Genetic Programming-Based Code Generation for Arduino

Wildor Ferrel[1]
Departamento Académico de Ingeniería Electrónica
Universidad Nacional de San Agustín de Arequipa
Arequipa, Perú

Luis Alfaro[2]
Departamento Académico de Ingeniería de Sistemas
Universidad Nacional de San Agustín de Arequipa
Arequipa, Perú

***Abstract*—This article describes a methodology for writing the program for the Arduino board using an automatic generator of assembly language routines that works based on a cooperative coevolutionary multi-objective linear genetic programming algorithm. The methodology is described in an illustrative example that consists of the development of the program for a digital thermometer organized on a circuit formed by the Arduino Mega board, a text LCD module, and a temperature sensor. The automatic generation of a routine starts with an input-output table that can be created in a spreadsheet. The following routines have been automatically generated: initialization routine for the text LCD screen, routine for determining the temperature value, routine for converting natural binary code into unpacked two-digit BCD code, routine for displaying a symbol on the LCD screen. The application of this methodology requires basic knowledge of the assembly programming language for writing the main program and some initial configuration routines. With the application of this methodology in the illustrative example, 27% of the program lines were written manually, while the remaining 73% were generated automatically. The program, produced with the application of this methodology, preserves the advantage of assembly language programs of generating machine code much smaller than that generated by using the Arduino programming language.**

Keywords*—Genetic programming; Arduino mega board; multi-objective linear genetic programming; cooperative coevolutionary algorithm; automatic generation of programs; Arduino based thermometer

I. Introduction

Arduino is an open-source, free hardware, microcontroller-based electronic board that has a series of analog and digital pins that can be used to connect sensors, peripheral devices, or actuators [1]. The program, which the user stores in the Arduino memory, allows this board to perform various functions such as controller functions, measurement instrument functions, communications equipment functions, etc.

Due to their relatively easy-to-use hardware and software, Arduino boards, in addition to being applied to everyday tasks, are also being applied in scientific instruments such as the measurement of transendothelial/epithelial resistance [2], in the analysis of the production volume of breast milk [3], in the measurement of the methane content of biogas samples [4], etc., which means that there is significant interest from many users and researchers in the use of this free software platform.

There are various Arduino boards such as Arduino Uno, Arduino Mega, Arduino Nano, Arduino Leonardo, Arduino Micro, etc. Because it is one of the fastest Arduino boards on the market [5] and due to the amount of digital and analog pins it has, in this work the Arduino Mega board is used, which is built based on the ATmega 2560 microcontroller with AVR architecture. The main features of the Arduino Mega 2560 are: it has 54 digital input/output pins, 16 analog inputs, 4 UARTs (serial ports in hardware), a 16 MHz crystal oscillator, a USB connection.

Arduino board programming is done through free software that is now accessible On-Line: Arduino Web Editor [6]. To program this board, knowledge of the Arduino programming language is required, which is similar to C++ [7]. In our research work, a program development methodology is proposed for the Arduino Mega board based on program synthesis. Program synthesis aims to automatically produce a program from a specification called "user intent". There are many ways to represent the specification, it can be a sketch [8], a sequence [9], or a table of input-output examples [10][11]. In our work, the starting point for the automatic generation of a program is an input-output table.

The rest of this paper is organized as follows: Section II discusses the related work. In Section III, the theoretical foundations of the proposed methodology are summarized. In Section IV, the fitness evaluation algorithms are detailed. Section V describes the automatic generation of the illustrative example routines. In Section VI, the experimental work carried out is described. In Section VII, the conclusions and recommendations are presented.

II. Related Work

In the classification of program synthesis techniques, genetic programming is within the group of stochastic search techniques [12]. There are efforts to use genetic programming in general-purpose synthesizers [13] and microcontroller program synthesizers. Genetic programming that evolves programs in an imperative language is called linear genetic programming. There are two types of linear genetic programming [14]: a machine code genetic programming, where each instruction is directly executable by the CPU, and an interpreted linear genetic programming. Due to the large difference in the clock frequency of the computer and the

Arduino Mega microcontroller, we use interpreted linear genetic programming.

Dias and Pacheco [15] proposed to apply linear genetic programming in the automatic synthesis of programs in assembly language for the PIC 18F452 microcontroller, showing as examples, the implementation of optimal time control strategies in two cases: in the cart-centering problem and in the problem of balancing an inverted pendulum in a minimum amount of time. The authors in [16] used linear genetic programming in the automatic synthesis of assembly program for the PIC 18F452 microcontroller for optimized control of a water bath plant. In both research works [15][16], a classical evolutionary algorithm is used, and two simulators for the fitness evaluation of an individual have been organized: a simulator of the microcontroller CPU; and, based on its dynamic equations, a simulator of the plant.

Serruto and Casas [17] have proposed to apply linear genetic programming with multi-objective optimization for the automatic synthesis of programs for the AT89S52 microcontroller of 8051 architecture. The generated programs are: 4x3 matrix keyboard scan program, initialization program of the text LCD screen, and a character display program on the LCD screen. For the first case, the authors have proposed the fitness evaluation based on an exhaustive search of the bits of the result, and for the last two cases, the fitness evaluation is carried out by comparing the timing diagrams produced by the genetic program with the target timing diagrams. Serruto and Casas in [18] improved the program generator by introducing the cooperative coevolutionary algorithm, which allowed generating the 4x4 matrix keyboard scanning program and the character display program with a better hit rate. To apply the cooperative coevolutionary algorithm, a machine code program is considered to be made up of program segments, and each segment corresponds to a species. To calculate the fitness of an individual (program segment) a complete program is formed with the individual and the representatives of the other species.

In the proposed work, a methodology of programming of the Arduino Mega board using cooperative coevolutionary multi-objective linear genetic programming is described. The application of the methodology is shown in an illustrative example of developing the program for a digital thermometer circuit based on the Arduino Mega board.

III. Theoretical Fundamentals

A. Circuit with the Arduino Mega Board

The program that will be developed using genetic programming will run on the Arduino Mega board, which will be part of a circuit that will also include other devices connected to the digital or analog pins of the board. This circuit is a system based on the ATmega 2560 microcontroller. In Fig. 1 we show an example of a circuit made up of the Arduino Mega board, a text LCD module connected to digital pins, and a temperature sensor connected to an analog pin. The details of the connection are given in Table I. The program that will be elaborated, following the proposed methodology, as an illustrative example, will allow the circuit of Fig. 1 to function as a thermometer. The concepts on which the proposed methodology is based are inductive programming, linear genetic programming, multi-objective optimization, and cooperative coevolution. Next, these concepts are formulated adapting them to the problem of the synthesis of programs for a microcontroller.

B. Inductive Programming

In the inductive programming method, the starting point is an input/output table [11], from which, the inductive programming technique allows generating a program that makes each input correspond to the output given in the table, and also extrapolates values for other inputs. An input-output table is used in some functions in everyday spreadsheets (Flash-Fill in Microsoft Excel), in which a string processing program is automatically generated, from one or more examples provided by the user [10]. In [19] a formulation of the problem for programming-by-example is shown: Given a set of *M* input-output examples (desired input-output table):

$$(\mathrm{E}_0, S_0), (\mathrm{E}_1, S_1), \ldots, (\mathrm{E}_{M-1}, S_{M-1})$$

a *P* program must be found that performs all the transformations correctly:

$$P(\mathrm{E}_0) \rightarrow S_0;\ P(\mathrm{E}_1) \rightarrow S_1; \ldots;\ P(\mathrm{E}_{M-1}) \rightarrow S_{M-1}$$

To find the *P* program, in the proposed work a multi-objective cooperative coevolutionary linear genetic programming algorithm is used.

