Mutation Testing for Physical Computing

Qianqian Zhu
Delft University of Technology
Email: qianqian.zhu@tudelft.nl

Andy Zaidman
Delft University of Technology
Email: a.e.zaidman@tudelft.nl

Abstract—**Physical computing, which builds interactive systems between the physical world and computers, has been widely used in a wide variety of domains and applications, e.g., the Internet of Things (IoT). Although physical computing has witnessed enormous realisations, testing these physical computing systems still face many challenges, such as potential circuit related bugs which are not part of the software problems, the timing issue which decreasing the testability, etc.; therefore, we proposed a mutation testing approach for physical computing systems to enable engineers to judge the quality of their tests in a more accurate way. The main focus is the communication between the software and peripherals. More particular, we first defined a set of mutation operators based on the common communication errors between the software and peripherals that could happen in the software. We conducted a preliminary experiment on nine physical computing projects based on the Raspberry Pi and Arduino platforms. The results show that our mutation testing method can assess the test suite quality effectively in terms of weakness and inadequacy.**

I. Introduction

Physical computing creates a conversation between the physical world and the virtual world of the computer [1]. The recent confluence of embedded and real-time systems with wireless, sensor, and networking technologies is creating a nascent infrastructure for an educational, technical, economic, and social revolution. Fuelled by the recent adoption of a variety of enabling wireless technologies such as RFID tags, embedded sensor and actuator nodes, the Internet of Things (IoT) has stepped out of its infancy and is rapidly advancing in terms of technology, functionality, and size, with more real-time applications [2]. A good example of the IoT is wearable devices like fitness trackers that are ever getting more popular.

Modern embedded platforms, like those centred around the 8051 and Freescale micro-controller series, have seen a dramatic rise in speed and functionality. The Raspberry Pi and Arduino platforms, which were originally meant for education, are two of the most popular modern embedded platforms. They are both open-source electronics platforms based on easy-to-use hardware and software.

An equally important trend is *softwarization of hardware*. In the early days, hardware engineers had to build circuits by physically connecting electronic components using wire and soldering. More recently, *reconfigurable computing* tools provide the opportunity to compile programs written in high-level languages such as C and Java into a hardware architecture. A Raspberry Pi supports several programming languages including Python to control the General Purpose Input/output (GPIO) pins to communicate with the external devices. This means that developing a physical computing system has been simplified to the point where the hardware peripherals can easily be controlled via software without even knowing the hardware part. This trend also provides a great opportunity for applying methodologies of software engineering in physical computing, especially testing techniques.

As physical computing is maturing, testing these sensor-based applications, especially the processing programs, becomes essential. Essential, because compared to conventional software projects, the costs associated with failing physical computing systems are often even bigger, as bugs can result in real-life accidents. For example, a robotic arm might accidentally hurt the human if the programmer does not set up the initial state properly. Therefore, to develop a rigorous and sound physical computing system, a high-quality test suite becomes crucial. This brings us to *mutation testing*, a fault-based testing technique that assesses the test suite quality by systematically introducing small artificial faults [3]. It has been shown to perform well in exposing faults [4]–[6].

In this paper, we propose a novel mutation testing approach for physical computing systems enabling engineers to judge the quality of their tests in an accurate way. Specifically, we define a set of mutation operators based on *common mistakes* that we observed when developing physical computing systems. We present an initial evaluation of our approach on the Raspberry Pi and Arduino platforms.

II. Background and motivation

We introduce basic concepts related to *physical computing* and *mutation testing*. We then motivate why mutation testing should be applied to physical computing systems.

A. Physical computing

Most physical computing systems (and most computer applications in general) can be broken down into the following same three stages: input, processing, and output [1]. The input is about how computers sense the physical world via sensors and signals, such as buttons and speakers. While the output is where computers make changes to the world under people's desire through various actuators, like servos, motors and LEDs. The processing procedure requires a computer (usually an embedded platform) to read the inputs and turn them into outputs.

The **General Purpose Input/output (GPIO)** is the primary interface that micro-controllers including Raspberry Pi use to communicate with external devices. The pins available on a

processor can be programmed to be used to either accept input or provide output to external devices depending on user desires and application requirements. These pins support a variety of data handling methods, such as Analog-to-Digital conversion and interrupt handling. GPIO is also the main focus of our methodology.

Among different embedded system platforms, the ***Raspberry Pi*** is a popular one-chip computer which includes an ARM-compatible CPU, a GPU and a Secure Digital (SD) card module. Its recommended operating system for normal use is Raspbian, a free, Debian-based operating system optimised for the platform. Python is the recommended programming language and the *RPi.GPIO* library is used to configure GPIO pins.

Another popular micro-controller is ***Arduino*** which is also open-source and easy-to-use. Arduino boards support GPIO pins as well. The Arduino Software (IDE) runs on the Windows, MacOS, and Linux operating systems. Its programming language can be expanded through C++ libraries, and users wanting to understand the technical details can make the leap from Arduino to the AVR C programming language on which it is based. Similarly, users can also add AVR-C code directly into Arduino programs.

B. Mutation Testing

Mutation testing is defined by Jia and Harman [3] as a fault-based testing technique which provides a testing criterion called the *mutation adequacy score*. This score can be used to measure the effectiveness of a test set regarding its ability to detect faults [3]. The principle of mutation testing is to introduce syntactic changes into the original program to generate faulty versions (called *mutants*) according to well-defined rules (mutation operators) [7]. The benefits of mutation testing have been extensively investigated and can be summarised as [8]: 1) having better fault exposing capability compared to other test coverage criteria [4]–[6], 2) being an excellent alternative to real faults and providing a good indication of the fault detection ability of a test suite [9].

C. Characteristics of Physical Computing

Physical computing allows to build interactive physical systems through a combination of hardware and software. The following six major characteristics describe the uniqueness of physical computing [10]:

(1) Safety and security issue: physical computing systems are much more safety-critical than traditional software where small defects could have a tremendous impact on the reliability of systems upon which people's lives and living depend. Moreover, sensor networks interact closely with their physical environment and with people, posing additional security problems.

(2) Fault-tolerance: fault-tolerance is a crucial requirement for physical computing systems that manage to handle exceptions properly once a certain part does not work. E.g., sensors may fail due to surrounding physical conditions or when their energy runs out. It may be difficult to replace existing sensors;

the network must be fault-tolerant such that non-catastrophic failures are hidden from the application [11].

(3) Lack of knowledge: physical computing is a multi-disciplinary domain which requires developers to create high-level software system as well as low-level embedded systems solutions. However, most embedded systems developers have an electrical engineering background, therefore, might lack basic knowledge of software engineering, especially testing techniques; which could lead to error-prone code and low-quality tests.

(4) Circuit related bugs: this type of errors is mostly due to hardware configuration, such as shorts circuits, errors in sensors, undefined states (not pulling up resistors for the input processing); these bugs could be prevented or localised by testing each component at the unit level.

(5) Timing issue: in most cases, peripherals are activated or deactivated at a particular time, e.g., systems embedded with sonic sensors only start working when the distance meets a specific condition. The timing issue decreases testability of physical computing systems as it is hard to set up the real scenarios for testing.

(6) Slow execution speed: although there is a dramatic improvement in the power and functionality of modern embedded platforms, the execution speed of these embedded platforms is still not as comparable as PCs and servers. Thereby, the processing program must be carefully designed to avoid computationally-consuming algorithms.

Motivation. We can see that physical computing systems require extremely error-free and reliable code considering the *safety and security issue* and *fault-tolerance*. The *slow execution speed* of embedded platforms also demands a well-designed and cost-effective processing program to be deployed ubiquitously. Moreover, the testing procedure is of utmost importance to implement high-quality and error-free programs, as well as detect *circuit related bugs,* and make up developers' *lack of knowledge* of software engineering. Also, a weak test suite is not sufficient enough to detect the faults and cannot correctly handle the *timing issue*.

