A Developmental Method for Construction of Arbitrarily Large Sorting Networks and Adders

Michal Bidlo
Faculty of Information Technology, Brno University of Technology
Božetěchova 2, 612 66 Brno, Czech Republic
bidlom@fit.vutbr.cz

ABSTRACT

The paper deals with a non-traditional design method inspired by natural ontogenesis (an embryonic approach) for construction of combinational logic circuits. The general principle of the technique is based on a set of proper instructions known beforehand that are repeatedly applied on the embryo (a trivial instance of a problem) to construct more complex system. Genetic algorithm is used to find a suitable sequence of instructions – a prescription for the growth of the embryo. We will focus on the design of arbitrarily large sorting networks and adders because these circuits have shown to be good candidates for employing development. The system complexity can increase continually and infinitely. It is shown that by employing of this approach the genetic algorithm is able to (1) rediscover the principle of already known method and (2) find a novel algorithm, by means of which we can obtain better solutions in comparison with a conventional method.

Keywords

Genetic algorithm, development, digital circuits design, sorting network, binary adder

1. INTRODUCTION

Nowadays, biologically-inspired algorithms are widely used to solve many complex problems in computer science. Evolutionary design belongs to an emerging field, in which a lot of interesting results have been gained up to now [25]. Boolean circuits [21], artificial neural networks [30], computer programs [15] are the typical representatives, which were successfully evolved in the past years. Although several modifications of evolutionary algorithms were utilized (depending on a particular domain), only relatively small instances of the problem were obtained.

For instance, the problem of scale is probably the most restricting issue of the evolutionary design. As the system complexity increases (e.g. a number of inputs and outputs or a number of components needed for the implementation), the length of the genotypes increases in the case of the evolutionary approach. In consequence of this the search spaces become very large and then it is usually extremely difficult to design an effective search algorithm.

However, several methods have been developed in order to overcome the scaling problem, which may be divided in three classes: *evolution at the functional level* (for instance, see [20]), *incremental evolution* (e.g. Torreses's *divide-and-conquer* approach [28]) and *development* (an embryonic approach, e.g. [8, 10]). We will deal with development in this paper.

The objective of this paper is to propose a developmental method for the design of arbitrarily large combinational logic circuits and show that by employing of this approach we can obtain better solutions in comparison with a conventional algorithm or rediscover the principle of an already known technique (e.g. for sorting networks and adders).

The organization of the paper is as follows. Section 2 gives a more detailed view to the development and discusses common results obtained in the area of the evolutionary design by means of a developmental process. An outline of the circuits to be designed (sorting networks and adders) with commonly used construction methods and preceding results obtained by the evolutionary design with development is in section 3. In section 4 we describe the system employed for the developmental evolutionary design of sorting networks and adders. Experimental results of the research are given in section 5. Finally, in section 6 we evaluate the obtained results and give some conclusion remarks.

2. DEVELOPMENT

Let's now briefly describe the basic principles of development. Development is in its original form a biological process that in essence "emerges an organized structures from an initially very simple group of cells" [29]. It is a process of construction that emerges from the interplay between proteins, genes, cells and the environment, resulting in the formation of an organism. Central to development is construction and self-organization. All cellular behaviour is controlled by proteins, which are produced by genes. Cleavage divisions, pattern formation, morphogenesis, cellular differentiation and growth are the main processes involved in biological development [26].

Considering the evolutionary design in the computer science, development is usually understood as a genotype–phenotype mapping by means of some (complex) rules. We will utilize development mainly for the "growth" of the system complexity to overcome the problem of scale in the evolutionary design of combinational logic circuits. In such a case the genotype has to contain a prescription for the construction of the system from a simple initial instance of the problem – an embryo.

Models of development were surveyed in Chapter 2 of [19] and [26]. Here we will briefly recall only a class of models related (in some way) to our work – the evolutionary design of arbitrarily large combinational circuits.

In bioinspired hardware and software systems the genotype–pheno-type mapping is often implemented by means of *rewriting systems*. The first rewriting developmental (neuro) system was investigated by Kitano [17]. Later, among others, Boers and Kuiper have utilized L-systems to create the architecture of feed-forward artificial neural networks [4]. Haddow et al. have adopted L-system in order to evolve scalable digital circuits [11]. Three-dimensional mechanical objects have been designed by evolution that also utilized a variant of L-system in its genotype–phenotype map [13].

John Koza introduced an original method in which novel analog circuits have been constructed according to the instructions produced by genetic programming [15]. Among others activities Koza's team employed this technique for routine duplication of many patented inventions (see the list at [27]).

