Many-Objective Cooperative Co-evolutionary Linear Genetic Programming applied to the Automatic Microcontroller Program Generation

Wildor Ferrel Serruto[1], Luis Alfaro[2]
Departamento Académico de Ingeniería Electrónica[1]
Departamento Académico de Ingeniería de Sistemas [2]
Universidad Nacional de San Agustín de Arequipa, Perú

***Abstract*—In this article, a methodology for the generation of programs in assembly language for microcontroller-based systems is proposed, applying a many-objective cooperative co-evolutionary linear genetic programming based on the decomposition of a program into segments, which evolve simultaneously, collaborating with each other in the process. The starting point for the program generation is a table of input/output examples. Two methods of fitness evaluation are also proposed. When the objective is to find a binary combination, the authors propose fitness evaluation with an exhaustive search for the output of each bit of the binary combination in the genetic program. On the other hand, when the objective is to generate specific variations of the logical values in the pins of the microcontroller's port, the authors propose calculating the fitness, comparing the timing diagrams generated by the genetic program with the desired timing diagrams. The methodology was tested in the generation of drivers for the 4x4 matrix keyboard and character LCD module devices. The experimental results demonstrate that for certain tasks, the use of the proposed method allows for the generation of programs capable of competing with programs written by human programmers.**

Keywords*—*Many-objective optimization; cooperative coevolution; linear genetic programming; program synthesis; microcontroller-based systems

I. Introduction

A microcontroller is an integrated circuit with the basic characteristics of a computer. Microcontrollers perform specific tasks in various types of hardware, from general-consumption electronic hardware to industrial electronic hardware. There are microcontrollers of different architectures currently on the market: Intel 8051, Microchip PIC, Atmel AVR, ARM Holdings ARM, etc. In the current research project, an 8-bit 8051 architecture is employed, which is frequently used in embedded systems. For the tests, an AT89S52 microcontroller, belonging to this architecture, was utilized.

The electronic circuit within which the microcontroller functions is called a 'microcontroller-based system' (MBS) or an 'embedded system'. Depending on the function that it performs, the MBS includes other peripheral devices such as matrix keyboard, LCD screen, physical-magnitude sensors, switches, etc. During the functioning of a MBS, the microcontroller's central processing unit (CPU) executes the programs stored in machine language in the program memory.

The development process of a MBS includes the design of the hardware (the electronic circuitry of the MBS) and the software (the program that will execute the CPU of the microcontroller). Very frequently, the time invested in the elaboration of the software is an important fraction of the total development time of the system [1]. As mentioned in [2], when the developed software becomes more and more complex and its management becomes more difficult, the necessity arises to possess tools, which permit the generation of programs.

Program synthesis is a topic that has attracted the attention of many researchers. According to [3], the most common techniques that are currently used in program synthesis are: stochastic search, enumerative search, constraint solving, and programming based on deduction through examples. In the present project, a methodology is developed that permits one to generate automatically programs for the microcontroller in certain frequent tasks in MBSs, such as the scanning of matrix keyboard and the character display on the LCD module.

For the solution of complex problems, multiobjective optimization algorithms or cooperative co-evolutionary algorithms can be used. The former seek to minimize or maximize several objectives at the same time, such as the recent algorithm called Multi-Objective Grasshopper Optimization Algorithm (MOGOA), which is described in [4]. The latter divide the problem into subcomponents that evolve in parallel collaborating with each other, for instance the algorithm named Multi-Modal Optimization Enhanced Cooperative Coevolution (MMO-CC) explained in [5]. For the synthesis of programs for the MBSs we propose to establish several objectives, each of which corresponds to a bit of the result, and we also divide the program into segments that evolve in parallel. The proposed methodology is based on the application of the many-objective cooperative co-evolutionary linear genetic programming (MaOCCLGP), which is classified as a stochastic-synthesis technique.

Linear genetic programming (LGP) has been widely used in computer program generation for the solution of different problems, for example, symbolic regression problems [6], [7], robotic problems [8], [9], control problems [10], [11], etc. In the literature, research projects having to do with the

application of LGP in program generation for microcontrollers are scant. In [12] LGP is applied in the automatic synthesis of programs in assembly language for the Microchip PIC18F452 microcontroller in optimal-time control problems. In [13] the authors describe the generation of the 4x3 matrix keyboard scanning program, the initialization program of the LCD screen and the character display program on LCD screen using classical multiobjective LGP following the EMOEA and NSGA II algorithms. It is concluded that the application of the NSGA II algorithm in the generation of the LCD screen drivers is not satisfactory.

The novel contribution of the present project is the application of the MaOCCLGP in the generation of microcontroller programs for specific tasks of peripheral device management: 4x4 matrix keyboard scanning and display of the two-digit decimal number on the LCD module. These problems are considered more complex than those solved in the paper [13]. The performance of the proposed methodology has been evaluated by comparing the results produced by MaOCCLGP with the results of the application of EMOEA in the generation of the mentioned programs. Also, the programs generated by MaOCCLGP have been compared with those written by a human programmer.

The subsequent sections are organized in the following way: In Section II, the theoretical fundamentals necessary to be able to understand the proposed methodology are summarized. In Section III, the problem is laid out. In Section IV, the methodology for the automatic synthesis of programs for microcontrollers is formulated. In Section V, the experimental results and validation are analyzed. In Section VI conclusions regarding the research conducted, as well as recommendations for future research are given.

II. Theoretical Fundamentals

A. Linear Genetic Programming

Genetic programming, introduced by John Koza [14], is a technique that uses the principles of Charles Darwin's theory of evolution in order to produce automatically programs that perform a defined task. Linear genetic programming is a variant of genetic programming, which evolves sequences of instructions in an imperative-programming language or machine language.

The control parameters for the synthesis of a program with LGP are: register quantity, initial values of the registers, program size, and population size. In [15] some important conclusions relative to the control parameters are established: a small number of working registers can produce lack of fluency in the genetic program, while a very large number of working registers can unnecessarily increase the search space. It is recommended to start the registers with the input values instead of putting in constant values. Evolving programs of a fixed length is not recommended because this would not permit one to optimize the program size. For the population size, there is no special recommendation. In the current research project, the premise that large populations permit a greater diversity but require more processing time has been considered. For this reason, the authors tried to take large populations, taking precaution that the evolution time was not too great.

B. Many-Objective Evolutionary Optimization

In this work, in the generation of programs for microcontrollers, many-objective genetic programming is used. Toward this end, first, a general proposal of the multi-objective optimization problem and its relationship with the program-generation problem will be explained.

In multi-objective optimization, two or more objectives, which in some cases could be in conflict, are optimized simultaneously [16], [17]. In the *K*-objective optimization problem, the vector $X^* = (x_1^*, x_2^*, \dots, x_n^*)$ is searched for, which satisfies the inequality restrictions $g_i(X) \leq 0 \ (i = 1,2, \dots, m)$ and the equality restrictions $h_j(X) = 0 \ (j = 1,2, \dots, p)$, and minimizes or maximizes the objective function $F(X) = (f_1(X), f_2(X), \dots, f_K(X))$, where $X = (x_1, x_2, \dots, x_n)$ is the decision vector of *n* variables, and each one of the objective functions *f* performs the mapping $f: \mathbb{R}^n \to \mathbb{R}$.

