we also include the API usage patterns inside the invoked function. We extract API usage patterns by traversing the abstract syntax tree using tree-sitter. In the end, the total number of training data is 84,222 pairs.

2) Evaluation: We leverage the code attached to 35 forum posts that we have collected (see Section II) to create (*input feature*, *API usage patterns*) pairs for our benchmark. First, we manually identify the valid libraries used in each code from the #include directives and object declarations. A valid library refers to a library listed in the official Arduino library reference [2]. Second, we manually extract the API usage patterns of each identified library from the code. We manually performed the data collection to ensure the reliability of the benchmark. In the end, the total number of testing data is 31 pairs.

3) Models: We evaluate the following models to generate API usage patterns, Co2BERT (CodeBERT2CodeBERT, a Transformer sequence-to-sequence model [5] based on CodeBERT [7]), PLBART [8], CodeT5 [9], and DeepAPI [10]. We choose Co2BERT, PLBART, and CodeT5 because they have been proven to achieve competitive performance on the code generation task. We choose DeepAPI, which leverages an RNN architecture, to demonstrate that ArduinoProg is flexible and can be initialized with any model, not just Transformers.

4) Metrics: We leverage NDCG [27] (Normalized Discounted Cumulative Gain) to evaluate the generation performance. We use NDCG because it considers two factors. First, NDCG considers how close a result is to the ground truth using a relevancy function. Second, NDCG weights the relevancy score of a result by its ranking in the result list. The intuition is because results at the lower ranking are less valuable as such results are less likely to be examined by a user. Equation 5 shows the formula to calculate NDCG up to position K in the result list.

$$DCG@K = \sum_{i=1}^{K} \frac{2^{rel_i} - 1}{\log_2(i+1)} \tag{4}$$

$$NDCG@K = \frac{DCG@K}{ideal\ DCG@K} \tag{5}$$

NDCG is computed as DCG divided by the ideal DCG. Equation 4 computes DCG, where rel_i refers to the relevancy measurement. We use the following relevancy metrics to measure how close the generated sequence is to the ground truth.

- ROUGE-n [28] measures the relevancy based on the number of n-grams in the ground truth that appear in the result.
- METEOR [29] measures the relevancy based on the harmonic mean of unigram precision and recall. METEOR also considers the order of the generated result.

- LCS (Longest Common Subsequences) measures the relevancy by the longest common subsequences between the ground truth and result divided by the ground truth length.

Moreover, ideal DCG@K is computed by setting the value of rel_i with 1 (ideal value) for $i \in [1, K]$.

V. Empirical Results

A. Library Retriever

Table III shows the results of retrieval experiment using Library Retriever.

TABLE III
THE RESULTS OF PRECISION@K ON THE LIBRARY RETRIEVAL EXPERIMENT USING 5 DIFFERENT MODELS. THE PRECISION@K VALUES ARE REPORTED IN PERCENTAGE.

Setting	Query Preprocessing	Model	Precision@K (%)					
			With Fine-tuning			No Fine-tuning		
			K=1	K=5	K=10	K=1	K=5	K=10
1	Subword tokenizer	BM25				37.1	29.1	25.4
		CodeBERT	40.0	35.4	27.4	11.4	8.0	6.3
		RoBERTa	34.3	23.4	16.9	17.1	12.6	12.0
		BERT	48.6	28.0	21.4	22.9	16.0	14.0
		DistillBERT	37.1	36.0	29.4	25.7	19.4	18.0
2	Subword tokenizer + keyword extraction	BM25				**77.1**	**69.1**	**60.0**
		CodeBERT	**97.1**	67.4	**54.0**	68.6	33.1	22.0
		RoBERTa	88.6	62.9	44.0	62.9	34.9	25.1
		BERT	91.4	**68.6**	49.4	57.1	32.0	23.7
		DistillBERT	91.4	65.7	52.0	71.4	42.3	30.0
Average Improvement of keyword extraction			**52.1**	**35.4**	**26.1**	**44.6**	**25.3**	**17.0**
Average Improvement of fine-tuning			**23.9**	**23.6**	**17.9**			

First, the overall results show that the models that leverage the keyword extraction in the query preprocessing steps perform better than those without the keyword extraction. The keyword extraction can increase the performance of models fine-tuned with Triplet loss by 26.1% to 52.1% on average. For the models without fine-tuning, the improvements range from 17.0% to 44.6%. Such results demonstrate that irrelevant words in the query can lower the performance of the retrieval models. Further discussion is in Section VI.

