Instruction-Based Development: From Evolution to Generic Structures of Digital Circuits

Michal Bidlo and Jaroslav Škarvada
Brno University of Technology
Faculty of Information Technology
Božetěchova 2, 61266 Brno, Czech Republic
Tel.: +420 54114 1206, Fax.: +420 54114 1270
E-mail: bidlom@fit.vutbr.cz, skarvada@fit.vutbr.cz

March 8, 2008

Submitted for a Special edition
Paper for the EHW ISSUE OF THE KES JOURNAL
Guest editors Prof. Mircea Negoita, Prof. Tughrul Arslan

Abstract

Evolutionary techniques provide powerful tools to design novel solutions for hard problems in different areas. However, the problem of scale (i.e. how to create a large, complex solution) represents a significant obstacle for the evolution of complex extensive systems. The computational development represents one of the approaches in the evolutionary design techniques that tries to overcome the problem of scale. In this paper an instruction-based developmental method is presented for the evolutionary design of generic structures of digital circuits. The developmental system involves a set of application-specific instructions constituting programs in order to solve a given task. In particular, the goal is to construct generic structures of combinational circuits. An evolutionary algorithm is utilized for the design of these programs that represent a mapping from the genotypes to the phenotypes during the evolutionary process, i.e. the prescription for the construction of target circuits. Two case-studies are presented in order to demonstrate the successfulness of this approach: (1) the evolutionary design of generic combinational multipliers and (2) the evolutionary design of generic sorting networks.

1 Introduction

In conventional design of electronic circuits, various algorithms exist that are usually based on an appropriate model created by engineers (for example, Boolean algebra in case of the design of digital circuits or models of electronic components and their relations when an analog circuit should be created). In recent yaars, evolutionary design techniques have widely been used to solve many complex problems of in different scientific and engineering areas, including the design of electronic circuits. The main advantage of the application of evolutionary approaches in the design of circuits is that engineer specifies components the circuit should be composed of (i.e. building blocks), creates a suitable evolutionary algorithm, including a proper evaluation function, and the artificial evolutionary process searches automatically for a solution with respect to the specified criteria. Moreover, the engineer usually needs not to specify the whole model according to which he would synthesize the target circuit "by hand", using the conventional approach. The evolution explores the search space that is restricted only by the criteria specified by the designer and, during the evolutionary process based on the evaluation function, the target solution emerges. In fact, there may be less restrictions in the evolutionary design process than in the conventional design based on an engineering model. Therefore, the evolution may, in some cases, discover a totally new and innovative solution which is hard to analyze and which would usually not be possible to achieve using conventional approaches. For example, Thompson utilized an evolutionary algorithm for the design of a tone discriminator in the reconfigurable array FPGA XC6216 [18]. The goal was to maximize the difference of average output voltage for two different input signals (1kHz and 10kHz). The evolved solution was verified in other FPGA of the same type, however, the original function of the circuit in this FPGA

was not achieved. It means that the evolution involved internal physical properties of the FPGA that was used for evaluation of the evolving solutions. In other experiment, Sekanina and Růžička. presented an evolutionary design of image filters at the level of functional blocks using a software simulator [17]. The results they obtained exhibit better properties (considering the filtered image quality or implementation cost) in comparison with the best conventional solutions.

In general, the problem is that as the task to be solved by means of an evolutionary algorithm gets more complex, the amount of information that is needed to encode the candidate solutions in the chromosome increases. Therefore, the length of the chromosome increases, the search space becomes enormously large and the evolutionary algorithm is not able to explore it effectively. This issue is referred to as the *problem of scale*. This is particularly the difficulty in the traditional evolutionary algorithms as there is typically a one-to-one relationship between the genotype and the corresponding solution description. The development, a technique inspired by the biological phenomena of ontogeny, represents an approach to overcome the problem of scale, which is the objective of this paper.

In nature, the process of development is influenced by the genetic information of the organism and the environment in which the process is carried out. Cells use the mechanism of transcription and translation to read each gene and produce the string of amino acids that makes up a protein. Proteins activate or suppress synthesis of other genes, work as signals among cells, influence internal functions of the cells and perform many other important roles. Therefore, they control the growth, position and behavior of all cells [5].

In the area of the evolutionary design, the process of development (more precisely, computational development [14]) is usually considered as a non-trivial *genotype–phenotype mapping* in an evolutionary algorithm (EA). While genetic operators of the EA work over genotypes, the fitness calculation is applied on the phenotypes created by means of the developmental process.

In recent years various developmental systems were utilized in the area of the evolutionary design. For example, Kitano applied a rewriting developmental system combined with genetic algorithm for the design of neural networks [12]. Three-dimensional mechanical objects have been designed by evolution that utilized a variant of Lindenmayer system in its genotype–phenotype mapping [9]. Koza introduced an original method in which novel analog circuits have been constructed according to the instructions produced by genetic programming [10]. Gordon and Bentley utilized the interaction of artificial genes and proteins to model development in digital circuits [6]. Gruau proposed a genetic encoding scheme networks based on a cellular duplication and differentiation for the design of neural networks [7]. Miller and Thomson invented a developmental method for growing graphs and circuits using Cartesian genetic programming [15]). In order to evolve 3D shape and form Kumar used complex models of development inspired by genetic regulatory networks [14]. Tufte and Haddow presented the evolutionary design of cellular computing machines implemented inside a FPGA for the investigation of structural and functional properties generated by the development of a cellular automaton that are, in addition to the inter- and intra-cellular interaction, dependent on an external environmental information [19]. Sekanina and Bidlo created an instruction-based developmental mapping for the evolutionary design of growing generic sorting networks using an iterative process of repeated application of an evolved program [16].

In this paper an application-specific developmental model is proposed that is based on the approach introduced in [16]. A general instruction-based developmental system is presented for the design of digital circuits. The goal is to evolve programs consisting of application-specific instructions by means of which *generic* circuits structures could be constructed. Two case-studies are presented in order to demonstrate the successfulness of this approach: (1) the evolutionary design of generic multipliers and (2) the evolutionary design of generic sorting networks. The results obtained by means of this method are compared with the traditional solutions and the solutions obtained during our previous research.

The paper is structured as follows. Section 2 introduces the concept of the proposed instruction-based developmental system. The evolutionary system setup by means of which the experiments were conducted is described in Section 3. The development of generic multipliers using the proposed developmental method is depicted in detail in Section 4 together with the experimental results obtained by the evolutionary algorithm for this class of circuits. Section 5 contains a detailed description of the development of generic sorting networks and the results obtained in this category of experiments. Finally, concluding remarks are summarized in Section 6.

2 Concept of the Developmental System

For the experiments presented in the next sections the following setup of the developmental system was utilized. The construction of the circuit is performed by means of a developmental program. This

program, which is the subject of evolution, consists of simple instructions. The instructions are executed sequentially according to the program pointer (*pp*). The instructions make use of numeric literals $0, 1, \ldots, max\_value$, where max_value is specified by the designer at the beginning of the evolutionary process. In addition to the numeric literals, a set of variables and parameters integrated into the developmental system may be utilized. The variables are denoted symbolically $v_0, v_1, v_2, \ldots$ and their values are altered by the appropriate instructions during the execution of the program (developmental process). The parameters are denoted as $p_0, p_1, \ldots$ and their values — usually relating in some way to the size (e.g. number of inputs) of the circuit being developed — are specified by the designer at the beginning of the evolutionary process, during which they are invariable. Note, that the number of variables and parameters, initial values of the variables as well as the utilization of parameters for the development are determined specifically for a given application.

Table 1 describes the instruction set utilized for the development. The SET instruction assigns a value determined by a numeric literal, parameter or another variable to a specified variable. Instructions INC/DEC are intended for increasing/decreasing the value of a given variable (specified in first argument) by given numeric literal (specified in second argument). Simple loops inside the developmental program are provided by the REP instruction whose first argument determines the repetition count and the second argument states the number of instructions after the REP instruction to be repeated. Inner loops are not allowed, i.e. REP instructions inside the repeated code are interpreted as NOP (no operation) instructions. The GEN instruction generates a building block of the type specified in its argument.

Instruction	*Arguments*	*Description*
0: SET	*variable*, *value*	Assign numeric literal *value* to *variable*.
1: INC	*variable*, *value*	Increase *variable* by numeric literal *value*.
2: DEC	*variable*, *value*	If *variable* $\geq$ *value*, then decrease *variable* by numeric literal *value*.
3: REP	*count*, *number*	Repeat *count*-times *number* following instructions. *count* is variable and *number* is numeric literal. All REP instructions in the following code are interpreted as NOP instructions (inner loops are not allowed).
4: GEN	*block*	Generate block of type *block* on the actual position (row, col); increase *col* by 1.
5: NOP		An empty operation.