For the problem of the automatic synthesis of programs for microcontrollers, $X = (x_1, x_2, \dots, x_n)$ represents a program in machine language (instruction sequence). Given that the problem is confronted with the use of genetic programming, the objective function $F(X)$ is the fitness function of the genetic program that indicates the degree of similarity between the input/output table that is generated after running the genetic program and the desired table of input/output examples. The multi-objective genetic programming algorithm looks to make the table generated equal to the one desired. When this occurs, the fitness function will have the highest value. Consequently, the multi-objective optimization will look to maximize the fitness function.

For the comparison of objective vectors, Pareto's dominance concept is used, and in the search for the solution, Pareto's optimality concept is employed. Given two solutions X_i, X_j, it is said that X_i dominates X_j in accordance with Pareto's dominance $(X_i \succ X_j)$ if the following conditions are met:

$\forall m \in \{0,1, \dots, K-1\}: f_m(X_i) \geq f_m(X_j)$ and

$\exists m \in \{0,1, \dots, K-1\}: f_m(X_i) > f_m(X_j)$,

On the contrary, it is said that X_j is a non-dominated solution with respect to X_i. A decision vector X^* is Pareto-optimal if there is not another vector X, such that X dominates X^*. The set of Pareto's optimal vectors is also named Pareto front.

The method of solving multi-objective optimization problems using evolutionary algorithms is known by the name of 'multi-objective evolutionary optimization'.

Multi-objective evolutionary algorithms are oriented for working toward a maximum of three objectives [18]. When the quantity of objectives is greater than three, it is recommended to apply algorithms that go by the name of 'many-objective evolutionary algorithms', better known by the abbreviation MaOEA. In the two problems studied in the present work, the number of objectives is seven.

In order to overcome the problems presented by an increase in objectives, there are different methods available [19]. One of

these methods is based on the use of aggregation functions in order to differentiate solutions for many objectives. In the present work, the 'individual information aggregation' method is used. In the fitness function, the degree of similarity between the input/output table generated and the one desired will be diminished by a value proportional to the program size. In this way, the objective function will permit the differentiation of solutions. At the same time, its maximization will help to reduce the program size.

C. Cooperative Coevolution

Another way to solve complex problems by way of evolutionary algorithms is to use 'cooperative coevolution'. The cooperative coevolution algorithm (CCEA), which was formulated by Potter [20], is based on the "divide and conquer" strategy, and it consists of decomposing the initial problem into subcomponents, also called species, which evolve in parallel while collaborating amongst each other in the process. In a CCEA in order to calculate the fitness of an individual of a species, a complete solution is formed, combining the individual with the representatives selected from other species.

In the present work, the authors apply cooperative coevolution in the synthesis of programs for microcontrollers, for which several species are formed. Each species corresponds to a program segment. In order to form a complete solution, individuals taken from each species are concatenated (one individual per species, as is shown in Fig. 1).

D. Microcontroller based Systems (MBS)

The electronic circuitry of a MBS, depending on its application, includes other peripheral devices apart from the microcontroller that are connected to it through input/output lines. Each device possesses a way of managing and in some cases a complex protocol for communication with the microcontroller.

In the present work, in order to put the proposed methodology to the test, the MBS is composed of the microcontroller, a matrix keyboard connected to port P2 (Fig. 2), and a text LCD module connected to port P1 (Fig. 3). If one desires to connect the matrix keyboard to another port, it is necessary to verify that the port lines possess 'pull-up' resistors.

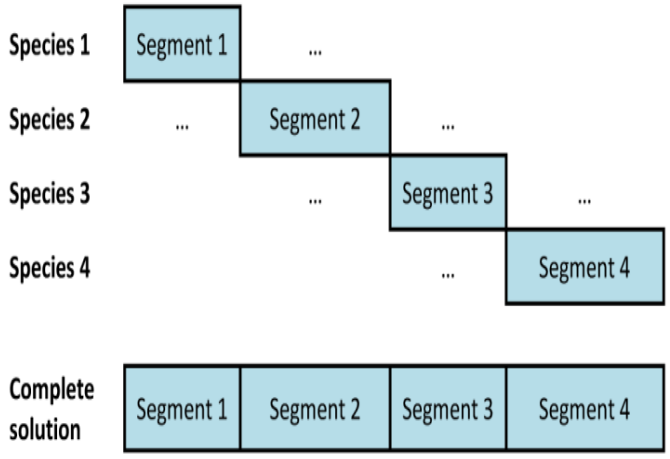

Fig. 1. Formation of a Complete Solution.

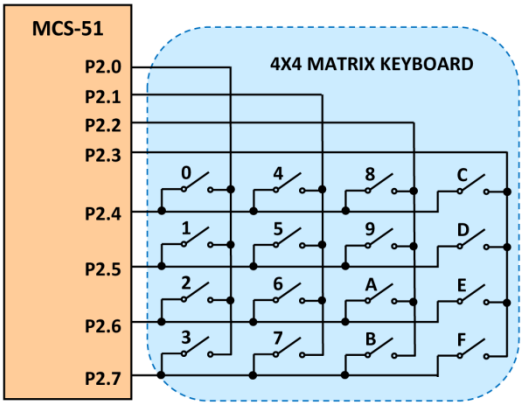

Fig. 2. 4x4 Matrix Keyboard Connection with the Microcontroller.

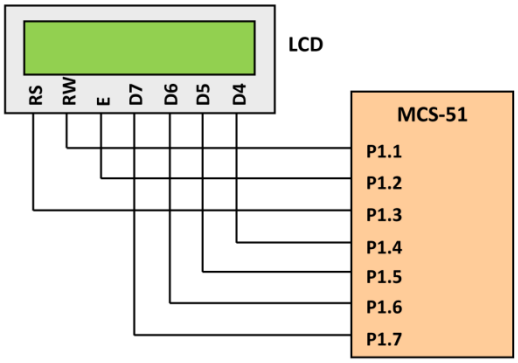

Fig. 3. LCD Module Connection with the Microcontroller with a 4-Bit Interface.

III. Formulation of the Problem

As the technique used in the proposed methodology is inductive programming, the starting point for the synthesis of a program is a table of input/output examples. The problem can be laid out, modifying the formulation given in [21], in the following manner: given a set of *M* input/output examples:

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

The objective is to find a program *P* that correctly transforms all the examples:

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

In this section, the problem is laid out regarding the generation of the matrix-keyboard scanning program and also that of the text-LCD-module display program.

A. Problem of Generation of 4x4 Matrix-Keyboard Scanning Program

In the 4x4 matrix keyboard of Fig. 2, the keys, represented as switches, have been numbered in hexadecimal code. In this system, when the user presses down on a key, the matrix-keyboard scanning program must identify the key, placing into the accumulator register (A) a binary combination which corresponds to the key pressed. The correspondence between the key number and the identifier is established in an input/output table. As an example, in Table 1, the identifiers of a 7-bit ASCII telephone keypad are shown. When there is not a key pressed, the identifier is 00h. It is assumed that only one key is pressed at a time or none.

