GETMic: Grammatical Evolution based Test Case Generation for Microcontroller

Joohee Kim
School of Computing
joohee0@kaist.ac.kr

Nayoung Oh
School of Computing
nyoh@kaist.ac.kr

Darae Lee
School of Computing
2darae@kaist.ac.kr

Yunseok Jeong
School of Computing
wjddbstjr03@kaist.ac.kr

Abstract

Microcontrollers like the Arduino or Raspberry Pi Pico are the lifeblood of embedded systems. Like any other programs, the programs loaded on these microcontrollers can be made much more robust with code coverage testing. However, testing for microcontrollers is complicated by the fact that we have to consider not only the software but also the hardware. Even if we run the exact same code on a microcontroller, the program behaves differently depending on how the hardware is configured and how the hardware interacts with the environment. To deal with this kind of consideration, we first 1) built a simple hardware simulator and 2) used grammatical evolution, a type of genetic algorithm, to encode the interaction with the hardware. We could successfully generate a sequence of interactions to the simulator (i.e. a test case) which maximizes the coverage of the target code, outperforming the random generation in terms of coverage and time-efficiency.[1]

1 Introduction

In the realm of embedded systems, microcontrollers such as the Arduino and Raspberry Pi Pico play a pivotal role, serving as the foundation for many applications. However, coverage testing, which can greatly enhance the robustness of the program, may be tricky for the microcontrollers. Unlike regular software, these programs are dependent on both the software and the hardware. Even if we use the same code on a microcontroller, the results can vary based on how the hardware is set up and interacts with the environment. This kind of characteristic brings a lot of complexity compared to other general code coverage tests. Through the project, we tried to implement the end-to-end test case generator that takes target code and creates test cases that maximize the coverage of that code.

[1]All codes available at https://github.com/darae-lee/GETMic

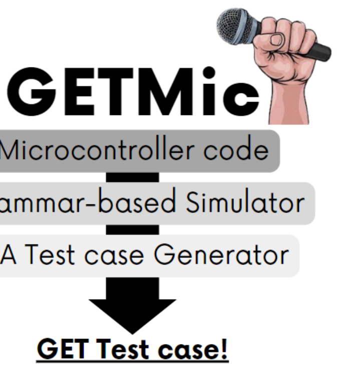

Figure 1: Overall flow of GETMic

2 Background and Related Works

2.1 Grammatical Evolution

Grammatical Evolution (GE) is an evolutionary algorithm, similar to a genetic algorithm (GA). It employs a Backus-Naur Form grammar definition in a genotype-to-phenotype mapping process. GE distinguishes between the search and solution spaces, enabling an unconstrained evolutionary search on simple variable-length binary strings (O'Neill and Ryan, 2001). Deiner and Frädrich use the grammar definition for test case generation for Scratch which requires user interaction similar to microcontrollers (Deiner et al., 2020).

2.2 Microcontroller

A microcontroller is a computer on a single integrated circuit that includes a low-power consuming CPU, RAM, and some I/O ports (Hussain et al., 2016). Usually, a microcontroller is programmable with supporting language (Nguyen et al., 2018), even controlling the I/O ports directly. Arduino and RasberryPi Pico series are popular open-source microcontrollers, widely used for education and prototyping. A microcontroller even supports the dynamic language, Python. MicroPython is a Python library supporting diverse open-source microcontrollers including ESP32 for Arduino nano series