Table 1: Instructions utilized for the development

In general, the developmental program may be interpreted as a set of subprograms, i.e. sequences of instructions that are executed independently. Each subprogram is identified by a unique index. The execution of a given subprogram during the developmental process is determined by a vector containing a sequence of indices corresponding to the subprograms identification. This vector can be considered as external (environmental) information for an additional control of the developmental process. We denote $prog(j)$ the j-th subprogram of the developmental program an $env(k)$ the k-th value of the environmental vector. Let us define a developmental step as a single execution of a program (subprogram). While the developmental program is stored in the chromosome of an evolutionary algorithm, the environmental vector may be specified by the designer with respect to the requirements of a given application. The principle of this concept is illustrated in Fig. 1.

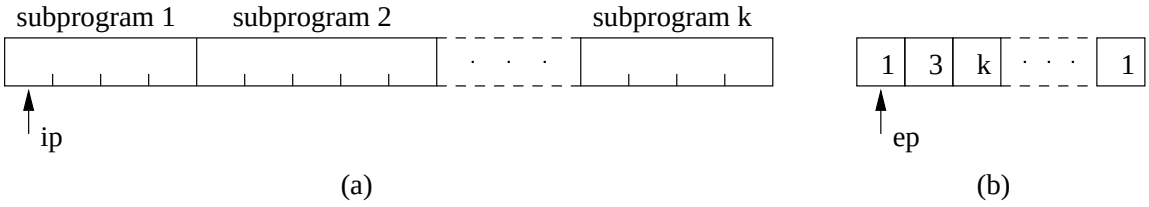

Figure 1: The concept of instruction-based developmental encoding controlled by the environment: (a) A set of subprograms, each of which is identified by a unique index. The instructions of a subprogram are executed sequentially according to the instruction pointer *ip*. (b) The environmental vector of subprogram indices for the additional control of the development. The environment is scanned sequentially according to the environment pointer *ep*.

In the next sections we present two different applications involving the proposed concept: Section 4

is devoted to the design of generic multipliers by means of the development driven by the environmental information and Section 5 introduces a novel developmental model for the construction of arbitrarily large sorting networks which does not utilize the environment.

3 Evolutionary System Setup

For the experiments presented in this paper a simple genetic algorithm was utilized in combination with the developmental system described in Section 2.

A chromosome consists of a constant-length linear array of the instructions, each of which is represented by the operation code and two arguments (the utilization of the arguments depends on the type of the instruction). The array contains the developmental program (or a set of subprograms stored in sequence). The utilization of a single developmental program or a set of subprograms depends on the application. Note, however, that the subprograms are not distinguished during the evolution, i.e. the chromosomes are handled as a uniform sequence of instructions during application of genetic operators. The population consists of 32 chromosomes which are generated randomly at the beginning of evolution. Tournament selection operator of base 2 is utilized.

Mutation of a chromosome is performed by a random selection of an instruction followed by a random choosing a part of the instruction (operation code or one of its arguments). If the operation code is mutated, entire new instruction will replace the original one and new arguments are randomly generated, otherwise one of arguments is mutated. Note that the mutation algorithm ensures proper arguments (variable or numeric literal) depending on the instruction type (see Table 1). Only one instruction per chromosome is mutated with the probability 0.04.

A special crossover operator was applied which exhibits a significant positive influence on the evolutionary process in comparison with standard one-point or uniform crossover or with the case when no crossover is utilized. Two parent chromosomes are selected and an instruction is selected randomly in each of them (i_1, i_2). A position (index) is chosen randomly in each of the two offspring (c_1, c_2). After the crossover, the first, respective the second offspring contains the original instructions from the first, respective the second parent with the exception of i_1, respective i_2, which is put at the position c_2 in the second offspring, respective c_1 in the first offspring. The crossover occurs with the probability 0.9 and is illustrated in Figure 2.

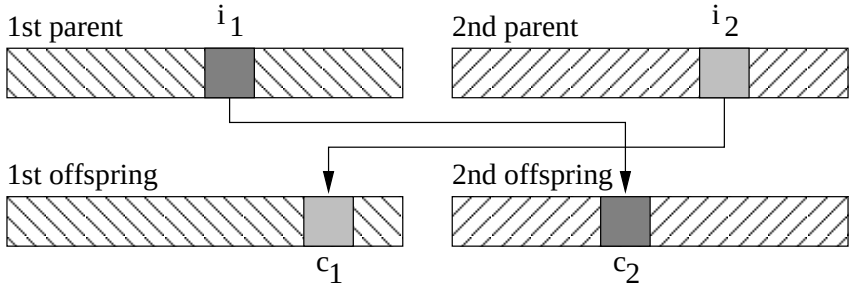

Figure 2: Crossover of two chromosomes. i_1, i_2 denote the instructions to be crossed, c_1, c_2 pose the offspring positions the instructions will be placed to.

Since the evaluation of candidate solutions is performed differently depending on the trait of a particular application, the fitness function will be described separately in the following section related to the design of generic multipliers and sorting networks. All experiments were conducted on a common PC equipped with a 2.0 GHz processor, 512 MB RAM and running Gentoo Linux, kernel 2.6.18-r6.

4 Development of Generic Multipliers

The development of generic multipliers is inspired by the construction of conventional combinational multipliers for which generic algorithms exist. Figure 3 shows a typical 4×4 combinational multiplier designed by means of the conventional approach [20]. It is evident that the first level of AND gates and the following sequence of adders are specific in comparison with the rest of the circuit, which poses a kind of irregularity. However, the rest of the circuit exhibits highly regular structure which can be created by means of an iterative algorithm utilizing variables. Moreover, the whole design can be easily parametrized

by the number of bits (width) of the operands. Therefore, this concept is assumed to be convenient for the design of generic multipliers using the development and the genetic algorithm.

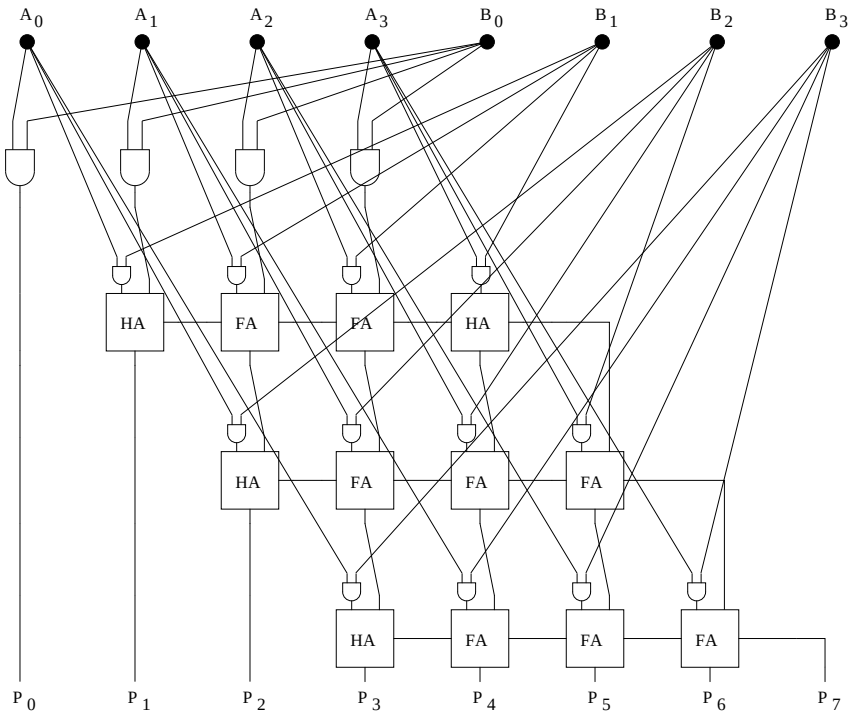

Figure 3: A 4×4 conventional combinational multiplier. $A_0 \ldots A_3, B_0 \ldots B_3$ represent the bits of the operands, $P_0 \ldots P_7$ denote the bits of the product.

