Functional Size Measurement Automation for IoT Edge Devices

Salma Salem[1], Hassan Soubra[2]

[1]German University in Cairo, New Cairo, Egypt
[2]ECE-École centrale d'électronique, Lyon, France

Abstract
The Internet of Things (IoT) was born when Edge devices were interconnected via the internet allowing them to send and receive data back and forth. Edge devices are actually embedded systems processing data in real-time close to the physical environment in which they are deployed. Functional Size Measurement (FSM) is a tool to measure functionality provided by a software and obtain technical indicators early in the design phase. The COSMIC method is an international standard for functional size measurement of a software. A study [2] discussed FSM for Internet of things (IoT) real-time embedded software and proposed a guideline using Arduino. Another study in [8] proposed the idea of Universal IoT metrics and automated some metrics proposed. This paper presents an approach to automatically measure COSMIC functional size of any IoT Edge Device code. This paper is based on the measurement procedure rules presented in [2], and it attempts to complete the tool presented in [8].

Keywords
Internet of Things (IoT), Edge Devices, FSM Automation, COSMIC ISO 19761, Functional Size

1. Introduction

The Internet of Things (IoT) is a collection of Edge Devices sending and receiving data via networks that needs to be processed and analysed.

Instead of sending IoT data systematically back to a datacenter or to the cloud, edge computing, a technique for computing on site where data is received or used, allows IoT data to be captured and processed at the edge.

IoT and edge computing work well together to quickly analyse data in real-time.

An IoT system typically operates by transmitting, receiving, and analysing data in a feedback loop perpetually. Humans or artificial intelligence and machine learning can perform analysis, either in near real-time or over a longer period of time.

When anything is described as smart, IoT is usually implied, e.g., autonomous vehicles, smart houses, smartwatches, virtual and augmented reality, and industrial IoT.

The COSMIC method is an International Standard: ISO/IEC 19761 Software Engineering – COSMIC – A functional size measurement (FSM) method and as greatly contributed in improving Software measurement and estimation processes in both research and industry [1]. COSMIC provides a standardized method for measuring the functional size of software from both the

IWSM/MENSURA 23, September 14-15, 2023, Rome, Italy
salma.abdallahsalem@student.guc.edu.eg (S. Salem); hsoubra@ece.fr (H. Soubra)

© 2023 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings (CEUR-WS.org)

CEUR-WS.org/Vol-3543/paper13.pdf

Management Information Systems perspective and real-time. Software functional size can be used for a number of purposes: to obtain system related technical indicators early in the design phase which in turn can impact performance. It can also be used to estimate development effort, manage project scope changes, measure productivity, benchmark and normalize quality and maintenance ratios.

In 2017, [2] introduced IoT and COSMIC based FSM for software running on the Arduino platform.

Many automated COSMIC FSM procedures have been proposed in the literature for other different domains e.g., aeronautical [3], automotive [4], quantum computing [5] and computer Hardware [6] [7].

This paper is based on the measurement procedure rules presented in [2], and it attempts to complete the IoT metrics automation tool presented in [8], by implementing a COSMIC FSM automation tool for IoT Edge devices.

This paper is organised as follow: Section 2 presents related works including overviews of [2] and [8]. While section 3 describes the Methodology of our automated tool, Section 4 introduces our tool proposed and the testing carried and the results obtained. Finally, conclusions and future work follow in Section 5.

2. Background

This section discusses related work on Functional size measurement for IoT as well as papers investigating IoT metrics.

The study in [2] utilized the COSMIC method to measure the Functional size of IoT Software by mapping the COSMIC method concepts to Arduino's. Two main elements are needed to map COSMIC to Arduino.

First, data group movements and the associated data manipulations that determine software functionality according to COSMIC are: ENTRY (E) and EXIT(X) data movements that allow data exchange with users across a boundary. READ (R) and WRITE(W) data movements that allow data exchange with the persistent storage.

Second, functional processes which are defined according to COSMIC as a set of data movements, representing an elementary part of the functional user requirements (FUR) for the software being measured that is unique within these FUR and that can be defined independently of any other functional process in these FUR. A functional process may have only one triggering Entry. Each functional process starts processing on receipt of a data group moved by the triggering Entry data movement of the functional process. The set of all data movements of a functional process is the set that is needed to meet its FUR for all the possible responses to its triggering Entry.