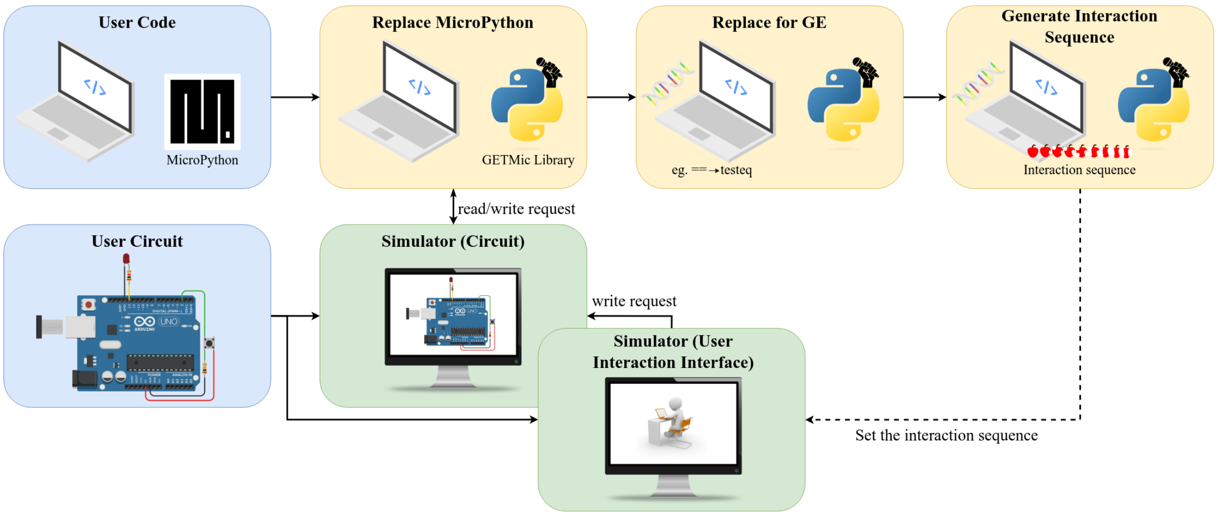

Figure 2: Overall System of GETMic

and RP2 for RasberryPi Pico (mic).

2.3 Testcase generation for microcontrollers

Research deals with test case generation for microcontrollers, including embedded systems with the microcontroller. Hossain et al. study interaction model generation in embedded systems, considering the dependency between interactions such as shard variable or physically coupled sensing (Hossain and Lee, 2019). The research focuses on the source code level interaction model, but it cannot capture circuit-level dependency which does not appear in the code. Therefore, we propose simulation as an alternative to interaction models. In the specific context of code coverage, Saha et al. develop code coverage checking systems for embedded systems considering the hardware limitations such as memory bound (Saha et al., 2021). However, Saha et al. do not deal with user interaction sequence as an input which is human-level input to control the microcontroller. Our approach is close to the work of Padmanabhan (Padmanabhan, 2022). Padmanabhan converts the circuit board into an event sequence graph to generate test cases. As Padmanabhan focuses on programming instruction for all 'circuit coverage', the approach cannot work for user-generated codes and corresponding circuits. Therefore, we propose our work GETMic which considers codes and circuits together to propose one sequence of user interactions for real-world testing.

3 Approach

GETMic consists of a simulator for simulating a circuit and user interaction, and a test case generation system using the simulator as shown in the figure 2.

3.1 Simulator

As existing simulators are developed to simulate circuits or debug the microcontroller code, the simulators are not fast enough to automatically simulate user interactions, and log the results. We design our simulator as a wrapper class of 'machine' in the MicroPython library. The simulator consists of two parts: a circuit part and a user interaction interface part.

3.1.1 Circuit

First, the circuit part simulates the electricity flow with the user-generated configuration of electrical components. The user can choose what and how many components to add to the circuit including a board, and how to connect them. Currently, we only support connection of the main signal port and one of voltage and ground pin. When the user adds interactable components, the corresponding user interaction grammar is generated and added to the interaction candidates. The detailed explanation about the grammar is in the section 3.2.1.

The circuit supports three operations: read, write, and update. First, read returns the current state of the board pin. As reading does not change the state of other components, it simply returns without update.