A building block represents a basic component of the circuit to be developed. The general structure of the block is shown in Figure 4a. Each building block contains three inputs from which one or two may be unused depending on the type of the block. There are two outputs at each building block from which one may be meaningless, i.e. permanently set to logic 0, depending on the block type. The outputs are denoted symbolically as $out0$ and $out1$. In case of the block containing only one output, $out0$ represents the effective output and $out1$ is permanently set to logic 0. The circuit is developed inside a grid (a rectangular array) which proved to be a suitable structure for the design of combinational multipliers (see Figure 4b). Figure 5 shows the set of building blocks utilized for the experiments presented in this section. The gate-level logic structures of the adders are illustrated in Figure 6 and 7 (marked by dotted rectangles). For the interconnection of the blocks the position (row, col) in the grid is utilized. The inputs of the blocks are connected to the outputs of the neighboring blocks or to the primary inputs of the circuit, determined by the references associated with the inputs of the blocks. Specifically, the inputs of the adders are connected to the appropriate neighbouring outputs in the grid and the inputs of the AND gates are connected to the primary inputs of the circuit via indices determined by the variables v_0 and v_1 (see Figure 5). For example, $out1(row, col-1)$ means that the input of the block at the position (row, col) in the grid is connected to the output denoted $out1$ of the block on its left-hand side. Let $A = a_0a_1a_2, B = b_0b_1b_2$ represent the primary inputs (operands A and B) of a 3×3 multiplier. For instance, an AND gate with $v_0 = 1$ and $v_1 = 2$ has its inputs connected to the second bit (a_1) of operand A and the third bit (b_2) of operand B. In case when v_0 or v_1 of an AND gate being generated exceeds the correct values, the appropriate input of that gate is set to logic 0. If (row, col) do not exceed the grid boundaries when generating a block, the block is generated at that position, otherwise no block is generated. After generating a block, col is increased by one. Considering the building blocks at the borders of the grid (for $row = 0$ or $col = 0$), where no blocks with valid outputs occur (for $row - 1$ or $col - 1$), the appropriate inputs of the blocks at (row, col) are set to logic 0. In this way, for example, full adder (Fig. 5f) at $(0, 0)$ is degraded to AND gate, the buffer (Fig. 5b) at $(1, 0)$ becomes the source of logic 0 etc.

The basic approach to the development of generic multipliers involves the developmental program consisting of a set of subprograms that are executed on demand with respect to the environmental vector (or environment) as introduced in Section 2. The meaning of the environment is to enable the system to

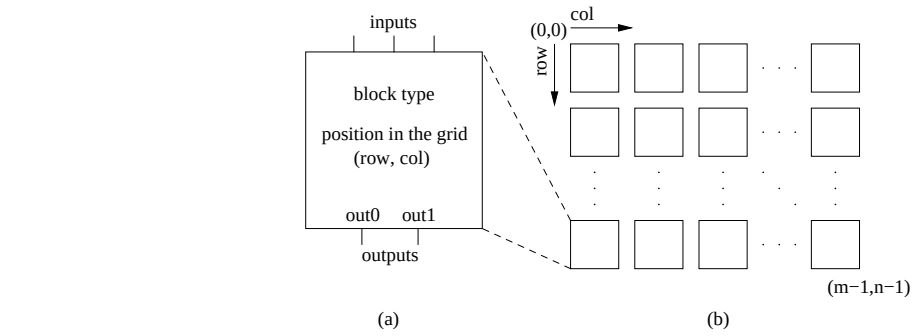

Figure 4: (a) Structure of a building block. (row, col) determines the position of the block in the grid – see part (b). The connection of the inputs depends on the type and position of the block. (b) Grid of the building blocks with m rows and n columns for the development of generic multipliers.

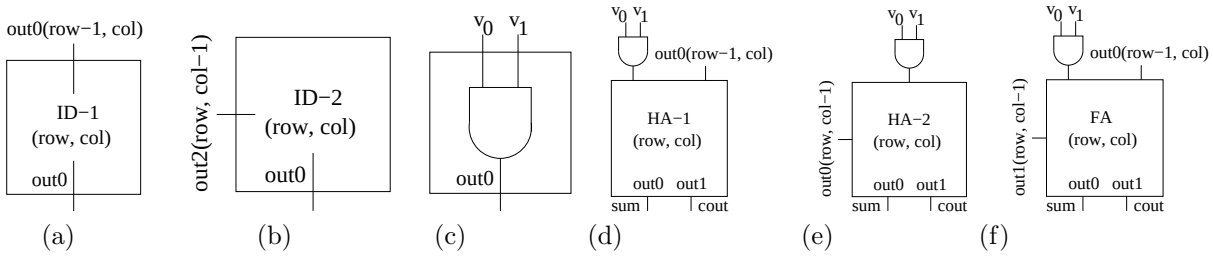

Figure 5: Building blocks for the development of combinational multipliers. (a, b) buffers, (c) AND gate, (d, e) half adders, (f) full adder. (row, col) denotes position in the grid. v_0 and v_1 determine indices of primary input bits. Connection of different inputs of the blocks is shown. Unused inputs and outputs are not depicted (they are considered as logic 0).

develop more complex structures which need not be fully regular. For example it is possible to construct the circuit in figure 3 by usage of 3 various subprograms (algorithms). First row (AND gates) can be constructed by algorithm 0, second row by algorithm 1 and all subsequent rows by algorithm 2. While the subprograms are executed independently, the variables and the parameter of the developmental system are shared by all the subprograms.

At the beginning of the evolutionary process the value of the parameter p_0 and the form of the environment *env* are specified by the designer. By the inspiration from the conventional multipliers the number of developmental steps needed for creating a working multiplier and the length of the environment will correspond to p_0. The developmental program is intended to operate over these data in order to develop a multiplier of a given size. The number of subprograms and the number of instructions they are composed of, are also specified a priori by the designer. The different sizes of multipliers are created by setting the parameter and adjusting the environment. Hence the circuit of a given size is always developed from scratch, i.e. the grid for generating the building blocks is empty before the development starts; it is a case of parametric developmental design. The following algorithm will be defined in order to handle the developmental process.

1. Initialize $row, col, v_0, v_1, v_2, v_3$ and e to 0.
2. Execute $env(e)$-th part of program.
3. Increase e and row by 1, set col to 0.
4. If neither e, nor row exceed, go to step 2.
5. Evaluate the resulting circuit.

In case of the development of the multipliers the parameter p_0 specifies the size of the multiplier to be developed and evaluated by the fitness function. The fitness function is computed as the number of output bits possessing the correct value after the circuit simulation. The experiments were conducted with the evolution of programs for the construction of 4×4 multipliers, i.e. the parameter $p_0 = 4$. There are $2^{4+4} = 256$ possible test vectors and the multipliers produce 8-bit results. Therefore, the maximum

fitness representing a working solution equals $256 \cdot 8 = 2048$. Note, however, that the building blocks from Figure 5 represent the elementary components of the circuit, only their function is involved during the fitness evaluation and no delay, fan-in or fan-out parameters of the target circuit are optimized during the development with respect to the properties of the building blocks. Two sorts of experiments have been performed (see chapter 4.1). If a working solution is not evolved in two millions of generations in case of the first sort, possibly in one million of generations in the second sort of experiments, the evolution is restarted with a new population. After the evolution the resulting program is verified in order to determine whether it is able to create larger multipliers, typically up to the size 14×14 bits. This size of circuit was determined experimentally, allowing to perform a sufficient number of developmental steps for demonstrating the correctness of the evolved program and keeping a reasonable verification time. If a program shows this ability, it will be considered as general.

4.1 Experimental Results and Discussion

In the first sort of experiments 3-part programs (6+12+12 instructions) were evolved utilizing the environment $env = (0, 1, 2, 2)$ for controlling the development. 1000 independent experiments were conducted from which 67 % working solutions (i.e. the programs for constructing 4×4 multipliers) were evolved and 18 % of them were classified as general programs. Figure 6 shows a multiplier together with detailed logic schemes of the building blocks (half adder from Fig. 5e and full adder from Fig. 5f) involved by the evolutionary algorithm. This multiplier was constructed by means of one of the most efficient programs that were evolved in this sort of experiments. The program is shown in Table 2. Let us go through the program in order to understand the developmental process.