TABLE I. INPUT/OUTPUT TABLE FOR THE 4X4 KEYBOARD

Key number	Identifiers of a 7-bit ASCII telephone keypad							
	Hex	Binary						
		S^6	S^5	S^4	S^3	S^2	S^1	S^0
0	31	0	1	1	0	0	0	1
1	34	0	1	1	0	1	0	0
2	37	0	1	1	0	1	1	1
3	2A	0	1	0	1	0	1	0
4	32	0	1	1	0	0	1	0
5	35	0	1	1	0	1	0	1
6	38	0	1	1	1	0	0	0
7	30	0	1	1	0	0	0	0
8	33	0	1	1	0	0	1	1
9	36	0	1	1	0	1	1	0
A	39	0	1	1	1	0	0	1
B	23	0	1	0	0	0	1	1
C	41	1	0	0	0	0	0	1
D	42	1	0	0	0	0	1	0
E	43	1	0	0	0	0	1	1
F	44	1	0	0	0	1	0	0
No pressed key	00	0	0	0	0	0	0	0

Therefore, the problem is laid out in the following way: using the proposed methodology, a 4x4 matrix-keyboard scanning program is generated, which complies with the input/output table given in Table 1.

The generated program is compared to a human-written program whose algorithm obeys the following reasoning: the pins that control the rows are configured as inputs, while the pins that control the columns as outputs. Only one column is activated, putting "0" in the corresponding pin, while all the other columns are deactivated with "1". The activation of a column allows one to verify if some key of this column is pressed through the reading of the pin of each row. If the read value is "1", this means that the key is not pressed, and if the value is "0", the key is pressed. This process is repeated for each column.

B. Problem of Generation of Two-Digit BCD Number Display Program in the Text LCD Screen

The text liquid-crystal screen (LCD) is a device that permits the visualization of text messages, where each character is shown in a dot matrix with a standard size of 5x7 dots [22]. Generally, the dots of the matrix darken in order to form the symbols. The control of the dot state of all screen matrices is carried out by a controller, which receives from the microcontroller the ASCII code of a character and displays it on the screen in the current position of the cursor. In this work, an LM016L text LCD screen is used, which has 2 lines of 16 characters per line, connected to port P1.

The LCD screen's signals, which are utilized for its connection with the microcontroller, are: the D7, D6, … , D0 data bus; and control signals E, RS, and R/W. Through the RS signal, the microcontroller indicates to the LCD screen if an instruction (RS=0) or a character (RS=1) is sent. With the R/W signal, the microcontroller determines the operation to run: reading (R/W=1) or writing (R/W=0). The E signal, called 'data enabling', initiates the operation with a falling edge.

The LCD screen's connection with the microcontroller can be performed by way of an 8-bit or 4-bit data interface. In the present work, authors use the 4-bit interface in which only the D7, D6, D5, and D4 lines are employed. Through these lines, the 8-bit commands and characters are sent. First, the high nibble is sent and later the low nibble. For each nibble, a falling edge is generated in E signal.

The problem can be formulated in the following way: using the proposed methodology, a program is generated, which permits one to visualize in the text LCD screen the two-digit decimal number that is found in the accumulator in packed BCD code. In this case, the input/output table has 100 rows. The input values are the BCD numbers from 00h to 99h. The output values are the timing diagrams that must be generated in the port pins for the visualization of the two-digit decimal number. In the input/output table, the timing diagrams are represented as a decimal or hexadecimal number chain.

One characteristic of the problems laid out is that the generated program must precisely comply with 100% of the input/output table. This is a difference between the application of the LGP to device driver generation and the application of the LGP to symbolic regression problems, where the result generated by the synthesized program can be approximated.

IV. PROPOSED METHODOLOGY

The methodology is based on the application of the many-objective cooperative co-evolutionary linear genetic programming, whose realization will be described in detail in this section.

A. Instruction Subset and Working Registers

The instruction set of the 8051-architecture-microcontroller CPU possesses 256 instructions, explained in the technical documentation of the microcontroller [24] [25], of which the instructions figuring in Table 2 will be used in the evolutionary process. In Table 2, it is observed that the used registers are: A (ACC o accumulator), B, R0, R7, and PX. PX is the register of input/output port P0, P1, P2, or P3, used in the peripheral device's connection. The registers A, B, and R0 function as working registers. The R7 register in genetic programs serves to store the input value with the objective that it can be retrieved by the working register. The operand #data is a number in a range from 0 to 255 called 'immediate value'. The operand 'bit address' is the address of a bit. In the evolutionary process, through bit addresses, the bits of the register A, B, and PX can be accessed. In the program's completion stage, the state register (PSW) is initialized.

B. Chromosome and Population Representation

Chromosome representation is very important in evolutionary algorithms. In LGP, the programs are represented by a linear sequence of instructions of an imperative programming language. For this work, an LGP chromosome representation with a dynamic size was adopted.

TABLE II. INSTRUCTIONS USED IN GENETIC PROGRAMS

MNE.	OPER.	MNE.	OPER.	MNE.	OPER.	MNE.	OPER.
ADDC	A,PX	INC	A	SETB	bit_addr	DEC	B
ADD	A,PX	RR	A	CLR	bit_addr	INC	B
ANL	A,PX	DEC	A	DEC	PX	ANL	B,A
ORL	A,PX	RRC	A	INC	PX	ORL	B,A
XRL	A,PX	RL	A	ORL	PX,A	XRL	B,A
SUBB	A,PX	RLC	A	ANL	PX,A	MOV	B,A
XCH	A,PX	SWAP	A	XRL	PX,A	MOV	B,R0
ADD	A,R0	CPL	A	MOV	PX,R0	ADD	A,#data
MOV	A,PX	CLR	A	MOV	PX,A	MOV	A,#data
ADDC	A,R0	ADD	A,B	INC	R0	ADDC	A,#data
ORL	A,R0	ADDC	A,B	DEC	R0	ORL	A,#data
ANL	A,R0	ORL	A,B	MOV	R0,A	ANL	A,#data
XRL	A,R0	ANL	A,B	MOV	R0,B	XRL	A,#data
SUBB	A,R0	XRL	A,B	MOV	R0,PX	SUBB	A,#data
XCH	A,R0	SUBB	A,B	MOV	A,R7	MOV	R0,#data
MOV	A,R0	XCH	A,B	MOV	B,R7	CLR	C
MUL	AB	MOV	A,B	MOV	R0,R7	SETB	C
				NOP			

According to [23], program synthesis using genetic programming presents two problems: epistasis and deceptiveness. Epistasis consists in the effects of the action of the instructions in a program being strongly interrelated. The deceptiveness pathology consists in the following: if the fitness is a scalar value, and there are two solutions with different fitness values, the solution with greater fitness is not necessarily the one that is closer to the correct solution. Selecting solutions only by the scalar fitness value can trap the search in a local optimum. In order to avoid the deceptiveness pathology, the authors use many-objective optimization, in which the fitness of an individual is a vector that will be used in the insertion of new individuals into the populations. What is more, in each species there will be two Pareto fronts, P1t and P2t, of variable size N1t and N2t, respectively.