Taking all the characteristics of physical computing systems together, the primary challenge for physical computing systems here is: *how to effectively and efficiently test these physical systems?* To deal with this challenge, we are seeking to apply software engineering methodologies to the physical computing domain. In particular, mutation testing, which is well-known for its high fault-revealing effectiveness, is a viable way to help developers design better quality test suites in this highly safety critical domain. Also, mutation testing, as a fault injection technique, is an ideal method for testing the fault tolerance mechanisms with respect to a specific set of inputs the physical computing systems are meant to cope with [12].

III. DESIGNING MUTATION OPERATORS

To integrate computing with the physical world via sensors and actuators, an essential component is an interface between the software (processing programs) and peripherals (sensors

and actuators). The proliferation of sensor and actuator networks in (civilian) applications requires new approaches to handle real-time, multimedia and multi-threaded communications, such as wireless sensor network [11] and cloud computing [13]. This leads to a more complex and error-prone integration part. Therefore, when designing a mutation operator for physical computing, our main goal is to narrow down the scope of the mutation process to parts of the code that affect the communication between the software and peripherals (digital circuits[1]), namely the GPIO interface. To derive the mutation operators that represents errors typically made by programmers during the implementation of the software, we first summarise *common mistakes* that could happen in the software based on our experience. Subsequently, we design a set of mutation operators for these common mistakes.

(1) output value errors: The output value is usually decided by a complex function which takes many elements such as feedback from the sensors and preferences of the user, into consideration. For example, an automatic watering system decides when to water the plants according to multiple environmental conditions, e.g., soil humidity and the amount of water configured by the user. Thus, the output could be wrong if there exists a bug in the function. More specifically, we only pay attention to the final output value generated by the function, i.e., whether the output value is high or low, regardless of the function details. For this type of the error, we derived the *OutputValueReplacement (OVR)* operator which replaces *HIGH* to *LOW* (and vice versa) in the output value.

(2) output setting omissions: Once a certain signal has been received/read by a peripheral, the output value should, in some cases, be reset to ensure that the peripheral can change states at a later stage. For example, a self-driving car should reduce engine output when detecting a wall, but the engine should engage again after clearing the wall. Accordingly, we designed the *OutputSettingRemoval (OSR)* operator which deletes the output setting function.

(3) pin number errors: The programmer may read information or send control signals using a wrong pin that she does not intend to operate. The problem typically arises during prototyping for two reasons: 1) the GPIO pins are usually on the PCB as a symmetric array without labels so that designers need to locate a pin by counting, and 2) the order of a pin on the PCB is typically different from its numerical ID in the software API, making the mapping error-prone. *PinNumberReplacement (PNR)* replaces the pin id with one of the surrounding pin ids.

(4) input value errors: There are usually two ways to obtain an input value. The simplest way is to check the input value at a point in time. This "polling" can potentially miss an input if the program reads the value at the wrong time. The other way of responding to a GPIO input is using edge detection. An edge is the name of a transition from HIGH to LOW (falling edge) or LOW to HIGH (rising edge). Quite often, we are more

---

[1]In this paper, we focus on the digital circuits, where two possible states, i.e., *HIGH* and *LOW*, are considered. As this is the fundamental circuit type compared to *analog*.

TABLE I
SUMMARY OF MUTATION OPERATORS

Mutation operator	Full name	Definition
OVR	Output Value Replacement	replace *HIGH* to *LOW* (and vice versa) in the output value
OSR	Output Setting Removal	delete the output setting function
PNR	Pin Number Replacement	replace the pin id with its surrounded pin ids
IVR	Input Value Replacement	replace *HIGH* to *LOW* (and vice versa) in the input value
EDR	Edge Detection Replacement	replace edge names among {*FALLING*, *RISING*, *BOTH*}
IOMR	I/O Mode Replacement	replace *IN* to *OUT* (and vice versa) in the mode setting
SIR	Setup Input Replacement	replace the input value from *PUD_UP* to *PUD_DOWN* (and vice versa) in setup function
SOR	Setup Output Replacement	replace the output value from *HIGH* to *LOW* (and vice versa) in setup function
SVR	Setup Value Removal	remove the initial value setting in setup function for both input and output modes

concerned by a change in state of an input than its value. One potential fault in the edge detection is to mix up the falling and rising edge. This problem is common due to the confusion brought by the variety of external devices, e.g., for the 7400 series logic chip, for instance, the 74LS107 JK flip-flop chip [14] triggers on a rising edge, while the 74HC74 D flip-flop chip [15] triggers on a falling edge. For input value mistakes, we defined the following two mutation operators:

- *InputValueReplacement (IVR)*: replaces *HIGH* to *LOW* (and vice versa) in the input value
- *EdgeDetectionReplacement (EDR)*: replaces *FALLING* to *RISING* (and vice versa) in the edge detection. However, sometimes, there is one more edge event called *BOTH* which covers both the falling and rising edge. In this case, the replacement happens among the three edge events, e.g., replace *FALLING* to *RISING* and *BOTH*.

(5) I/O pin mode errors: A GPIO pin allows to define each individual pin on the chip as being in *input* or *output* mode. As a side-effect of pin number mistakes, the programmer might set the pin I/O mode by mistake. Thus we designed the *I/OModeReplacement (IOMR)* operator that changes *IN* to *OUT* (or vice versa).

(6) initial setup value errors: If a pin is not "connected" to a peripheral, it will "float". In other words, the value that is read in is undefined because it is not connected to anything. It could frequently change values as a result of receiving mains interference. To get around this, GPIO modules usually provide an option to use a pull-up (*PUD_UP*) or pull-down (*PUD_DOWN*) resistor to set the default value of the input. Two potential errors in this context are (1) the omission of setting up the input value or (2) initializing it with the opposite value by mistake. Similarly, for the output mode, pins can have different default output values in a single GPIO module. The initial output value affects the initial state of the peripheral that the pin is connected to, which could lead to a breakdown or unexpected activation. For instance, if the pin connected to a motor is initially set to *HIGH*, then once the module is activated, the motor is immediately activated which is supposed to be activated when the switch is on. The potential errors in output value setup are similar to the input value

setup, i.e., the setup omission and the initial value mistakes. Accordingly, there are three mutation operators:

- *SetupInputReplacement (SIR)*: replace the input value from *PUD_UP* to *PUD_DOWN* (and vice versa) in setup function.
- *SetupOutputReplacement (SOR)*: replace the output value from *HIGH* to *LOW* (and vice versa) in setup function.
- *SetupValueRemoval (SVR)*: remove the initial value in setup function for both input and output modes.

Summary: We designed nine mutation operators (summarised in Table I) to replicate common communication errors in physical computing systems.

IV. TOOL IMPLEMENTATION

Various modern embedded platforms contain the GPIO module, such as Arduino, BeagleBone, PSoC kits and Raspberry Pi. In this paper, we chose Raspberry Pi and Arduino as the target platforms to implement the aforementioned mutation operators. One thing to note is that our approach should work with the other aforementioned platforms as well.

We have coined our mutation tool MUTPHY and implemented it in Python. The overall architecture of MUTPHY is shown in Figure 1. MUTPHY consists of two components, i.e., the mutation engine and the test executor. MUTPHY takes the program and its test suite as input. First, the mutation engine analyses the source code and marks all possible mutation points, and then the mutation generator produces all the mutants according to mutation operators. After that, the program and generated mutants together with the test suite go to the test executor where the mutation testing is performed: each mutant is executed against the test suite one by one. Finally, MUTPHY prints out the detailed mutant killable results. The main task of the code analyser is to analyse the test dependencies and parse the source code of the program for the mutation generator. The mutation generator contains all the mutation operators and the details of the mutants including the mutation location (line number) and the mutation operator type.

As Raspberry Pi and Arduino are the target platforms, we have created two variants of MUTPHY. The main differences between the two variants are inherent to the programming languages that are supported by two platforms. For Raspberry Pi, the code analyser of MUTPHY needs to parse Python, as Python is Raspberry Pi's recommended programming language. As Arduino only supports C/C++, we created a C/C++ code analyser in MUTPHY for Arduino. Moreover, we considered *pytest* [16], a non-boilerplate alternative to Python's standard *unittest* testing framework [17], as the test executor for both the Raspberry Pi and Arduino platforms , as it can also handle other popular Python testing libraries, e.g. *unittest* and *doctests* [18].