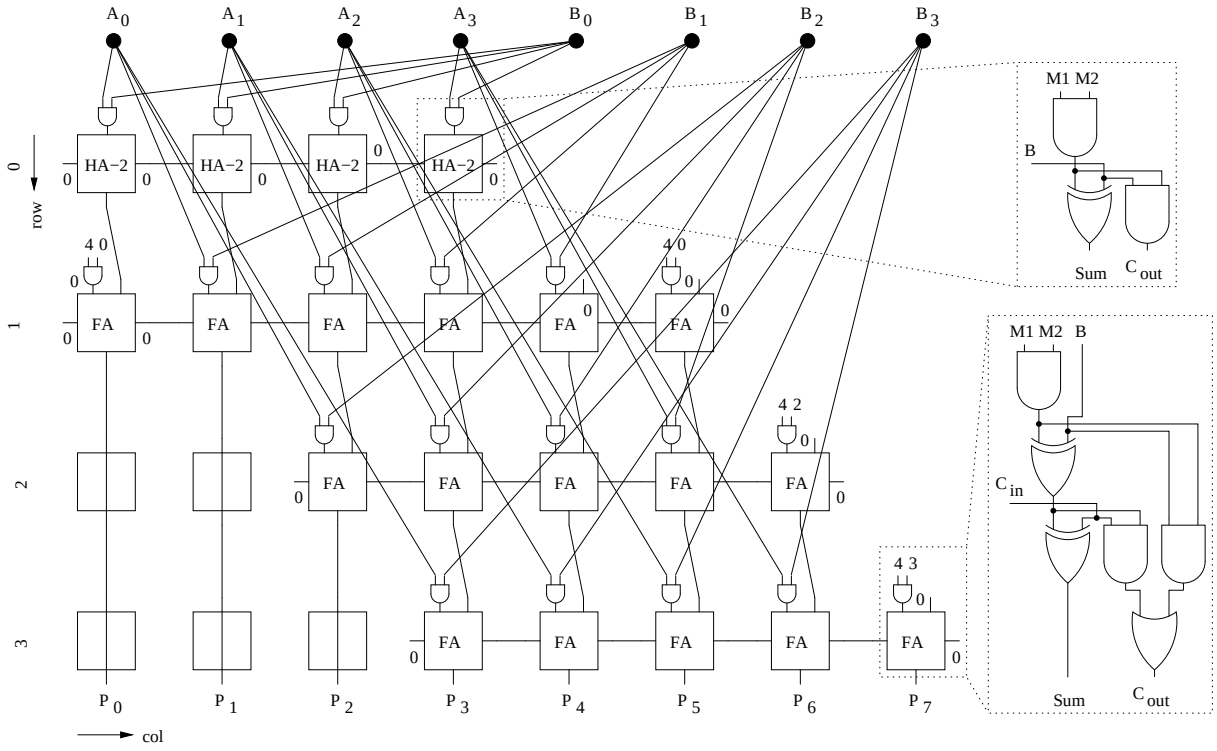

Figure 6: A 4×4 multiplier created by means of evolved program shown in Table 2 using the environment $env = (0, 1, 2, 2)$. $A_0 \ldots A_3, B_0 \ldots B_3$ represent the bits of the operands, $P_0 \ldots P_7$ denote the bits of the product. Logic schemes of the half adder and full adder-based building blocks utilized by the evolved program are shown. M1 and M2 denote the multiplicands whose partial product represents the first operand of the full adder, B denotes the second operand, C_{in} poses the input carry, Sum and C_{out} represent the resulting sum and output carry.

At the beginning of the development, the following setup is specified by the designer: $p_0 = 4, env = (0, 1, 2, 2)$. The following initialization is performed by the system: $v_0 = 1, v_1 = 0, v_2 = 0, v_3 = 0, row = 0, col = 0, e = 0$.

At this point $env(e) = 0$, therefore Part 0 of the program will be executed. The instruction 0 should repeat zero times instructions 1 and 2 (because $v_1 = 0$), therefore, this code has no effect. Instruction

Line	Part 0	Part 1	Part 2
0:	REP v_1 2	GEN FA	REP v_1 2
1:	GEN FA	SET v_3 0	REP v_0 2
2:	INC v_0 1	SET v_0 v_3	GEN ID-1
3:	REP p_0 2	INC v_1 1	GEN ID-1
4:	GEN HA-2	REP p_0 2	INC v_0 1
5:	INC v_0 1	GEN FA	INC v_1 1
6:		INC v_0 1	REP p_0 1
7:		SET v_1 0	SET v_0 v_2
8:		GEN FA	REP p_0 2
9:		DEC v_1 0	GEN FA
10:		INC v_1 1	INC v_0 1
11:		REP v_0 2	GEN FA

Table 2: Evolved general program by means of which the multiplier from Figure 6 was created. In this case $p_0 = 4$, the program consists of 3 parts executed according to the environment $env = (0, 1, 2, 2)$.

3 will repeat four times (because $p_0 = 4$) instructions 4 and 5 which create row 0 of the multiplier (blocks HA-2) with the inputs of AND gates of these blocks connected to the primary inputs (operands of the multiplier) specified by the actual values of v_0 and v_1. While v_1 retains 0, v_0 is increased by 1 by instruction 5 and *col* is increased by 1 automatically by the system in each pass (in general, after executing a GEN instruction). Since there are no more instructions to be executed in Part 0, the system increases *row* and e by 1 and the construction of row 0 of the circuit is finished. Note that the variables hold their actual values, i.e. $v_0 = 4$ and the others equal 0.

Now, $env(e) = 1$ for $e = 1$, therefore, Part 1 of the program will be executed in order to develop row 1 of the multiplier. Instruction 0 of Part 1 generates full adder (FA block), where the inputs of AND gate of this block should be connected to bits 4 and 0 of the operands (according to the variables $v_0 = 4, v_1 = 0$). Note, since v_0 exceeds the operand width, the first input of AND gate of this FA block will be considered as logic 0 causing permanent logic 0 at the output of the AND gate, i.e. the AND gate of this block is meaningless (see Fig. 6). Instructions 1 and 2 actually set v_0 to 0. Then, v_1 is increased by 1 by instruction 3. Instructions 4, 5 and 6 generate four FA blocks with the inputs of AND gates of these blocks connected to the appropriate operand bits. Note that instruction 7 sets v_1 to 0 which, in fact, voids the result of instruction 3. An FA block is generated by instruction 8 (again, its AND gate is meaningless). Instruction 9, decreasing v_1 by 0, has no effect, v_1 is increased by 1 by instruction 10 and instruction 11 is meaningless since there is no instructions to repeat. Row 1 is completed with the actual values of $v_0 = 4, v_1 = 1$ and other variables possessing zeros.

The row 2 of the circuit will be constructed using Part 2 of the program according to the next environment value $env(e) = 2$ for $e = 2$. Instruction 0 initiates a loop repeating once instructions 1 and 2. Instruction 1 is interpreted as no operation because inner loops are not allowed and instruction 1 generates an ID-1 block. In addition, instruction 3 creates one more ID-1 block in the next column. Value of v_0, respective v_1 is increased by one by instruction 4, respective 5. In fact, the only effect of the loop initiated by instruction 6, repeating instruction 7, is to set v_0 to 0 (according to v_2 which equals 0). This operation actually voids the result of instruction 4. Four FA blocks are generated by instruction 9 inside the loop started by instruction 8. Instruction 9, which is also a part of the loop body, determines the connection of the inputs of AND gates generated inside these blocks. The last instruction 11 generates an FA block with a redundant AND gate. Now row 2 is finished. The variables $v_0 = 4, v_1 = 2$ and the other ones equal 0.

According to $env(e) = 2$ for $e = 3$ the last row of the circuit will be generated by executing Part 2 of the program. The development proceeds in the same way as described in the previous paragraph, considering the values of variables resulted from the previous developmental step.

This program showed the ability to construct generic multipliers. Note that, in general, for p_0-bit operands the environment would have the form $env = (0, 1, 2, \ldots, 2)$ containing $p_0 - 2$ twos and p_0 developmental steps would be needed to construct a working multiplier.

It is evident that the multiplier shown in Fig. 6 could be optimized considering the inputs of the building blocks. For instance, half adders in row 0 of the circuit can be replaced by simple AND gates since the first input of these adders are permanently connected to logic 0. Similarly, full adders at positions (1, 1), (1, 4), (2, 2) and (3, 3) actually represents half adders and full adders at positions (1, 0), (1, 5), (2, 6) and (3, 7) can be replaced by identity functions. In fact, the circuit corresponds to the

conventional multiplier after this optimization (compare with Figure 3).

The second sort of experiments was devoted to evolutionary design of 1-part developmental program consisting of 10 instructions. A new form of the environment was specified in order to demonstrate the adaptation of the program being evolved to the new conditions of creating generic multipliers. Again, 1000 independent experiments were conducted from which 97 % working solutions were obtained. 85 % of the evolved programs were classified as general. An evolved 4×4 multiplier adapted to the new environment $env = (0, 0, 0, 0)$ is shown in Figure 7. Table 3 shows the appropriate developmental program. This program showed the ability to construct generic multipliers. Note that, in general, for p_0-bit operands the environment would have the form $env = (0, \ldots, 0)$ containing p_0 twos and p_0 developmental steps would be needed to construct a working multiplier.