C. Variation Operator

In each generation, each Pareto front will be kept sorted according to the sum value of fitness vector elements. In the selection of parents, (0.75·N1t + 0.25·N2t)·0.4 parent couples are selected. Upon selecting a parent, its index in the list is found with the following formula:

$$i = trunc((N-1)(1-a)^3)$$

where N is the length of the list, the function *trunc* returns the integer smaller value of a real number, and a is a real random number uniformly distributed in the interval$0 \leq a < 1$. Therefore, the result i has a greater probability of finding itself among the lower indices, where individuals of greater fitness are found.

Once the parent couples are selected, to each couple the variation operator is applied, which consists of: crossover (with 0.3 probability) or mutation (with 0.7 probability). The crossover operator consists of the exchange of tails (with 0.3 probability) or the exchange of instructions (with 0.7 probability).

In the crossover operation, the exchange of instructions is performed in randomly-selected positions of the parent sequences, and for the exchange of tails, the same position in the parent sequences is randomly selected.

In a sequence, the mutation is performed, with the same probability, in one of the following ways:

1) Removing a randomly-selected instruction.

2) Inserting, in a randomly-selected position, a randomly-generated instruction.

3) Exchanging two consecutive instructions.

4) Adding to the end of the sequence a randomly-generated instruction.

5) Changing a randomly-selected instruction with a randomly-generated instruction.

D. Fitness Evaluation

From now on, $P = [I_0, I_1, \dots, I_i, \dots, I_{N-1}]$ represents a genetic program.

1) Exhaustive fitness: When, in the table of input/output examples, the output is a binary combination, the authors propose calculating the fitness through a search of the output of each bit of the binary combination after the execution of each instruction of the genetic program and in each bit of the working registers. This way of calculating the fitness has called 'exhaustive fitness'.

In order to describe the exhaustive fitness evaluation, the authors use the representation of the input/output table of Fig. 4, where each column of the target output in binary representation is a combinational function, corresponding to a bit of the output combination.

In [26] a way of executing a genetic program in order to improve the output quality is proposed. The approach is oriented toward the genetic program based on trees. The authors apply part of this approach, taking into consideration that, in place of trees, there is a sequence of instructions *P*. The approach consists of forming a table in which each row corresponds to an instruction of the genetic program, and each column corresponds to a row of the input/output table. To this table, the name 'register value matrix' (RVM) is given.

Input	Target output (decimal or hexadecimal)	Target output in the binary representation				
		S^{K-1}	…	S^t	…	S^0
E_0	S_0	$S_0{}^{K-1}$	…	$S_0{}^t$	…	$S_0{}^0$
…	…					
E_j	S_j	$S_j{}^{K-1}$	…	$S_j{}^t$	…	$S_j{}^0$
…	…					
E_{M-1}	S_{M-1}	$S_{M-1}{}^{K-1}$	…	$S_{M-1}{}^t$	…	$S_{M-1}{}^0$

Fig. 4. Representation of the Input/Output Table (IOT).

Each cell of the RVM matrix possesses 24 bits (the bit numbers from 0 to 7 are for register A, from 8 to 15 for B, and from 16 to 23 for R0). Subsequent the execution of an instruction, the contents of working registers R0, B, and A are stored in the RVM matrix. In this way, after the execution of the genetic program for all inputs of the input/output table, the RVM matrix is completely full.

The authors adopt the following representations: RVM_{ij} is the RVM cell that contains the working-register values subsequent to the execution of the I_i instruction for input E_j, so RVM_{ij}^b represents the bit b of the said cell. If in the RVM matrix, the values of i and b remain fixed, and j is made to vary in the interval $(0 \le j \le M-1)$, then RVM_{ij}^b is a combinational function that only depends on the input value of the input/output table.

In the calculation of the exhaustive fitness (described in Algorithm 1) for each combinational function S^t of the table in Fig. 4, the most similar combinational function RVM_{ij}^b is searched for. The location of said function is given by the specific values of i and b. Therefore, K being the number of bits of the output binary combination, the fitness vector will be calculated with the formula:

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

$$f^t = \max_{\substack{0 \le b \le 23 \\ 0 \le i \le N-1}} \sum_{j=0}^{M-1} \{RVM_{ij}^b \odot S_j^t\} \tag{1}$$

$$fsum = \sum_{t=0}^{K-1} f^t \tag{2}$$

The symbol "$\odot$" represents the nor-exclusive logical operation that returns "1" if the input values are equal, and returns "0" otherwise. The summation symbol is the arithmetic sum operation.

The best bit location is specified with the pair (i, b), where i is the index of the instruction in the sequence, and b is the bit number in the RVM matrix cell. Subsequent the fitness evaluation, the bit location matrix (BLM) contains the bit location for all output bits:

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

Based on the BLM matrix, the effective program size is calculated:

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

In order to perform program size optimization, the vector f_{op} is used:

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

$$f_{op}^t = f^t - \alpha \cdot NE \tag{5}$$

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

where α is a parameter which the authors refer to as "instruction penalty", which can take any real value greater than 0 and less than 1/ (*maximum expected size of the program* +1). The evolutionary process upon maximizing f_{op}^p minimizes NE, which means that the program size is optimized.

The authors informally distinguish between the generated program and the synthesized program. To the sequence of instructions obtained as a result of the evolutionary process, the name 'synthesized program' is given. A generated program is obtained by adding some instructions to the synthesized program. For example, at the beginning of the program, it is necessary to add instructions in order to put in the initial register values. After the synthesis of a program, completion of the program must be performed in order to obtain the generated program. The structure of the generator with exhaustive fitness evaluation is shown in Fig. 5.

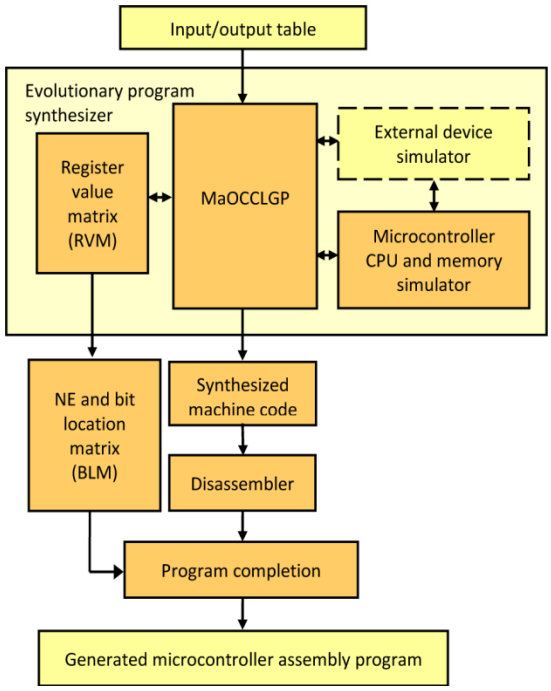

Fig. 5. Block Diagram of the Generator with Exhaustive Fitness.

Algorithm 1. Exhaustive fitness evaluation.