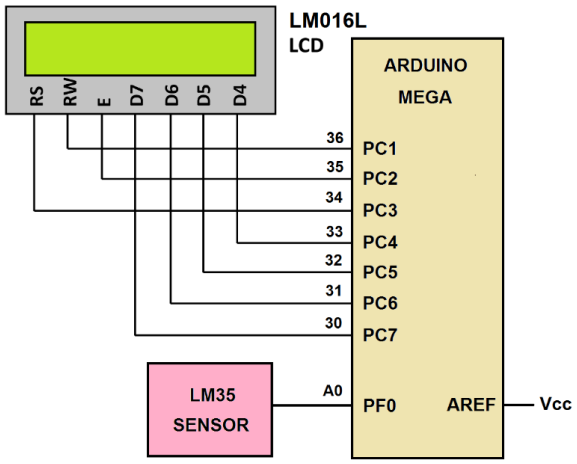

Fig. 1. Circuit Example based on the Arduino Mega.

TABLE I. Connections in the Circuit Example based on the Arduino Mega

Device	Device Line	Arduino Mega Pin	ATmega 2560 Microcontroller Pin
LM35 sensor	Output	A0	PF0/ADC0
LM016L LCD Module	RS	34	PC3/A11
	E	35	PC2/A10
	RW	36	PC1/A9
	D4	33	PC4/A12
	D5	32	PC5/A13
	D6	31	PC6/A14
	D7	30	PC7/A15

C. Linear Genetic Programming

Linear genetic programming (LGP) is the branch of genetic programming that evolves sequences of instructions from an imperative programming language or machine language [20]. In the automatic generation of routines for Arduino Mega, a subset of the ATmega 2560 microcontroller instruction set is used [21][22]. For the generation of programs, which do not interact with the input/output ports, the instructions in Table II are used. When generating programs that produce timing diagrams on Port C, in addition to the instructions in Table II, the instructions in Table III must be used. If the timing diagrams are generated on another port, then in the instructions in Table III, the operand "PORT C" must be changed to the corresponding Port.

TABLE II. INSTRUCTIONS USED IN THE SYNTHESIS OF PROGRAMS

Instruction	Instruction	Instruction	Instruction
NOP	AND R0,R20	INC R20	ASR R1
ADD R0,R1	AND R1,R0	DEC R0	ASR R20
ADD R0,R20	AND R1,R20	DEC R1	SWAP R0
ADD R1,R0	AND R20,R0	DEC R20	SWAP R1
ADD R1,R20	AND R20,R1	TST R0	SWAP R20
ADD R20,R0	ANDI R20,K	TST R1	BST R0,b
ADD R20,R1	OR R0,R1	TST R20	BST R1,b
ADC R0,R1	OR R0,R20	CLR R0	BST R20,b
ADC R0,R20	OR R1,R0	CLR R1	BLD R0,b
ADC R1,R0	OR R1,R20	CLR R20	BLD R1,b
ADC R1,R20	OR R20,R0	SER R20	BLD R20,b
ADC R20,R0	OR R20,R1	MUL R0,R1	SEC
ADC R20,R1	ORI R20,K	MUL R1,R20	CLC
SUB R0,R1	EOR R0,R1	MUL R0,R20	SEN
SUB R0,R20	EOR R0,R20	MULS R20,R20	CLN
SUB R1,R0	EOR R1,R0	LSL R0	SET
SUB R1,R20	EOR R1,R20	LSL R1	CLT
SUB R20,R0	EOR R20,R0	LSL R20	SEH
SUB R20,R1	EOR R20,R1	LSR R0	CLH
SUBI R20,K	COM R0	LSR R1	MOV R0,R1
SBC R0,R1	COM R1	LSR R20	MOV R0,R20
SBC R0,R20	COM R20	ROL R0	MOV R1,R0
SBC R1,R0	NEG R0	ROL R1	MOV R1,R20
SBC R1,R20	NEG R1	ROL R20	MOV R20,R0
SBC R20,R0	NEG R20	ROR R0	MOV R20,R1
SBC R20,R1	CBR R20,K	ROR R1	LDI R20,K
SBCI R20,K	INC R0	ROR R20	
AND R0,R1	INC R1	ASR R0	

TABLE III. ADDITIONAL INSTRUCTIONS USED IN THE SYNTHESIS OF PROGRAMS

Instruction
OUT PORTC,R0
OUT PORTC,R1
OUT PORTC,R20
SBI PORTC,b
CBI PORTC,b

D. ATmega 2560 Microcontroller used Registers

In the microcontrollers of the AVR architecture [23], there are 32 general-purpose registers with names R0, R1, R2,…, R31. These registers are mapped to memory occupying the lowest 32 addresses. The working registers used in the evolutionary process, in genetic programs, are R0, R1, and R20. Register R20 has been included to use the instructions with immediate addressing ANDI, ORI, SUBI, LDI that only work with registers R16 to R31. In code conversion problems, at the completion stage of the program, K registers are used from register R5 to register R(K + 4), where K is the number of bits in the result. In the generation of programs that interact with an input-output Port, the PORT, DDR, and PIN registers of this Port are used. All genetic programs use the SREG register that stores the status of the program.

E. Multi-Objective Evolutionary Optimization

In this work, to find the program P, multi-objective optimization is used. For this purpose, we form K objective functions. In the generation of code conversion routines, each objective function corresponds to one bit of the objective code. In the generation of routines that produce timing diagrams in an input/output Port, each objective function corresponds to a pin of the Port. Next, we formulate the multi-objective optimization problem, adapted to the synthesis problem of a microcontroller program, based on the formulation given in [24]:

Maximize

$$f(P) = (f^0(P), f^1(P), \dots, f^{K-1}(P))$$

Subject to condition

$$P \in U$$

where $P = [I_0, I_1, \dots, I_{N-1}]$ is a genetic program in machine language, I_i is an instruction from Table II or Table III, U is the feasible set, $f(P) = (f^0(P), f^1(P), \dots, f^{K-1}(P))$ is the vector of objective functions.

The target vector $f(P)$ indicates the degree of similarity of the input-output table, generated after running the genetic program P, and the desired input-output table. The multi-objective genetic programming algorithm aims that the generated table is equal to the desired one. When this occurs, the objective functions have the highest value. Thus, in the microcontroller program synthesis problem, studied in this work, multi-objective optimization seeks to maximize the vector of objective functions.

F. Multi-Objective Cooperative Coevolutionary Linear Genetic Programming Algorithm (MOCCLGPA)

The automatic generation of a routine for Arduino is a complex problem, so in addition to multi-objective optimization, in the proposed work, a cooperative coevolutionary algorithm is used. Cooperative coevolutionary algorithms are based on decomposing the problem into subcomponents also called "species" that evolve in collaboration with each other [25].

To apply cooperative coevolution in program synthesis, a machine code program is considered to be made up of program segments, and each segment corresponds to a species. The calculation of the fitness of an individual of a species is carried out by previously forming a complete solution (genetic program) combining the individual (program segment) with the selected representatives of the other species as detailed in [18]. In the proposed work, the number of species is 10 and the representatives are the two best individuals of each species. The Multi-Objective Cooperative Coevolutionary Linear Genetic Programming Algorithm (MOCCLGPA) used in the proposed work is Algorithm 1 taken and modified from [18].

In Algorithm 1, each time the fitness of a genetic program is evaluated, the number of evaluations *n* is increased, the values of *f, fsum, fop,* and *fopsum* are determined, and the *Pbest* program is updated if a better one was found. The sorting of a list is done according to the value of the scalar *fopsum*. The insertion of an individual in a list is carried out using the vector *fop* and the concept of Pareto dominance. The algorithms for the selection of representatives, selection of parents, and variation are described in [18]. At the end of Algorithm 1, the synthesized *Pbest* program, after the completion operation, becomes the generated program.