Second, the write can occur due to two reasons.

```
<test case> ::= <interaction> | <test case> <interaction>
<interaction> ::= <digital interaction> | <analog sensor interaction>
<digital interaction> ::= set <digital sensor> to 0 | set <digital sensor> to 1
<digital sensor> ::= button
<analog sensor interaction> ::= set <analog sensor> to <value>
<analog sensor> ::= temperature sensor | potentiometer | photoresistor
<value> ::= <digit> <digit> <digit> <digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Figure 3: GETMic Grammar

First, when a direct write request to the port occurs, a write operation happens to the port in the board. Second, when a user interaction request arrives, the user interaction can invoke a write request to the component with corresponding values. The detailed explanation about the mapping is in the section 3.2.1. Regardless of the source or write operation, it will proceed to update operation.

The circuit updates the corresponding state of the component. Then, the total circuit states are updated by calculating the new voltage state of each port, starting from each starting point of the electricity flow.

3.1.2 User Interaction Interface

The user interaction interface redirects the interaction requests to the circuit. As a user interaction happens regardless of circuit states or updates, we generate a separate thread for the user interaction interface. The interaction interface keeps reading the initially given interaction sequence, sends a write request to the circuit part, and waits for a while mimicking the human action delay. In addition, the interface supports converting machine-readable requests into human-readable actions, so that a user can follow the generated user interaction sequence.

3.2 Test case generation

Test case generation was performed using a GE. GE consists of representation, recombination, and fitness function.

3.2.1 Representation using Grammar

The representation encodes the test case as the chromosome. In the genotype-to-phenotype mapping process, the grammar defined for microcontrollers (see Figure 3) is employed, and the mapping is processed by the simulator of GETMic.

The genotype is represented as a list of tuples, each containing two integers (*codons*). The length of the genotype corresponds to the length of the user interaction to be generated by GE. Each tuple represents the target component for the interaction and the action to be performed.

$$genotype = [(x_0, y_0), .., (x_n, y_n)]$$
$$x, y \in integer\,[0, 2048]$$

The mapping of genotype to phenotype depends on the types and numbers of components set up in the simulator board. The first genotype, by default, does nothing. Given the first codon *x* and *n* interactable components, the target component is determined as follows:

$$target\,component = x\,mod\,(n + 1)$$

Similarly, given the second codon *y* and *m* actions, the action for the target component is determined as follows:

$$action = y\,mod\,m$$

Consider the following example genotype generated for `Button.py`.

$$[(231, 375), (347, 1478), (692, 1262)]$$

In the subject code `Button.py`, there is a single digital component (*button at pin 2*), to which *1* is mapped. Note that in every case, 0 is reserved for doing nothing. The possible actions for the button are *press* and *unpress*, to which each *0*, *1* is mapped. The decoding of a test case looks as follows:

$$231\,mod\,2 = 1 \rightarrow BTN\,at\,pin2$$
$$375\,mod\,2 = 1 \rightarrow unpress$$
$$347\,mod\,2 = 1 \rightarrow BTN\,at\,pin2$$
$$1478\,mod\,2 = 0 \rightarrow press$$
$$692\,mod\,2 = 0 \rightarrow do\,nothing$$

Therefore the phenotype of the generated test case is mapped as follows:

$$[unpress\, BTN\, at\, pin2, press\, BTN\, at\, pin2,$$

$$do\, nothing]$$

The decoding of analog components is done differently since analog components receive input values. For clarity, let's consider the genotype generated for `IfStatementConditional.py`:

$$[(1577, 1722), (165, 1060), (1990, 1658)]$$

In the code `IfStatementConditional.py`, there is a single analog component *(Potentiometer sensor at pin 0)*. The maximum value that the Potentiometer sensor can receive is configured to 1024; thus the modulus is 1024. The decoding of a test case looks as follows:

$$1577\, mod\, 2 = 1 \rightarrow Potentiometer\, at\, pin\, 0$$

$$1722\, mod\, 1024 = 698 \rightarrow set\, value\, 698$$

$$\ldots$$

The phenotype of the generated test case is mapped as follows:

$$[set\, value\, 698\, Potentiometer\, at\, pin\, 0,$$

$$set\, value\, 36\, Potentiometer\, at\, pin\, 0,$$

$$do\, nothing]$$

3.2.2 Recombination

Crossover and mutation were used as recombination. In crossover, the two chromosomes are mixed by randomly selecting the split index. Mutation occurs for each element of the chromosome list, which converts each element into a random interaction. Recombination occurs with a fixed probability.

3.2.3 Fitness function

General GA-based test case generation calculates fitness for each target branch and finds a test case that optimizes it. Therefore, optimization code must be executed once for each branch. However, in the case of GETMic, since the microcontroller loop code runs infinitely, one chromosome should cover all branches at once. Therefore, the fitness function is designed as follows.