V. EMPIRICAL EVALUATION

To assess the efficacy of our mutation testing approach, we conducted an experimental study using two embedded system platforms, i.e. Raspberry Pi and Arduino. We proposed the following research questions to steer our experimental study:
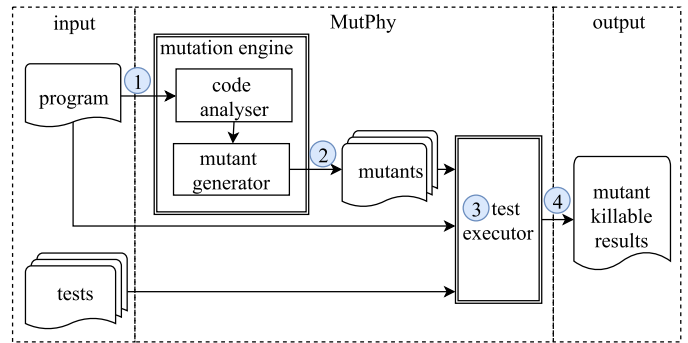

Fig. 1. Overview of MUTPHY architecture and workflow

- **RQ1:** *How effective is* MUTPHY *in evaluating the existing test suite?* With this research question, we evaluate to what extent MUTPHY can effectively evaluate the quality of the existing test suite.
- **RQ2:** *How efficient is* MUTPHY *in generating non-equivalent mutants?* As we designed the mutation operators based on common mistakes made by programmers, this might lead to potential *redundant* mutation operators which are subsumed by others. **RQ2** addresses the efficiency of MUTPHY in generating non-equivalent mutants.
- **RQ3:** *Is it possible to kill all non-equivalent surviving mutants by adding extra test cases?* This research question focuses on non-equivalent surviving mutants and aims to assess whether our approach enables engineers to write a better test suite.

For **RQ1**, we determined the effectiveness of our approach based on the number of non-equivalent surviving mutants. Also, we compared our results to test coverage. To answer **RQ2**, we manually analysed the generated mutants to determine whether the mutant is equivalent to the original program. For **RQ3**, we analysed the non-equivalent surviving mutants in detail and tried to manually engineer new test cases to kill these mutants.

A. Case Studies with Raspberry Pi

In the first part of the experiment, we use five Raspberry Pi based projects for evaluating MUTPHY. For these five projects, four are obtained from GitHub, and one is from industry (Guangzhou Kompline Electronics). The four open source projects have been manually selected from GitHub under the Raspberry Pi topic using the following process: we (1) sorted by stars (from high to low), (2) checked whether they contain "GPIO" as a keyword, (3) verified that they are implemented in Python, and (4) examined whether they can be successfully built, and (5) inspected whether they contain a test suite. Since our main focus is the GPIO interface, we only apply mutation operators on the files that use the GPIO library. Table II summarises the main characteristics of the selected projects.

When answering the **RQ**s in the next sections, we will start with **RQ2**, as we need to analyse non-equivalent mutants to calculate the mutation score which is part of **RQ1**.

TABLE II
SUBJECTS BASED ON RASPBERRY PI

Project	File	LOC^2	#Tests	Coverage^3
RPLCD	gpio.py	99	35	71%
hcsr04sensor	sensor.py	93	6	96%
jean-pierre	buzzer.py	20	21	41%
gpiozero	mock.py	312	302	97%
four-wheel robot	arm.py	179	11	93%
	chassis.py	158	4	100%
Total		861	379	82.8%

1) RPLCD: The project *RPLCD* [19] is a Python 2/3 Raspberry PI Character LCD library for the Hitachi HD44780 [20] controller. The main peripheral of this system is a LCD module.

TABLE III
MUTANTS RESULT OF *RPLCD*

MOP	#Generated	#Covered	#Alive	#Killed	#Equiv.	MS
OVR	13	10	13	0	0	0
OSR	11	10	11	0	1	0
PNR	47	41	47	0	0	0
IVR	0	0	0	0	0	-
EDR	0	0	0	0	0	-
IOMR	1	1	1	0	0	0
SIR	0	0	0	0	0	-
SOR	0	0	0	0	0	-
SVR	0	0	0	0	0	-
Overall	72	62	72	0	1	0

Using MUTPHY, we generated 72 mutants for the *RPLCD* project. This project mainly uses the *GPIO.output* method to write data to the LCD board, thus, only four types of mutation operators can be applied to the system: OVR, OSR, PNR and IOMR. The details of all generated mutants are presented in Table III. We can see from Table III that only one equivalent mutant is generated by MUTPHY. This equivalent OSR mutant is located in a statement that, under the existing test configuration, cannot be reached. Thus, for this LCD controlling system, the efficiency of MUTPHY in generating non-equivalent mutants is promising (**RQ2**).

While the statement coverage is 71%, the mutation score is zero. Furthermore, 86.1% of mutants are covered by the test suite, but none of the mutants is actually killed. Why then is the mutation score of this project so low? We found that the developers replaced the *RPi.GPIO* module of the system under test with mock objects; this allows the tests to be executed without a Raspberry Pi. As a side effect, the developers did not assess the communication between the software and peripherals for this system. The above findings indicate that compared to statement coverage, the mutation score can better represent how a test suite examines the behaviour of GPIO pins (**RQ1**).

To kill the mutants (**RQ3**), we first removed the mock objects for the *RPi.GPIO* module and executed the test suite

^{2}The line of code (LOC) is measured by sloccount [21].

^{3}The test coverage is here is statement coverage measured by Coverage.py [22].

on an actual Raspberry Pi. This modification led to 21 PNR and 1 IOMR mutants killed. Then, we analysed whether the remaining mutated statements are covered by the tests or not. As shown in Table III, we found 85.2% non-equivalent mutants to be covered by the test suite. However, the existing test suite only calls the functions in *gpio.py* file, but does not check the behaviour of the GPIO pins. To address this drawback of the existing test suites, we added five test cases to examine all the pins once their states changed. To capture the state change sequence of GPIO pins, we introduced new mock objects. Different from the system developers' mock objects, we used mock objects to increase the observability of the system under test. For instance, one method in *gpio.py* file called *pulse_enable()*, that sends a pulse signal to tell the LCD board to process the data. The method *pulse_enable()* calls *GPIO.output* three times in one pin generating a LOW-HIGH-LOW signal. Without a mock object of method *GPIO.output*, it is hard to tell what happens to this pin after this function call, as the starting and the ending states are both LOW. With the additional five test cases, all the non-equivalent mutants are killed.

2) hcsr04sensor: The *hcsr04sensor* project [23] is a Python module for measuring distance and depth with a Raspberry Pi and HC-SR04 Ultrasonic Module [20], which uses sonar to determine the distance to an object, just like bats or dolphins do. The sensor first emits ultrasound at 40,000 Hz, which travels through the air and if there is an object or obstacle on its path, the ultrasound will bounce back to the module. Considering the travel time and the speed of the sound, it calculates the distance. The HC-SR04 Ultrasonic Module has 4 pins: Ground, VCC, Trig and Echo.

TABLE IV
MUTANTS RESULT OF *hcsr04sensor*

MOP	#Generated	#Covered	#Alive	#Killed	#Equiv.	MS
OVR	3	3	1	2	0	0.67
OSR	3	3	1	2	0	0.67
PNR	31	31	7	24	0	0.77
IVR	2	2	0	2	0	1
EDR	0	0	0	0	0	-
IOMR	2	2	0	2	0	1
SIR	0	0	0	0	0	-
SOR	0	0	0	0	0	-
SVR	0	0	0	0	0	-
Overall	41	41	9	32	0	0.78

Table IV details the generated mutants for *hcsr04sensor*. In total, MUTPHY generated 41 mutants. For this system, the Raspberry Pi controls the HC-SR04 Ultrasonic Module by writing to the Trig pin and reading from Echo. As such, this control program mainly adopts *GPIO.output* and *GPIO.input* methods. This results in five types of mutants from OVR, OSR, PNR, IVR and IOMR operators. There are no equivalent mutants generated by our proposed mutation operators; this indicates MUTPHY has high efficiency in generating non-equivalent mutants (**RQ2**).