Algorithm 1. Multi-objective cooperative coevolutionary linear genetic programming algorithm

S_t is a species (program segment) $(t = 0, \dots, T-1)$
Each species has two non-Pareto-dominated fronts P_{1t} and P_{2t} and two temporary lists Q_t and D_t
Pbest is the best program found so far
Nlimit is the limit number of evaluations

1. **for** $t = 0$ **to** $T - 1$ **do**
2. Random generation of P_{1t} and P_{2t}
3. Selection of representatives of S_t
4. **end for**
5. **for** $t = 0$ **to** $T - 1$ **do**
6. Fitness evaluation of each individual of P_{1t} and P_{2t}
7. Sorting of P_{1t}
8. Sorting of P_{2t}
9. **end for**
10. **while** *n* < *Nlimit* **do**
11. **for** $t = 0$ **to** $T - 1$ **do**
12. Selection of representatives of S_t
13. $Q_t \leftarrow$ Parent selection (P_{1t}, P_{2t})
14. $Q_t \leftarrow$ Variation (Q_t)
15. **end for**
16. **for** $t = 0$ **to** $T - 1$ **do**
17. Fitness evaluation of each individual of P_{1t}, P_{2t} and Q_t
18. **end for**
19. **for** $t = 0$ **to** $T - 1$ **do**
20. Insertion of the best from Q_t in P_{1t} and those discarded in D_t
21. Insertion of the best from D_t in P_{2t}
22. Sorting of P_{1t}
23. Sorting of P_{2t}
24. **end for**
25. **end while**

G. Structure of the Automatic Routine Generator

The automatic routine generator consists of an evolutionary program synthesizer and a program completion block. In turn, the synthesizer is made up of the program that implements the MOCCLGPA algorithm and a simulator of the ATmega 2560 microcontroller CPU. The fitness evaluation, in the MOCCLGPA algorithm, and the completion of the program, is carried out according to the type of routine that is generated.

IV. Fitness Evaluation

A. Fitness Evaluation in the Conversion of One Code to Another

Authors in [17] proposed to evaluate the fitness through an exhaustive search of each bit of the binary representation of the output. In the algorithm description, the output value corresponding to the E_j input, expressed in binary, is represented as $\left(S_j^{K-1}, \dots S_j^t, \dots S_j^0\right)$. In this way, each output bit S^t is a combinational function.

The algorithm uses the *RVM* register value matrix that contains the value of the three working registers R0, R1, and R20, at the end of each I_i instruction of the *P* program, and for each value of the E_j input, therefore, the matrix has three dimensions. Each element of the *RVM* matrix is a binary value that we represent RVM_{ij}^b where the index *i* corresponds to the instruction number in the genetic program, *j* corresponds to the input E_j in the input-output table, and *b* is the number of bit in the range 0 to 23 (bit numbers 0 to 7 correspond to register R0, 8 to 15 to R1, and 16 to 23 to R20). Algorithm 2 describes the fitness evaluation of a genetic program in the generation of a code conversion program.

For each combinational function S^t corresponding to an output bit, with the following formula, the most similar combinational function is found in *RVM* matrix:

$$f = (f^0, f^1, \dots, f^{K-1}) \; \forall \; t = 0, \dots, K-1$$
$$f^t = \max_{\substack{0 \le b \le 23 \\ 0 \le i \le N-1}} \sum_{j=0}^{M-1} \left\{ RVM_{ij}^b \odot S_j^t \right\} \quad (1)$$

where the operator ⊙ represents the nor-exclusive operation. The result of this operation, which can be "0" or "1", is then considered an integer value that participates in the arithmetic sum represented by the summation symbol.

Algorithm 2. Fitness evaluation of a genetic program of code conversion

IOT is the input-output table with the outputs in binary representation:

$$\left(\mathrm{E}_0, (S_0^{K-1}, \dots S_0^t, \dots S_0^0)\right),$$
$$\dots$$
$$\left(\mathrm{E}_j, (S_j^{K-1}, \dots S_j^t, \dots S_j^0)\right),$$
$$\dots$$
$$\left(\mathrm{E}_{M-1}, (S_{M-1}^{K-1}, \dots S_{M-1}^t, \dots S_{M-1}^0)\right),$$

$fmax = K \cdot M$ is the maximum value of *fsum*.
$P = [I_0, I_1, \dots, I_i, \dots, I_{N-1}]$ is the genetic program to evaluate.
Pbest is the best program found so far of *NEbest* size with fitness *fsumbest*
RVM is the array of register values

1. Clear (*RVM*)
2. **for** each E_j of *IOT* **do**
3. SREG ← 80H
4. R0 ← E_j
5. R1 ← E_j
6. R20 ← E_j
7. **for** *i* = 0 **to** *N*-1 **do**
8. Execute (I_i)
9. RVM_{ij} ← (*R20*) (*R1*) (*R0*)
10. **end for**
11. **end for**
12. Calculate *f*, *fsum*, *BLM*, *NE*, *fop* and *fopsum* using formulas (1), (2), (3), (4), (5) and (6) respectively
13. **if** (*fsum* > *fsumbest*) **or** ((*fsum* = *fmax*) **and** (*NE* < *NEbest*)) **then**
14. *Pbest* ← *P*; *NEbest* ← *NE*; *fsumbest* ← *fsum*
15. **end if**
16. Return *BLM*, *NE*, *fop*, *fopSum*

To find out if the maximum value has been reached, the sum of all the elements of fitness *f* is calculated:

$$fsum = \sum_{t=0}^{K-1} f^t \quad (2)$$

The best bit location for each element of fitness *f* is stored in the bit location matrix (*BLM*):

$$BLM = [(i^0, b^0), \dots, (i^t, b^t), \dots, (i^{K-1}, b^{K-1})] \quad (3)$$

The effective size of the program is:

$$NE = 1 + \max_{0 \le t \le K-1} (i^t) \quad (4)$$

For smaller programs to have better fitness we use the formula:

$$f_{op} = \left(f_{op}^0, f_{op}^1 \dots, f_{op}^{K-1}\right) \forall\, t = 0, \dots, K-1$$

$$f_{op}^t = f^t - \alpha \cdot NE \quad (5)$$

α has been assigned a value of 0.001.

The fitness f_{op} is used in insertion operations of individuals on Pareto fronts. On Pareto fronts the sorting operations are based on the scalar value:

$$fopSum = \sum_{t=0}^{K-1} f_{op}^t \quad (6)$$

B. Fitness Evaluation in the Generation of Timing Diagrams without Input Values

We represent the timing diagrams of the pins of a microcontroller Port as a string of decimal values in which two consecutive values are not equal.

The algorithm for fitness evaluation of a program that generates timing diagrams proposed in [17], adapted to the AVR architecture, is shown in Algorithm 3, where *L* represents the number of values in the timing diagrams and *K* is the number of pins where the timing diagrams are generated. When the generated timing diagrams *G* are updated, a new value is recorded only if it is different from the previous one by at least one bit. Each component of the fitness vector corresponds to the timing diagram of a Port pin. After the execution of the genetic program, the generated timing diagrams (*G*) are compared with the target timing diagrams (*S*) in binary representation with the formula:

$$f = (f^0, f^1, \dots, f^{K-1}) \text{ for all } p = 0, \dots, K-1$$

$$f^p = \sum_{d=0}^{L-1} \left\{ (L-d)\left(G_d^p \odot S_d^p\right) \right\} \quad (7)$$

where (*L* - *d*) is a weight assigned to *d* time. The first bits have greater weight compared to the last ones. This allows that, in the evolutionary process, the correct values are established, with greater probability, starting with the previous times and ending with the later ones, to improve the speed of convergence of the algorithm.

