ArduinoProg: Towards Automating Arduino Programming

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

***Abstract*—Writing code for Arduino poses unique challenges. A developer 1) needs hardware-specific knowledge about the interface configuration between the Arduino controller and the I/O hardware, 2) identifies a suitable driver library for the I/O hardware, and 3) follows certain usage patterns of the driver library in order to use them properly. In this work, based on a study of real-world user queries posted in the Arduino forum, we propose ArduinoProg to address such challenges. ArduinoProg consists of three components, i.e., Library Retriever, Configuration Classifier, and Pattern Generator. Given a query, Library Retriever retrieves library names relevant to the I/O hardware identified from the query using vector-based similarity matching. Configuration Classifier predicts the interface configuration between the I/O hardware and the Arduino controller based on the method definitions of each library. Pattern Generator generates the usage pattern of a library using a sequence-to-sequence deep learning model. We have evaluated ArduinoProg using real-world queries, and our results show that the components of ArduinoProg can generate accurate and useful suggestions to guide developers in writing Arduino code.**
Demo video: bit.ly/3Y3aeBe
Tool: https://huggingface.co/spaces/imamnurby/ArduinoProg
Code and data: https://github.com/imamnurby/ArduProg

***Index Terms*—arduino programming, information retrieval, code generation, deep learning**

I. Introduction

Arduino projects typically consist of three main components: a controller, one or more I/O hardware (e.g., temperature sensor) that are connected to the controller to interact with the environment, and software code that define the logic of the system. Writing Arduino code poses unique challenges because of the involvement of both hardware and software components.

First, Arduino coding may require a developer to configure the interface between the Arduino controller and the I/O hardware. The interface configuration between the code and physical hardware should be consistent to ensure the system works properly. For instance, in Fig. 1, a developer connects a servo to interface number 10 in the physical space. Consequently, the developer should also configure the Servo object in the code using interface number 10 (myServo.attach(pin=10)). Choosing the correct interface for an I/O hardware may require domain-specific knowledge that a developer may not be familiar with [1], [2].

Second, there is a wide range of driver libraries available. Each driver library may have its own set of API usage patterns, as illustrated by the few lines of code on the top of Fig. 1. Developers may not be familiar with such usage patterns due to the large number of libraries and usage patterns available.

```
Servo  myServo;
myServo.attach(pin=10);
myServo.write(10)
```

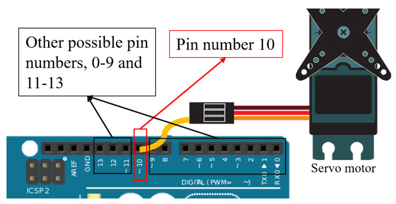

Fig. 1. An example of how the setup in the physical space and code should match. The servo should be connected to the interface number 10 because the servo motor is initialized using interface number 10 in the code.

To address the aforementioned challenges, we propose ArduinoProg, a tool to assist developers in identifying relevant driver libraries, as well as suggesting suitable interface configurations and usage patterns based on natural language queries. Based on our preliminary study of Arduino forum posts [3], we have developed ArduinoProg as an integration of three main components, i.e., Library Retriever, Configuration Classifier, and Pattern Generator. First, Library Retriever utilizes NLP techniques to analyze the query's grammatical structure, extract keywords, and retrieve relevant library names. Second, Configuration Classifier extracts method definitions from each library, encode them into feature vectors using a deep learning model, and predicts the interface configuration using a classifier. Third, Pattern Generator employs a deep learning sequence-to-sequence model to generate the usage patterns of each library.

We have evaluated ArduinoProg on real-world queries by leveraging various machine learning models to instantiate each components. Library Retriever can achieve 0.64-0.97 and Pattern Generator yields 0.32-0.73 in terms of Normalized Discounted Cumulative Gain; the performance of Configuration Classifier ranges from 0.79-0.85 in terms of Area Under the Receiver Operating Characteristics Curve. Such results indicate that ArduinoProg can possibly produce useful suggestions to guide developers in writing Arduino code.

In summary, our contributions are as follows: 1) we propose ArduinoProg, the first tool that can identify the relevant I/O hardware libraries and suggest its pin configurations and API usage patterns, and 2) we open source the implementation of ArduinoProg, along with the training and testing data.

II. Related Works

Several empirical studies have been conducted in the field of embedded development [1], [4] . Makhshari and Mesbah [1] confirm the challenging nature of IoT (Internet of Things) embedded development, emphasizing the need for hardware-specific knowledge. Uddin et al. [4] discover that microcontroller configuration is among the top three most discussed IoT-related topics on StackOverflow.

Several works have been proposed to recommend relevant API usage patterns based on natural language queries [5]–[8]. However, solely generating API usage patterns is insufficient for Arduino applications, as developers need to configure the interface in the physical space and code. By leveraging our tool, developers can discover the API usage patterns and also the necessary configurations for a library, providing a comprehensive solution for Arduino application development.