In another approach, Gordon and Bentley have utilized the interaction of artificial genes and proteins to model the development in digital circuits [8]. CAM Brain machine [9] and POEtic platform [6] are examples of those systems that use cellular automata-based development.

Miller and Thomson have invented a developmental method for growing graphs and circuits using Cartesian genetic programming in order to evolve similar constructors to ours (referred to as iterators in [24]). Because they worked at a very low level of abstraction (as configuration bits of a hypothetical reconfigurable hardware) no general constructor has been found for their task, i.e. the design of large even parity circuits. However, other researchers have successfully evolved completely general solutions to the even-parity problem; for instance Huelsbergen, who has worked at the machine code level [14].

In order to evolve 3D shape and form Kumar has used complex and so realistic models of development inspired by genetic regulatory networks [19]. Bentley has invented fractal proteins for the same purpose. A fractal protein is a finite square subset of Mandelbrot set, defined by three artificial codons that form the coding region of a gene in the genome of a cell [3].

These methods have illustrated various approaches to the development, however, only a few of them were successful with designing large real-world systems.

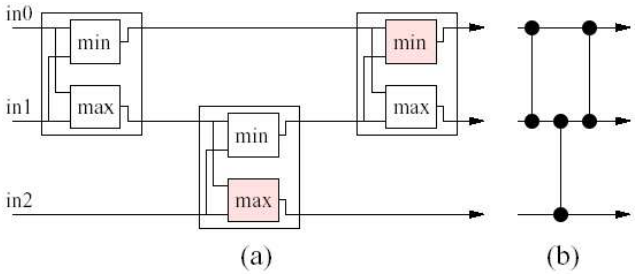

Figure 1: (a) A 3-input sorting network consists of 3 comparators, each of which contains components for computing minumim and maximum. (b) Alternative symbol. The sorting network may be described by a sequence of pairs containing the indices of inputs of individual comparators, for this example: $(\mathbf{0},\mathbf{1})(\mathbf{1},\mathbf{2})(\mathbf{0},\mathbf{1})$.

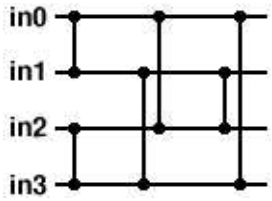

Figure 2: An example of 3-delay sorting network. The groups of independent comparators are $(\mathbf{0},\mathbf{1})(\mathbf{2},\mathbf{3})$, $(0,2)(1,3)$ and $(1,2)(0,3)$. Comparator $(0,3)$ is redundant, so it can be removed.

3. CONSIDERED TARGET CIRCUITS

We are considering the design of arbitrarily large sorting networks and adders because these types of combinational logic circuits have shown to be good candidates for employing development.

3.1 Sorting Networks

The concept of sorting networks was introduced in 1954; rigorous theory is summarized in [18]. Consider a *compare–swap* operation that compares and possibly swaps the values of its two operands (a, b), so that we obtain a pair (a, b) satisfying $a \leq b$ after execution. A sorting network is a sequence of compare–swap operations that depends only on the number of elements to be sorted [18]. Let's call the compare–swap operation as a comparator. An advantage of sorting networks against the classical sorting algorithms is that the number of comparators is fixed for a given number of inputs. Thus they can be easily implemented in hardware. Figure 1 shows an example of a three-input sorting network and its alternative symbol.

The two main aspects determining the quality of sorting network are the number of comparators (the fewer comparators, the lower implementation cost) and delay (the lower delay, the faster sorting). Let's define the delay of a sorting network as the number of groups of independent comparators (i.e. the groups of comparators, whose input indices are mutually different). Such the comparators may be performed in parallel. We denote a comparator of a sorting network as *redundant* if it does not swap any input values during the

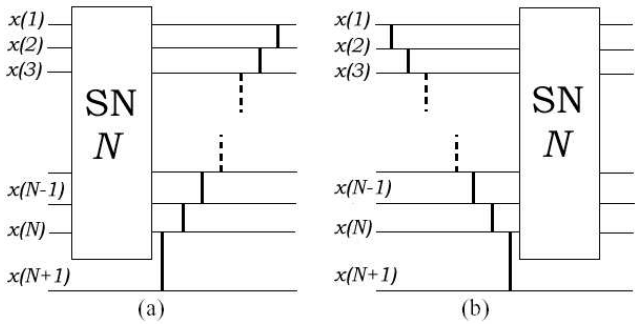

Figure 3: Methods for creating larger sorting networks from smaller structures: (a) straight insertion, (b) selection