Takeaway 1: Our keyword extraction in the preprocessing step is useful. It can improve the precision by 17.0% to 52.1% at various values of K.

Second, the results indicate that fine-tuning using Triplet loss on Transformer-based models (CodeBERT, RoBERTa, BERT, DistillBERT) is beneficial. All fine-tuned models perform better than those without. The average improvements range from 17.9% to 23.9%.

Takeaway 2: Fine-tuning using Triplet loss is beneficial because it can improve the precision of the Transformer-based models by 17.9% to 23.9% at various values of K.

Third, the overall results show that Transformer-based models fine-tuned using Triplet loss outperform BM25 in setting (1). However, BM25 can yield competitive results in some cases, particularly in setting (2). The Transformer-based models consistently outperform BM25 at K=1, but BM25's performance becomes competitive at K=5. Moreover, at K=10, BM25 yields better results than the Transformer-based models. These findings suggest that BM25 and the Transformer-based models may complement each other when deployed in real-world scenarios.

Takeaway 3: BM25 and the Transformer-based models can yield competitive results in the setting (2). BM25 and the Transformer-based models may complement each other.

B. Configuration Classifier

Table IV shows the results of the hardware classification experiments on Configuration Classifier.

TABLE IV
THE AUC RESULTS ON THE HARDWARE CLASSIFICATION EXPERIMENT USING 3 DIFFERENT CLASSIFIERS ON 2 TESTING DATA.

Testing Set	Classifier	Average AUC per Class				Average AUC
		UART	SPI	I2C	EXPLICIT	
475-libraries	Dense	0.89	0.83	0.96	0.99	0.92
	SVM	0.92	0.77	0.96	0.96	0.91
	RF	**0.96**	**0.88**	**0.98**	**0.99**	**0.95**
real-queries	Dense	0.66	0.68	0.91	0.90	0.79
	SVM	**0.82**	**0.81**	**0.87**	**0.89**	**0.85**
	RF	0.69	0.69	0.90	0.91	0.80

As seen in Table IV, RF (RandomForest) yields the best performance among the other classifiers in the 475-libraries split with average AUC value of 0.95. On the other hand, SVM performs the best on the real-queries split with average AUC value of 0.85. Furthermore, all classifiers have lower AUC values in the real-queries split compared to the 475-libraries split. The possible reason for this discrepancy is that the real-queries split contains libraries from categories other than those within the training data. We confirm that the real-queries split includes 96 libraries from the sensor, 63 libraries from the device control, and 26 from the display, while the training data only consists of the libraries that belong to the sensor and signal input-output categories.

Takeaway 4: RF yields the best performance among the other classifiers on the 475-libraries split with an AUC value of 0.95. On the other hand, SVM achieves the best result on the real-queries split with 0.85 AUC value.

C. Pattern Generator

Table III shows the results of API usage pattern generation using Pattern Generator.

In terms of NDCG-ROUGE@K (NDCG@K measured with ROUGE score), the best performing model is Co2BERT with 0.73, 0.69, and 0.69 at K=1, K=5, and K=10, respectively. A high NDCG-ROUGE@K means that the individual APIs in the ground truth sequences are most likely to appear in the generated API sequences (unigram recall of the APIs in the ground truth). Deep API and Co2BERT yield competitive results at various K values in terms of NDCG-LCS@K. The interpretation of NDCG-LCS@K is similar to NDCG-ROUGE@K. However, NDCG-LCS@K considers the longest common subsequences between the ground truth sequences

TABLE V
THE NDCG@K RESULTS FOR THE API USAGE PATTERN GENERATION ON 4 DIFFERENT MODELS USING 3 RELEVANCY METRICS, I.E., ROUGE, LCS (LONGEST COMMON SUBSEQUENCES), AND METEOR.