$$Fitness = \sum_i max((approachLevel_i + branchDistance_i), 0)$$

Each run of the loop code updates the local fitness(= approach level + branch distance) of each branch to the minimum value. The fitness of the chromosome is the sum of local fitness for each branch. The minimum value of local fitness was set to 0 by the max operator so that local fitness was not affected by each other. It is optimized to minimize the fitness value, and if fitness is 0, it means that it is a 100% coverage test case in which all branches are covered.

4 Experimental Setup

4.1 Subjects

We set the target microcontroller as Arduino nano series. Table 2 lists the subject Arduino codes selected for generating test cases. The computation of lines of code and the count of branches is derived from the segments of code corresponding to the loop body in each Arduino code.

Within each subject code, various interaction components—including a button (*digital*), potentiometer sensor(*analog*), and temperature sensor (*analog*)-are utilized. Users interact with these components during the Arduino execution process.

`Button.py` (featuring simple button input), `IfStatementConditional.py` (simple analog input), `StateChangeDetection.py` (state changes through a button push counter), `SwitchCase.py` (analog input with multiple branches) , `LoveOMeter.py` (analog input with multiple branches), `SegmentDisplay.py` (ascending or descending counter based on the state of a button) are obtained from Arduino built-in examples.

To test on test cases with multiple interactable components, `WarmButton.py` (button component added to `LoveOMeter.py`) and `ComplexButton.py` (three buttons) are made by the authors.

Subjects	**Lines of Code**	**No. of Branches**	**Input Types and No.**
Button.py	7	2	digital 1
IfStatementConditional.py	8	2	analog 1
StateChangeDetection.py	17	5	digital 1
SegmentDisplay.py	14	4	digital 1
SwitchCase.py	11	4	analog 1
ComplexButton.py	27	8	digital 3
LoveOMeter.py	30	4	analog 1
WarmButton.py	33	8	digital 1, analog 1

Table 1: Subject Arduino Codes

Target Code	Avg. Branch Coverage(%)		No. reach to 100%		Avg. Trials		Avg. Execution time(sec)	
	GE	RG	GE	RG	GE	RG	GE	RG
Button.py	100.0	100.0	10	10	10.0	1.9	0.08	0.18
IfStatementConditional.py	100.0	100.0	10	10	10.0	1.5	0.12	0.16
StateChangeDetection.py	100.0	100.0	10	10	18.0	6.8	0.19	0.56
SegmentDisplay.py	100.0	100.0	10	10	12.0	6.8	0.23	0.74
SwitchCase.py	100.0	97.5	10	9	70.0	147.6	0.59	12.92
ComplexButton.py	98.8	91.3	9	3	381.0	824.8	6.83	110.85
LoveOMeter.py	97.5	80.0	9	2	486.0	848.4	9.13	110.04
WarmButton.py	95.0	83.3	7	0	709.0	1000.0	13.25	131.71

Table 2: Comparison of grammatical evolution (GE) and random generation (RG)

4.2 Implementation and Configuration

The GE and simulator are executed using Python run time version 3.11.5. In the GE, the population size is set to 10. Fitter parents are selected with a size of 0.2 relative to the population size. The stopping criterion is determined by either a fixed run of 100 generations or reaching zero fitness. The crossover rate is configured at 0.9, and the mutation rate is set to 0.1.

The length of user interaction, representing the interaction generated per test case, is dynamically configured based on the complexity of the target codes. Specific configuration of user interaction length is detailed in Table A. Notably, `LoveOMeter.py`, `WarmButton.py`, and `ComplexButton.py` have been configured to generate 10-12 user interactions per test case. This adjustment is made to accommodate the intricate branch conditions and the substantial number of branches present in these particular code segments.

4.3 Evaluation

The test case generation was iterated 10 times with a different random seed number to account for its stochastic nature. To evaluate the performance of the implemented GE, we established a baseline using randomly generated test cases configured with the same user interaction length as employed by the GE. `Coverage.py` version 7.3.2 is used to measure the coverage of generated test cases. Coverage measurement is exclusively done on the loop body, which is the main section of the Arduino code of interest in testing. In addition, computational expense is evaluated by measuring the execution time.

5 Result and Analysis

5.1 Result

Table 2 shows the test case generation result of our GE engine and random generation (RG). For all the cases, our GE performed better than random generation in terms of both coverage and the execution time. GE ensured branch coverage higher than 95% for every case while random generation showed the 80% range for some of the cases. Also, GE has reached 21 more 100% coverage than random generation out of 80 runs.

5.2 Analysis

5.2.1 Digital vs. Analog

We can observe that as the target code becomes more complex, GE completely outperforms random generation. Here, the complexity is mostly dominated by the types and the number of components included, especially the types. This is because when analog components are added, they extremely broaden the search space compared to digital ones. If a target code uses multiple analog components, GE engine will become incomparable to random generation.