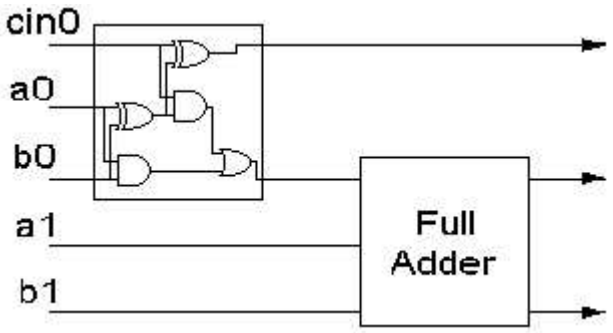

Figure 4: Construction of arbitrarily large binary adders by means of full adders as the building blocks. Logic structure of the full adder is shown.

complete test of the sorting network. Such a comparator can be removed from the sorting network without any loss of functionality. For instance, the sorting network in Figure 2 possesses the delay value 3 and the comparator $(0, 3)$ is redundant.

We will focus on optimizing of these properties of resulting sorting networks. Table 1 summarizes the properties of currently best-known structures. The (13–16)-sorting networks were discovered using evolutionary algorithms [5, 12, 16, 15]. Because of the scaling problem, the direct evolutionary design of larger sorting networks is very difficult today. The two methods shown in Figure 3 may be considered as the conventional approaches to the construction of larger sorting network from the smaller one.

3.2 Adders

In general, an adder may be considered as an abstract component with n inputs and m outputs. Usually n comprises $2w$ bits of the numbers to be added and one input carry and m equals $w + 1$ in boolean logic. Then w denotes the *width* of the adder. The common implementations of adders (e.g. carry-propagate adders, ripple-carry adders etc.) constitute a class of well-scalable combinational logic circuits and hence they might be suitable for the developmental method described in Section 4.

As an example we give some references dealing with digital circuits design which (at least partially) showed an ability of the evolutionary algorithm to design adders. First of all, Miller et al. utilized Cartesian genetic programming [23] for the evolutionary design of digital circuits (including adders and multipliers) at the gate level [21, 22]. Gordon modelled autocatalytic gene networks for pattern formation which is subsequently mapped to the circuit synthesis (a developmental approach) [8, 7]. Shanthi et al. showed in [1] that when the Developmental cartesian genetic programming approach [24] is applied to partitioned digital circuits, it is possible to evolve 8x8 adders or multipliers for example.

All the methods mentioned above create the digital circuits with a given number of inputs. However, in the next section we propose the approach to the construction of arbitrarily large circuits by means of an *evolved program*. Figure 4 shows one of the common approaches to the construction of arbitrarily large binary adders (the carry-propagate adders in this case).

4. THE DEVELOPMENTAL METHOD

As noted above, the main goal of this paper is to propose a method, which enables us to construct arbitrarily large combinational logic circuits. The sorting networks and adders showed to be the proper candidates for demonstrating the functionality of the designed approach.

We employ genetic algorithm to evolve a sequence of instructions (a program), by means of which an initial simple instance of the problem (an embryo) will *grow* continually and infinitely. The program represents the *rules* of the development. Let's call the sequence of instructions as a *constructor*. As the size of the circuit grows (i.e. the number of inputs and outputs), its complexity grows too (i.e. the number of components needed for the implementation of a desired function and their interconnections). The general format of an instruction is of the form (*opcode* $arg1$ $arg2$), where *opcode* represents operational code of the instruction, $arg1$ and $arg2$ are its arguments. The meaning of arguments depends on the type of the instruction (instructions with no operands are allowed; then both $arg1$ and $arg2$ arrays can be arbitrary). The form of instruction set is an application-specific problem and it showed to be one of the most crucial matter of the algorithm. We are interested mainly in such the constructors that are able to build combinational logic circuits of arbitrary size (i.e. by applying the constructor repeatedly on the embryo, we gain a fully functional circuit with all the functionality occuring in the precursor). Such the constructor is referred to as a *general constructor*. In this paper, we require the general constructor to be able build fully functional circuits with at least 28 input bits. A *developmental step* is understood as an application of the constructor on the embryonal structure to construct more complex system. After its application the number of input bits of the circuit increases of the size of the developmental step.