The mapping of COSMIC elements to Arduino elements is made in [2] :

- In the Arduino code, the setup() and loop() functions are considered as functional processes and each of them has merely one triggering Entry. The setup() function is triggered upon power up or reset of the system. The loop() is triggered upon the completion of setup() and it runs perpetually or until the powering off of the system. Furthermore, all data

movements needed for functional user requirements take place in both of these functions. Declaring pins as input or output (sensor or actuator) occurs in the setup() functions and then, in the perpetual loop() function, data flows between the system and the sensors or actuators which are considered to be functional users of the software.

- The mapping of the Movement of data flows can be summarized as follows:

  1. Entry represents the one and only one flow of a single data group from the functional user (e.g., sensor) across a boundary to a functional process.
  2. Exit represents the one and only one flow of a single data group from the functional process across a boundary to a functional user (e.g., actuators).
  3. Read represents the movement of one and only one single data group from persistent storage (e.g., EEPROM) to functional processes.
  4. Write represents the movement of one and only one single data group from functional processes persistent storage (e.g., EEPROM).

Hence, each INPUT pin used as an argument in a function is considered as a COSMIC Entry. Meanwhile,each OUTPUT pin used in as an argument in a function is a COSMIC Exit. Further more, each function call used for reading data from EEPROM counts as a COSMIC Read. Meanwhile, each function call used to write data to EEPROM counts as COSMIC Write.

The study in [8] proposed the idea of Universal IoT metrics which tends to group related metrics in specific classes based on the ISO 25023 standard. In addition, [8] presented a tool to automatically measure one of the universal metrics defined in the paper, namely the Functional Suitability metric. nevertheless, the tool in [8] does not integrate COSMIC FSM.

Lastly, the authors in [9] used a set of five research questions on the present knowledge bases for IoT metrics to perform a review of the literature based on studies published between January 2010 and December 2021. According to the many components and features of an IoT system, 158 IoT metrics were discovered and divided into 12 categories. The 12 criteria were arranged into an ontology to encompass the total performance of an IoT system. The findings indicate that 43% of the studies covered network metrics the most, and that 37% of those studies had the maximum amount of metrics. This study can offer recommendations for choosing metrics for IoT systems as well as insightful information on areas for enhancement and optimisation.

3. Methodology

This section presents the prototype tool implemented for automating COSMIC functional size measurement for IoT Edge devices relying on the measurement procedure presented in [2] and it follows the automatic metrics retrieval approach implemented in [8].

The tool is divided into two modules as depicted in Figure 1. The first module retrieves the source code from the micro-controller. The Second module measures the FSM of the retrieved source code in Comic Functional Points (CFP). The Message Queueing Telemetry Transport (MQTT) protocol which is an extremely lightweight publish/subscribe messaging transport

protocol is used in our tool for sending the retrieved code from the first to the second module as well as publishing the size in CFP of the retrieved code. Hence, each module of the tool acts as an MQTT client.

Initially, a connection between the micro-controller and a MQTT client should be established, this connection is aimed to send the code of the micro-controller via the internet to another MQTT client (Module 2 of our tool proposed) to measure automatically the COSMIC functional size of the retrieved code in CFP. HiveMQ MQTT cloud broker is used.

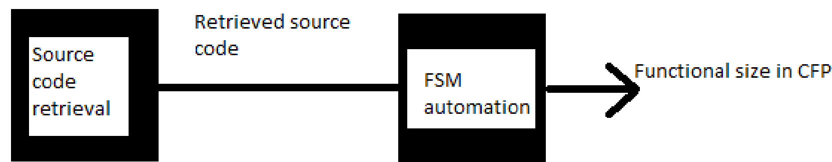

Figure 1: The two Modules of the tool proposed

3.1. Module 1

The main objective of Module 1 is to retrieve the source code running on the micro-controller and to send it using MQTT. Since the programming languages used on Edge devices are not a high level language, the source code cannot be retrieved automatically (e.g., as opposed to JavaScript through invoking a .body() function call to retrieve the functions of the source code). Hence, in order to retrieve the source code of Edge devices (e.g., Arduino, ESP), 4 potential approaches are explored:

- Approach 1: This approach requires manual manipulations from the user's side. First, the user has to retrieve the .elf file from the Arduino Ide by ticking the compilation option in preferences submenu found in the file menu. Secondly the user has to use either AVR or Xtensa tool using specific commands to convert the .elf file to assembly and next the user has to install MQQT CLI and establish connection with a client using specific commands that can be scripted and copy and paste manually the assembly code in a message and send it via CLI. To that end, this approach was excluded due to the multiple manual steps that are needed by the user.