For **RQ1**, although 100% of the mutants are covered, 22% of the mutants are *not* detected by the test suite. Looking at

the existing test suite, we found that the test suite checked all the initial settings of each GPIO pins, but lacks tests to (1) examine the pins' state changes during the execution and (2) the final states after tearing down. For this project, it is important to clean up the Trig and Echo pins after use, because otherwise the distance cannot be accurately calculated by a new request to the ultrasonic sensor. This gives another indication that mutation score is a better metric of test suite quality than statement coverage, which only reveals insufficient tests for the system.

Regarding **RQ3**, we observe seven PNR mutants that are still alive; all originating from the *GPIO.cleanup* function. To kill these mutants, we need to add two additional assertions at the point just after the pins are torn down which means the pins are not used anymore. Once the pins are torn down, they cannot be read from or written to anymore, so the assertions expect exceptions when trying to read those pins.

The other two alive mutants, one of type OVR and one other of type OSR, are located on the same line, more precisely when calling the *GPIO.output* function. Similar to the *pulse_enable()* method in project *RPLCD*, this *GPIO.output* function is meant to send a LOW value, the first stage of the pulse signal. We follow a similar strategy in that we try to introduce mock objects to increase the observability, but this modification led to a syntax error: a local variable *sonar_signal_on* is referenced before assignment. Through further investigation, we found that this local variable is only assigned right after the Echo pin detects a HIGH signal via the *GPIO.input* function, while in the situation with mocks, the *GPIO.input* function is not actually invoked. This leaves us in the situation that if we do not introduce mock objects, the state change of this *GPIO.output* function cannot be observed, while if we do introduce mock objects, there is a syntax error.

The aforementioned observation is a case of a *snarled method*, a term coined by Feathers to describe a method dominated by a single large, indented section [24]. Feathers suggest to perform an *extract method* refactoring to move all the statements related to the pulse signal into a separate method [24]. In doing so, we create a function *pulse_enable()* and we separate responsibilities of this *snarled method*. As a result, we can easily test the state change caused by the target *GPIO.output* function without affecting the remaining part. For these two mutants, it is hard to derive new tests to kill them without refactoring the original production code. Through refactoring, the statement where the mutants are located is moved from a long method to a short one, thus, improving the observability of the state change made by the statement. This raises an interesting speculation: the testability of the production code [25] could have an influence on the test suite's mutation score. In Voas et al.'s work [26], they proposed that software testability could be defined for different types of testing, such as data-flow testing and mutation testing. Their work inspires us to explore the relationship of software testability and mutation testing in the future work.

3) jean-pierre: The project *jean-pierre* [27] is a little DIY robot based on the Raspberry Pi Zero W [28]. It uses a camera

to scan food barcodes: it fetches information about the product from the OpenFoodFacts API [29] and adds it to a grocery list that the user can manage from a web interface. Once an object is successfully added to the grocery list, a buzzer makes two beeps. This system consists of three components: a Raspberry Pi Zero W, a Raspberry Pi Camera Module [30] and a buzzer. The main use of the GPIO pins in this project is to control the buzzer (*buzzer.py* file).

TABLE V
MUTANTS RESULT OF *jean-pierre*

MOP	#Generated	#Covered	#Alive	#Killed	#Equiv.	MS
OVR	2	0	2	0	0	0
OSR	2	0	2	0	0	0
PNR	6	0	6	0	0	0
IVR	0	0	0	0	0	-
EDR	0	0	0	0	0	-
IOMR	1	0	1	0	0	0
SIR	0	0	0	0	0	-
SOR	0	0	0	0	0	-
SVR	0	0	0	0	0	-
Overall	11	0	11	0	0	0

As the buzzer only has one function, i.e., *beep()*, it mainly adopts the *GPIO.output* function. When running our tool, 11 mutants are generated (shown in Table V). For **RQ2**, no equivalent mutant is generated, which shows MUTPHY's high efficiency in generating non-equivalent mutants. For **RQ1**, we can see that the mutation score is 0 while the statement coverage is 41%. Although the statement coverage is 41%, none of the generated mutants is covered by the test suite. Closer inspection revealed that there are no tests in the existing test suite that are specifically designed to test the communication of the software and the buzzer. We can see that the mutation score enables to evaluate how the test suite examines the integration part of the software and peripherals in physical computing systems, while the test coverage cannot.

To kill the mutants (**RQ3**), we first added a test case to cover the mutants without assertions. Once the mutated statements are covered, i.e., the statement coverage reaches 100%, the six alive PNR and one alive IOMR mutants are killed. These seven mutants can easily be detected once the mutated GPIO pins are invoked, because the *RPi.GPIO* module throws exceptions if these pins are either not initialised or initialised incorrectly. For instance, GPIO8 pin is called without initialisation, or GPIO9 pin is written to HIGH after being initialised to input mode. Then, to kill the remaining four surviving mutants, we again introduced mock objects to assess each state change made by the *GPIO.output* function. By designing effective test oracles to test the state change of the GPIO pins using mock objects, all the mutants are killed.

4) gpiozero: The project *gpiozero* [31] is a simple interface to GPIO devices with Raspberry Pi, which requires minimal boilerplate code to get started. This project is developed by the Raspberry Pi Foundation. This library provides many simple and obvious interfaces for the essential components, such as LED, Button, Buzzer, sensors, motors and even a few simple add-on boards.

MOP	#Generated	#Covered	#Alive	#Killed	#Equiv.	MS
OVR	2	2	0	2	0	1
OSR	1	1	0	1	0	1
PNR	68	68	0	68	0	1
IVR	6	6	0	6	0	1
EDR	19	19	0	19	0	1
IOMR	14	14	0	14	0	1
SIR	8	8	0	8	0	1
SOR	2	2	0	2	0	1
SVR	5	5	1	4	1	1
Overall	125	125	1	124	1	1

Table VI shows the 125 mutants generated by MUTPHY. For **RQ2**, there is only one equivalent mutant generated by MUTPHY. This equivalent mutant of type SVR stems from the initial value being removed from the setup function, yet with the default output value being the same as the initial value, there is an equivalence. For **RQ1**, the mutation score of this project is 1, which shows the existing test suite is adequate to detect all the mutants. One necessary condition for such a high mutation score is high test coverage. We can see that the statement coverage of the existing test suite is 97% and all the mutated statements are covered by the test suite. Moreover, there are 302 test cases in the existing test suite. Looking at the tests in detail, we found that each test case not only examines the basic information of the pin under test, i.e., the pin number and the pin state, but also other possible settings of the pin, e.g., I/O pin mode and resistor state. As the mutation score of this project has already achieved 1, there is no need for us to add extra tests to enhance the test quality (**RQ3**). From project *gpiozero*, we can conclude that the test suite can indeed achieve 100% mutation score when the GPIO pins are taken into consideration in tests and test oracles are carefully designed.