IOT is the input/output table.
$fmax = K \cdot M$ is the maximum value that can have *fsum*.
$P = [I_0, I_1, \dots, I_i, \dots, I_{N-1}]$ is the program to evaluate.
Pbest is the best program found until the moment with size *NEbest* and with fitness *fbest*
RVM is the register value matrix

1. Delete(RVM)
2. **for** each *Ej* of IOT **do**
3. (A, B, R0, R7) ← Initial values;
   Ports ← Initial values; PSW ← 00H
4. **for** i = 0 **to** N-1 **do**
5. Execution(*Ii*) for *Ej*
6. RVM$_{ij}$ ← (*R0*) (*B*) (*A*)
7. **end for**
8. **end for**
9. *f*, *fsum*, BLM, NE, *fop* and *fopsum* are calculated with the formula (1), (2), (3), (4), (5) y (6) respectively
10. **if** (*fsum* > *fbest*) or ((*fsum* = *fmax*) and (*NE* < *NEbest*)) **then**
11. *Pbest* ← *P*; *NEbest* ← *NE*; *fbest* ← *fsum*
12. **end if**
13. Return BLM, NE, *fop*, *fopsum*

a) Program Completion

When the stop condition is met, the evolutionary algorithm returns the synthesized program and the BLM matrix. The completion of the program is performed in the following way:

- Removing instructions with indices from *NEbest* to N-1.
- Inserting a MOV instruction after each instruction pointed by the BLM matrix in order to store the register that contains the result bit in the memory temporarily.
- Adding "*MOV C, bit-address*", "*MOV bit-address, C*" instructions in order to put the result bits together in the accumulator register.
- Adding instructions, before the synthesized sequence, in order to establish the initial values in registers A, B, R0, R7, PX, and PSW.

2) Fitness when the program generates timing diagrams according to input values: In this case, the genetic program must generate determined timing diagrams in the pins of the microcontroller's port, depending on the accumulator-register value. For this purpose, it is necessary to evaluate the degree of similarity between the two timing diagrams. In Fig. 6, the comparison of the timing diagrams is illustrated, where the quantity of intervals along the timing diagrams is L=6.

In Fig. 7 the input/output table is represented, which determines the timing diagrams that must be generated depending on the input value that is in the accumulator. In this table, $S_{j,d}$ represents the timing diagrams in all pins of the port in interval d when the input is E_j.

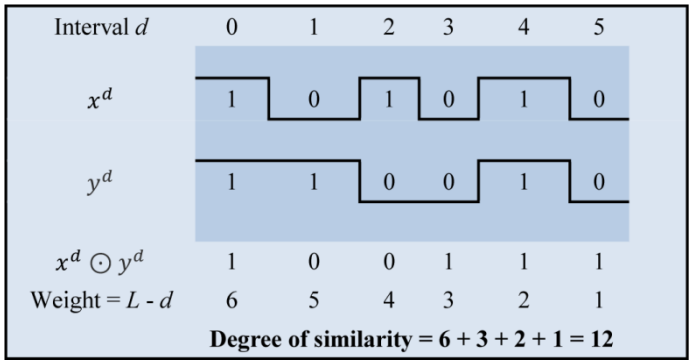

Fig. 6. Comparison of Two Timing Diagrams.

Input	**Target timing diagrams (decimal or hexadecimal)**				
	0	…	*d*	…	*L*-1
E_0	$S_{0,0}$	…	$S_{0,d}$	…	$S_{0,L-1}$
…					
E_j	$S_{j,0}$	…	$S_{j,d}$	…	$S_{j,L-1}$
…					
E_{M-1}	$S_{M-1,0}$	…	$S_{M-1,d}$	…	$S_{M-1,L-1}$

Fig. 7. Representation of the Input/Output Table when Timing Diagrams are Generated Depending on the Input Value.

If the number of pins where the timing diagrams are generated is K, then each $S_{j,d}$ value of the target timing diagram and each $G_{j,d}$ value of the generated timing diagram is expressed in the binary representation, where each bit corresponds to a port pin:

$$S_{j,d} = \begin{bmatrix} S_{j,d}^{(K-1)} \\ \vdots \\ S_{j,d}^{p} \\ \vdots \\ S_{j,d}^{0} \end{bmatrix}; \qquad G_{j,d} = \begin{bmatrix} G_{j,d}^{(K-1)} \\ \vdots \\ G_{j,d}^{p} \\ \vdots \\ G_{j,d}^{0} \end{bmatrix}$$

In order to find the fitness vector of the timing diagrams generated by the genetic program, first the fitness vector is calculated for each E_j input and each pin, determining the degree of similarity of the timing diagrams with the following formula:

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

$$f_j^p = \sum_{d=0}^{L-1}\left\{(L-d)\left(G_{j,d}^p \odot S_{j,d}^p\right)\right\} \tag{7}$$

Where $(L-d)$ is the weight of the interval d.

Then the fitness for all inputs is calculated:

$$f = \sum_{j=0}^{M-1} f_j \tag{8}$$

Thus, for the fitness vector $f = (f^0, f^1, \dots, f^{K-1})$ one also calculates the sum:

$$f_{sum} = \sum_{p=0}^{K-1} f^p \tag{9}$$

For each E_j input, there is a vector $VNI_j = \left[i_{j,0}, i_{j,1}, \dots, i_{j,d}, \dots, i_{j,L-1}\right]$ with the indices of instructions which produce the changes in the timing diagrams. For the input E_j, the scalar $NUI_j = \max_d\left(\left[i_{j,0}, i_{j,1}, \dots, i_{j,d}, \dots, i_{j,L-1}\right]\right)$ is calculated, which is the index of the instruction that produced the last value in the timing diagram. Also, the effective size of the program is found:

$$NE = 1 + \max_{0 \le j \le M-1}\left(NUI_j\right) \tag{10}$$

Based on all the VNI_j vectors, the authors form the VDIF vector of size L, in which each element is equal to the difference between the maximum and minimum values of all the values in each VNI_j vector position:

$$VDIF_d = \max_{0 \le j \le M-1}\left(i_{j,d}\right) - \min_{0 \le j \le M-1}\left(i_{j,d}\right) \tag{11}$$

Using NE and $DIFmax = \max_{0 \le d \le L-1}(VDIF_d)$, the fitness $f_{op} = \left(f_{op}^0, f_{op}^1 \dots, f_{op}^{K-1}\right)$ and its sum f_{opsum} are calculated by the following equations:

$$f_{op}^p = f^p - \alpha \cdot (DIFmax + NE) \tag{12}$$

$$f_{opsum} = \sum_{p=0}^{K-1} f_{op}^p \tag{13}$$

Algorithm 2 shows the fitness evaluation when timing diagrams are generated according to input values.

The evolutionary process upon maximizing f_{op}^p minimizes NE and DIFmax. This means that program size is optimized.

At the end of the process, DIFmax must be zero, which ensures that the vectors of the instruction indices that produce the changes in the timing diagrams are equal for all inputs.

The structure of a generator with fitness evaluation when timing diagrams are generated is shown in Fig. 8.

Algorithm 2. Fitness evaluation of the program that generates timing diagrams according to input values.