During the execution of the program, a *VNI* vector is formed with the indices of the instructions, whose execution has produced a change in some Port pin:

Algorithm 3. Fitness evaluation of a genetic program for the generation of timing diagrams without input values

$S = (S_0, , \dots S_d, \dots, S_{L-1},)$ are the target timing diagrams.
$G = (G_0, , \dots G_d, \dots, G_{L-1},)$ are the generated timing diagrams.
Each value of S is represented in binary $S_d = [S_d^{K-1} \quad \dots \quad S_d^0]$
Each value of G is represented in binary $G_d = [G_d^{K-1} \quad \dots \quad G_d^0]$
$P = [I_0, I_1, \dots, I_i, \dots, I_{N-1}]$ is the genetic program to be evaluated
$fmax = K \cdot (L+1) \cdot L/2$ is the maximum value of *fsum*.
Pbest is the best program found so far of *NEbest* size with fitness *fsumbest*

1. SREG ← 80H
2. PORT ← 00H
3. DDR ← FFH
4. R0 ← 0
5. R1 ← 0
6. R20 ← 0
7. Clear (*G*)
8. Clear (*VNI*)
9. **for** *i* = 0 **to** *N* - 1 **do**
10. Execute (*Ii*)
11. Update (*G*)
12. Update (*VNI*)
13. **end for**
14. Calculate *f*, *fsum*, *NE*, *fop* y *fopsum* using formulas (7), (2), (9), (5) and (6) respectively
15. **if** (*fsum* > *fsumbest*) **or** ((*fsum* = *fmax*) **and** (*NE* < *NEbest*)) **then**
16. *Pbest* ← *P*; *NEbest* ← *NE*; *fsumbest* ← *fsum*
17. **end if**
18. Return *NE*, *fop*, *fopSum*

$$VNI = [i_0, i_1, \dots, i_d, \dots, i_{L-1}] \quad (8)$$

The effective size of the program is:

$$NE = 1 + \max_{0 \le d \le L-1}(i_d) \quad (9)$$

C. Fitness Evaluation in the Generation of Timing Diagrams According to Input Values

Algorithm 4 describes the fitness evaluation of a program that generates timing diagrams according to input values. In the algorithm, L represents the number of values in each timing diagram, M is the number of input values, and K is the number of pins where the timing diagrams are generated. To compare the generated timing diagrams with those desired for the E_j input, the formula is:

$$f_j = \left(f_j^0, f_j^1, \dots, f_j^{K-1}\right) \ \forall p = 0, \dots, K-1$$

$$f_j^p = \sum_{d=0}^{L-1}\left\{(L-d)\left(G_{j,d}^p \odot S_{j,d}^p\right)\right\} \quad (10)$$

The fitness vector f is equal to the sum of all the fitness vectors f_j. For the fitness vector f the sum is calculated:

$$f = (f^0, f^1, \dots, f^{K-1})$$

$$f_{sum} = \sum_{p=0}^{K-1} f^p \quad (11)$$

Algorithm 4. Fitness evaluation of a genetic program for the generation of timing diagrams according to input values.

In the input-output table (*IOT*) each output is a sequence of values:

$$\left(E_0, (S_{0,0}, \dots S_{0,d}, \dots S_{0,L-1})\right),$$
$$\dots$$
$$\left(E_j, (S_{j,0}, \dots S_{j,d}, \dots S_{j,L-1})\right),$$
$$\dots$$
$$\left(E_{M-1}, (S_{M-1,0}, \dots S_{M-1,d}, \dots S_{M-1,L-1})\right),$$

Each value in the sequence is represented in binary:

$$S_{j,d} = [S_{j,d}^{K-1} \quad \dots \quad S_{j,d}^0]$$

$P = [I_0, I_1, \dots, I_i, \dots, I_{N-1}]$ is the genetic program to be evaluated

$fmax = M \cdot K \cdot (L+1) \cdot \frac{L}{2}$ is the maximum value of *fsum*.

Pbest is the best program found so far of *NEbest* size with fitness *fsumbest*

1. Clear (*f*)
2. **for** each input E_j **do**
3. DDR ← FFH
4. PORT ← 00H
5. SREG ← 80H
6. R0 ← E_j;
7. R1 ← E_j;
8. R20 ← E_j;
9. Clear (*Gj*);
10. Clear (*VNIj*)
11. **for** i = 0 to N-1 **do**
12. Execute (I_i)
13. Update (*Gj*)
14. Update (*VNIj*)
15. **end for**
16. f_j is calculated with the formula (10)
17. $f \leftarrow f + f_j$
18. **end for**
19. Calculate *fsum, NE, fop, fopsum* using formulas (11), (12), (13) and (14) respectively
20. **if** (*fsum* > *fsumbest*) **or** ((*fsum* = *fmax*) **and** (*NE* < *NEbest*)) **then**
21. *Pbest* ← *P*; *NEbest* ← *NE*; *fsumbest* ← *fsum*
22. **end if**
23. Return *NE, fop, fopsum*

As in the generation of timing diagrams without input values, for each input E_j there is a vector of indices to instructions:

$$VNI_j = \left[i_{j,0}, i_{j,1}, \dots, i_{j,d}, \dots, i_{j,L-1}\right]$$

based on which the scalar $NUI_j = \max_{0 \le d \le L-1}\left(i_{j,d}\right)$ is calculated which is the index of the instruction that produced the last change in the timing diagrams for E_j. Therefore, the effective size of the program is:

$$NE = 1 + \max_{0 \le j \le M-1}\left(NUI_j\right)$$

To prevent that, for different E_j inputs, the vectors VNI_j are different, and the program sizes are also different, based on all the vectors VNI_j, the *VDIF* vector of size *L* is formed, in which each element of position *d* is equal to the difference between the maximum value and the minimum value of all the values of that position in the vectors VNI_j:

$$VDIF_d = \max_{0 \le j \le M-1}\left(i_{j,d}\right) - \min_{0 \le j \le M-1}\left(i_{j,d}\right) \quad (12)$$

Then it is calculated:

$$DIFmax = \max_{0 \le d \le L-1}(VDIF_d),$$

When the evolutionary process ends, $DIFmax$ must be equal to 0. Using NE and $DIFmax$, the fitness vector f_{op} and its sum f_{opsum} are calculated with the following formulas:

$$f_{op} = \left(f_{op}^0, f_{op}^1 \dots, f_{op}^{K-1}\right)$$

$$f_{op}^p = f^p - \alpha \cdot (DIFmax + NE) \quad (13)$$

$$f_{opsum} = \sum_{p=0}^{K-1} f_{op}^p \quad (14)$$

V. Automatic Routine Generation for Arduino MEGA

A. Generation of the Routine for Determining the Temperature Value (ADC_BIN)

As can be seen in the circuit in Fig. 1, the LM35 temperature sensor is connected to the A0 analog input of the Arduino Mega, which in the ATmega 2560 microcontroller is an input to the analog-digital converter (ADC). The ADC converts the voltage provided by the sensor into an integer that we represent as *ADCvalue*.