- Approach 2: This approach consist of saving the code as a pre-requisite in the EEPROM and the FSM tool includes a function that automatically retrieves the code from EEPROM. This approach can be easily automated. However, it has its own drawbacks. For instance, the EEPROM has limited amount of storage size and hence codes with size exceeding a

certain threshold will not fit in the EEPROM. In addition to the fact that the EEPROM might be used by the program itself to store and retrieve functional or calibration data needed by the Edge device. In addition, certain rules have to be followed by the user: the Edge device programmer should pass the starting address in the function to be called to read the source code from the EEPROM as well as the Lines of codes including the separators used. Secondly, the Edge device programmer must store after every line of code a special character, e.g., “!” , since the EEPROM is byte addressable and an indicator on where every line of code starts and where it ends is needed. Finally, the Edge device programmer has to store another special character, e.g., “@”, in the EEPROM to mark the end of the file. See example in Figure 2.

```
File    Edit    View

[{ "void setup () {", "!","Serial.begin(9600);", "!","}", "!","void loop () {", "!","Serial.println(IOT);", "!","}", "!","@" }].
```

Figure 2: Arduino code example using the added special characters saved on the EEPROM

- Approach 3: This approach consists of saving the code as a pre-requisite on an external SD- card and the FSM prototype tool proposed includes a function that automatically retrieves the code from the SD-card. Using an SD-card is as a practical solution when it comes to storage, however, it requires extra Hardware connections to the micro-controller of the Edge device.

- Approach 4: This approach consists of storing the source code in the file system of the micro-controller, our tool includes a function which parameters should be passed by the user which is the source code and the function automatically saves it in the file system. Next, another function can be called without passing any parameters to retrieve the source code from the file system and send it via MQTT to the Module 2 of the tool. The code has to be passed as a String and line separators ”\n” have to be added to indicate the end of each line, as shown in Figure 3.

```
"Void setup() {" +"\n"+ "Serial.begin(9600);"+ "\n" + "}" + "\n" + "void loop(){" + "\n" +"Serial.print(IoT);"+"\n" + "}"
```

Figure 3: Arduino code example using a String and line separator ”\n”

Table 1 provides a comparison summary of the four approaches proposed to retrieve the source code form the IoT Edge device based on: space availability, extra hardware resources required, pre-requisites needed and feasibility of full automation.

For instance, when using boards that do not support File systems such as Arduino, we recommend following Approach 2 in case of a small size code (in Bytes) or to use an external SD-card

(Approach 3) if the source code exceeds a certain size in Bytes. Conversely, we recommend Approach 4 when using ESP boards since ESP boards support File systems which storage capacity is sufficient for any code size without neither acquiring extra hardware connection to store the code in an external SD-card nor facing the limitations of memory storage capacity of the EEPROM.

Approach comparison				
Approach number	Space availability	Extra Hardware resources	Pre-requisites needed	Fully automation
Approach 1	-	No	Multiple	Not possible
Approach 2	Very limited	No	Code stored in the EEPROM	Possible
Approach 3	Less limited	Required	Code stored on the SD-card	Possible
Approach 4	Less limited	No	No	Possible

Table 1
Comparison summary between the 4 approaches proposed.

Once the source code has been extracted, the connection between the MQTT client and MQTT broker is done and next the source code is published via the network to another MQTT client. To ensure the re-usability of our functions, they are wrapped in a class enabling them to be imported: this requires Wi-Fi credentials as well as the Hive MQTT credentials to initialize the class. In addition, functions relevant to the approach used should be called with the required parameters in the specific format, as described earlier, and the function will automatically publish the source code to a specific code to be received by the second MQTT client (Module 2) to measure the functional size in CFP of the source code retrieved.