5) four-wheel robot: This subject is a four-wheel robot, which has been designed and developed for industrial use (as shown in Figure 2). The robot is capable of moving pie-shaped objects from one place to another. During the movement, the robot may optionally rotate the object by at most 2π rad, and the four wheels can move it in six directions (as presented in Figure 3). The robot includes one Raspberry Pi 2, five photoelectric sensors, two DC motors, four stepper motors and three servos. The photoelectric sensors are mainly used to align the robot in specific positions (e.g., the starting point and the destination) based on differently coloured regions. The four stepper motors are responsible for the movement of the four wheels. As for the two DC motors, one drives the vertical movement of the robotic arm; the other is for the rotation of the arm. The three servos are used to control the action of the claw to grab the pie-shaped objects. The control system of the robot consists of two parts, the chassis (*chassis.py*) and the arm (*arm.py*). The chassis part has 13 functions, and the arm part consists of 13 functions. The entire system's footprint comprises 337 lines of code. To set up a safe environment for testing, there is one test track with black and white lines
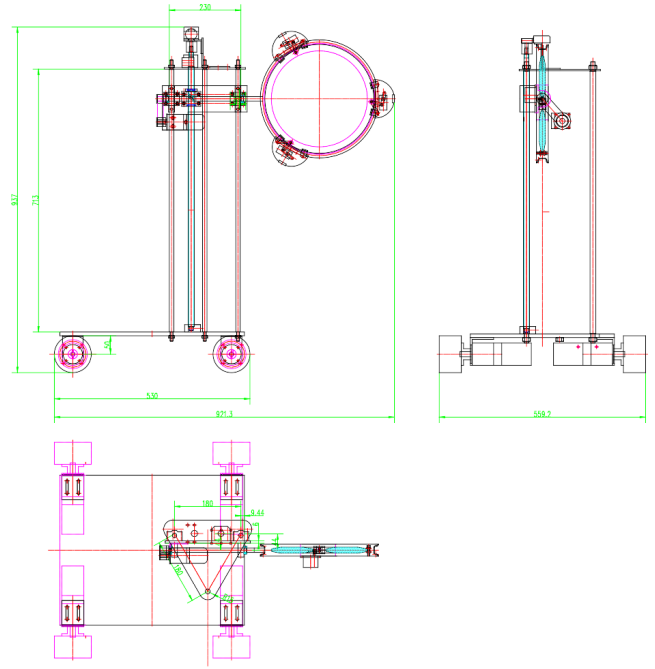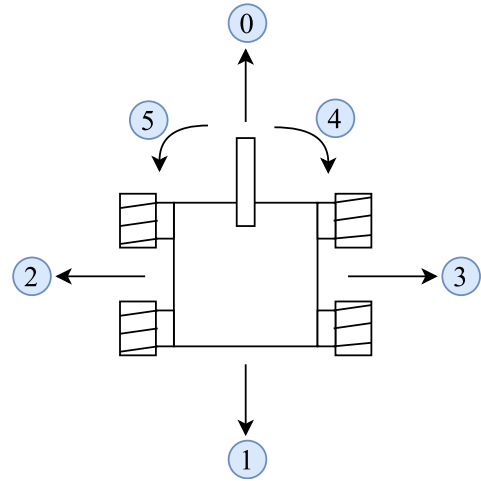

Fig. 2. Three-view diagrams of four-wheel robot

Fig. 3. Movement directions of four-wheel robot

designed for the robots. All the test cases are based on this test track. The test suite for the four-wheel robot system consists of 15 test cases totalling 243 lines of code. The statement coverage of the test suite is 96.5%.

Using MUTPHY, we generated 371 mutants. The summarised result of all generated mutants is presented in Table VII. For **RQ2**, we found there are 10 equivalent mutants generated by MUTPHY. Similar to project *gpiozero*, all the equivalent mutants are of type SVR, where the initial value assignment in the setup function is removed. The cause of the equivalence is also similar: the initial default value is the same as the explicitly set initial value. Although these mutants are equivalent to the original program, explicitly setting the initial value in the setup function is still recommended because different embedded platforms have different default values and

TABLE VII
MUTANTS RESULT OF FOUR-WHEEL ROBOT

MOP	#Generated	#Covered	#Alive	#Killed	#Equiv.	MS
OVR	32	32	10	22	0	0.69
OSR	32	32	12	20	0	0.63
PNR	235	235	20	215	0	0.91
IVR	21	20	4	17	0	0.81
EDR	0	0	0	0	0	-
IOMR	19	19	0	19	0	1
SIR	3	3	3	0	0	0
SOR	13	13	7	6	0	0.46
SVR	16	16	15	1	10	0.17
Overall	371	370	71	300	10	0.83

Fig. 4. Diagram of line-follower robot

setting the initial value can avoid unexpected initial states. In conclusion, for the four-wheel robot system, the efficiency of MUTPHY in generating non-equivalent mutants is high (97.3%).

For **RQ1**, the overall mutation score is 0.83, which is lower than the statement coverage (96.5%). The three mutation operators with the highest mutation score are IOMR (1), PNR (0.91) and IVR (0.81). The first two mutation operators are easier to be killed than the others because these mutants can be detected once the mutated GPIO pins are invoked: in most cases, these pins are not initialised or initialised correctly (e.g., replace the output mode to the input mode). The 20 alive mutants from PNR are because of insufficient assertions in the tests suite; these missing assertions are needed to check the mutated statements. For IVR, as the input pins of the robots are connected to photoelectric sensors that are used to align the robot, most IVR mutants are easily killed if the robot does not reach the specific position by reading the un-mutated photoelectric sensors' states. For the four alive IVR mutants, one is due to uncovered statements; the other three are due to poor test design.

The three mutation operators with lowest mutation score are SIR (0), SVR (0.08) and SOR (0.46). The reason why none of the SIR mutants is killed is that the corresponding pins are connected to the peripherals (in particular, the photoelectric sensors) with very high resistors; this means the replacement of initial input value (*PUD_UP* or *PUD_DOWN*) cannot affect the overall potential. These alive mutants cannot be killed in this case, and even adding new tests would not make a difference. For SVR, the five alive non-equivalent mutants are due to insufficient assertions of the tests suite: the existing test suite does not examine all the initial states of the GPIO pins. The low mutation score of the SOR operator is due to inadequate tests that do not examine the initial states of the GPIO pins once the program starts.

The mutation score of mutants generated from OVR and OSR are 0.69 and 0.63, which is lower than we expected. The alive mutants of these two operators are due to meaningless feedback produced by the control program, and the test oracles are based on these feedback messages. For instance, the function *lift()* in *arm.py* lifts the arm for a given direction (up or down) and a period. Once the *lift()* call is finished, the function returns the input direction. This kind of feed-
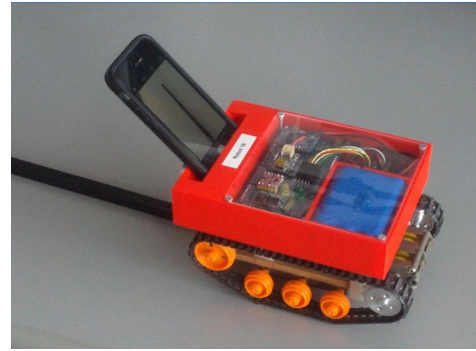
back does not reflect the actual states of the GPIO pins. Thus, the corresponding tests can never fail. To kill these surviving mutants, we replaced *GPIO.output* functions with mock objects to assess intermediate states of the target pins. For the five mutants that are located in the method *lift()*, introducing mock objects enables to effectively detect these mutants. However, the 17 other mutants cannot be easily killed by making use of mock functions. These 17 mutants reside in complicated methods with loops and input detections. Similar to project *hcsr04sensor*, the intermediate changes cannot be easily captured and observed by introducing mock objects, as the sequence of the method calls is uncertain (another case of a *snarled method* [24]). Thus, we need to refactor the original control program by moving the related *GPIO.output* function calls into new methods; this enabled us to design accurate test oracles to examine the state changes.

For **RQ3**, we managed to kill the 51 non-equivalent alive mutants by adding and improving test cases. The remaining 20 non-equivalent surviving mutants cannot be killed by simply adding tests. Among the 20 mutants, 17 mutants can be killed by refactoring the production code. This observation strengthens our earlier assumption that the mutation score could be influenced by the testability of the production code. The other three non-killable SIR mutants are caused by the peripherals. More precisely, for the affected circuits the overall potential cannot be changed by pulling up or down resistor, as the resistor of peripherals is too high to be changed by the Raspberry Pi's function. This type of stubborn mutants is unique to physical computing systems when compared to conventional software; it also increases the difficulty of testing physical computing systems. We suggest to classify this type of stubborn mutants as equivalent mutants, as the peripherals are part of the system, and generally, this part is not likely to change once the system is built up.

B. Case Studies with Arduino

The second part of our experiment targets the Arduino platform. The Arduino based system is taken from a lab session of an Embedded Software course for second-year undergraduate students at Delft University of Technology. The system is a robot that uses a camera instead of light or IR sensors to follow a line. It is shown in Figure 4 and is composed of of three components, each with a different role:

1) Smartphone: the camera of the smartphone is mounted on the robot makes images of the floor in front of the robot where the line should be detected;
2) Laptop: the laptop runs the Robot Operating System (ROS) core [32] and performs line detection on the images of the smartphone;
3) Arduino-based robot: the robot has to follow the line on the ground. This part includes one LCHB-100 H-bridge [33], one Arduino Mega ADK [34], one HC-05 Bluetooth dongle and one HC-SR04 ultrasonic sensor.