III. ArduinoProg

A. Design Background

The design of ArduinoProg is motivated by our prior work [3], where we studied real-world user questions from the Arduino Discussion Forum. This study provided three key insights. First, driver libraries have specific API usage patterns. Because many driver libraries available, it becomes necessary to have a component that can automatically generate such patterns. This finding motivates the need of Pattern Generator. Second, we discovered that driver libraries can be categorized into four configuration categories based on the communication protocol of the I/O hardware: address-based (I2C), Serial (SPI), Asynchronous (UART), and Explicit. Furthermore, method definitions of hardware within the same communication protocol often share similar tokens, as depicted in Fig. 2. This finding allows us to formulate the problem of inferring interface configuration to classifying method definitions into the correct configuration category, thus motivating the design of Configuration Classifier. Last, we found that both the interface configuration and API usage pattern of a driver library are independent of other driver libraries. Therefore, we can first identify the I/O hardware in the input query and then individually predict the interface configuration and generate the API usage pattern for each identified I/O hardware. This insight motivates the design of Library Retriever.

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 to get the sensor value from the register memory of the hardware. Both method definitions involves similar tokens such as "wire", "beginTransmission", and "rv" .

B. Workflow

Fig. 3 illustrates the high-level workflow of ArduinoProg. Given a natural language query, ① Library Retriever recommends relevant libraries. For each library, ② Configuration Classifier predicts the interface configuration and ③ Pattern Generator generates the API usage patterns.

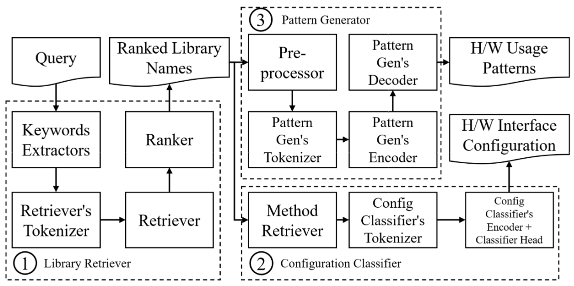

Fig. 3. The components of ArduinoProg.

The workflow of ArduinoProg consists of two phases, i.e., training and inference. In the training phase, we train a set of models to perform the required tasks. In the inference phase, ArduinoProg is ready to interact with a developer.

Training. First, we train Retriever in ① to suggest relevant driver libraries based on the input query. We leverage Triplet Loss [9] for training the retrieval model. Each training instance is a triplet, comprising an anchor, a positive sample, and a negative sample. The anchor corresponds to a library name, while the positive sample is a library name that is relevant to the anchor. In contrast, the negative sample is an irrelevant library name. In this context, a relevant library is the driver library that supports same I/O hardware. The training objective is to minimize the distance between the anchor and positive sample in the vector space, while maximizing the distance between the anchor and negative sample.

Second, we train Configuration Classifier's Encoder and Classifier Head in ② to predict the interface configuration given the method definitions of a driver library. One training instance is a set of method definitions paired with the ground truth configuration category. Given one training instance, Configuration Classifier's Tokenizer transforms the method definitions into a list of subword token ids. Configuration Classifier's Encoder converts each token id into a feature vector. Classifier Head takes the the list of feature vectors to predict the configuration category. Subsequently, the loss is by comparing the predicted category with the ground truth using Cross Entropy [10]. The computed loss is used to update the model's parameters.

Third, we train Pattern Generator's Encoder and Pattern Generator's Decoder in ③ to generate the usage patterns of a driver library. One training instance is a library name paired with its ground truth usage pattern. Given one training instance, Preprocessor appends the the hardware classname to the library name. Then, Pattern Generator's tokenizer transforms the processed library name into a list of subword token ids. Pattern Generator's Encoder converts each token id into a feature vector. Pattern Generator's Decoder takes the list of feature vectors to generate the usage patterns. The model's parameters are updated by computing the loss between the generated usage pattern and the ground truth using Cross Entropy as the loss function.

Inference. Given a natural language query, ① Library Retriever leverages Keywords Extractor to omit unimportant tokens (e.g., conjunctions) and output a list of keywords (i.e., tokens that are most likely to appear in a library name). Keywords Extractor extracts the noun phrases (a root noun + its modifiers) from the query by analyzing its grammatical structure using dependency parser For example, given a query "Using BMP280 sensor to control a DC motor", the noun phrases are "BMP280 sensor" and "DC motor". For each noun phrase Keywords Extractor extracts the token that corresponds to an I/O hardware name by checking whether a token consists of alphanumeric letters. We use such a heuristic since most hardware names comprise alphanumeric letters (e.g., BMP280). If our heuristic fails to extract a hardware name, Keywords Extractor uses the root noun of the noun phrase as the keyword, e.g., "motor" in "dc motor".