3.2. Module 2

Module 2's main functionality, implemented as a JavaScript MQTT client, is to automatically measure the COSMIC size of the source code retrieved. Module 2 (the second MQTT client) receives the source code from the first MQTT client (Module 1) as a String. This code is analyzed and the COSMIC Entry, Exit, Read and Write data movements are identified in the code. JavaScript is used because it is a high-level language that supports String comparison which is needed for the Data movement identification.

The libraries that are taken into consideration in the current implementation of the prototype tool proposed are:

- Serial library: used for serial communication between the micro-controller and a serial monitor or another micro-controller. Hence, its function calls will either represent an Entry or an Exit.
- EEPROM library: used from reading and writing data to and from persistent storage. Hence it functions calls either represent a Read or a Write.
- Liquid Crystal: used to control print data on an LCD (Actuator). Hence, its function calls are considered as an Exit.

- WiFi : Enables connection with between edge devices and internet network . Hence, its function calls will either represent an Entry or an Exit.

3.3. Use of the prototype tool proposed

The usage of our prototype tool is not restricted on a certain IoT domain since Module 2 of the tool relies on parsing the source code retrieved from the micro-controller and checking line by line for the existence of any data group movements. It is also worth mentioning that the function call responsible for source code retrieval and sending it via MQTT to the second module will not infer with other tasks running on the Micro-controller since the function is called once in the Set-up function and will not run perpetually so no need to endure extra costs or re-schedule the tasks running .Hence,the steps required to use our prototype tool are:

1. Register to HiveMQ cloud and create a cluster. Next, create 2 web Clients and save their credentials.
2. Use the credentials of the first web client in the second module of the tool to establish a connection with the MQTT broker and subscribing to the topics of interest: Lines of code, CFP, Entry, Exit, Read, Write.
3. Use the credentials of the second web client as well as wifi credentials in the IoT Edge device to establish a connection with the MQTT broker and calling the function dedicated for sending the code with the appropriate parameters previously mentioned relying on any of the 3 automatable approaches described earlier to the first web client to measure the COSMIC functional size in CFP. The details of the measurement results are displayed in the console. It is possible for the IoT Edge device to subscribe to the same topics as the JavaScript client if the measurement result is needed to be displayed on the IoT Edge device (e.g., displaying the COSMIC size of the Edge device's code on its LCD).

Hence, it is only required from the user to choose the most convenient approach required for source code retrieval in module 1 of the tool based on the parameters mentioned above which are the type of the board, the size of the source code running , possibility to add extra-hardware connections ,afterwards, the user has to merely call the function corresponding for the approach that was adopted provided that the prerequisites of the approach are already met and that the connection using the Wi-Fi credentials is established. The MQTT client of the second module has to be connected to the MQTT broker before attempting to send the source code via the first module since the second module receives the code published from the first module.

4. Testing and Results

4.1. Automated COSMIC FSM testing

In order to verify the accuracy of the automated functional size measurement of Module 2, the example code used in [2], see Figure 5, was used for testing since its functional size is also measured and the detailed measurement results are presented in the research paper. According to [2], the COSMIC size measurement of the example code is 17 CFP and 3 Entry and 14 Exit

data movements are identified.

The automated measurement result of the example code produced by Module 2 of our prototype tool proposed is shown in Figure 4: the prototype tool has correctly and accurately identified 3 Entry and 14 Exit data movements for a total COSMIC functional size of 17 CFP.

```
Connected
Received message: my/test/topic/CFP  JS CFP is 17
Received message: my/test/topic/CFP/ENTRY  JS ENTRY is 3
Received message: my/test/topic/CFP/EXIT  JS EXIT is 14
Received message: my/test/topic/CFP/READ JS READ is 0
Received message: my/test/topic/CFP/WRITE JS WRITE is 0
```

Figure 4: Automated COSMIC functional size results of Module 2.