The students are required to implement the control program for the Arduino board and the line detection program based on ROS in groups of two. We collected implementations from four groups (the average LOC is 122.5 measured by sloccount [21]), and then the teaching assistant was asked to design test suites for those implementations. The main purpose of the test suites are to examine the five behaviours of the robot, i.e., going straight, turning left, turning right, stopping when there is an obstacle in front and stopping when no image is received. However, since the implementations of different groups are different from each other, we have to adjust the details of the tests to make them pass for further mutation testing. The statement coverage of the test suite is 100%.

1) Test Environment: The testing system is expected to be as isolated as possible from the program under test. In particular, the testing system should monitor the GPIO signals while keeping the code untouched. However, since the requirements of the student codes do not include the testing part, most of them cannot be tested without altering the codes. The reasons are as follows: first of all, the Arduino platform does not support multi-process nor multi-threading and thus only allows one main loop during execution. For the line-follower robot, the Arduino control program needs to be running continuously to receive operation signals from the PC as a client. Secondly, the test execution should be independent of the control program as a second process. In order to not introduce another process, we have to alter the students' code by adding test cases in the same program. This results in modifications and uncertainties in the control program. Therefore, we worked around the software limitation by adding a hardware monitor as shown in Figure 5. More specifically, we used another Arduino board (Arduino Uno [35]) to monitor the pin states of the control board of the line-follower robot.

The hardware monitor picks up two types of signals from the system under test.

- **Pulse Width Modulation (PWM) signals for the two DC motors for the wheels**. Each DC motor occupies a pair of PWM channels for the two rotational directions (controlled via LCHB-100 H-bridge). Therefore, the two DC motors take four PWM channels in total. We program the monitor hardware to sample the signals from the four channels at regular intervals. Thus, we can know whether the signal is high or low at each interval. We then approximate the duty cycle by calculating the ratio between the number of high signals and that of all signals. For instance, there are 100 high signals out of 500
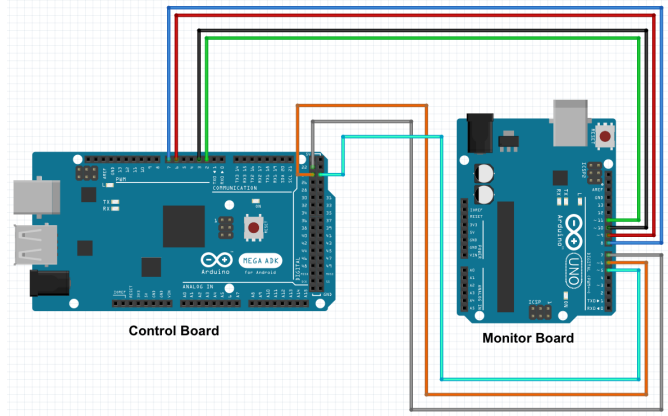

Fig. 5. Layout of test setup of line-follower robot

TABLE VIII
MUTANTS RESULT OF LINE-FOLLOWER ROBOT

MOP	#Generated	#Covered	#Alive	#Killed	#Equiv.	MS
OVR	38	38	26	12	0	0.32
OSR	34	34	25	9	0	0.26
PNR	298	298	184	114	0	0.38
IVR	0	0	0	0	0	-
EDR	0	0	0	0	0	-
IOMR	36	36	19	17	0	0.47
SIR	4	4	2	2	0	0.50
SOR	3	3	2	1	0	0.33
SVR	3	3	3	0	0	0.00
Overall	416	416	261	155	0	0.37

detected in five seconds for one PWM channel. Thus, the approximated duty cycle is 20%.

- **Standard digital signals from the ultrasonic distance sensor**. The sensor (HC-SR04) has a trigger pin and an echo pin. The trigger pin is used to emit ultrasound at 40,000 Hz, and the ultrasound signal is received in the echo pin. A test may override the echo signal of the sensor to create a simulated situation in which the robot detects a wall or an obstacle. The trigger pin is programmed to send an ultrasound continuously in this robot, which is independent of the simulation, so we use a single channel to emulate the echo signal.

To fully automate the testing process, we removed the chassis part from the Arduino-based robot, which does not influence the states of the PWM channels but prevents the robot from moving physically. Because our test oracles are based on the PWM signals of the DC motors to examine the robot's behaviour without the information of the physical location. For example, we designed the assertion for the robot turning left as *right_fwd_pwm > right_fwd_pwm*, where the forward PWM signal of the right motor is greater than that of the left motor. As a consequence, the whole mutation testing process is automated and requires no human observations.

2) Result: The overall mutation scores of the four implementations are quite similar, i.e., 0.34, 0.36, 0.39 and 0.40. The test suites examine the five movements of the robots; they are almost the same for the four student projects that

we consider. Table VIII summarises the mutants for these four implementations. We observe that 416 mutants have been generated. We did not find equivalent mutants amongst the generated mutants (**RQ2**). This is likely due to the control program of the Arduino being quite simple: it is mainly a signal receiver for the ROS core. The key program, the image processing program, on the other hand, is located on the PC side.

For **RQ1** we note that while the statement coverage of the test suite is 100%, the overall mutation score is 0.37. Further investigation of the test suite leads us to the fact that the existing test suite lacks assertions to examine all the target pins. In fact, the test suite only checks the states of two pins which control the forward direction of the motors (i.e. the 1FWD and 2FWD ports in the LCHB-100 H-bridge). Ideally, the test suite should check the four pins connected to the other ports of the LCHB-100 H-bridge.

To kill the alive mutants (**RQ3**), we added four assertions in each test case to ensure the correct states of the pins connected to the LCHB-100 H-bridge that controls the movement of the motors. This improvement resulted in 201 mutants being killed. However, there are still 60 mutants surviving after the modification. These 60 mutants are hard to kill due to the limitations of our test environment setup. Among the 60 stubborn mutants, 20 mutants are related to a pin that the hardware monitor did not track. This pin is to control an LED which students mostly used for debugging purposes. These 20 mutants could be killed if we monitor the states of the LED pin and add specific assertions for it. The remaining 40 mutants are hard to kill because our test environment can only monitor the pin states of the robot. This means that we cannot further check the other settings of the pins, e.g., the pin mode and resistor state, as we can do in the Raspberry Pi platform. This type of stubborn mutants is different from the previously observed stubborn mutants in the *hcsr04sensor* and *four-wheel robot* projects, where the stubbornness was due to software testability issues. As mentioned in Section V-B1, limitations of the Arduino platform prevent us from touching the codebase of the control program directly. The adoption of the hardware monitor treats the system as a black box; this restricts the features that we can test in this system, such as the internal settings of the pins. For this line-following robot, 90.4% non-equivalent surviving mutants can be killed by adding extra test cases, while the rest mutants are not killable due to test setups.

C. Summary

Based on the case studies on the Raspberry Pi and Arduino platforms, we evaluated our method in terms of the efficiency in generating non-equivalent mutants (**RQ2**) and the effectiveness in evaluating the test suite quality (**RQ1**). Moreover, we also manually analysed non-equivalent surviving mutants to explore whether the mutation score can be improved by implementing new or improved tests (**RQ3**). In this section, we summarise all results of all subjects involved in our experimental study (as shown in Table IX) and answer the three research questions in the light of our observations.

TABLE IX
MUTANTS RESULT OF ALL SUBJECTS

MOP	#Generated	#Covered	#Alive	#Killed	#Equiv.	MS
OVR	90	85	52	38	0	0.42
OSR	83	80	51	32	1	0.39
PNR	685	673	264	421	0	0.61
IVR	29	28	4	25	0	0.86
EDR	19	19	0	19	0	1.00
IOMR	73	72	21	52	0	0.71
SIR	15	15	5	10	0	0.67
SOR	18	18	9	9	0	0.50
SVR	24	24	19	5	11	0.38
Overall	1036	1014	425	611	12	0.60

Table IX indicates that there are 1036 mutants generated in total, with the PNR mutants comprising 66.1% of the total. The EDR mutants are easiest to kill, while the OSR and SVR mutants are most difficult to kill. For **RQ2**, the overall percentage of non-equivalent mutants is 98.8%, which is quite promising. The equivalent mutants mainly stem from SVR (one from project *gpiozero* and ten from project *four-wheel robot*). However, the equivalent versions without the initial value setup are not recommended since different embedded platforms have different default values. Explicitly setting the initial value in the setup function can avoid unexpected initial states. The other equivalent one arises from OSR, which is due to dead code (see project *RPLCD* in Section V-A1). Besides, three SIR mutants are non-killable which are caused by the circuit of the peripherals. We considered these mutants as equivalent mutants in the context of physical computing systems. Even taking the three SIR mutants into consideration, the non-equivalent mutants still comprise 97.5% of the total number of mutants, showing MUTPHY has high efficiency in generating non-equivalent mutants.