If the reference voltage on the ADC is Vcc = 5V, and the resolution is 10 bits; then the ADC converts the voltage range from 0V to 5V, proportionally, in the integer range from 0 to 1023. When the LM35 temperature sensor is configured to produce 10mV per °C, to convert the output value of the ADC in temperature value, in degrees centigrade, the following formula is used:

$$Temperature(°C) = \frac{500 \cdot ADCvalue}{1024} \quad (15)$$

The LM35 sensor, as configured, allows a temperature measurement range of 0° C to 150°C. To simplify the input-output table, we set the temperature range from 0°C to 99°C. In a spreadsheet, formula (15) is calculated for all the ADC

output values in the range from 0 to 203, obtaining the column "Temperature (°C)" of Table IV. Then, these values are rounded to the nearest one, obtaining the column "Rounded Temperature". The input-output table for the routine generator is made up of the columns "ADC Value" (Input) and "Rounded Temperature" (Output) of Table IV.

The evolutionary process follows Algorithm 1 with the fitness evaluation described in Algorithm 2. In the completion stage, to the synthesized program *Pbest*, instructions are inserted and added with the information from the bit location matrix (*BLM*). For each (i^t, b^t) pair of *BLM*, after the instruction with index i^t, a MOV instruction is inserted to copy the register containing the bit b^t into a temporary register. For each bit, a different register is used from register R5 to register R(K + 4). At the end of the synthesized program *Pbest*, clear instruction of the R0 register is concatenated followed by pairs of BLD and BST instructions that copy the bits stored in the temporary registers into the R0 register. The completion of the program ends by inserting at the beginning, the instructions that place the initial values in the registers as indicated in Algorithm 2 on lines 3-6.

Before the execution of the ADC_BIN routine, the output value of the ADC must be placed in the R0 register. After executing the ADC_BIN routine, the temperature value rounded to the nearest one is obtained in the same R0 register as a number in natural binary code in the range from 0 to 99.

B. Generation of the Conversion Routine from a Natural Binary Number to Unpacked 2-Digit BCD (BIN_BCD2)

The routine converts a natural binary number in the range 0 to 99 to unpacked BCD code. The input-output table for the generator of this routine is shown in Table V, where for the numbers from 0 to 99, after separating the tens digit and the units digit, the operation *Tens* · 256 + *Units* is calculated placing the tens digit in the high byte and the units digit in the low byte that corresponds to the representation of the number in unpacked BCD code.

The evolutionary process follows Algorithm 1 with the fitness evaluation described in Algorithm 2. The completion of the program is done similarly to that carried out in the generation of the ADC_BIN routine, with the difference that now the most significant bits are placed in the register R1.

To invoke the BIN_BCD2 routine, the natural binary value is placed in register R0. The result of the conversion is obtained in registers R1 and R0. In register R1 the BCD tens digit is obtained and in register R0, the BCD units digit.

C. Generation of the "Home" Command Routine for the LCD Screen (LCD_HOME)

As can be seen in the circuit of Fig. 1 and Table I, the LCD text display module is connected to digital pins on the Arduino Mega board that correspond to Port C of the ATmega 2560 microcontroller. The "Home" routine performs the command to return the LCD screen to the initial state causing the cursor to return to the first left position of the first row. For the execution of this command, the microcontroller must generate in Port C the timing diagrams corresponding to the command with hexadecimal code 02 of the LCD screen controller.

According to the datasheet of the LCD module, in the timing diagrams, the RS and RW signals have value "0", while the enable signal E, for each nibble of the command, during a time interval has value "1" and in the next interval, the value "0". Therefore, in Table VI, at times 1 and 2 when the lines D7-D4 have a value of 0 (high nibble of the hexadecimal value 02), E = 1 at time 1, and E = 0 at time 2. It is important that at the beginning (time 0) and at the end (time 5) all signals are deactivated with a value of "0". The row "Port C" of Table VI is obtained by expressing the value of the Port C pins in decimal representation. The input-output table is made up of the row "Time" (Input) and the row "Port C" (Output) of Table VI.

The evolutionary process follows Algorithm 1 with the fitness evaluation of a genetic program according to Algorithm 3. When the stop condition is met, the synthesized program is *Pbest* of *NE* size.

The completion of the program consists of inserting at the beginning the instructions that place the initial values in the registers as indicated in Algorithm 3 on lines 1-6.

TABLE IV. OBTAINING THE INPUT-OUTPUT TABLE FOR THE GENERATION OF THE ADC_BIN ROUTINE

ADC value	Temperature (°C)	Rounded Temperature
0	0.0000	0
1	0.4883	0
2	0.9766	1
3	1.4648	1
4	1.9531	2
…		
201	98.1445	98
202	98.6328	99
203	99.1211	99

TABLE V. INPUT-OUTPUT TABLE FOR THE BIN_BCD2 ROUTINE GENERATION

Temperature	Unpacked BCD
0	0
1	1
2	2
…	
97	2311
98	2312
99	2313

TABLE VI. OBTAINING THE INPUT-OUTPUT TABLE FOR THE GENERATION OF THE "HOME" COMMAND ROUTINE

Time		0	1	2	3	4	5
D7-D4	PC7-PC4	0	0	0	2	2	0
RS	PC3	0	0	0	0	0	0
E	PC2	0	1	0	1	0	0
RW	PC1	0	0	0	0	0	0
-	PC0	-	-	-	-	-	-
Port C		0	4	0	36	32	0

In the generation of routines that produce timing diagrams on a Port, such as Port C, to overcome timing problems, after each instruction with the PORTC operand the CALL DELAY instruction is inserted.

D. Generating the LCD Screen Initialization Routine (LCD_INI)

According to the datasheet of the LCD screen module, to configure the module with a 4-bit interface and non-visible cursor, the microcontroller must generate timing diagrams on Port C corresponding to the hexadecimal sequence of commands: 33, 32, 28, 0C.

As in the generation of the "Home" routine, the RS and RW signals must have a value of "0" and the E signal for each nibble of command must have a value of "1" during a time interval, and then it must change to "0". In Table VII, the obtaining of the decimal sequence corresponding to the timing diagrams for the initialization of the LCD screen is shown. The input-output table consists of the row "Time" (Input) and the row "Port C" (Output) of Table VII.

The evolutionary process follows Algorithm 1 with the fitness evaluation according to Algorithm 3. When the stop condition is met, the synthesized program is *Pbest* of *NE* size. The program is completed with the insertion of the instructions that place the initial values in the registers, as indicated in lines 1-6 of Algorithm 3, at the beginning of the program. As in the generation of the LCD_HOME routine, after each instruction with the PORTC operand the CALL DELAY instruction is inserted.

E. Generating the Symbol Writing on the LCD Screen Routine (BCD1_LCD)

This routine allows the display of symbols on the LCD screen such as decimal digits from 0 to 9, space, decimal point, and capital letter C. To be displayed, the symbols must be sent to the LCD screen in ASCII code. As an example, Table VIII shows the obtaining of the timing diagrams in Port C to display the decimal digit "1". Proceeding similarly with all symbols, the timing diagrams are obtained in columns S0 to S5 of Table IX. The input-output table for the routine generator consists of the column "Code" (Input) and the columns from S0 to S5 (Output).