Model	NDCG@K								
	ROUGE			LCS			METEOR		
	K=1	K=5	K=10	K=1	K=5	K=10	K=1	K=5	K=10
Co2BERT	**0.73**	**0.69**	**0.69**	**0.65**	0.60	0.59	**0.53**	0.45	0.42
PLBART	0.52	0.47	0.46	0.43	0.35	0.34	0.33	0.23	0.21
CodeT5	0.52	0.46	0.45	0.43	0.34	0.33	0.32	0.22	0.22
Deep API	0.71	**0.69**	**0.69**	0.64	**0.62**	**0.62**	**0.53**	**0.46**	**0.45**

and the generated API sequences instead of individual APIs recall. The value of NDCG-LCS@K is lower than NDCG-ROUGE@K, meaning that the models also tend to produce APIs that do not appear in the ground truth sequence. We confirm such a fact by measuring NDCG-METEOR@K. NDCG-METEOR@K is more strict than NDCG-ROUGE@K; it gives a penalty if the order in the generated API sequences is incorrect. The results in terms of NDCG-METEOR@K are similar to NDCG-LCS@K; Deep API and Co2BERT perform competitively. On the other hand, both PLBART and CodeT5 do not perform well in all the NDCG metrics.

Takeaway 5: Co2BERT and Deep API perform competitively in terms of NDCG@K across different relevancy measurements. On the other hand, PLBART and CodeT5 do not perform well.

VI. DISCUSSION

In this section, we discuss how irrelevant words in the query can affect the performance of Transformer-based models in Library Retriever. Next, we scrutinize the strengths and pitfalls of BM25 in recommending relevant hardware. Then, we discuss the performance of Configuration Classifier using various classifiers that have been evaluated. After that, we analyze why the generated API usage patterns from Pattern Generator differ from ground truths.

A. Library Retriever

TABLE VI
THE EFFECT OF IRRELEVANT WORDS ON THE RANKING AND SIMILARITY SCORES OF THE SAMPLED LIBRARIES.

Query	(Ranking)-Prediction	Cosine Similarity
(1) bmp280	(1)-Adafruit_BMP280_Library	0.97
	(2)-DFRobot_BMP280	0.97
	(3)-Arduino-BMP388	0.74
(2) with bmp280	(1)-Adafruit_BMP280_Library	0.83
	(2)-DFRobot_BMP280	0.82
	(3)-Arduino-BMP388	0.68
(3) above ground level with bmp280	(2)-Adafruit_BMP280_Library	0.70
	(1)-DFRobot_BMP280	0.73
	(3)-Arduino-BMP388	0.63
(4) calculating altitude above ground level with bmp280	(23)-Adafruit_BMP280_Library	0.48
	(12)-DFRobot_BMP280	0.51
	(14)-Arduino-BMP388	0.49

Table VI demonstrates that irrelevant words can lower the similarity scores between a query and a set of libraries. As the number of irrelevant words increases, the scores become lower. Moreover, adding irrelevant words may change the prediction ranking at some points, such as in query (3) and query (4). Such results indicate that some words may have a greater impact on the retrieval results than others.

Next, we discuss why BM25 yields good results. The majority of the queries contain a hardware name, and it is worth noting that library names also often contain a hardware name. For example, the library name "Adafruit_BMP280_Library" contains "BMP280". As BM25 relies on lexical matching, it can perform well on such queries. However, for the queries that do not specifically mention a hardware name, BM25 fails to return relevant libraries. An example of such a query is "Need help with coding IR," where IR refers to the type of the sensor, i.e., infrared, rather than a specific sensor name.

B. Configuration Classifier

Table IV indicates that the dense classifier works well on all classes in the 475-libraries split, with a 0.92 AUC score. However, the results are lower on the real-queries split, i.e., 0.79. In the previous section, we mentioned that the real-queries split contains libraries from more diverse categories than the 475-libraries split. Because the performance of the dense layer in the 475-libraries split is high (i.e., 0.89 on UART and 0.83 on SPI), the bad results on the real-queries split may indicate that the dense classifier overfits certain categories. The same trend is also observed when using the RF classifier.

C. Pattern Generator

We select some representative examples of the API usage patterns generated by the best-performing model to understand the results better. Figure 9 shows several examples of the generated API usage patterns.