4.2. Code retrieval and measurement testing

The second test consists of establishing the connection between our JavaScript client and the IoT Edge Device and verifying that the Edge Device is able to send the source code properly to the JavaScript client. For this purpose, the JavaScript client subscribed to a new topic namely (test/topic) which is also used by the IoT Edge Device (the Arduino client in this example) to publish its source code. The code in Figure 6 is used to retrieve the source code from the EEPROM and publish it to the topic named (test/topic) to be received by the JavaScript client.

The connection was established successfully and the Arduino client was able to send its source code to the JavaScript client, a simple code example that prints the word "IoT" to the serial monitor is stored in the EEPROM (approach 2) and the Arduino client was able to retrieve the code from the EEPROM and send it to the JavaScript client. The simple code example used is shown in Figure 7. It is important to note that the code sent by the Arduino broker did not include the comments to save space, since we followed the second approach that requires as a prerequisite storing in advance the source code (without the comments) in the EEPROM. See Figure 8.

Finally, the code sent by the Arduino client is measured automatically by the JavaScript client and then the results are published as shown in Figure 9.

```
// include the LCD library
#include <LiquidCrystal.h>
// initialize the library with the numbers of the interface pins
LiquidCrystal lcd(12, 11, 5, 4, 3, 2);
//initiate the value of the sensor
int sensorVal=0;
//define the setup function
void setup()
{ // open a serial connection to display values
Serial.begin(9600);
// set up the LCD's number of columns and rows:
lcd.begin(16, 2);}
//define the loop function
void loop() {
// read the value on AnalogIn pin 0
int sensorVal = analogRead(A0);
// send the 10-bit sensor value out the serial port
Serial.print("sensor Value: ");
Serial.print(sensorVal);
// convert the ADC reading to voltage
float voltage = (sensorVal / 1024.0) * 5.0;
// Send the voltage level out the Serial port
Serial.print(", Volts: ");
Serial.print(voltage);
// convert the voltage to temperature in degrees C
// the sensor changes 10 mV per degree
// the datasheet says there's a 500 mV offset
// ((voltage - 500mV) times 100)
float temperature = (voltage - .5) * 100;
// Send the temperature in degrees C out the Serial port
Serial.print(", degrees C: ");
Serial.println(temperature);
// set the cursor to column 0, line 0 and print out Hello
lcd.setCursor(0, 0);
lcd.print("Hello");
// set the cursor to column 0, line 1 and print out Paris Temp
//and the temperature in degrees C
lcd.setCursor(0, 1);
lcd.print("Paris Temp: ");
lcd.print(temperature);
 delay(1000);}
```

Figure 5: Arduino source code example used in [2].

4.3. Real life application test and discussion

Once the functionality of the prototype tool proposed has been tested with simple code examples, the source code from [10] was used as a real-life IoT application test for its containment of multiple functions calls used to connect or send data via the WiFi network which is essential in any IoT applications. The Edge device running the code measures the temperature and sends the values via the Internet to a web page, see Figure 10. The testing results showed that: the COSMIC functional size of the real-life application example obtained automatically by the tool is 43 CFP. The automated results have been verified following the accuracy protocol in [11] and the accuracy of the results produced by the prototype tool proposed for the real-life application example is 93.4%. While Table 2 shows the detailed data movements identification per functional process, Table 3 summarizes the results of the total COSMIC FSM size of the test in CFP.

Testing the real life example code inferred the following key points about the tool accuracy and usability :

```
void sendmessageEEPROM(int address, int size){
    int addr=address;
   int arraysize=size;
  String temp ="";
  int count;
  int j;
  for (int i = 0; i < arraysize; ++i)
    { count =0;
      j=0;
      //byte Charl = qsid[i].length();
     while (char(EEPROM.read(addr +j))!='!'){
       count++;

      temp += char(EEPROM.read(addr +j));
       j++;

      }
      temp= temp +"\n";
      addr= addr+ count+1;

     // Serial.println(temp);    //for debugging if needed
      if(char(EEPROM.read(addr)=='@' )){
        break;
      }

 }
 Serial.println(temp);
 int length= temp.length()+1;
char buf [length+1];
   temp.toCharArray(buf,length);
  // char *buf=temp;
   delay(500);
  client.publish("test/Topic", buf);
```