TABLE VII. OBTAINING THE INPUT-OUTPUT TABLE FOR THE GENERATION OF THE LCD_INI ROUTINE

Time		**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**
D7-D4	**PC7-PC4**	0	3	3	3	3	3	3	2	2	2	2	8	8	0	0	12	12	0
RS	**PC3**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
E	**PC2**	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	0
RW	**PC1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
-	**PC0**	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Port C		0	52	48	52	48	52	48	36	32	36	32	132	128	4	0	196	192	0

TABLE VIII. OBTAINING THE TIMING DIAGRAMS TO DISPLAY THE DECIMAL DIGIT "1" ON THE LCD SCREEN

Time		**0**	**1**	**2**	**3**	**4**	**5**
D7-D4	**PC7-PC4**	0	3	3	1	1	0
RS	**PC3**	0	1	1	1	1	0
E	**PC2**	0	1	0	1	0	0
RW	**PC1**	0	0	0	0	0	0
-	**PC0**	-	-	-	-	-	-
Port C		0	60	56	28	24	0

TABLE IX. OBTAINING THE INPUT-OUTPUT TABLE FOR THE BCD1_LCD ROUTINE GENERATION

Symbol	**Code**	**ASCII (Hex)**	**S0**	**S1**	**S2**	**S3**	**S4**	**S5**
0	0	30	0	60	56	12	8	0
1	1	31	0	60	56	28	24	0
2	2	32	0	60	56	44	40	0
3	3	33	0	60	56	60	56	0
4	4	34	0	60	56	76	72	0
5	5	35	0	60	56	92	88	0
6	6	36	0	60	56	108	104	0
7	7	37	0	60	56	124	120	0
8	8	38	0	60	56	140	136	0
9	9	39	0	60	56	156	152	0
Space	10	20	0	44	40	12	8	0
.	11	2E	0	44	40	236	232	0
C	12	43	0	76	72	60	56	0

The evolutionary process is developed according to Algorithm 1 with the fitness evaluation according to Algorithm 4. The completion of the program is carried out with the insertion, at the beginning of the program, of the instructions that perform the operations indicated in rows 3-8 of Algorithm 4. As explained before, after each instruction with the PORTC operand, the CALL DELAY instruction is inserted. To invoke the BCD1_LCD routine, the input value (symbol code) must be in the R0 register.

VI. Experimental Work

In the design of systems based on the Arduino Mega board, when the circuit includes devices such as LCD screen module, matrix keyboard, seven-segment indicators, sensors, and others, the methodology proposed in this article for developing the program consists of executing the following steps: 1) the global task is divided, if possible, into subtasks that are expressed through an input-output table; 2) the input-output tables are made with the help of a spreadsheet; 3) the routines described by input-output tables are automatically generated by the generator implemented based on the algorithms shown in previous sections; 4) the main program and routines that were not described with tables are manually written in assembly language. The main program has two parts: the first run only once and sets the system configuration, the second is a loop that repeats indefinitely. For the implementation of the digital thermometer, the main program, shown in Fig. 2(a), which was manually written, mainly contains register transfer instructions (MOV) and routine call instructions (CALL). The routines whose names are in green have been written manually, while the routines with names in red have been generated automatically. Within the infinite loop, the tasks are sequentially executed: place the LCD screen cursor in the initial state, read the ADC value, convert the ADC value into natural binary temperature value, convert the natural binary value into two-digit unpacked BCD code, display the tens digit, display the units digit, and display the letter C.

When the entire assembly language program is written manually, it is necessary to write routines to perform multiplication and/or division operations in the determination of temperature, and routines that perform the operations for handling the LCD screen. In the proposed methodology, these operations are carried out at the stage of making the input-output tables in a spreadsheet; therefore, the application of the proposed method is less complex compared to the manual writing of the entire program in assembly language.

Fig. 2(b) shows the program written in Arduino programming language for the circuit in Fig. 1, with the modification of connecting the RW signal of the LCD module to the ground. Because it is a high-level programming language, writing the program in this language is less complex than writing according to the proposed methodology. In the program in Fig. 2(b), the setup function configures the system and is executed only once; the loop function repeats indefinitely. In the loop function, the following operations are carried out: placing the LCD screen cursor in the first position of the first row, reading the microcontroller ADC value, calculating the temperature, displaying the temperature, and displaying the letter C.

```
.INCLUDE
"M2560DEF.INC"
LDI R16,HIGH(RAMEND)
OUT SPH,R16
LDI R16,LOW(RAMEND)
OUT SPL,R16
CALL SYS_INI
CALL ADC_INI
CALL LCD_INI
LOOP:
CALL ADC_HOME
CALL ADC_READ
MOV R0,R24
CALL ADC_BIN
CALL BIN_BCD2
MOV R2,R0
MOV R0,R1
CALL BCD1_LCD
MOV R0,R2
CALL BCD1_LCD
LDI R24,0X0C
MOV R0,R24
CALL BCD1_LCD
JMP LOOP
```

(a)

```
int ADCvalue;
float Temper;
#include <LiquidCrystal.h>
LiquidCrystal lcd(34, 35, 33, 32, 31, 30);
void setup() {
 lcd.begin(16, 2);
}
void loop() {
 lcd.setCursor(0, 0);
 ADCvalue = analogRead(0);
 Temper=round((500.0*ADCvalue)/1024);
 lcd.print(Temper,0);
 lcd.print("C");
}
```

(b)

Fig. 2. Program for the Digital Thermometer for the Arduino Mega Board. (a) In Assembly Programming Language following the Proposed Methodology (Only the main Program). (b) In Arduino Programming Language.

As can be seen, the two programs in Fig. 2 have similar structures. The programmer who follows the proposed methodology must have basic knowledge about the assembly programming language and the microcontroller architecture to manually write the main program, system configuration routine, ADC initialization routine, ADC reading routine, and a delay routine. All the other routines, which represent a significant percentage of the entire program (73%), have been automatically generated. The complete program, obtained with the proposed methodology, is shown in Fig. 3, with the routines automatically generated on a light blue background. Compiling this program generates a 614-byte machine code. On the other hand, writing the program in the Arduino programming language requires knowledge of this language, which is similar to the C++ language, and its compilation produces a 3788-byte machine code, which means that the use of the proposed methodology preserves the advantage of assembly language to produce smaller machine code compared to other programming languages.

In the stage of convergence and stability tests of the algorithms used in the proposed methodology, the generator of each routine has been executed 10 times. Table X shows the most important characteristics of the tests carried out in the generation of these routines. As can be seen in Table X, when the input-output table has a large number of rows (ADC_BIN routine), or when the table has several output columns (BCD1_LCD routine) the hit rate may be low (30%); this also occurs when the output has a low correlation or is not correlated with the input. For the generator of the BIN_BCD2 routine, the hit rate is the highest (100%) since the input and the output are quite correlated. The test parameters are: The size of the initial population of each species is 100, the number of species (program segments) is 10, the number of