Case 1
Hardware: Servo (actuator)
Ground Truth
attach -write
Predictions
(1) attach - write
(2) attach - writeMicroSeconds
(3) attach - write - detach

Case 2
Hardware: NovaSDS011 (dust sensor)
Ground Truth
begin - setWorkingMode - setDutyCycle - queryData
Prediction
(1) begin - setWorkingMode - getVersionDate - setDutyCycle - queryData
(2) begin - setWorkingMode - setDutyCycle - queryData
(3) begin - setWorkingMode - getVersionDate - queryData

Fig. 9. The examples of the generated API usage patterns. The API calls are separated by "-".

We find that some of the generated API usage patterns are valid, even though they are different from the ground truth. The reason is that a library may have multiple valid patterns,

as shown by the example in Case 1 of Figure 9. The API usage pattern in Case 1 is used to control a servo. The attach method initializes the connection to the controller. The subsequent API call can be either the write or writeMicroSeconds method, depending on which mechanism is preferred to control the servo. Optionally, the servo can be detached using the detach method after being attached. Although the second and third usage patterns are different from the ground truth, these usage patterns are still valid.

Moreover, the generated usage patterns can be different due to calling optional APIs, as shown in Case 2 in Figure 9. We find that the APIs can be categorized into two based on their functionality: core and optional. In Case 2, the API usage pattern is begin, setWorkingMode, and queryData. "begin" is used to initialize the connection to the controller, setWorkingMode is used to configure the working mode of the sensor, and queryData is used to get the reading value from the sensor. These APIs correspond to the core functionality and should be invoked sequentially. On the other hand, getVersionDate and setDutyCycle are optional APIs; getVersionDate is used to get the firmware version, while setDutyCycle is used to overwrite the default configuration of the duty cycle. These APIs can be called in between the core APIs. If these optional APIs are removed, the top-3 predictions become identical.

VII. Related Works

A. Developing Embedded System

Makhsari and Mesbah [3] scrutinize the challenges of programming embedded systems in the context of IoT environments by analyzing the bugs and interviewing IoT developers. The study [3] confirms that developers often find difficulty when developing applications that involve hardware because it often requires domain-specific knowledge. In the Arduino domain, Booth et al. [1] conducted a user study by asking developers to develop Arduino applications. The study demonstrated that 80% of circuit-related problems are caused by miswiring and missing electronic components. Moreover, both experienced and beginner developers faced a problem despite their level of expertise. As a consequence, a number of tools have been proposed to help developers develop Arduino applications. TAC (Trigger-Action-Circuits) [11] allows developers to specify hardware setups automatically using hand-crafted templates. However, TAC is not scalable because developers should manually create such templates for each hardware. VirtualWire [12] allows developers to configure hardware connections through software by leveraging a specialized device. Similarly, CircuitSense [13] and SchemaBoard [14] can visualize existing hardware connections in software using specialized hardware. Such approaches [12], [14] only help developers to test and debug problems; developers still rely on their domain-specific knowledge to correctly configure the hardware. Several interactive tutorial-based approaches have also been proposed to guide developers to set up hardware correctly, such as [30]–[33]. However, such approaches still hand-crafted tutorials.

B. Recommending API Usage Patterns

Various techniques have been proposed to mine and generate API usage patterns, such as MAPO [34], UP-miner [35], and PAM [36]. MAPO and UP-miner leverage pattern mining technique, while PAM use probabilistic model as the core technique. FACER [37] recommends API usage patterns based on code-examples. BIKER [38], [39] recommends API usage patterns by leveraging the similarity of StackOverflow posts and API documentation. Yuan et al. [40], [41] propose API recommendation system for Android applications. Recently, many deep learning-based models have been adapted to learn API usage patterns. RecipeGen++ generates API sequences specifically for trigger action programs [42], [43]. Ling et al. develop a graph neural network using collaborative filtering to recommend relevant API usage patterns [44]. DeepAPI [10] leverage Recurrent Neural Network and attention mechanism to generate API usage patterns.