S contains the target timing diagrams.
$fmax = K \cdot (L+1) \cdot \frac{L}{2}$ is the maximum value that can have *fsum*.
$P = [I_0, I_1, \ldots, I_i, \ldots, I_{N-1}]$ is the program to evaluate.
Pbest is the best program found until the moment with size *NEbest* and with fitness *fbest*

1. Delete(*f*)
2. **for** each input E_j **do**
3. (A, B, R0, R7) ← Initial values;
   Ports ← Initial values; PSW ← 00H
4. Delete(Gj); Delete(VNIj)
5. **for** i = 0 to N-1 **do**
6. Execution(I_i)
7. Update(Gj); Update(VNIj);
8. **end for**
9. For the input E_j, f_j is calculated with equation (7)
10. $f \leftarrow f + f_j$
11. **end for**
12. *f*, *fsum*,*NE*, *fop*, *fopsum* are calculated with the equations (8), (9), (10), (11) y (12)
13. **if** (*fsum* > *fbest*) or ((*fsum* = *fmax*) and (*NE*< *NEbest*)) **then**
14. *Pbest* ← *P*; *NEbest* ← *NE*; *fbest* ← *fsum*
15. **end if**
16. Return *NE*, *fop*, *fopsum*

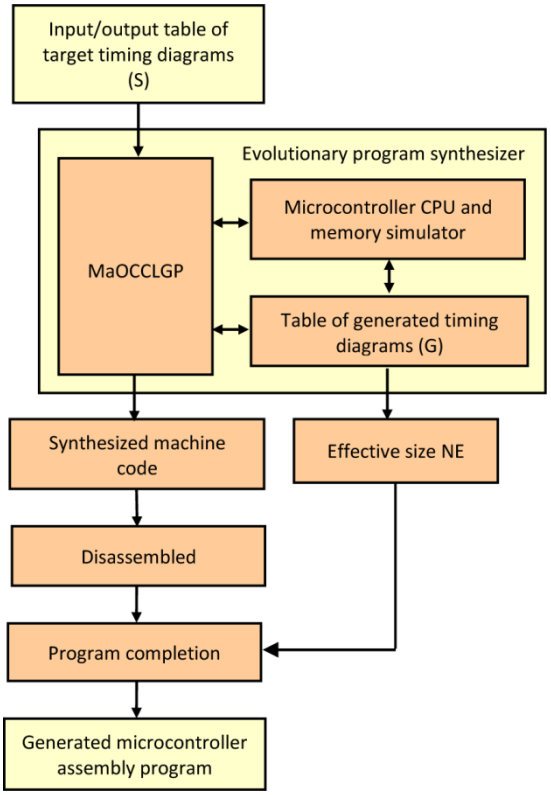

Fig. 8. Block Diagram of the Timing Diagram Generator.

E. Many-Objective Cooperative Co-evolutionary Linear Genetic Programming (MaOCCLGP)

In the MaOCCLGP algorithm, each program segment evolves like a species. In each species, there are two populations, P1t and P2t, which are non-dominated Pareto fronts and two auxiliary lists Qt and Dt. The algorithm begins by randomly generating populations P1t and P2t in each species (see Algorithm 3). Next, a determined quantity of representatives of each species is selected. The representatives are the best individuals on the P1t list, starting from the individual with index 0. Using the representatives, the fitness of individuals of each species is calculated. Each P1t and P2t list is sorted using the *fopsuma* value of each individual.

While the stop condition is not fulfilled, the following operations are performed: in each species, the representatives are selected; next, the parent individuals are selected, to which the variation operator is applied in order to obtain the descendants in list Qt; then, in each species the fitness of each individual of P1t, P2t, and Qt is evaluated; subsequently, in each species the best individuals of Qt in P1t are inserted, putting individuals cast aside in Dt; the best individuals of Dt in P2t are inserted; finally, P1t and P2t are sorted.

In the insertion operations, the algorithm described in [27] is followed, using the *fop* fitness of each individual in such a way that P1t and P2t are Pareto fronts. To maintain diversity a parent is taken from P1t with a probability of 0.75 or from P2t with a probability of 0.25.

Algorithm 3. Many-Objective cooperative co-evolutionary linear genetic programming.

1. **for** each species **do**
2. Random generation of P1t y P2t;
3. **end for**
4. Selection of representatives of each species
5. **for** each species **do**
6. Fitness_evaluation (P1t)
7. Fitness_evaulation (P2t)
8. Sorting(P1t), Sorting(P2t)
9. **end for**
10. **while** the stop condition is not reached **do**
11. **for** each species **do**
12. Selection of representatives
13. Qt ← Parent_selection (Pt1, Pt2)
14. Qt ← Variation(Qt)
15. **end for**
16. **for** each species **do**
17. Fitness_evaluation (P1t),
18. Fitness_evaluation (P2t),
19. Fitness_evaluation (Qt),
20. **end for**
21. **for** each species **do**
22. Insertion of the best from Qt in P1t and those cast aside in Dt;
23. Insertion of the best from Dt in P2t
24. Sorting(P1t), Sorting(P2t)
25. **end for**
26. **end while**

The fitness evaluation is performed in the following manner: for example, if the quantity of species is 4 and the quantity of representatives in each species is 2, the representatives are denoted R00 and R01 of species 0, R10 and R11 of species 1, R20 and R21 of species 2, and R30 and R31 of species 3. Therefore, in order to evaluate the fitness of individual Ix of species 2, the fitness of each of the two programs is calculated, the first consisting of the concatenation of R00, R10, Ix, and R30, and the second of R01, R11, Ix, and R31. The fitness of Ix is the *fop* fitness of the program with greater *fopsuma* value. It is necessary to indicate that, in the given example, a total of eight different programs could be formed with representatives and the Ix individual. Nevertheless, due to processing time constraints, only two programs were formed.

V. Results and Validation

To evaluate the performance of the methodology each generator was executed ten times with a limit number of evaluations (LNE). It is possible that in the evolutionary process, a program satisfies the input/output table before reaching the LNE limit. If this is the case, the generator continues running and optimizing the size of the synthesized program until reaching the LNE limit.

In order to analyze the results, the following criteria are used: hit rate (HR), minimum number of instructions (MNI), minimum code size after the compilation (number of bytes) (MCS), and minimum number of clock cycles (MNCC).

In [13], the EMOEA algorithm has been applied in the generation of the following programs: 1) 4x3 matrix keyboard scanning program, 2) initialization program of the LCD screen, and 3) character display on LCD module program. In the present work MaOCCLGP is applied in the generation of the programs: 1) 4x4 matrix keyboard scanning program, and 2) two-digit decimal number display on LCD module program.

The evaluation of the performance of the methodology proposed in this article has been carried out in two ways:

1) The EMOEA algorithm that was used in [13] has been applied to the two cases studied in the present work and has been compared with the application of MaOCCLGP. The results are shown in Table 3.

2) The programs generated by MaOCCLGP have been compared with programs written by human as shown in Table 4. In this case, the test parameters given in Table 5 were used. Human-written programs taken as a reference were elaborated following the algorithms described in microcontroller assembly language programming courses.

In Table 3, it can be seen that in the generation of the 4x4 matrix keyboard scanning program, the MaOCCLGP algorithm has a hit rate much higher than that of EMOEA. In the generation of the program for BCD number display on the LCD screen, the MaOCCLGP algorithm produced a smaller program.