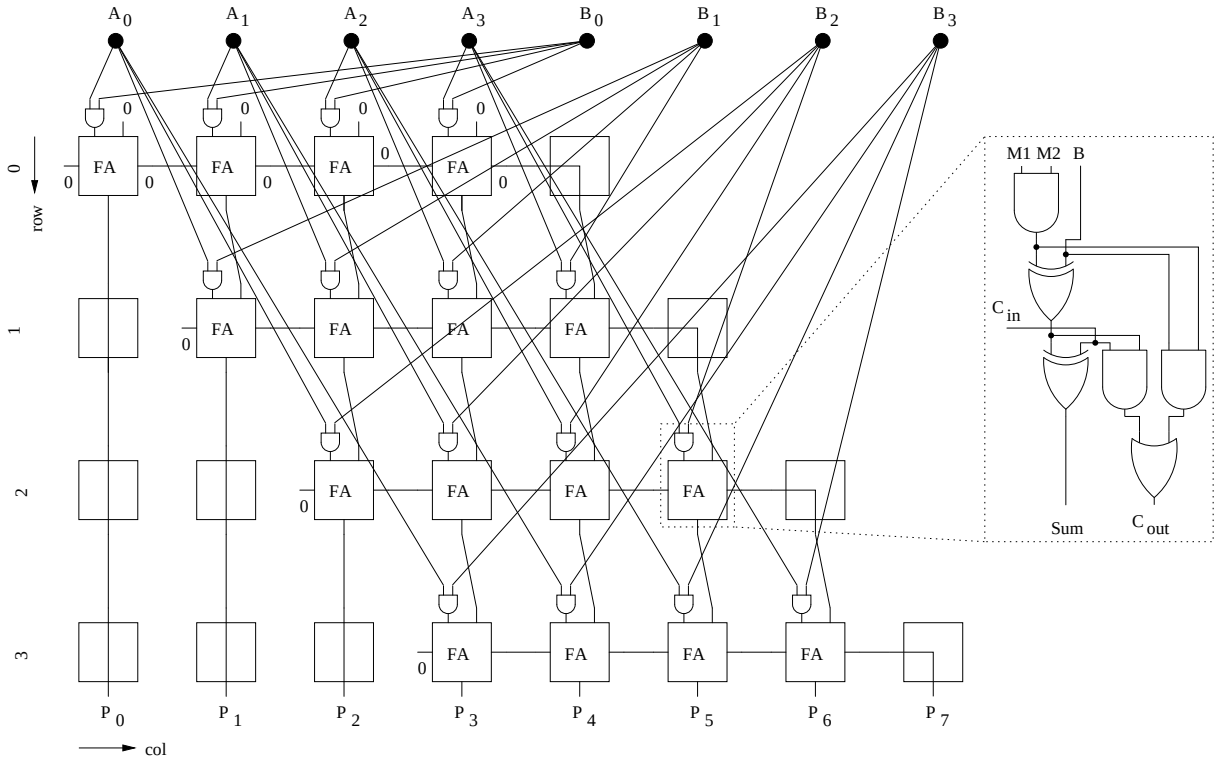

Figure 7: A 4×4 multiplier created by means of evolved program shown in Table 3 adapted to the environment $env = (0, 0, 0, 0)$. $A_0 \ldots A_3, B_0 \ldots B_3$ represent the bits of the operands, $P_0 \ldots P_7$ denote the bits of the product. Logic scheme of the fundamental full adder-based building block (see Fig. 5f) utilized by the evolved program is shown. M1 and M2 denote the multiplicands whose partial product represents the first operand of the full adder, B denotes the second operand, Sum and C_{out} represent the sum and output carry of the full adder.

0: REP v_1 1	4: INC v_0 1	8: SET v_0 v_2
1: GEN ID-1	5: INC v_3 0	9: GEN ID-2
2: REP p_1 2	6: INC v_1 1	
3: GEN FA	7: SET v_3 v_0	

Table 3: Evolved general program by means of which the multiplier from Figure 7 was created. In this case $p_0 = 4$, there is only one program part operating in the environment $env = (0, 0, 0, 0)$.

Experiments for the evolution of 3×3 multipliers were conducted, however, no general solution was obtained. Although basic AND gates and ID functions were available in the set of building blocks, they were rarely used in the design and adders were generated instead. This behavior could be explained by predominating occurence of adders which pushes the evolution to design regular structures, utilizing the properties of the building blocks and their interconnection. The evolved programs exhibit certain degree of redundancy, which is caused by the determination of the program length based on the conventional design. Therefore, there is an additional possibility for reducing the search space. A very good success rate

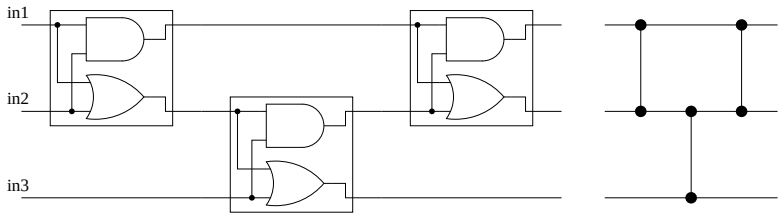

Figure 8: (a) A three-input sorting network consists of three comparators. The blocks marked by rectangles represent the comparators. (b) Alternative symbol of the sorting network.

Inputs	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*
Delay	0	1	3	3	5	5	6	6	7	8	8	9	10	10	10	10
Comparators	0	1	3	5	9	12	16	19	25	29	35	39	45	51	56	60

Table 4: The number of comparators and delay of the best currently known sorting networks

was observed both in the case of the evolution of initial solutions and the occurence of general programs among these solutions after verification especially in the second sort of experiments, which indicates the suitability of the proposed representation to evolve generic structures. However, the selection of building blocks represents a crucial issue for successful evolution of this kind of circuits. Both the programs presented herein showed the ability to construct generic multipliers, which has never been seen before in the field of the evolutionary design.

An environment was integrated into the developmental model in order to allow the system to construct irregular structures (inspired by the conventional multipliers). The system demonstrated a capability of adaptation to another environment that allowed designing generic multipliers exhibiting a high level of regularity in their structures using a program consisting of only one part. Moreover, the adaptation was observed to many other irregular environments and even to random binary environments (i.e. the environments consisting only of values 0 and 1), retaining the ability of the system to develop *generic* multipliers by means of a single program, whose parts are executed according to the environment. Note that this feature is significantly influenced by the grid chosen for representing the circuits and by the general structure and properties of a building block, particularly the facility of degradation of more complex blocks (e.g. full adders) to simpler blocks (e.g. AND gates, ID functions etc.) according to the inputs of the blocks. However, this is significant information with respect to the future research. For example, the development of generic combinational multipliers possessing exactly that structure shown in Figure 3 would not be possible without applying the environment. A variety of building blocks exist which could be involved in the design process in order to develop more complex generic circuits exhibiting irregularities. Therefore, the approach utilizing a form of environment suggests a big space deserving of the subsequent investigation.

5 Development of Generic Sorting Networks

A sorting network is defined as a sequence of compare–swap operations (comparators) that depends only on the number of elements to be sorted, not on the values of the elements. A *compare–swap* of two elements (a, b) compares and exchanges a and b so that we obtain $a \leq b$ after the operation.

The main advantage of the sorting network is that the sequence of comparisons is fixed. Thus it is suitable for parallel processing and hardware implementation, especially if the number of sorted elements is small. Figure 8 shows an example of a 3-input sorting network and the structure of the comparator.

The number of compare–swap components and delay are two crucial parameters of any sorting network. By delay we mean the minimal number of groups of compare–swap components that must be executed sequentially. Designers try to minimize the number of comparators, delay or both parameters. A sorting network is possible to optimize by removing redundant comparators. The redundant comparators does not swap its input values during the complete test of the sorting network, therefore these comparators can be removed from the network without the loss of functionality. Table 4 shows the number of comparators and delay of some of the best currently known sorting networks. Some of these networks were designed (or rediscovered) using evolutionary techniques [2, 4, 3, 8, 11, 10].