VIII. Conclusions

Writing Arduino applications can be challenging due to two main reasons. Firstly, developers may be required to configure hardware that they are not familiar with. Secondly, hardware libraries usually have specific usage patterns that developers may not remember. To address these challenges, we propose a framework called ArduinoProg for automating Arduino programming. This framework has three main components: Library Retriever, Configuration Classifier, and Pattern Generator. Library Retriever recommends relevant hardware libraries based on natural language queries, Configuration Classifier recommends the hardware configuration by classifying method definitions found in the libraries' implementation files to a configuration class, and Pattern Generator generates sample API usage patterns using a sequence-to-sequence deep learning model. We evaluate each component of ArduinoProg using various models and real-world queries, and our results show that they are promising for guiding Arduino developers.

Our study also suggests that further improvements can be explored, such as incorporating more diverse features into Configuration Classifier, adapting various ranking or clustering techniques in Library Retriever, model training with more real-world queries and Arduino documents, and evaluation with actual developers. However, we leave such improvements as future work.

IX. Data Availability

The implementation of ArduinoProg, labelling results, and datasets are available on https://github.com/imamnurby/ArduProg.

Acknowledgment

We thank the anonymous reviewers for their insightful feedback on our paper. This research / project is supported by the Ministry of Education, Singapore, under its Academic Research Fund Tier 2 (Award No.: MOE2019-T2-1-193). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of the Ministry of Education, Singapore.

REFERENCES

[1] T. Booth, S. Stumpf, J. Bird, and S. Jones, "Crossed wires: Investigating the problems of end-user developers in a physical computing task," in *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. ACM, 2016, pp. 3485–3497. [Online]. Available: https://doi.org/10.1145/2858036.2858533

[2] "Arduino library list - arduino libraries," https://www.arduinolibraries.info/, (Accessed on 01/20/2023).

[3] A. Makhshari and A. Mesbah, "Iot bugs and development challenges," in *43rd IEEE/ACM International Conference on Software Engineering*. IEEE, 2021, pp. 460–472. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00051

[4] "Arduino forum," https://forum.arduino.cc/, (Accessed on 01/20/2023).

[5] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems*, 2017, pp. 5998–6008. [Online]. Available: https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[6] C. Lai and T. Kuo, "Available capacity computation model based on long short-term memory recurrent neural network for gelled-electrolyte batteries in golf carts," *IEEE Access*, vol. 10, pp. 54 433–54 444, 2022. [Online]. Available: https://doi.org/10.1109/ACCESS.2021.3123789

[7] 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," in *Findings of the Association for Computational Linguistics*, ser. Findings of ACL, vol. EMNLP 2020. Association for Computational Linguistics, 2020, pp. 1536–1547. [Online]. Available: https://doi.org/10.18653/v1/2020.findings-emnlp.139

[8] W. U. Ahmad, S. Chakraborty, B. Ray, and K. Chang, "Unified pre-training for program understanding and generation," in *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, 2021, pp. 2655–2668. [Online]. Available: https://doi.org/10.18653/v1/2021.naacl-main.211

[9] Y. Wang, W. Wang, S. R. Joty, and S. C. H. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2021, pp. 8696–8708. [Online]. Available: https://doi.org/10.18653/v1/2021.emnlp-main.685

[10] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep API learning," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 631–642. [Online]. Available: https://doi.org/10.1145/2950290.2950334

[11] F. Anderson, T. Grossman, and G. W. Fitzmaurice, "Trigger-action-circuits: Leveraging generative design to enable novices to design and build circuitry," in *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*. ACM, 2017, pp. 331–342. [Online]. Available: https://doi.org/10.1145/3126594.3126637

[12] W. Lee, R. Prasad, S. Je, Y. Kim, I. Oakley, D. Ashbrook, and A. Bianchi, "Virtualwire: Supporting rapid prototyping with instant reconfigurations of wires in breadboarded circuits," in *TEI '21: Fifteenth International Conference on Tangible, Embedded, and Embodied Interaction*. ACM, 2021, pp. 4:1–4:12. [Online]. Available: https://doi.org/10.1145/3430524.3440623

[13] T. Wu, B. Wang, J. Lee, H. Shen, Y. Wu, Y. Chen, P. Ku, M. Hsu, Y. Lin, and M. Y. Chen, "Circuitsense: Automatic sensing of physical circuits and generation of virtual circuits to support software tools," in *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*. ACM, 2017, pp. 311–319. [Online]. Available: https://doi.org/10.1145/3126594.3126634