In Table 4, it is observed that in the two analyzed cases, the hit rate is high (80%). Comparing the best generated keyboard-scan program with the written by human program, it is observed that regarding minimum length, the generated program is 13% bigger than the human-written program. Regarding code size, the generated program is 3% bigger, and with regard to clock cycles, it is 71% more spread out. The difference between the minimum-length and code-size percentages is due to the fact that the human-written program makes use of conditional jump instructions, which possess 3 bytes. The significant difference between the minimum-length and clock-cycle-number percentages is due to the fact, that in the generated program all the instructions are executed. On the other hand, in the human-written program, due to the jump instructions, not all instructions are run. Upon comparing the generated program and the written by human program for BCD-number-display on LCD screen, it is observed that the human-written program has slightly-higher percentages in the three criteria of MNI 22%, MCS 2%, and MNCC 10%, which demonstrates the advantage of the generated program in the solution of this problem.

TABLE III. Comparison of the Performance of the EMOEA and MaOCCLGP Algorithms

Generated Program	Algorithm	LNE x10^6	HR (%)	MNI	MCS	MNCC
4x4 matrix keyboard scanning in 7-bit ASCII code	EMOEA	3	10	57	105	828
	MaOCCLGP	3	80	61	115	840
Two-digit BCD number display on LCD module	EMOEA	1	100	25	48	324
	MaOCCLGP	2	80	23	46	348

TABLE IV. Comparison of Programs Generated by MaOCCLGP and Programs Written by Human

Program	N°	HR (%)	MNI	MCS	MNCC
Generated program for 4x4 matrix keyboard scanning in 7-bit ASCII code	P1	80	61	115	840
Written by human program for 4x4 matrix keyboard scanning in 7-bit ASCII code	P2	-	54	111	492
Generated program for two-digit BCD number display on LCD module	P3	80	23	46	348
Written by human program for two-digit BCD number display on LCD module	P4	-	28	47	384

TABLE V. Test Parameters for MaOCCLGP

Parameter	P1	P3
Initial population size of each species	200	200
Number of species	10	10
Number of representatives	2	2
Initial size of the program segment (min – max)	2-4	2-4
LNE – Limit number of evaluations (x10^6)	3	2
α	0.001	0.01

TABLE VI. PROGRAM EXAMPLES

P1	P2	P3	P4

mov a,#0
mov b,a
mov r0,a
mov p2,#ffh
mov psw,#0
clr 163
inc p2
mov a,#15
xch a,p2
add a,p2
mov 38,a
subb a,b
add a,#232
addc a,#81
mov 35,a
addc a,#206
mov 37,a
orl a,#131
cpl a
rl a
subb a,p2
mov p2,a
addc a,p2
cpl a
mov 34,a
setb 165
addc a,b
rl a
orl a,p2
mov 36,a
dec a
anl a,p2
dec a
swap a
add a,p2
add a,#232
cpl a
subb a,p2
subb a,p2
mov 32,a
subb a,p2
add a,#177
xch a,p2
xrl b,a
addc a,p2
mov 33,a
mov a,#0
mov c,5
mov 224,c
mov c,10
mov 225,c
mov c,23
mov 226,c
mov c,30
mov 227,c
mov c,33
mov 228,c
mov c,47
mov 229,c
mov c,51
mov 230,c
	mov p2,#0feh
	jb p2.4,n1
	mov a,#'1'
	ret
	n1: jb p2.5,n2
	mov a,#'4'
	ret
	n2: jb p2.6,n3
	mov a,#'7'
	ret
	n3: jb p2.7,n4
	mov a,#'*'
	ret
	n4:mov p2,#0fdh
	jb p2.4,n5
	mov a,#'2'
	ret
	n5: jb p2.5,n6
	mov a,#'5'
	ret
	n6: jb p2.6,n7
	mov a,#'8'
	ret
	n7: jb p2.7,n8
	mov a,#'0'
	ret
	n8:mov p2,#0fbh
	jb p2.4,n9
	mov a,#'3'
	ret
	n9: jb p2.5,n10
	mov a,#'6'
	ret
	n10: jb p2.6,n11
	mov a,#'9'
	ret
	n11: jb p2.7,n12
	mov a,#'#'
	ret
	n12:mov p2,#0f7h
	jb p2.4,n13
	mov a,#'a'
	ret
	n13: jb p2.5,n14
	mov a,#'b'
	ret
	n14: jb p2.6,n15
	mov a,#'c'
	ret
	n15: jb p2.7,n16
	mov a,#'d'
	ret
	n16: mov a,#0
	ret
		mov a,#0
		mov b,a
		mov r0,a
		mov r7,a
		mov p1,#0ffh
		mov psw,#0
		mov r0,#60
		mov p1,r0
		clr 146
		anl a,#252
		orl a,#13
		xch a,p1
		clr 146
		mov p1,r0
		clr 146
		mov acc,r7
		clr 229
		swap a
		orl a,#12
		xch a,p1
		mov a,#6
		clr 146
		anl p1,a
			mov r0,a
			anl a,#0fh
			orl a,#30h
			push acc;
			mov a,r0
			swap a
			anl a,#0fh
			orl a,#30h;
			acall send_data
			pop acc
			acall send_data
			send_data:
			mov r0,a
			anl a,#0f0h
			add a,#00ch
			mov p1,a
			anl a,#0f0h
			add a,#008h
			mov p1,a
			mov a,r0
			swap a
			anl a,#0f0h
			add a,#00ch
			mov p1,a
			anl a,#0f0h
			add a,#008h
			mov p1,a
			ret

In Table 6, examples of generated (P1, P3) and human-written programs (P2, P4) are shown. In the generated programs, the instructions in boldface were obtained as a result of the evolutionary process, while those in normal font were put into the program during the completion stage. All programs have been tested in the Proteus software from Labcenter Electronics. The programs that manage the LCD screen have been tested when the clock frequency of the microcontroller is 100 kHz.

VI. CONCLUSIONS AND SUGGESTIONS

In this work, a new method of generating programs in assembly language for microcontroller-based systems was described. The method uses many-objective cooperative co-evolutionary linear genetic programming.

When the objective is to find a binary combination, for the fitness evaluation exhaustive fitness was proposed, which, for each bit of the binary combination, searches for an output in the genetic program at the bit level, permitting a faster convergence of the evolutionary algorithm. On the other hand, when the objective is to generate variations in the logical values in the pins of the microcontroller's input/output port, the fitness evaluation was proposed, which is based on the comparison of the generated timing diagrams with the target timing diagrams.

The methodology was tested for the 8051 architecture in two examples that are frequently found in microcontroller-based systems: 4x4 matrix-keyboard scanning program and two-digit BCD-number display program on the text LCD screen.

The comparison of the generated and human-written programs for the case of BCD-number display on LCD screen shows an advantage of the generated program in the three analyzed criteria. This procedure possesses the following particularity: when it is human-written, it is developed in two stages. First, the BCD-code number is converted into ASCII code, and later the ASCII characters are displayed on the LCD screen. On the other hand, when the program is generated, the generator can carry out the task in a direct manner, without their explicitly being a conversion of the BCD code into ASCII code.

One disadvantage of the proposed methodology is that for certain cases the input/output table can be too big. In those cases, making use of the counterexamples-driven genetic programming, described in [28], is suggested for future research.