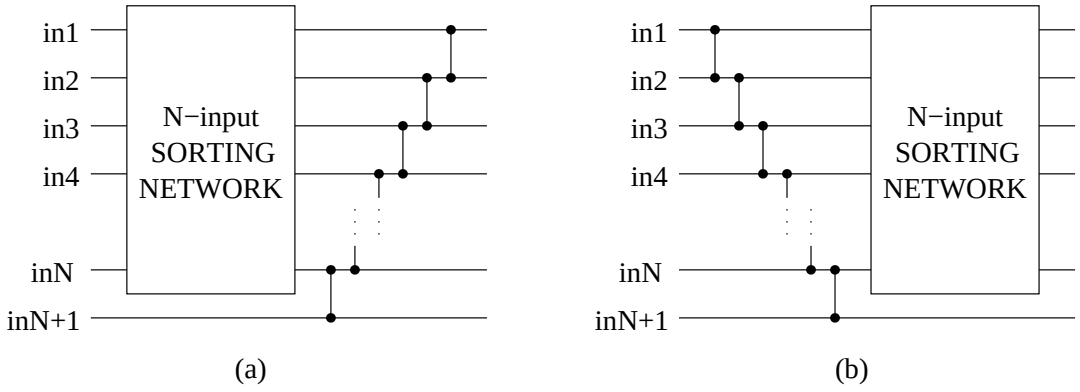

Figure 9: Making (N+1)-sorters from N-sorters: (a) insertion and (b) selection principle

Sorting networks are usually designed for a fixed number of inputs. It is also valid for the mentioned evolutionary approaches. Note, however, that the evolutionary approach is not scalable. Some conventional approaches exist for designing *arbitrarily* large sorting networks. Figure 9 shows two principles for constructing a sorting network for $N + 1$ inputs when an N-input network is given [13].

- Insertion – the (N+1)st input is inserted into a proper place after the first N elements have been sorted – Fig. 9a.
- Selection – the largest input value can be selected before we proceed to sort the remaining ones – Fig. 9b.

Although the sorting networks created by means of insertion or selection principle are not optimal for a particular N, these methods can be treated as generic construction algorithms for this class of circuits. Moreover, they showed to be suitable approaches for the evolutionary design using a developmental encoding. Therefore, we well consider these techniques as a basis for comparing the evolved solutions. It is evident that these conventional approaches enable to "grow" the sorting networks to arbitrary size from a smaller instance. The evolutionary development of generic sorting networks presented in this section is inspired by those conventional methods.

Similar approach, involving an instruction-based developmental encoding for the design of arbitrarily large sorting networks, was introduced in [16], which differs from the developmental system presented herein in the following aspects: (1) The form of the embryo influence the form of the developed sorting networks. (2) Only two types of instructions were utilized for the development: copy and copy-and-modify. Neither variables nor parameters were involved. Loops were not used explicitly, the number of comparators to be processed was specified by the instruction argument and the width of the sorting network being developed. (3) The comparators were represented by the indices of wires they should be connected to. The new comparators were created by copying the existing comparators and possibly modifying their input indices using the instructions of the evolved program.

The novel developmental system for the construction of generic sorting networks is based on the concept introduced in Section 2. The following application-specific setup was utilized for the design of the sorting networks. There are six variables inside the developmental system; parameters are not involved explicitly. Only a single program is evolved, i.e. no environmental information is utilized. Since the sorting networks are intended to be able to "grow", the development runs in steps by means of repeated application of a single program. After each application of the program some comparators are generated next to the existing ones in order to create larger sorting network. A developmental step is defined as a single application of the evolved program. A finite number of the developmental steps represents a developmental sequence. Let us define the size of a developmental step as the difference of the number of inputs of two resulting sorting networks following immediately in the developmental sequence. A comparator is considered as a basic building block for the development of the sorting networks. Four types of comparators are utilized in the evolutionary process which differs in the "width", i.e. the number of wires of the sorting network they are connected over - see Fig. 10a. The structure for the construction of the sorting networks consists of a one-dimensional array in which each element can contain one comparator (see Fig. 10b). A 2-input, 3-input and 4-input embryo is utilized for the development of the sorting networks as shown in Fig. 10c). The embryo is stored in the first e elements

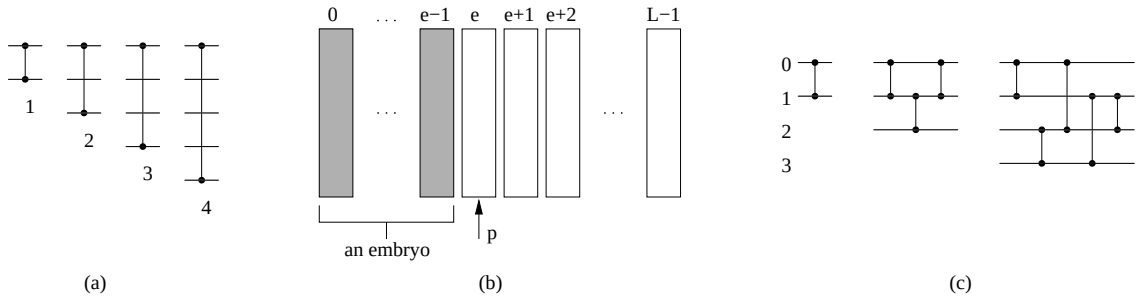

Figure 10: The concept of development of generic sorting networks: (a) The building blocks are represented by the comparators of the width 1, 2, 3 and 4 respectively. (b) The array where the building blocks are generated to during the development. The gray elements denote the building blocks of the embryo. (c) The embryos of the width 2, 3 and 4 respectively, utilized for the development.

of the array depicted in gray and is invariable during the evolutionary and developmental process. The position of the comparators in the sorting network (i.e. the connection to the particular wires) is specified by the value of an arbitrary of the variables v_0, v_1, v_2 or v_3. For example, consider the comparator of width 2 from Fig. 10a and let its position be determined by the variable $v_0 = 1$. Then the first input of this comparator is connected to the wire 1 and the second input to the wire 3. During the development of a sorting network the comparators are generated sequentially into the free positions of the array pointed by the index pointer p according to the program, which is the subject of evolution. Note that the comparator is generated only if p does not exceed the array boundary of L elements and the connection of the comparator does not exceed the width (the number of inputs) of the sorting network being developed. Although the developmental system does not utilize any parameters (constants that are invariable during the developmental process), the variables are initialized before proceeding each developmental steps to the specific values which were determined experimentally. The variables are initialized as follows: $v_0 = 0, v_1 = w - 2, v_2 = w - 2, v_3 = w - 3, v_4 = w - 4, v_5 = w$, where w denotes the number of inputs (width) of the sorting network to be developed in the forthcoming developmental step. Note that the initial values of the variables may be changed during the development by the appropriate instructions of the evolved program.

The candidate solutions are evaluated typically for three developmental steps, i.e. for each chromosome in the population a finite developmental sequence is created consisting of three sorting networks of different sizes depending on the width of the embryo and the size of the developmental step. Each sorting network in the developmental sequence is evaluated by computing a fitness value as the number of correct output bits for all binary test vectors. The fitness value of a chromosome is calculated as the sum of the fitness values of all sorting networks in the developmental sequence. For example, let $w_e = 2$ be the number of inputs of the embryo, $w_s = 2$ be the size of the developmental step and $n = 3$ be the number of developmental steps to be conducted. Then the total fitness value of a chromosome is calculated as $F_t = f_4 + f_6 + f_8$, where f_k denotes the fitness value of the k-input sorting network ($k = w_e + w_s$, $w_e + 2w_s$ and $w_e + 3w_s$). In this case the maximal fitness value of a chromosome is $F_m = 4 \cdot 2^4 + 6 \cdot 2^6 + 8 \cdot 2^8 = 2048$. Similarly to the development of combinational multipliers presented in Section 4, a comparator is considered as elementary sorting network component. Neither delay nor the number of comparators are optimized during evaluation of the candidate solutions; the selected resulting programs have subsequently been analyzed in order to identify the best sorting networks developed by means of the evolved programs.

5.1 Experimental Results and Discussion

In case of the development of generic sorting networks three sorts of experiments were conducted which differ by the size of the developmental step and the embryo utilized. In all the experiments presented in this section each chromosome of the evolutionary algorithm contains a single program consisting of eight instructions.