Retriever's Tokenizer convert each keyword into a list of token ids. Retriever converts the list of token ids to a vector and computes the similarity between this vector with the available library name vectors. The result is an unordered list of K x m library names, where K is the top-K results and m is the number of keywords. Finally, Ranker outputs a ranked list of K library names.

Given a library name, ② Configuration Classifier leverages Method Retriever to retrieve the library's .cpp files and then extracts the method definitions. The subsequent steps are the same as in the training phase, but without updating the model's parameters. After predicting the configuration category, we map the category to the interface configuration using a predefined dictionary.

Similarly, ③ Pattern Generator takes the library name as the input. The remaining steps are the same as in the training phase, but the without updating the model's parameters.

C. Tool Usage

Envisioned Users. ArduinoProg caters to both developers who are familiar with I/O hardware and those who are not. For developers who already know the I/O hardware they want to use, ArduinoProg enables them to identify the driver library for that I/O hardware, infer its interface configuration, and generate its API usage pattern. Furthermore, by describing the high-level functionality (e.g., "sensor to measure temperature"), ArduinoProg allows developers who are uncertain of using which I/O hardware to identify the I/O hardware that perform the described functionality.

Usage. Developers can access https://huggingface.co/spaces/imamnurby/ArduinoProg to use ArduinoProg, then input a suitable query and ArduinoProg will return a set of results. Developers can also locally train and deploy ArduinoProg by following the instructions in https://github.com/imamnurby/ArduProg.

IV. Evaluation

A. Experimental Setting

Library Retriever. As we explained in Section III, the training data is a triplet of (anchor, positive sample, negative sample) We create the Triplet as follows. We crawl 4,309 library names from the library reference [11]. We iterate through each library name, extract the hardware name, then remove the frequent tokens. For each anchor, we sample 50 positive and negative samples. The number of training triplets is 234,577. For the testing set, we use 35 posts that we collected in the prior study [3]. We use the question title as the query as it often summarizes the question's main idea [12].

We evaluate Library Retriever by instantiating Retriever (see ① in Fig. 3) using CodeBERT [13], RoBERTa [14], and BM25 [15].

Configuration Classifier. For the training data, we crawl the hardware libraries listed in the library reference [11]. We extract the method definitions using tree-sitter [16]. For the ground truths, we manually annotate the libraries with its supported communication protocol. For the testing data, we feed the real-world posts that we collected in the prior study [3] to Library Retriever, take the top 10 libraries for each query, and extract the method definitions. The number of testing instances is 290.

We evaluate Configuration Classifier by instantiating Configuration Classifier's Encoder using CodeBERT [13] and Classifier Head using Dense Layer, SVM (Support Vector Machine) [17], and RF [18] (Random Forest) (see ② in Fig. 3).

Pattern Generator. We crawl 106,777 client files from GitHub and extract the API usage patterns of I/O hardware in those client files by traversing the abstract syntax tree using tree-sitter [16]. The number of training instances is 84,222. For the testing data, we leverage the codes attached in the forum posts that we collected in the prior study [3]. We manually extract the API usage patterns in each code. The number of testing instances is 31.

We evaluate Pattern Generator by instantiating Pattern Generator's Encoder and Decoder (see ③ in Fig. 3) using CodeBERT [13], CodeT5 [19], and DeepAPI [20].

B. Evaluation Metrics

NDCG. We use Normalized Discounted Cumulative Gain (NDCG) to evaluate Library Retriever and Pattern Generator. NDCG measures how close a result is to the ground truth using a relevancy function and weights the relevancy score by its ranking in the result list. NDCG ranges from 0 to 1. For the relevancy score, we use binary for evaluating Library Retriever and ROUGE score and Longest Common Subsequences (LCS) for evaluating Pattern Generator.

AUC. We use AUC (Area Under the Receiver Operating Characteristics) to evaluate Configuration Classifier. AUC measures how well a classifier can differentiate each target class of a classifier. AUC ranges between 0 and 1.

C. Baselines

There is no comparable tool for automating Arduino programming because as far as we know, ArduinoProg is the first tool that can generate both hardware configuration and

API usage patterns. We only evaluate and compare variants of ArduinoProg by instantiating its components using various machine learning models.

D. Results

Library Retriever. Table I shows the evaluation results of Library Retriever. Overall, CodeBERT performs the best among the other models, with the average NDCG 0.78, followed by BM25 and RoBERTa. Without performing keyword extraction, the performance of CodeBERT, RoBERTa, and CodeBERT degrades by 0.42, 0.43, and 0.39 on average. Such results demonstrate that our keyword extraction can significantly improve the performance.

TABLE I
THE EVALUATION RESULTS OF LIBRARY RETRIEVER.