For **RQ1**, compared to the statement coverage, the mutation score generated by our method can be a better indicator of test suite quality. More specifically, the mutation score can evaluate how well the test suite examines the integration part of the software and peripherals in physical computing systems, something the statement coverage does not allow. Except for project *gpiozero*, all the non-equivalent alive mutants reveal the inadequate test cases in the existing test suite. This is especially true for project *RPLCD*, for which the mutation score is 0, while the statement coverage is 71%.

For **RQ3**, 94.2% of the mutants, in most cases, it is possible to kill all non-equivalent surviving mutants by adding extra test cases, which again supports **RQ1** that mutation score can effectively evaluate the existing test suite. The exception being 59 mutants. The Raspberry Pi case studies account for 19 of these mutants: 2 mutants from project *hcsr04sensor* and 17 mutants from project *four-wheel robot*. Killing these mutants would require refactoring the production code to increase the observability of state changes. This implies that test quality is not the only factor to determine the mutation score, as the testability of the production code can also impact the mutation score. Moreover, introducing mock objects is a double-edged sword. If the mock objects are used properly, the behaviour of

the GPIO pins cannot be examined, e.g., replacing the whole *RPi.GPIO* module to mock objects in project *RPLCD*. While proper use of mock objects can improve the observability of intermediate state changes to derive high-quality tests (see project *hcsr04sensor* and project *four-wheel robot*). For Arduino, 40 mutants remain not-killed as our test setups are unable to assess the internal settings of the system. A deeper analysis of these 40 mutants reveals that factors such as the testability of the software under test and the test setup influence the mutation score. We would like to explore these potential factors in the future work to further understand mutation testing and thus improve it.

VI. THREATS TO VALIDITY

External validity: First, our results are based on the Raspberry Pi and Arduino platforms; these results might be different when using other embedded platforms. Second, concerning the subject selection, we only chose nine physical computing systems in total to evaluate our approach. Unfortunately, few physical computing systems on the Raspberry Pi and Arduino platforms with up-to-date test suites are publicly available.

Internal validity: The main threat to internal validity for our study is the implementation of MUTPHY for the experiment. To reduce internal threats to a large extent, we carefully reviewed and tested all code for our study to eliminate potential faults in our implementation. Another threat to internal validity is the detection of equivalent mutants through manual analysis. However, this threat is unavoidable and shared by other studies that attempt to detect equivalent mutants.

Construct validity: The main threat to construct validity is the measurement we used to evaluate our methods. We used the percentage of non-equivalent mutants and the mutation score as key metrics in our experiment, both of which have been widely used in other studies on mutation testing.

VII. RELATED WORK

There has been a great deal of work on verification and validation of embedded systems (not limited to physical computing systems) in literature. The main methodologies are static analysis (e.g., [36], [37]), dynamic analysis (e.g., [38], [39]), formal verification (e.g., [40], [41]), black-box testing (e.g., [42], [43]), and white-box testing (e.g., [44], [45]).

Most related to our approach are *software-implemented fault injection (SWIFI)* techniques that inject faults pre-runtime at machine code level (e.g., by changing the content of memory/registers based on specified fault models) to emulate the consequences of hardware faults [46]. One of the earliest SWIFI techniques was presented by Segall et al. [47]. Their technique's initial results showed usefulness in reducing the fault injection complexity and validation of the system. Later, in 1995, Kanawati et al. [48] proposed a flexible software-based fault and error injection system, which is useful in evaluating the dependability properties of complex systems. More recently, Arlat et al. [12] compared physical and software-implemented fault injection techniques. As shown in their results, these two types of fault injection techniques are

rather complementary, while SWIFI approaches are preferable mainly due to high controllability, repeatability and cost-effectiveness. All the above works focus on hardware testing, and more specifically, the kernel layer. None of them considers the communication between the software and peripherals in physical computing systems.

Concerning the application of mutation testing in embedded systems, Zhan et al. [49], He et al. [50] and Stephan et al. [51] have addressed the notion of *Simulink model mutations*. They proposed a set of mutation operators explicitly for Simulink that target the run-time properties of the model, such as signal addition operators. Moreover, Enoiu et al. [52] investigated mutation-based test generation for PLC embedded software using model checking. In their work, they designed six mutation operators for PLC embedded software relying on commonly occurring faults in IEC 61131-3 software [53], [54]. Different from our approach, all these works target mutation testing at the model level, and can only be applied to one specific type of software, e.g. *Simulink*. Our approach, on the other hand, is based on source code, and can thus potentially apply to all kinds of embedded system platforms.

VIII. CONCLUSION & FUTURE WORK

Physical computing systems come with their own set of challenges. This paper focuses on the challenge of testing these physical computing systems, with a particular focus on assessing the quality of the tests that validate the interactions between the software and the physical components. We zoom in on common mistakes that occur in these interactions and propose a novel mutation testing approach with nine mutation operators targeting these common interaction mistakes.

Our results have shown encouraging results in uncovering weaknesses in existing tests. As such, our mutation testing approach enables to guide engineers to test systems more effectively and efficiently. More specifically, for our nine case study systems our mutation testing tool generated a total of 1036 mutants of which 41% were not killed by the original test suite (and 1.2% of the overall mutants being equivalent mutants). Adding tests or reinforcing existing tests made it possible to kill 94% of the non-equivalent surviving mutants.

Our paper makes the following contributions:

- a generic mutation testing approach for physical computing systems;
- a mutation testing tool named MUTPHY working on the Raspberry Pi and Arduino platforms;
- a preliminary experiment on nine physical computing systems[4];

Future work. In the future, we aim to conduct additional case studies on more realistic physical computing systems. Also, we would like to explore the complementarity between traditional mutation operators and our newly designed, yet very specific mutation operators. Finally, we also aim to explore the relationship between testability and mutation score.

---

[4]All the tools, scripts and metadata for this experimental study are available in our GitHub repository [55].

REFERENCES

[1] D. O'Sullivan and T. Igoe, *Physical computing: sensing and controlling the physical world with computers*. Course Technology Press, 2004.

[2] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of things (iot): A vision, architectural elements, and future directions," *Future generation computer systems*, vol. 29, no. 7, pp. 1645–1660, 2013.

[3] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Trans. on Softw. Engineering*, vol. 37, no. 5, pp. 649–678, 2011.

[4] A. P. Mathur and W. E. Wong, "An empirical comparison of data flow and mutation-based test adequacy criteria," *Software Testing, Verification and Reliability*, vol. 4, no. 1, pp. 9–31, 1994.

[5] P. G. Frankl, S. N. Weiss, and C. Hu, "All-uses vs mutation testing: an experimental comparison of effectiveness," *Journal of Systems and Software*, vol. 38, no. 3, pp. 235–253, 1997.

[6] N. Li, U. Praphamontripong, and J. Offutt, "An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage," in *ICST workshops*. IEEE, 2009, pp. 220–229.

[7] J. Offutt, "A mutation carol: Past, present and future," *Information and Software Technology*, vol. 53, no. 10, pp. 1098–1107, 2011.

[8] Q. Zhu, P. Annibale, and A. Zaidman, "A systematic literature review of how mutation testing supports test activities," *PeerJ Preprints*, 2016. [Online]. Available: https://doi.org/10.7287/peerj.preprints.2483v1

[9] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *International Conference on Software Engineering*. IEEE, 2005, pp. 402–411.