The first sort of experiments dealt with the development of arbitrary even- input sorting networks from a two-input embryo when the size of the developmental step was set to 2. 1000 independent experiments were conducted from which 90 % finished successfully in 40000 generations of the evolutionary algorithm and 98 % of the evolved programs were classified as general. Figure 11a shows a sequence of sorting networks developed by the three steps of the best evolved program shown in Fig. 11b. The sorting

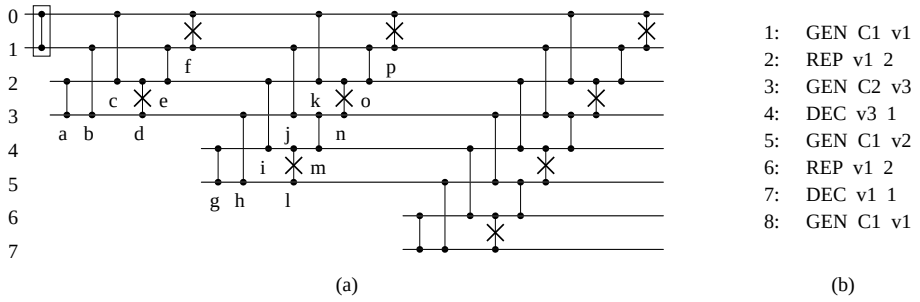

Figure 11: (a) Even-input sorting networks developed from a two-input embryo using the developmental step of size 2. The crossed comparators are redundant in the sorting network, therefore, they can be removed from the network without any loss of its functionality. (b) The evolved general program for the development of these sorting networks.

networks have been constructed from a two-input embryo by the following process. Let $w_e = 2$ denote the number of inputs of the embryo, $s = 2$ denote the size of the developmental step and $w = v_e + i \cdot s$ denote the width of the sorting network to be developed in the i-th developmental step. Recall the values of appropriate variables involved in the evolved program are initialized as $v_1 = w - 2, v_2 - 2, v_3 - 3$. For the first developmental step ($i = 1$), $w = 2 + 1 \cdot 2 = 4$, therefore, $v_1 = 2, v_2 = 2$ and $v_3 = 1$. Considering these initial values the first instruction 1 from Fig. 11b generates the comparator of width 1 labeled as a in Fig. 11a which is connected to the wires denoted by indices 2 and 3. The instruction 2 initiates a loop repeating 2 times (since $v_1 = 2$) two following instructions. During the first pass of this loop instruction 3 generates comparator b whose connection to the sorting network is determined by $v_3 = 1$ and instruction 4 decreases v_3 by one, i.e. $v_3 = 0$ at the moment. Similarly, comparator c is generated in the second pass of the loop considering the actual value of v_3. Note that negative values are not allowed, therefore, the execution of instruction 4 during the second pass of the loop has no effect in this step. Instruction 5 generates comparator d with respect to the value of $v_2 = 2$. Instruction 6 initiates a loop to be repeated 2 times (since $v_1 = 2$) and the two following instructions 7 and 8 result in generating comparators e, f in each pass of this loop. The first developmental step is now finished. At the beginning of the second developmental step the variables are initialized to the new values with respect to actual w. During the second developmental step comparator g is generated by the instruction 1 and comparators from h to k are generated by the loop initiated by instruction 2. Then comparator l is generated by instruction 5 and comparators $m - p$ are generated by the loop initiated by instruction 6. The next developmental steps proceed in the same way the consequence of which is the "growth" of the sorting network. Note that this program was verified for generality, i.e. arbitrary even-input sorting network can be created. However, the analysis of the developed sorting networks indicates that there are redundant comparators in these networks which can be removed without the loss of its functionality. Therefore, these sorting networks are optimized both from the point of view of the number of comparators and delay. Note that the redundant comparators in Fig. 11 are crossed.

In the second sort of experiments sorting networks were developed from a three-input embryo considering the size of the developmental step $s = 3$. Therefore, 6-input, 9-input, 12-input etc. sorting networks could be designed by means of the evolved programs. From 1000 independent experiments conducted 88 % of working programs were evolved from which 99 % were classified as general. Figure 12 shows the best and most interesting result obtained in this sort of experiments. The evolved program, which was classified as general, produces sorting networks without any redundant comparators. Moreover, there are both even-input and odd-input sorting networks in a single developmental sequence (because of the size of the developmental step $s = 3$). This result represents the first case of observing such a behavior that was not achieved in the developmental system introduced in [16]. Note that the structure of the sorting networks and the evolved program is very similar in comparison with that shown in Fig. 11. In addition, the algorithm from Fig. 11 (without any modifications) showed the ability to construct sorting networks with the size of the developmental step $s = 3$. The only difference is the dashed line drawn comparator shifted before its predecessor in each developmental step (caused by different variable in instruction 5 determining the connection of the comparator to be generated, see Fig. 12) which, however, results in better delay in comparison with the solution constructed by means of the program from Fig. 11b.

The goal of the third sort of experiments was to develop arbitrary even-input sorting networks con-

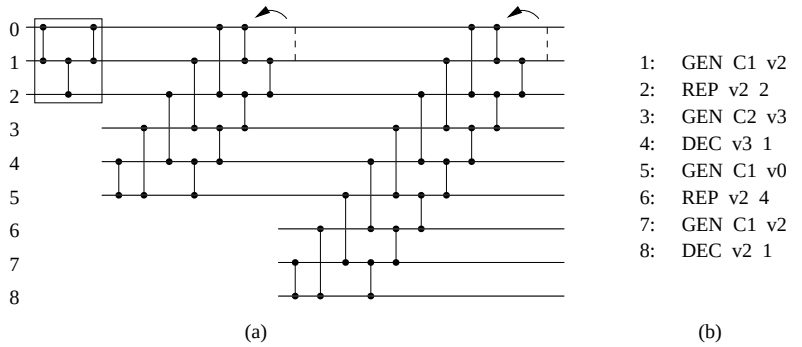

Figure 12: (a) Sorting networks developed from a three-input embryo using the developmental step of size 3. (b) The evolved general program. Note that this solution constructs sorting networks without any redundant comparators.

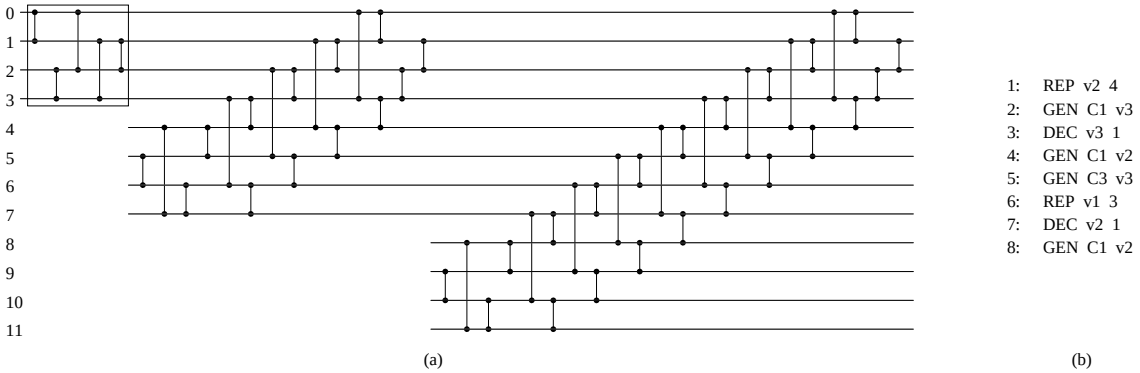

Figure 13: (a) Even-input sorting networks developed from a four-input embryo using the developmental step of size 4. Note that only the effective (non-redundant) comparators are shown. (b) The evolved general program.

sidering the size of the developmental step $s = 4$ and a four-input embryo. Since there are 8-input, 12-input etc. sorting networks in the developmental sequence considering the four-input embryo and the developmental step of size 4, only two developmental steps were performed for the fitness calculation because of very time-consuming evaluation of such large sorting networks. 500 independent experiments were conducted from which 34 % evolved a working general solution, i.e. 100 % successfulness of the evolved programs. Figure 13 shows one of the best evolved program together with the optimized sorting networks developed by means of it.

Tables 5 and 6 summarize the number of comparators and delay of the developed sorting networks and their optimized variants for selected numbers of inputs of the sorting networks. It is evident that all the sorting networks presented in this paper (Figs. 11, 12 and 13) exhibit better properties from the both point of view of the number of comparators and delay in comparison with the general conventional principle of the same type (straight-insertion sort). Note that the optimized sorting networks created using the developmental step of size 2 corresponds to the best sorting networks developed in [16]. Moreover, general programs were evolved herein for the developmental step of sizes 3 and 4 that were not achieved in [16] and these sorting networks also exhibit better properties in comparison with the conventional solution. In case of the step of size 3 a general program was evolved which even constructs sorting networks without any redundant comparators. The results presented in this section suggest that this instruction-based developmental model is more robust and flexible in comparison to the system introduced in [16].