representatives per species is 2, the initial size of the program segment has a minimum value of 2 and maximum value 4, the coefficient α in the fitness evaluation is 0.001. The programs have been compiled in AtmelStudio software and simulated in Proteus software using the Simulino Mega model. In the simulation or the implementation, if necessary, in the DELAY procedure, the waiting time can be varied by modifying the initial value placed in register R21 or R22.

```
.INCLUDE "M2560DEF.INC"
 LDI R16,HIGH(RAMEND)
 OUT SPH,R16
 LDI R16,LOW(RAMEND)
 OUT SPL,R16
 CALL SYS_INI
 CALL ADC_INI
 CALL LCD_INI
LOOP:
 CALL LCD_HOME
 CALL ADC_READ
 MOV R0,R24
 CALL ADC_BIN
 CALL BIN_BCD2
 MOV R2,R0
 MOV R0,R1
 CALL BCD1_LCD
 MOV R0,R2
 CALL BCD1_LCD
 LDI R24,0X0C
 MOV R0,R24
 CALL BCD1_LCD
 JMP LOOP
SYS_INI:
 IN R20,MCUCR
 ANDI R20,0XEF
 OUT MCUCR,R20
 LDI R20,255
 OUT DDRC,R20
 LDI R20,0X00
 OUT PORTC,R20
 RET
ADC_READ:
 LDS R22,ADCSRA
 ORI R22,(1<<ADSC)
 STS ADCSRA,R22
SALTO:
 LDS R22,ADCSRA
 SBRC R22,ADSC
 RJMP SALTO
 LDS R24,ADCL
 LDS R25,ADCH
 RET
DELAY:
 IN R21,SREG
 PUSH R21
 LDI R21,1;*
SAL1:
 LDI R22,10;*
SAL2:
 LDI R23,255
SAL3:
 DEC R23
 BRNE SAL3
 DEC R22
 BRNE SAL2
 DEC R21
 BRNE SAL1
 POP R21
 OUT SREG,R21
 RET
ADC_INI:
 LDI R22,0X00
 STS ADMUX,R22
 LDS R22,ADCSRB
 ANDI R22,0B11110111
 STS ADCSRB,R22
 LDI R22,0B10000111
 STS ADCSRA,R22
 ANDI R22,0B11011111
 STS ADCSRA,R22
 RET
ADC_BIN:
 LDI R20,0X80
 OUT SREG,R20
 MOV R20,R0
 MOV R1,R0
 LDI R20,250
 MUL R0,R20
 INC R1
 MOV R5,R1
 MOV R6,R1
 MOV R7,R1
 MUL R0,R20
 IN R21,SREG
 MOV R8,R1
 MUL R1,R20
 MOV R9,R1
 MOV R6,R1
 MOV R10,R1
 MOV R7,R1
 MOV R11,R1
 MOV R8,R1
 CLR R0
 CLR R0
 BST R5,1
 BST R5,0
 BLD R0,0
 BLD R0,0
 BST R6,2
 BST R6,1
 BLD R0,1
 BLD R0,1
 BST R7,3
 BST R7,2
 BLD R0,2
 BLD R0,2
 BST R8,4
 BST R8,3
 BLD R0,3
 BLD R0,3
 BST R9,5
 BST R9,7
 BLD R0,4
 BLD R0,4
 BST R10,6
 BST R10,7
 BLD R0,5
 BLD R0,5
 BST R11,7
 BST R11,7
 BLD R0,6
 BLD R0,6
 RET
BIN_BCD2:
 LDI R20,0X80
 OUT SREG,R20
 MOV R20,R0
 MOV R1,R0
 ROR R1
 MOV R5,R0
 MOV R9,R0
 MOV R10,R0
 MOV R11,R0
 MOV R12,R0
 INC R1
 LDI R20,51
 MUL R1,R20
 MUL R1,R20
 CLR R1
 BST R13,0
 BLD R1,0
 BST R14,1
 BLD R1,1
 BST R15,2
 BLD R1,2
 BST R16,3
 BLD R1,3
 RET
LCD_HOME:
 LDI R20,0XFF
 OUT DDRC,R20
 MOV R13,R1
 MOV R14,R1
 MOV R15,R1
 MOV R16,R1
 MUL R0,R20
 MUL R1,R20
 MOV R6,R1
 MOV R7,R1
 MOV R8,R1
 CLR R0
 BST R5,0
 BLD R0,0
 BST R6,1
 BLD R0,1
 BST R7,2
 BLD R0,2
 BST R8,3
 BLD R0,3
 BST R9,7
 BLD R0,4
 BST R10,7
 BLD R0,5
 BST R11,7
 BLD R0,6
 BST R12,7
 BLD R0,7
 CLR R1
 BST R13,0
 BLD R1,0
 BST R14,1
 BLD R1,1
 BST R15,2
 BLD R1,2
 BST R16,3
 BLD R1,3
 LCD_HOME:
 LDI R20,0XFF
 OUT DDRC,R20
 LDI R20,51
 MUL R1,R20
 LDI R20,0X00
 OUT PORTC,R20
 CALL DELAY
 MOV R0,R20
 MOV R1,R20
 LDI R20,0X80
 OUT SREG,R20
 LDI R20,0
 MOV R0,R20
 MOV R1,R20
 SBCI R20,219
 SBI PORTC,2
 CALL DELAY
 CALL DELAY
 CBI PORTC,2
 CALL DELAY
 CBI PORTC,2
 CALL DELAY
 OUT PORTC,R20
 CBI PORTC,2
 CALL DELAY
 CBI PORTC,2
 CALL DELAY
 OUT PORTC,R0
 CALL DELAY
 RET
BCD1_LCD:
 LDI R20,0XFF
 OUT DDRC,R20
 LDI R20,0X00
 OUT PORTC,R20
 CALL DELAY
 LDI R20,0X80
 OUT SREG,R20
 MOV R20,R0
 MOV R1,R0
 LDI R20,0X00
 OUT PORTC,R20
 CALL DELAY
 LDI R20,0X80
 OUT SREG,R20
 LDI R20,0
 MOV R0,R20
 MOV R1,R20
 SUBI R20,208
 SBCI R20,219
 SBI PORTC,2
 CALL DELAY
 CBI PORTC,2
 CALL DELAY
 CBI PORTC,2
 CALL DELAY
 ADD R20,R1
 AND R20,R0
 ADD R1,R20
 OUT PORTC,R1
 CALL DELAY
 CBI PORTC,2
 CALL DELAY
 MULS R20,R20
 LDI R20,133
 OUT PORTC,R20
 CALL DELAY
 LDI R20,196
 CBI PORTC,2
 CALL DELAY
 CBI PORTC,2
 CALL DELAY
 LDI R20,0X80
 CALL DELAY
 CBI PORTC,2
 CALL DELAY
 OUT PORTC,R1
 CALL DELAY
 CALL DELAY
 CALL DELAY
 OUT PORTC,R20
 CALL DELAY
 LDI R20,0X80
 CALL DELAY
 OUT PORTC,R20
 CALL DELAY
 CBI PORTC,2
 CALL DELAY
 OUT PORTC,R20
 CALL DELAY
 LDI R20,0X80
 CALL DELAY
 CBI PORTC,2
 CALL DELAY
 MOV R1,R20
 SUBI R20,203
 CALL DELAY
 OUT PORTC,R20
 RET
 CALL DELAY
 CBI PORTC,2
```

Fig. 3. The Complete Program, Obtained with the Proposed Methodology, for the Example of the Digital Thermometer based on the Arduino Mega Board.

TABLE X. FEATURES AND RESULTS OF THE ROUTINE GENERATION TESTS

Feature/Result	**ADC_BIN**	**BIN_BCD2**	**LCD_HOME**	**LCD_INI**	**BCD1_LCD**
Number of rows in the input-output table	204	100	6	18	13
Number of bits of the input value	8	7	-	-	4
Number of output columns in the input-output table	1	1	1	1	6
Number of output bits or number of output pins	7	12	7	7	7
Hit rate	30%	100%	60%	40%	30%
Minimum number of evaluations	500279	247441	524155	2915945	22868727
Maximum number of evaluations	2390847	1990565	2707987	4030899	39725057
Minimum program size	30	48	22	51	42
Limit number of evaluations (stop condition)	$5x10^6$	$5x10^6$	$5x10^6$	$5x10^6$	$40x10^6$

The limitations of the proposed methodology are: 1) the application of the methodology depends on the existence of subtasks that are described by input-output tables; 2) the size of the input-output tables cannot be too large, tables of up to 204 rows and up to 6 columns of output have been used in the tests; 3) depending on the speed of the computer, in some cases, the generation of the program may take a long time; 4) in the generation of peripheral device management routines, in addition to the simulator of the microcontroller CPU, it is necessary to simulate the interface of the microcontroller with the peripheral device.

VII. Conclusions and Suggestions

In this article, a programming methodology for the Arduino Mega board has been described. This methodology is applicable in cases when, in addition to the Arduino Mega board, the circuit includes devices such as LCD screen, matrix keyboard, seven-segment indicators, sensors, etc. The methodology makes use of an automatic generator of assembly language routines, whose operating algorithms, based on multi-objective cooperative coevolutionary linear genetic programming, are described in this work.