Model	NDCG@K			Average
	K=1	K=5	K=10	
BM25	0.77	0.71	**0.64**	0.71
RoBERTa	0.89	0.68	0.53	0.70
CodeBERT	**0.97**	**0.74**	0.62	**0.78**

Configuration Classifier. Table II shows the evaluation results of Configuration Classifier. The overall results indicate that SVM yields the best performance, with an average AUC of 0.85. For Dense and Random Forest, the low AUC values are caused by UART and SPI classes. The possible reason is that UART and SPI classes have fewer training samples (i.e., 19 and 45) than I2C and Digital/Analog I/O (i.e., 159 and 98).

TABLE II
THE EVALUATION RESULTS OF CONFIGURATION CLASSIFIER.

Model	Average AUC per Class				Average
	UART	SPI	I2C	Explicit	
Dense	0.66	0.68	**0.91**	0.90	0.79
SVM	**0.82**	**0.82**	0.88	0.89	**0.85**
RF	0.69	0.69	0.90	**0.91**	0.80

Pattern Generator. Table III shows the evaluation results of Pattern Generator. Overall, CodeBERT and DeepAPI yield competitive performance in terms of NDCG-ROUGE@K and NDCG-LCS@K. A high NDCG-ROUGE@K and NDCG-LCS@K mean that the APIs in the ground truths are likely to appear in the generated API sequences .

TABLE III
THE EVALUATION RESULTS OF PATTERN GENERATOR.

Model	NDCG@K					
	ROUGE			LCS		
	K=1	K=5	K=10	K=1	K=5	K=10
DeepAPI	0.71	**0.69**	**0.69**	0.64	**0.64**	**0.64**
CodeBERT	**0.73**	**0.69**	**0.69**	**0.65**	0.60	0.59
CodeT5	0.52	0.46	0.45	0.34	0.33	0.32

V. CONCLUSION

We propose ArduinoProg to address the challenges when writing Arduino code. Given a query in natural language, ArduinoProg recommends the relevant libraries, the interface configurations, and the usage patterns of each library. Our evaluation results using real-world queries indicate that ArduinoProg can possibly produce useful suggestions to guide developers in writing Arduino code. In the future, we intend to release the tool to the Arduino community and conduct a user study to gather feedback and further enhance its functionality.

Acknowledgment. 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] A. Makhshari and A. Mesbah, “Iot bugs and development challenges,” in *International Conference on Software Engineering*. IEEE, 2021, pp. 460–472.

[2] T. Booth, S. Stumpf, J. Bird, and S. Jones, “Crossed wires: Investigating the problems of end-user developers in a physical computing task,” in *Conference on Human Factors in Computing Systems*. ACM, 2016, pp. 3485–3497.

[3] I. N. B. Yusuf, D. B. A. Jamal, and L. Jiang, “Automating arduino programming: From hardware setups to sample source code generation,” in *Mining Software Repositories*. IEEE, 2023, pp. 453–464.

[4] G. Uddin, F. Sabir, Y. Guéhéneuc, O. Alam, and F. Khomh, “An empirical study of iot topics in iot developer discussions on stack overflow,” *Empir. Softw. Eng.*, vol. 26, no. 6, p. 121, 2021.

[5] 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 *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2019, pp. 1075–1079.

[6] 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.

[7] 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.

[8] 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.

[9] F. Schroff, D. Kalenichenko, and J. Philbin, “Facenet: A unified embedding for face recognition and clustering,” in *Computer Vision and Pattern Recognition*. IEEE Computer Society, 2015, pp. 815–823.

[10] C. M. Bishop, *Pattern recognition and machine learning, 5th Edition*, ser. Information science and statistics. Springer, 2007.

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

[12] 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.

[13] 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: EMNLP*. Association for Computational Linguistics, 2020, pp. 1536–1547.

[14] 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.

[15] S. E. Robertson, S. Walker, S. Jones, M. Hancock-Beaulieu, and M. Gatford, “Okapi at TREC-3,” in *Text REtrieval Conference*, ser. NIST Special Publication, vol. 500-225. National Institute of Standards and Technology, 1994, pp. 109–126.

[16] “Github - tree-sitter/tree-sitter: An incremental parsing system for programming tools,” https://github.com/tree-sitter/tree-sitter, (Accessed on 11/18/2022).

[17] C. Cortes and V. Vapnik, “Support-vector networks,” *Mach. Learn.*, pp. 273–297, 1995.

[18] T. K. Ho, “Random decision forests,” in *International Conference on Document Analysis and Recognition. Volume I*. IEEE Computer Society, 1995, pp. 278–282.

[19] 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 *Empirical Methods in Natural Language Processing*, 2021, pp. 8696–8708.

[20] X. Gu, H. Zhang, D. Zhang, and S. Kim, “Deep API learning,” in *Foundations of Software Engineering*, 2016, pp. 631–642.