6 Conclusions

In this paper a new approach to the computational development in the area of evolutionary design was proposed: a general instruction-based model. The goal was to present an ability of the evolutionary

Inputs	*8*	*9*	*10*	*12*	*14*	*15*	*16*	*18*	*20*	*21*	*22*	*24*	*26*	*27*	*28*
Conventional	28	36	45	66	91	105	120	153	190	210	231	276	325	351	378
Evolved: step 2	31		49	71	97		126	161	199		241	287	337		391
	22		**35**	**51**	**70**		**92**	**117**	**145**		**176**	**210**	**247**		**287**
Evolved: step 3		29		51		79		113		153		199		251	
Evolved: step 4	29			69			125		197			285			389
	23			**53**			**95**		**149**			**215**			**293**

Table 5: The number of comparators of the evolved sorting networks for different sizes of the developmental step in comparison with the conventional straight- insertion sorting networks. The bold values represent the number of comparators of the optimized sorting networks (after removing the redundant comparators). Note that the sorting networks created using the developmental step of size 3 do not contain any redundant comparators.

Inputs	*8*	*9*	*10*	*12*	*14*	*15*	*16*	*18*	*20*	*21*	*22*	*24*	*26*	*27*	*28*
Conventional	13	15	17	21	25	27	29	33	37	39	41	45	49	51	53
Evolved: step 2	15		20	25	30		35	40	45		50	55	60		65
	9		**12**	**15**	**18**		**21**	**24**	**27**		**30**	**33**	**36**		**39**
Evolved: step 3		12		17		22		27		323		37		42	
Evolved: step 4	14			28			46		68			94			124
	9			**15**			**21**		**27**			**33**			**39**

Table 6: Delay of the evolved sorting networks for different sizes of the developmental step in comparison with the conventional straight-insertion sorting networks. The bold values represent the delay of the optimized sorting networks (after removing the redundant comparators). Note that the sorting networks created using the developmental step of size 3 do not contain any redundant comparators.

developmental system to design generic structures of slacable digital circuits. Two case-studies were presented: (1) the evolution of generic multipliers and (2) the evolution of generic sorting networks.

In case of the development of multipliers a specific form of environment was integrated into the developmental model representing an external control of the developmental process which is intended as a tool enabling the design of irregular structures. Moreover, the environment was utilized in order to demonstrate adaptation of the development, retaining its ability to design generic multipliers. The experiments confirmed the capability of adaptation in connection with the proposed circuit representation. General programs were evolved for the construction of multipliers which exhibit a high degree of regularity in the circuit structure. Since the multipliers of different sizes are constructed every time from scratch by means of an evolved program, utilizing the bit-width of the operands as a parameter for determining the circuit structure, it is a case of parametric developmental design. Note that several developmental steps are needed to construct a single working circuit.

An iterative approach was utilized for the design of generic sorting networks from the embryo. Therefore, the sorting network is able to "grow" to theoretically arbitrary size. General programs were evolved for the sizes of the developmental steps 2, 3 and 4 that was not achieved before. Moreover, the evolution showed an ability to design innovative solutions in all those cases. The best general programs working with the developmental step of size 2 and 4 produce sorting networks that exhibit the same qualities (i.e. the number of comparators and delay) like the best solutions developed in [16]. Although the sorting networks developed with the step of size 3 do not achieve the qualities of the best solutions, their number of comparators and delay are better in comparison with the conventional solution. However, the evolved program constructs solutions with no redundant comparators.

Similar model based on this instruction-based development was utilized also for the evolutionary design of efficient carry-save multipliers [1]. Considering the manner of the development of sorting networks presented herein, the resulting developed solution is fully functional after each developmental step which poses a significant difference in the approach compared to the development of multipliers. This demonstrates that the instruction-based model is possible to use in a wide range of applications of various traits.

The open questions, however, regard to the choice of suitable building blocks for a given application. In case of the multipliers, relatively complex building blocks (half- and full adders) had to be utilized

for the successful design which was inspired by the structure of the conventional multipliers. No general solution has yet been evolved at a lower-level representation of the circuit. The proposed evolutionary developmental system was implemented using a software simulator. In order to speed-up the automatic design process, hardware realization could be useful. The instruction-based development actually represent a simple application-specific machine language that could be executed directly (or after a few modifications) on processors that are commonly available in some FPGAs. The regular representation of the circuits to be developed is evidently suitable for implementation using the configurable logic blocks of the FPGAs. If this hardware implementation was realized, the candidate circuits would be possible to evaluate in shorter time in comparison with the software simulation, which would additionally speed-up the evolution. These issues represent the ideas for our next research.

Acknowledgements

This research was supported by the Grant Agency of the Czech Republic under contract No. 102/07/0850 *Design and hardware implementation of a patent-invention machine*, No. 102/05/H050 *Integrated Approach to Education of PhD Students in the Area of Parallel and Distributed Systems* and the Research Plan No. MSM 0021630528 *Security-Oriented Research in Information Technology.*

References

[1] M. Bidlo. Evolutionary design of generic combinational multipliers using development. In *Proc. of the International Conference on Evolvable Systems: From Biology to Hardware (ICES 2007), Lecture Notes in Computer Science vol. 4684*, pages 77–88. Springer, 2007.

[2] S.-S. Choi and B.-R. Moon. A hybrid genetic search for the sorting network problem with evolving parallel layers. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 258–265, San Francisco, California, USA, 2001. Morgan Kaufmann.

[3] S.-S. Choi and B. R. Moon. Isomorphism, normalization, and a genetic algorithm for sorting network optimization. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 327–334, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc.

[4] S.-S. Choi and B. R. Moon. More effective genetic search for the sorting network problem. In *Proc. of the Genetic and Evolutionary Computation Conference GECCO 2002*, pages 335–342, New York, US, 2002. Morgan Kaufmann.

[5] B. A. et al. *Essential Cell Biology, 2nd edition.* Garland Science/Taylor & Francis Group, 2003.

[6] T. G. W. Gordon and P. J. Bentley. Towards development in evolvable hardware. In *Proc. of the 2002 NASA/DoD Conference on Evolvable Hardware*, pages 241–250, Washington D.C., US, 2002. IEEE Press.

[7] F. Gruau. Neural network synthesis using cellular encoding and the genetic algorithm, PhD thesis. Technical report, Laboratoire de l'Informatique du Parallelisme, Ecole Normale Superieure de Lyon, France, 1994.

[8] W. D. Hillis. Co-evolving parasites improve simulated evolution as an optimization procedure. *Physica D*, 42(1–3):228–234, June 1990.

[9] G. S. Hornby and J. B. Pollack. The advantages of generative grammatical encodings for physical design. In *Proc. of the 2001 Congress on Evolutionary Computation*, pages 600–607. IEEE Press, 2001.

[10] J. R. Koza et al. *Genetic Programming III: Darwinian Invention and Problem Solving.* Morgan Kaufmann, San Francisco, 1999.

[11] H. Juillé. Evolution of non-deterministic incremental algorithms as a new approach for search in state spaces. In *Proc. of 6th Int. Conference on Genetic Algorithms*, pages 351–358. Morgan Kaufmann, 1995.

[12] H. Kitano. Designing neural networks using genetic algorithms with graph generation system. *Complex Systems*, 4(4):461–475, 1990.

[13] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching (2nd ed.)*. Addison Wesley, 1998.

[14] S. Kumar. Investigating computational models of development for the construction of shape and form, PhD thesis. Technical report, Department of Computer Science, University College London, 2004.

[15] J. F. Miller and P. Thomson. A developmental method for growing graphs and circuits. In *Proc. of the 5th Conf. on Evolvable Systems: From Biology to Hardware (ICES 2003), Lecture Notes in Computer Science, vol. 2606*, pages 93–104, Berlin DE, 2003. Springer-Verlag.

[16] L. Sekanina and M. Bidlo. Evolutionary design of arbitrarily large sorting networks using development. *Genetic Programming and Evolvable Machines*, 6(3):319–347, 2005.

[17] L. Sekanina and R. Růžička. Easily testable image operators: The class of circuits where evolution beats engineers. In *Proc. of the 2003 NASA/DoD Conference on Evolvable Hardware*, pages 135–144. IEEE Computer Society Press, 2003.

[18] A. Thompson. Silicon evolution. In *Proc. of Genetic Programming GP'96*, pages 555–452. MIT Press, 1996.

[19] G. Tufte and P. C. Haddow. Extending artificial development: Exploiting environmental information for the achievement of phenotypic plasticity. In *Proc. of the 7th International Conference on Evolvable Systems: From Biology to Hardware (ICES 2007), Lecture Notes in Computer Science, vol. 4684*, pages 297–308, Berlin Heidelberg New York, 2007. Springer.

[20] J. F. Wakerly. *Digital Design: Principles and Practice*. Prentice Hall, New Jersey, US, 2001.