[10] J. A. Stankovic, I. Lee, A. Mok, and R. Rajkumar, "Opportunities and obligations for physical computing systems," *Computer*, vol. 38, no. 11, pp. 23–31, 2005.

[11] S. Tilak, N. B. Abu-Ghazaleh, and W. Heinzelman, "A taxonomy of wireless micro-sensor network models," *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 6, no. 2, pp. 28–36, 2002.

[12] J. Arlat, Y. Crouzet, J. Karlsson, P. Folkesson, E. Fuchs, and G. H. Leber, "Comparison of physical and software-implemented fault injection techniques," *IEEE Transactions on Computers*, vol. 52, no. 9, pp. 1115–1133, 2003.

[13] T. Dillon, C. Wu, and E. Chang, "Cloud computing: issues and challenges," in *Advanced Information Networking & Applications*. IEEE, 2010, pp. 27–33.

[14] Texas Instruments, "74LS107 JK flip-flop Data Sheet," http://www.utm.edu/staff/leeb/logic/74ls107.pdf, [Online; accessed 14-June-2017].

[15] ——, "74HC74 D flip-flop Data Sheet," http://www.utm.edu/staff/leeb/logic/74ls74.pdf, [Online; accessed 14-June-2017].

[16] "pytest," https://docs.pytest.org/en/latest/, [Online; accessed 30-October-2017].

[17] "unittest," https://docs.python.org/3/library/unittest.html, [Online; accessed 30-October-2017].

[18] "doctest," https://docs.python.org/3/library/doctest.html, [Online; accessed 30-October-2017].

[19] D. Bargen, "RPLCD," https://github.com/dbrgn/RPLCD, [Online; accessed 30-January-2018].

[20] "HCSR04 Manual," https://www.linuxnorth.org/raspi-sump/HC-SR04Users_Manual.pdf, [Online; accessed 30-January-2018].

[21] D. A. Wheeler, "SLOCCount," https://www.dwheeler.com/sloccount/, [Online; accessed 25-September-2017].

[22] "Coverage.py," https://https://coverage.readthedocs.io, [Online; accessed 30-January-2018].

[23] A. Audet, "hcsr04sensor," https://github.com/alaudet/hcsr04sensor, [Online; accessed 30-January-2018].

[24] M. Feathers, *Working effectively with legacy code*. Prentice Hall Professional, 2004.

[25] L. Moonen, A. van Deursen, A. Zaidman, and M. Bruntink, "On the interplay between software testing and evolution and its effect on program comprehension," in *Software Evolution*, T. Mens and S. Demeyer, Eds. Springer, 2008, pp. 173–202.

[26] J. M. Voas and K. W. Miller, "Software testability: The new verification," *IEEE software*, vol. 12, no. 3, pp. 17–28, 1995.

[27] M. Cargnelutti, "Jean-Pierre," https://github.com/matteocargnelutti/jean-pierre, [Online; accessed 31-January-2018].

[28] "Raspberry Pi Zero W," https://www.raspberrypi.org/products/raspberry-pi-zero-w/, [Online; accessed 31-October-2017].

[29] "OpenFoodFacts API," https://world.openfoodfacts.org/, [Online; accessed 31-October-2017].

[30] "Raspberry Camera Module V2," https://www.raspberrypi.org/products/camera-module-v2/, [Online; accessed 31-October-2017].

[31] RPi-Distro, "GPIO Zero," https://github.com/RPi-Distro/python-gpiozero, [Online; accessed 30-January-2018].

[32] Open Source Robotics Foundation, "Robot Operating System," http://www.ros.org/, 10 2014, [Online; accessed 16-February-2018].

[33] "LCHB-100 H-bridge," https://www.robotshop.com/media/files/pdf/lchb-100.pdf, [Online; accessed 27-February-2018].

[34] Arduino, "Arduino Mega ADK," https://store.arduino.cc/arduino-mega-adk-rev3, [Online; accessed 27-February-2018].

[35] ——, "Arduino Uno," https://store.arduino.cc/arduino-uno-rev3, [Online; accessed 25-September-2017].

[36] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner, "Thorough static analysis of device drivers," *ACM SIGOPS Operating Systems Review*, vol. 40, no. 4, pp. 73–85, 2006.

[37] J. W. Voung, R. Jhala, and S. Lerner, "Relay: static race detection on millions of lines of code," in *Proc. of the Joint Meeting of the European Software Engineering Conference and the Int'l Symp. on Software Engineering (ESEC/FSE)*. ACM, 2007, pp. 205–214.

[38] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model checking programs," *Automated Software Engineering*, pp. 203–232, 2003.

[39] V. V. Rubanov and E. A. Shatokhin, "Runtime verification of linux kernel modules based on call interception," in *Int'l Conf. Software Testing, Verification and Validation (ICST)*. IEEE, 2011, pp. 180–189.

[40] K. G. Larsen, M. Mikucionis, B. Nielsen, and A. Skou, "Testing real-time embedded software using uppaal-tron: an industrial case study," in *Proc. Int'l Conf. on Embedded Software*. ACM, 2005, pp. 299–306.

[41] J. Kim, I. Kang, J.-Y. Choi, and I. Lee, "Timed and resource-oriented statecharts for embedded software," *IEEE Transactions on Industrial Informatics*, vol. 6, no. 4, pp. 568–578, 2010.

[42] W.-T. Tsai, L. Yu, F. Zhu, and R. Paul, "Rapid embedded system testing using verification patterns," *IEEE software*, pp. 68–75, 2005.

[43] A. Sung, B. Choi, and S. Shin, "An interface test model for hardware-dependent software and embedded os api of the embedded system," *Computer Standards & Interfaces*, vol. 29, no. 4, pp. 430–443, 2007.

[44] H. Lu, W. Chan, and T. Tse, "Testing context-aware middleware-centric programs: a data flow approach and an RFID-based experimentation," in *Int'l Symp. Foundations of Software Engineering (FSE)*. ACM, 2006, pp. 242–252.

[45] Q. Zhang and I. G. Harris, "A data flow fault coverage metric for validation of behavioral hdl descriptions," in *Proc. Int'l Conf on Computer-aided design*. IEEE, 2000, pp. 369–373.

[46] E. Fuchs, "An evaluation of the error detection mechanisms in mars using software-implemented fault injection," *Dependable Computing Conference*, pp. 73–90, 1996.

[47] Z. Segall, D. Vrsalovic, D. Siewiorek, D. Ysskin, J. Kownacki, J. Barton, R. Dancey, A. Robinson, and T. Lin, "Fiat-fault injection based automated testing environment," in *Proc. 18th Int. Symposium on Fault-Tolerant Computing*. IEEE, 1988, p. 394.

[48] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham, "Ferrari: A flexible software-based fault and error injection system," *IEEE Transactions on computers*, vol. 44, no. 2, pp. 248–260, 1995.

[49] Y. Zhan and J. A. Clark, "Search-based mutation testing for simulink models," in *Proceedings of the 7th annual conference on Genetic and evolutionary computation*. ACM, 2005, pp. 1061–1068.

[50] N. He, P. Rümmer, and D. Kroening, "Test-case generation for embedded simulink via formal concept analysis," in *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*. IEEE, 2011, pp. 224–229.

[51] M. Stephan, M. H. Alalfi, and J. R. Cordy, "Towards a taxonomy for Simulink model mutations," in *Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2014, pp. 206–215.

[52] E. P. Enoiu, D. Sundmark, A. Čaušević, R. Feldt, and P. Pettersson, "Mutation-based test generation for plc embedded software using model checking," in *IFIP International Conference on Testing Software and Systems*. Springer, 2016, pp. 155–171.

[53] Y. Oh, J. Yoo, S. Cha, and H. S. Son, "Software safety analysis of function block diagrams using fault trees," *Reliability Engineering & System Safety*, vol. 88, no. 3, pp. 215–228, 2005.

[54] D. Shin, E. Jee, and D.-H. Bae, "Empirical evaluation on fbd model-based test coverage criteria using mutation analysis," in *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2012, pp. 465–479.

[55] Q. Zhu, "MutPhy GitHub Repository," https://zenodo.org/badge/latestdoi/136980504, [Online; accessed 11-June-2018].