[14] Y. Kim, H. Lee, R. Prasad, S. Je, Y. Choi, D. Ashbrook, I. Oakley, and A. Bianchi, "Schemaboard: Supporting correct assembly of schematic circuits using dynamic in-situ visualization," in *UIST '20: The 33rd Annual ACM Symposium on User Interface Software and Technology*. ACM, 2020, pp. 987–998. [Online]. Available: https://doi.org/10.1145/3379337.3415887

[15] C. Rosen and E. Shihab, "What are mobile developers asking about? A large scale study using stack overflow," *Empir. Softw. Eng.*, vol. 21, no. 3, pp. 1192–1223, 2016. [Online]. Available: https://doi.org/10.1007/s10664-015-9379-3

[16] B. Xu, Z. Xing, X. Xia, and D. Lo, "Answerbot: automated generation of answer summary to developersź technical questions," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2017, pp. 706–716. [Online]. Available: https://doi.org/10.1109/ASE.2017.8115681

[17] G. Chen, C. Chen, Z. Xing, and B. Xu, "Learning a dual-language vector space for domain-specific cross-lingual question retrieval," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016, pp. 744–755. [Online]. Available: https://doi.org/10.1145/2970276.2970317

[18] F. Schroff, D. Kalenichenko, and J. Philbin, "Facenet: A unified embedding for face recognition and clustering," in *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*. IEEE Computer Society, 2015, pp. 815–823. [Online]. Available: https://doi.org/10.1109/CVPR.2015.7298682

[19] S. E. Robertson, S. Walker, S. Jones, M. Hancock-Beaulieu, and M. Gatford, "Okapi at TREC-3," in *Proceedings of The Third Text REtrieval Conference*, ser. NIST Special Publication, vol. 500-225. National Institute of Standards and Technology (NIST), 1994, pp. 109–126. [Online]. Available: http://trec.nist.gov/pubs/trec3/papers/city.ps.gz

[20] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," *CoRR*, vol. abs/1810.04805, 2018. [Online]. Available: http://arxiv.org/abs/1810.04805

[21] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "Distilbert, a distilled version of BERT: smaller, faster, cheaper, and lighter," *CoRR*, vol. abs/1910.01108, 2019. [Online]. Available: http://arxiv.org/abs/1910.01108

[22] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized BERT pretraining approach," *CoRR*, vol. abs/1907.11692, 2019. [Online]. Available: http://arxiv.org/abs/1907.11692

[23] D. M. W. Powers, "Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation," *CoRR*, vol. abs/2010.16061, 2020. [Online]. Available: https://arxiv.org/abs/2010.16061

[24] "Treesitter introduction," https://tree-sitter.github.io/tree-sitter/, (Accessed on 01/20/2023).

[25] T. K. Ho, "Random decision forests," in *Third International Conference on Document Analysis and Recognition*. IEEE Computer Society, 1995, pp. 278–282. [Online]. Available: https://doi.org/10.1109/ICDAR.1995.598994

[26] C. Cortes and V. Vapnik, "Support-vector networks," *Mach. Learn.*, vol. 20, no. 3, pp. 273–297, 1995. [Online]. Available: https://doi.org/10.1007/BF00994018

[27] K. Järvelin and J. Kekäläinen, "Cumulated gain-based evaluation of IR techniques," *ACM Trans. Inf. Syst.*, vol. 20, no. 4, pp. 422–446, 2002. [Online]. Available: http://doi.acm.org/10.1145/582415.582418

[28] C.-Y. Lin, "ROUGE: A package for automatic evaluation of summaries," in *Text Summarization Branches Out*. Barcelona, Spain: Association for Computational Linguistics, Jul. 2004, pp. 74–81. [Online]. Available: https://aclanthology.org/W04-1013

[29] S. Banerjee and A. Lavie, "METEOR: an automatic metric for MT evaluation with improved correlation with human judgments," in *Proceedings of the Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization@ACL*. Association for Computational Linguistics, 2005, pp. 65–72. [Online]. Available: https://aclanthology.org/W05-0909/