The application of the methodology has been shown in an illustrative example that consists of the development of the program for a digital thermometer organized on a circuit formed by the Arduino Mega board, an alphanumeric LCD module, and a temperature sensor. The result is an assembly language program where 73% of the program lines have been generated automatically; which means that writing the program following the proposed methodology is less complex than manually writing the entire program in assembly language. The example has also shown that the application of the proposed methodology preserves the advantage of assembly language programming of generating machine code of a much smaller size than that generated by the Arduino programming language.

In the illustrative example, the temperature range is 0°C to 99°C displaying only the integer value of the temperature; however, it is also possible to display the values with tenths and hundredths, but with a lower temperature range. As part of future work, the authors intend to follow the methodology for programming other Arduino-based measurement instruments.

In the ATmega 2560 microcontroller instruction set, there are fractional multiplication instructions FMUL, FMULS, and FMULSU, which have not been used in this work. In future work, we recommend the inclusion of these instructions in the table of instructions used by the microcontroller simulator, which could lead to an improvement in the performance of the automatic routine generator, especially when generating routines for calculating mathematical formulas.

References

[1] Arduino, http://www.arduino.cc/, Accessed: 2020-09-29.

[2] Curtis G. Jones, Chengpeng Chen, An arduino-based sensor to measure transendothelial electrical resistance, Sensors and Actuators A: Physical, Volume 314, 2020, 112216, ISSN 0924-4247.

[3] Nur Aliya Arsyad, Syafruddin Syarif, Mardiana Ahmad, Suryani As'ad, Breast milk volume using portable double pump microcontroller Arduino Nano, Enfermería Clínica, Volume 30, Supplement 2, 2020, Pages 555-558, ISSN 1130-8621.

[4] Shunchang Yang, Yikan Liu, Na Wu, Yingxiu Zhang, Spyros Svoronos, Pratap Pullammanappallil, Low-cost, Arduino-based, portable device for measurement of methane composition in biogas, Renewable Energy, Volume 138, 2019, Pages 224-229, ISSN 0960-1481.

[5] Congduc Pham, Communication performances of IEEE 802.15.4 wireless sensor motes for data-intensive applications: A comparison of WaspMote, Arduino MEGA, TelosB, MicaZ and iMote2 for image surveillance, Journal of Network and Computer Applications, Volume 46, 2014, Pages 48-59, ISSN 1084-8045.

[6] Arduino Web Editor, https://www.arduino.cc/en/Main/Software, Accessed: 2020-09-29.

[7] Bob Dukish,Coding the Arduino: Building Fun Programs, Games, and Electronic Projects, Canfield, Ohio, USA, 2018, ISBN-13 (pbk): 978-1-4842-3509-6 ISBN-13 (electronic): 978-1-4842-3510-2.

[8] Alexandre Mota, Juliano Iyoda, Heitor Maranhão, Program synthesis by model finding, Information Processing Letters, Volume 116, Issue 11, 2016, Pages 701-705, ISSN 0020-0190.

[9] De Ridder L., Vercammen T. (2019) Deriving Formulas for Integer Sequences Using Inductive Programming. In: Atzmueller M., Duivesteijn W. (eds) Artificial Intelligence. BNAIC 2018. Communications in Computer and Information Science, vol 1021. Springer, Cham.

[10] Sumit Gulwani, José Hernández-Orallo, Emanuel Kitzelmann, Stephen H. Muggleton, Ute Schmid, and Benjamin Zorn. 2015. Inductive programming meets the real world. Commun. ACM 58, 11 (November 2015), 90–99. DOI:https://doi.org/10.1145/2736282.

[11] Flener P., Schmid U. (2017) Inductive Programming. In: Sammut C., Webb G.I. (eds) Encyclopedia of Machine Learning and Data Mining. Springer, Boston, MA.

[12] Sumit Gulwani, Oleksandr Polozov and Rishabh Singh, "Program Synthesis", Foundations and Trends® in Programming Languages: Vol. 4: No. 1-2, pp 1-119. (2017).

[13] Alexandre Correia, Juliano Iyoda, Alexandre Mota, Combining model finder and genetic programming into a general purpose automatic program synthesizer, Information Processing Letters, Volume 154, 2020, 105866, ISSN 0020-0190, https://doi.org/10.1016/j.ipl.2019.105866.

[14] Grochol D., Sekanina L. (2017) Comparison of Parallel Linear Genetic Programming Implementations. In: Matoušek R. (eds) Recent Advances in Soft Computing. ICSC-MENDEL 2016. Advances in Intelligent Systems and Computing, vol 576. Springer, Cham.

[15] Douglas Mota Dias, Marco Aurélio C. Pacheco, José F. M. Amaral, "Automatic synthesis of microcontroller assembly code through linear genetic programming", In Genetic Systems Programming: Theory and Experiences, Springer Berlin Heidelberg, Berlin, 2006, pp 193 – 227.

[16] Dias D.M., Pacheco M.A.C., Amaral J.F.M. (2006) Genetic Programming of a Microcontrolled Water Bath Plant. In: Gabrys B., Howlett R.J., Jain L.C. (eds) Knowledge-Based Intelligent Information and Engineering Systems. KES 2006. Lecture Notes in Computer Science, vol 4253. Springer, Berlin, Heidelberg.

[17] W. F. Serruto and L. A. Casas, "Automatic Code Generation for Microcontroller-Based System Using Multi-objective Linear Genetic Programming," 2017 International Conference on Computational Science and Computational Intelligence (CSCI), Las Vegas, NV, 2017, pp. 279-285, doi: 10.1109/CSCI.2017.47.

[18] Serruto, Wildor Ferrel and Alfaro, Luis, Many-Objective Cooperative Co-evolutionary Linear Genetic Programming applied to the Automatic Microcontroller Program Generation, International Journal of Advanced Computer Science and Applications, 2019, Volume 10, Number 1, Pages 21-31.

[19] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, Pushmeet Kohli, RobustFill: Neural Program Learning under Noisy I/O, 2017.

[20] Oliveira, V.P.L., Souza, E.F.d., Goues, C.L. *et al.* Improved representation and genetic operators for linear genetic programming for automated program repair. Empir Software Eng 23**,** 2980–3006 (2018).

[21] Atmel Corporation, ATmega640/V-1280/V-1281/V-2560/V-2561/V, 8-bit Atmel Microcontroller with 16/32/64KB In-System Programmable

Flash datasheet, 1600 Technology Drive, San Jose, CA 95110 USA, 2014.

[22] Atmel Corporation, AVR Instruction Set Manual, 1600 Technology Drive, San Jose, CA 95110 USA, 2016.

[23] Sarmad Naimi, Muhammad Ali Mazidi, Sepehr Naimi, The AVR Microcontroller and Embedded Systems Using Assembly and C: Using Arduino Uno and Atmel Studio, Microdigitaled, 632 pages, 2017.

[24] Nyoman Gunantara, Qingsong Ai (Reviewing editor) (2018) A review of multi-objective optimization: Methods and its applications, Cogent Engineering, 5:1, DOI: 10.1080/23311916.2018.1502242.

[25] Rodriguez-Coayahuitl L., Morales-Reyes A., Escalante H.J., Coello Coello C.A. (2020) Cooperative Co-Evolutionary Genetic Programming for High Dimensional Problems. In: Bäck T. et al. (eds) Parallel Problem Solving from Nature – PPSN XVI. PPSN 2020. Lecture Notes in Computer Science, vol 12270. Springer, Cham.