5.2.2 Execution time

From a perspective of execution time, GE approach was observed to be 1.3x to 24.8x faster than random generation. This can be attributed to two factors.

One is that the GE engine itself is more efficient at finding solutions as we can observe from the average number of trials required to find the solution. Except for the first 4 targets in Table 2, which are simple enough for them to be solved within around 1 generation, GE took 0.46x to 0.71x less trial than random generation. These results are actually rather understated. If we use a larger budget or no limit at all, the difference between them will be more extreme, and therefore the difference in execution time will be very large(especially for `WarmButton.py`).

However, even in the case where random generation has fewer trials, we can see that the execution time is actually smaller in GE. This is due to the second factor related to how we find the solution. In GE, we find the best solution using the fitness function and calculate the coverage only once for that best solution. On the other hand, in a random generation, coverage is calculated for every single trial. Since fitness calculation is a much cheaper operation than coverage calculation, GE is inevitably faster than random generation. These two factors combine to make GE dominate random generation in terms of execution time.

6 Conclusion

We found difficulties in test case generation for microcontrollers which is the need for hardware consideration and the representation of the test case. To tackle these challenges, we employed two strategies respectively: a simulator and grammatical evolution. Combining two approaches, we were able to successfully define user interactions, simulate the execution of a real microcontroller program, and discover test cases that maximize branch coverage through a genetic algorithm. The comparison with random generation confirmed the superiority of the GE approach in terms of both coverage performance and time efficiency.

References

Micropython.

Adina Deiner, Christoph Frädrich, Gordon Fraser, Sophia Geserer, and Niklas Zantner. 2020. Search-based testing for scratch programs. In *Search-Based Software Engineering: 12th International Symposium, SSBSE 2020, Bari, Italy, October 7–8, 2020, Proceedings 12*, pages 58–72. Springer.

Muhammad Iqbal Hossain and Woo Jin Lee. 2019. Integration testing based on indirect interaction for embedded system. *International Journal of Reconfigurable and Embedded Systems*, 8(2):86.

Altaf Hussain, Muhammad Hammad, Kamran Hafeez, and Tabinda Zainab. 2016. Programming a microcontroller. *International Journal of Computer Applications*, 155(5):21–26.

Trieu Nguyen, Sune Zoëga Andreasen, Anders Wolff, and Dang Duong Bang. 2018. From lab on a chip to point of care devices: The role of open source microcontrollers. *Micromachines*, 9(8):403.

M. O'Neill and C. Ryan. 2001. Grammatical evolution. *IEEE Transactions on Evolutionary Computation*, 5(4):349–358.

Mani Padmanabhan. 2022. Test case generation for arduino programming instructions using functional block diagrams. *Trends in Sciences*, 19(8):3472–3472.

Anirban Saha, Raju Udava, Mallikarjun Bidari, Mahadeva Prasad, Venkata Raju, and Tushar Vrind. 2021. Trafic—a systematic low overhead code coverage tool for embedded systems. In *2021 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)*, pages 1–6. IEEE.

A Appendix

Target Code	Interaction Length	Best Test case
Button.py	3	press BTN at pin2, press BTN at pin2, None do_nothing
IfStatementConditional.py	3	set_value_698 Potentiometer sensor at pin0, set_value_36 Potentiometer sensor at pin0, None do_nothing
StateChangeDetection.py	3	press BTN at pin2, unpress BTN at pin2, None do_nothing
SegmentDisplay.py	3	press BTN at pin13, unpress BTN at pin13, None do_nothing
SwitchCase.py	3	set_value_308 Photoresistor sensor at pin0, set_value_412 Photoresistor sensor at pin0, set_value_754 Photoresistor sensor at pin0
ComplexButton.py	10	press BTN at pin12, press BTN at pin10, unpress BTN at pin12, press BTN at pin11, press BTN at pin11, press BTN at pin12, None do_nothing, unpress BTN at pin10, unpress BTN at pin12, unpress BTN at pin11
LoveOMeter.py	12	set_value_653 Temperature sensor at pin0, None do_nothing, set_value_153 Temperature sensor at pin0, set_value_654 Temperature sensor at pin0, None do_nothing, set_value_775 Temperature sensor at pin0, set_value_509 Temperature sensor at pin0, set_value_149 Temperature sensor at pin0, set_value_657 Temperature sensor at pin0, set_value_690 Temperature sensor at pin0, set_value_154 Temperature sensor at pin0, set_value_312 Temperature sensor at pin0
WarmButton.py	12	unpress BTN at pin1, press BTN at pin1, set_value_418 Temperature sensor at pin0, set_value_154 Temperature sensor at pin0, None do_nothing, press BTN at pin1, press BTN at pin1, set_value_149 Temperature sensor at pin0, unpress BTN at pin1, set_value_180 Temperature sensor at pin0, set_value_987 Temperature sensor at pin0, None do_nothing

Table 3: GETMic-generated test case for each target code