[30] Y. Kim, Y. Choi, D. Kang, M. Lee, T. Nam, and A. Bianchi, "Heyteddy: Conversational test-driven development for physical computing," *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, vol. 3, no. 4, pp. 139:1–139:21, 2019. [Online]. Available: https://doi.org/10.1145/3369838

[31] J. U. Davis, J. Gong, Y. Sun, P. K. Chilana, and X. Yang, "Circuitstyle: A system for peripherally reinforcing best practices in hardware computing," in *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*. ACM, 2019, pp. 109–120. [Online]. Available: https://doi.org/10.1145/3332165.3347920

[32] J. Warner, B. Lafreniere, G. W. Fitzmaurice, and T. Grossman, "Electrotutor: Test-driven physical computing tutorials," in *The 31st Annual ACM Symposium on User Interface Software and Technology*. ACM, 2018, pp. 435–446. [Online]. Available: https://doi.org/10.1145/3242587.3242591

[33] D. Drew, J. L. Newcomb, W. McGrath, F. Maksimovic, D. Mellis, and B. Hartmann, "The toastboard: Ubiquitous instrumentation and automated checking of breadboarded circuits," in *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*. ACM, 2016, pp. 677–686. [Online]. Available: https://doi.org/10.1145/2984511.2984566

[34] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "MAPO: mining and recommending API usage patterns," in *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference*, ser. Lecture Notes in Computer Science, vol. 5653. Springer, 2009, pp. 318–343. [Online]. Available: https://doi.org/10.1007/978-3-642-03013-0_15

[35] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang, "Mining succinct and high-coverage API usage patterns from source code," in *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Computer Society, 2013, pp. 319–328. [Online]. Available: https://doi.org/10.1109/MSR.2013.6624045

[36] J. M. Fowkes and C. Sutton, "Parameter-free probabilistic API mining across github," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 254–265. [Online]. Available: https://doi.org/10.1145/2950290.2950319

[37] S. Abid, S. Shamail, H. A. Basit, and S. Nadi, "FACER: an API usage-based code-example recommender for opportunistic reuse," *Empir. Softw. Eng.*, vol. 26, no. 5, p. 110, 2021. [Online]. Available: https://doi.org/10.1007/s10664-021-10000-w

[38] L. Cai, H. Wang, Q. Huang, X. Xia, Z. Xing, and D. Lo, "BIKER: a tool for bi-information source based API method recommendation," in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2019, pp. 1075–1079. [Online]. Available: https://doi.org/10.1145/3338906.3341174

[39] Q. Huang, X. Xia, Z. Xing, D. Lo, and X. Wang, "API method recommendation without worrying about the task-api knowledge gap," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 293–304. [Online]. Available: https://doi.org/10.1145/3238147.3238191

[40] W. Yuan, H. H. Nguyen, L. Jiang, and Y. Chen, "Libraryguru: API recommendation for android developers," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ACM, 2018, pp. 364–365. [Online]. Available: https://doi.org/10.1145/3183440.3195011

[41] W. Yuan, H. H. Nguyen, L. Jiang, Y. Chen, J. Zhao, and H. Yu, "API recommendation for event-driven android application development," *Inf. Softw. Technol.*, vol. 107, pp. 30–47, 2019. [Online]. Available: https://doi.org/10.1016/j.infsof.2018.10.010

[42] I. N. B. Yusuf, L. Jiang, and D. Lo, "Accurate generation of trigger-action programs with domain-adapted sequence-to-sequence learning," in *International Conference on Program Comprehension*. ACM, 2022, pp. 99–110. [Online]. Available: https://doi.org/10.1145/3524610.3527922

[43] I. N. B. Yusuf, D. B. A. Jamal, L. Jiang, and D. Lo, "Recipegen++: an automated trigger action programs generator," in *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2022, pp. 1672–1676. [Online]. Available: https://doi.org/10.1145/3540250.3558913

[44] C. Ling, Y. Zou, and B. Xie, "Graph neural network based collaborative filtering for API usage recommendation," in *28th IEEE International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 2021, pp. 36–47. [Online]. Available: https://doi.org/10.1109/SANER50967.2021.00013