Figure 6: Code snippet implementing Source code retrieval from the EEPROM (Approach 2)

```
Void Setup () {

//Establish Serial connection with the PC monitor with baud rate value 9600
Serial.begin(9600);

}

Void loop () {

//Print the word "IoT" to the serial monitor

Serial.print("IoT")

}
```

Figure 7: Code example used for testing the code retrieval and sending

```
Received message: test/Topic void setup() {
Serial.begin(9600);
}
void loop() {
Serial.println(IOT);
}
```

Figure 8: Source Code received by the JavaScript client

```
Received message: my/test/topic/linesofcode  JS LOC6
Received message: my/test/topic/CFP  JS CFP is 4
Received message: my/test/topic/CFP/ENTRY  JS ENTRY is 2
Received message: my/test/topic/CFP/EXIT  JS EXIT is 2
Received message: my/test/topic/CFP/READ JS READ is 0
Received message: my/test/topic/CFP/WRITE JS WRITE is 0
```

Figure 9: Functional size measurement results (in CFP) of the simple code example

- The prototype tool currently is able to identify multiple function calls included in the basic libraries used for communication (e.g., Serial, Wifi) as well as basic function calls used for interacting with functional users (i.e. sensors and actuators). Nevertheless, measuring functional size accurately requires detailed analysis of every function call found in every library which is a difficult task due to the big number of these libraries.
- The current version of the prototype tool restricts the way of writing the Edge device code. To clarify, the tool assumes when defining a function that every ("{") is written in the same line where the function name, conditional statements or loops are written. In addition, comments are assumed to be written in separate lines from the actual codes. Otherwise, the prototype tool will not be able to identify all COSMIC Data Movements successfully. For instance, in the real-life example temperature sensing code, the tool failed in identifying 3 data movements identifications:
  - 1X: Serial.print("http://"); because // was considered a comment by the tool;
  - 1E: float reading = analogRead(LM35); and 1X:client.println(""); had comments in the same line. H

Hence, the main limitation lies in the numerous number of libraries supported by Arduino or ESP boards dedicated merely for certain functional users (i.e sensors or actuators). This causes the inability of our prototype proposed in identifying data group movements related to these libraries' function calls.

Table 2: Manual and automated data movements identification

Functional process	Data group movements identified Manually	Data group movements identified Automatically
Setup()		
	1X:WiFiServer server(80);	1X:WiFiServer server(80);
	1E:Trigger (setup function call)	1E:Trigger (setup function call)
	1X:Serial.begin(115200);	1X:Serial.begin(115200);
	1X:delay(10);	1X:delay(10);
	1X:Serial.println();	1X:Serial.println();
	1X:Serial.println();	1X:Serial.println();
	1X:Serial.print(”Connecting to ”);	1X:Serial.print(”Connecting to ”);
	1X:Serial.println(ssid);	1X:Serial.println(ssid);
	1X:WiFi.begin(ssid,password);	1X:WiFi.begin(ssid,password);
	1E:while (WiFi.status() != WL-CONNECTED)	1E:while (WiFi.status() != WL-CONNECTED)
	1X:delay(500);	1X:delay(500);
	1X:Serial.print(”.”);	1X:Serial.print(”.”);
	1X:Serial.println(””);	1X:Serial.println(””);
	1X:Serial.println(”WiFi connected”)	1X:Serial.println(”WiFi connected”)
	1X:server.begin();	1X:server.begin();
	1X:Serial.println(”Server started”);	1X:Serial.println(”Server started”);
	1X:Serial.print(”Use this URL to connect:”);	1X: Serial.print(”Use this URL to connect: ”)
	1X:Serial.print(”http://”);	**UNIDENTIFIED BY THE TOOL**
	1X:Serial.print(WiFi.localIP());	1X:Serial.print(WiFi.localIP());
	1X:Serial.println(”/”);	1X:Serial.println(”/”);
loop()		
	1E: Trigger (setup complete)	1E: Trigger (setup complete)
	1X,1E:WiFiClient client = server.available();	1X,1E:WiFiClient client = server.available();
	1X:Serial.println(”new client”);	1X:Serial.println(”new client”);
	1E:while(!client.available())	1E:while(!client.available())
	1X:delay(1);	1X:delay(1);
	1E:String request = client.readStringUntil(’”);	1E:String request = client.readStringUntil(’”);
	1X:Serial.println(request);	1X:Serial.println(request);
	1E: float reading = analogRead(LM35);	**UNIDENTIFIED BY THE TOOL**
	1X:client.println(”HTTP/1.1 200 OK”);	1X:client.println(”HTTP/1.1 200 OK”);
	1X:client.println(”Content-Type: text/html”);	1X:client.println(”Content-Type: text/html”);
	1X:client.println(””);	**UNIDENTIFIED BY THE TOOL**
	1X: client.println(”<!DOCTYPE HTML>”);	1X: client.println(”<!DOCTYPE HTML>”);
	1X: client.println(”<html>”);	1X: client.println(”<html>”);