Based on the results shown in Table 4, it can be concluded that the microcontroller programs in assembly language, generated following the proposed methodology, are capable of competing with programs written by a human programmer in the solution of the specific tasks. However, it is necessary to point out that currently limitations exist, meaning that there are tasks for which the generator did not manage to produce a program that complies with 100% of the input/output table, for example, natural binary number display on LCD screen, which would be interesting area for future research to help improve the methodology.

The proposed perspective can be applied to the automatic generation of routines of other peripheral devices: graphic LCD screen, 7-segment indicators, physical-magnitude sensors, etc. Likewise, the methodology can be extended to other 8-bit architectures like PIC or AVR.

REFERENCES

[1] Kamal, Raj, Embedded systems: architecture, programming and design. 1st ed. Boston: McGraw-Hill Higher Education, 2008.

[2] Rainer Leupers. Code generation for embedded processors. In Proceedings of the 13th international symposium on System synthesis (ISSS '00). IEEE Computer Society, Washington, DC, USA, 173-178. 2000.

[3] S. Gulwani, O. Polozov, and R. Singh. Program Synthesis. Foundations and Trends® in Programming Languages, vol. 4, no. 1-2, pp. 1–119, 2017.

[4] Alaa Tharwat, Essam H. Houssein, Mohammed M. Ahmed, Aboul Ella Hassanien, Thomas Gabel. MOGOA algorithm for constrained and unconstrained multi-objective optimization problems. Applied Intelligence (2017), Volume 48, Issue 8, p.2268-2283, August 2018.

[5] X. Peng, Y. Jin and H. Wang, "Multimodal Optimization Enhanced Cooperative Coevolution for Large-Scale Optimization," in IEEE Transactions on Cybernetics. 2018. doi: 10.1109/TCYB.2018.2846179

[6] Douglas Mota Dias, Marco Aurélio C. Pacheco. Toward a quantum-inspired linear genetic programming model. In Proceedings of the Eleventh conference on Congress on Evolutionary Computation (CEC'09). IEEE Press, Piscataway, NJ, USA, 1691-1698. 2009.

[7] Guilherme C. Strachan, Adriano S. Koshiyama, Douglas M. Dias, Marley M. B. R. Vellasco, Marco A. C. Pacheco. Quantum-Inspired Multi-gene Linear Genetic Programming Model for Regression Problems. In Proceedings of the 2014 Brazilian Conference on Intelligent Systems (BRACIS '14). IEEE Computer Society, Washington, DC, USA, 152-157. 2014.

[8] Jens Busch, Jens Ziegler, Christian Aue, Andree Ross, Daniel Sawitzki, and Wolfgang Banzhaf. 2002. Automatic Generation of Control Programs for Walking Robots Using Genetic Programming. In Proceedings of the 5th European Conference on Genetic Programming (EuroGP '02), James A. Foster, Evelyne Lutton, Julian F. Miller, Conor Ryan, and Andrea Tettamanzi (Eds.). Springer-Verlag, Berlin, Heidelberg, 258-267.

[9] Wolff K., Nordin P. (2003) Learning Biped Locomotion from First Principles on a Simulated Humanoid Robot Using Linear Genetic Programming. In: Cantú-Paz E. et al. (eds) Genetic and Evolutionary Computation — GECCO 2003. GECCO 2003. Lecture Notes in Computer Science, vol 2723. Springer, Berlin, Heidelberg

[10] Li, Ruiying; Noack, Bernd R.; Cordier, Laurent; Borée, Jacques; Harambat, Fabien. Drag reduction of a car model by linear genetic programming control. Experiments in Fluids, Volume 58, Issue 8, article id.103, 20 pp. (ExFl Homepage). 2017.

[11] Li, Ruiying & Noack, Bernd & Cordier, Laurent & Jacques, Boree & Kaiser, Eurika & Harambat, Fabien. (2017). Linear genetic programming control for strongly nonlinear dynamics with frequency crosstalk.

[12] 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, pp 193 – 227, 2006.

[13] Wildor Ferrel Serruto, Luis Alfaro Casas. Automatic Code Generation for Microcontroller-Based System Using Multi-objective Linear Genetic Programming. Proceedings of the 2017 International Conference on Computational Science and Computational Intelligence (CSCI'17: 14-16 December 2017, Las Vegas, Nevada, USA), Publisher: IEEE Computer Society.

[14] J.R. Koza, Genetic Programming – On the Programming of Computer Programs by Natural Selection. MIT Press, Cambridge, MA, 1992.

[15] Markus F. Brameier, Wolfgang Banzhaf, Linear genetic programming, On Genetic and Evolutionary Computation, Publisher Springer, US, 2007.

[16] Eckart Zitzler, Marco Laumanns, Stefan Bleuler, "A tutorial on evolutionary multiobjective optimization", Swiss Federal Institute of Technology (ETH) Zurich, Computer Engineering and Networks Laboratory (TIK), Zurich, Switzerland 2004.

[17] Kalyanmoy Deb, "Multi-objective optimization using evolutionary algorithms", John Wiley & Sons, LTD, pp. 239-286, New York, USA, 2001.

[18] Shelvin Chand, Markus Wagner, Evolutionary Many-Objective Optimization: A Quick-Start Guide. Article in Surveys in Operations Research and Management Science, December 2015

[19] Bingdong Li, Jinlong Li, Ke Tang, and Xin Yao. Many-Objective Evolutionary Algorithms: A Survey. ACM Comput. Surv. 48, 1, Article 13 (September 2015), 35 pages. 2015.

[20] Potter, M.A., De Jong, K.A.: Cooperative Coevolution: An Architecture for Evolving Coadapted Subcomponents. Evolutionary Computation 8 (2000) 1–29

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

[22] Ampire Co., Ltd. Specifications for LCD Module. 2001.

[23] T. Weise, M. Wan, K. Tang and X. Yao, "Evolving exact integer algorithms with Genetic Programming," 2014 IEEE Congress on Evolutionary Computation (CEC), Beijing, 2014, pp. 1816-1823. doi: 10.1109/CEC.2014.6900292

[24] "8-bit Microcontroller with 8K bytes in-system programmable flash AT89S52", Atmel Corporation, 2008.

[25] "Atmel 8051 microcontrollers hardware manual", Atmel Corporation, 2007.

[26] Ignacio Arnaldo, Krzysztof Krawiec, and Una-May O'Reilly. 2014. Multiple regression genetic programming. In Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation (GECCO '14). ACM, New York, NY, USA, 879-886. DOI: https://doi.org/10.1145/2576768.2598291

[27] Rui Liu, Sang-you Zeng, Lixin Ding, Lishan Kang, Hui Li, Yuping Chen, et al., "An efficient multi-objective evolutionary algorithm for combinational circuit design", First NASA/ESA Conference on Adaptive Hardware and Systems (AHS'06), Istanbul, pp. 215-221, 2006.

[28] Iwo Błądek, Krzysztof Krawiec, Jerry Swan. Counterexample-Driven Genetic Programming: Heuristic Program Synthesis from Formal Specifications. Evolutionary Computation. Massachusetts Institute of Technology. Volume 26, Issue 3, p.441-469, Fall 2018