	1X:client.print(”Celcius temperature =”);	1X:client.print(”Celcius temperature =”);
	1X:client.print(temperatureC);	1X:client.print(temperatureC);
	1X:client.println(”Farenheight temperature =”);	1X:client.println(”Farenheight temperature =”);
	1X:client.print(temperatureF);	1X:client.print(temperatureF);
	1X:client.println(”Updated”);	1X:client.println(”Updated”);
	1X: client.print(”Not Updated”);	1X: client.print(”Not Updated”);
	1X:client.println(” ”);	1X:client.println(” ”);
	1X:client.println(”<button>Update Temperature</button> ”);	1X:client.println(”<button>Update Temperature</button> ”);
	1X:client.println(”</html>”);	1X:client.println(”</html>”);
	1X:delay(1);	1X:delay(1);
	1X:Serial.println(”Client disconnected”);	1X:Serial.println(”Client disconnected”);
	1X:Serial.println(””);	1X:Serial.println(””);

Table 3: Summary of testing results

Summary of testing results			
Functional processes	Size obtained manually (in CFP)	Size obtained automatically (in CFP)	Measurement Accuracy
Setup()	20	19	95 %
loop()	26	24	92.3 %
Total Functional Size of the software	46	43	93.4 %

```
#include <ESP8266WiFi.h>

const char* ssid = "Your SSID";
const char* password = "Your Wifi Password";

int LM35= A0;     //Analog channel A0 as used to measure temperature
WiFiServer server(80);

void setup() {
 Serial.begin(115200);
 delay(10);
 // Connect to WiFi network
 Serial.println();
 Serial.println();
 Serial.print("Connecting to ");
 Serial.println(ssid);

 WiFi.begin(ssid, password);

 while (WiFi.status() != WL_CONNECTED) {
  delay(500);
  Serial.print(".");
 }
 Serial.println("");
 Serial.println("WiFi connected");

 // Start the server
 server.begin();
 Serial.println("Server started");

 // Print the IP address on serial monitor
 Serial.print("Use this URL to connect: ");
 Serial.print("http://");    //URL IP to be typed in mobile/desktop browser
 Serial.print(WiFi.localIP());
 Serial.println("/");

}

void loop() {
 // Check if a client has connected
 WiFiClient client = server.available();
 if (!client) {
  return;
 }

 // Wait until the client sends some data
 Serial.println("new client");
 while(!client.available()){
  delay(1);
 }

 // Read the first line of the request
 String request = client.readStringUntil('\r');
 Serial.println(request);
 client.flush();

 // Match the request
 float temperatureC;
 float temperatureF;
 int value = LOW;
 if (request.indexOf("/Tem=ON") != -1)  {
    float reading = analogRead(LM35);     //Analog pin reading output voltage by Lm35
    float temperatureC=LM35/3.1;         //Finding the true centigrate/celsius temperature
    //Serial.println("CENTI TEMP= ");
    //Serial.print(temperatureC);               //Print centigrade temperature on Serial Monitor
    float temperatureF= ((temperatureC ) * 9.0 / 5.0) + 32.0;
    //Serial.println("FARE TEMP= ");
    //Serial.print(temperatureF);              //Print farenheight temperature on Serial Monitor

  value = HIGH;
 }

 // Return the response
 client.println("HTTP/1.1 200 OK");
 client.println("Content-Type: text/html");
 client.println(""); //  do not forget this one
 client.println("<!DOCTYPE HTML>");
 client.println("<html>");

 client.print("Celcius temperature =");
 client.print(temperatureC);
 client.println("Farenheight temperature =");
 client.print(temperatureF);

 if(value == HIGH) {
  client.println("Updated");
 } else {
  client.print("Not Updated");
 }
 client.println("<br><br>");
 client.println("<a href=\"/Tem=ON\"\"><button>Update Temperature</button></a><br />");
 client.println("</html>");
 delay(1);
 Serial.println("Client disonnected");
 Serial.println("");

}
```

Figure 10: Temperature sensing code real-life example

5. Conclusion

The Internet of Things (IoT) is a collection of Edge devices interconnected via the internet. Edge devices process data in real-time close to the physical environment in which they are deployed. Functional Size Measurement is an omnipotent tool to measure functionality provided by a software and obtain technical indicators early in the design phase.

This paper presented a prototype tool to automate the COSMIC Functional size measurement of IoT Edge devices. The measurement automation is based on the rules proposed in [2]. The prototype tool proposed has two modules. The first Module is responsible for retrieving the source code from the IoT Edge device. The second Module measures automatically the COMSIC functional size of the source code retrieved. Four different approaches were proposed and discussed to automatically retrieve the source code from the IoT Edge device. Some approaches are more adapted to Arduino platforms and others could be used with ESP platforms.

Tests of the prototype tool proposed were carried out to validate its functionality: the tool was first tested of a very simple Arduino example with an accuracy of 100% and next it was tested on an ESP based real-life example with an accuracy of 93.4%.

In the future, we intend to improve Module 2 by including more Arduino and ESP libraries to the measurement rules to identify the Data movements concerned and measure their size in

CFP accordingly. In addition, a more user-friendly interface for our JavaScript client will be proposed by integrating it to the tool proposed in [8].

References

[1] Abran, Alain, Charles Symons, Christof Ebert, Frank Vogelezang, and Hassan Soubra. ”Measurement of software size: Contributions of cosmic to estimation improvements.” In The International Training Symposium, At Marriott Bristol, United Kingdom, pp. 259-267. 2016.

[2] Soubra, Hassan and Abran, Alain, ”Functional size measurement for the internet of things (IoT) an example using COSMIC and the arduino open-source platform.” Proceedings of the 27th International Workshop on Software Measurement and 12th International Conference on Software Process and Product Measurement. 2017.

[3] Soubra, Hassan, Laury Jacot and Steven Lemaire. Manual and automated Functional Size Measurement of an aerospace Realtime Embedded system: a case study based on SCADE and on COSMIC ISO 19761. 2015.

[4] H. Soubra, A. Abran, and M. Sehit, Functional Size Measurement for Processor Load Estimation in AUTOSAR, Springer International Publishing (Switzerland), Lecture Notes on Business Information Processing – LNBIP230, pp. 1-16, 2015.

[5] K. Khattab, H. Elsayed, and H. Soubra, Functional Size Measurement of Quantum Computers Software, in: Proceedings of the 31st IWSM-Mensura, Izmir, Turkey, 2022.

[6] Soubra, Hassan, Yomna Abufrikha, and Alain Abran. ”Towards Universal COSMIC Size Measurement Automation.” In IWSM-Mensura. 2020.

[7] Darwish, Ahmed, and Hassan Soubra. ”COSMIC Functional Size of ARM Assembly Programs.” In IWSM-Mensura. 2020.

[8] H. Soubra,Towards Universal IoT Metrics Automation, in: Proceedings of the 31st IWSM-Mensura, Izmir, Turkey, 2022.

[9] Moulla, Donatien Koulla, Ernest Mnkandla, and Alain Abran. Systematic Literature Review of IoT Metrics. Applied Computer Science 19 (1), 64-81. https://doi.org/10.35784/acs-2023-05. 2023.

[10] https://www.engineersgarage.com/lm35-with-nodemcu/.

[11] Soubra, Hassan, Alain Abran, and Amar Ramdane-Cherif. ”Verifying the accuracy of automation tools for the measurement of software with COSMIC–ISO 19761 including an AUTOSAR-based example and a case study.” In 2014 Joint Conference of the International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement, pp. 23-31. IEEE, 2014.