4.1 Sorting Networks

Let's consider the sorting networks at the compare–swap operation level, i.e. the comparator is the basic building block for the evolutionary design. The initial configuration of the developmental system for the sorting networks is shown in Figure 5. The embryo pointer (ep) denotes the comparator

Sorting networks designed individually																
N	**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
Number of comparators	0	1	3	5	9	12	16	19	25	29	35	39	45	51	56	60

Table 1: Properties of currently best-known sorting networks

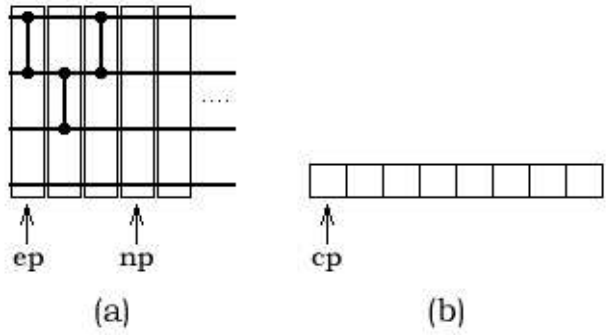

Figure 5: Initialization of the development: (a) growing sorting network (an embryo), (b) constructor represented by a chromozome of the genetic algorithm

that is actually processed by the instruction pointed by the instruction pointer (cp). The new comparator will be placed on the position denoted by the next-position pointer (np). At the beginning of each developmental step (an application of the constructor), the next-position pointer (np) is initialized by the index of the first free position of the embryo. The instructions in the constructor are executed serially until the last instruction is applied or the end of the embryo is reached. After executing of an instruction the pointers ep, cp and np are updated.

The function of the comparator is fixed, so only its inputs may be modified. With respect to that a suitable instruction set has to be chosen. The *copy* and *copy-and-modify* instructions have shown to be suitable for the sorting network design. Each of theses types of instruction is used in two variants in the developmental system. The variation of these types lies in updating of the embryo pointer (ep). Table 2 shows the instruction set of the developmental system for the sorting networks.

4.2 Adders

Binary adders constitute a class of combinational logic circuits, which are, for instance, in comparison with the sorting networks, easier to design (e.g. adders have smaller number of output bits). As we are representing the building block of the sorting network by a comparator (which can be understood as the elementary 2-input sorting network), the building block of a general adder might be represented by the one-bit full adder. However, such an element haven't shown to be suitable enough (even at gate level) for the evolutionary design of adders because the construction of larger circuits is then very easy and does not lead to non-trivial results. Therefore, our goal is to minimize the amount of knowledge provided by the building blocks and embryo and

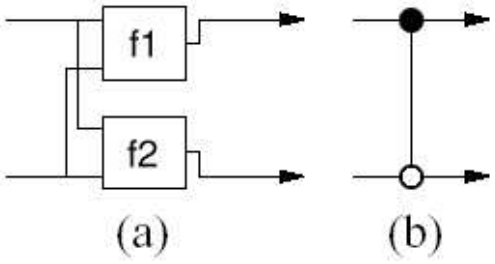

Figure 6: The form of the building block at the gate level: (a) logic structure, (b) symbolic notatin. This circuit can be represented by a 4-tuple (i, j, f_1, f_2), where i and j are the indices of inputs, f_1 and f_2 represent functions. One-input functions are also allowed.

Symbol	Meaning
●	$f = i_1\ AND\ i_2$
○	$f = i_1\ OR\ i_2$
$\oplus$	$f = i_1\ XOR\ i_2$
$-$	$f = i_1(i_2)$

Table 3: Symbols of the logic functions utilized for the adder design. i_1 and i_2 are the inputs, f represents the result of the function.

let the genetic algorithm evolve the most of the circuit structure (more exactly, the prescription for its construction) automatically.

Let's consider all the parts of adders to be logic gates, i.e. the construction program will operate at the gate level. Note that only two-input gates are utilized. The general form of a building block is shown in Figure 6. The combinational logic circuit is described by a sequence of the building blocks. Table 3 shows the logic functions and their symbols utilized for the construction of adders.

The evolving program processes the building blocks of the circuit in the same manner as in the case of the sorting networks. In addition, "empty" embryo is allowed for fully automatic design using evolutionary algorithm. For instance, such the embryo is encoded in a 4-tuple $(0, 0, -, -)$.

The instruction set has to be adjusted to the building blocks and circuit to be designed. Besides the input indices, the building block contains two functions and there are instructions in the instruction set, which are able to set or change these functions. Table 4 shows the instruction set utilized for the developmental rules in the evolutionary design of adders at the gate level.

Op. code	Name	Arg1	Arg2	Meaning
0	ModifyS	p	q	$i_1 = (i_1 + p) \bmod w, i_2 = (i_2 + q) \bmod w, cp = cp + 1,$ $np = np + 1$
1	ModifyN	p	q	$i_1 = (i_1 + p) \bmod w, i_2 = (i_2 + q) \bmod w, cp = cp + 1,$ $ep = ep + 1, np = np + 1$
2	CopyS	p	–	copy $w - p$ comparators, $cp = cp + 1, np = np + w - p$
3	CopyN	p	–	copy $w - p$ comparators, $cp = cp + 1, ep = ep + w - p,$ $np = np + w - p$

Table 2: The instruction set of the developmental system for the sorting networks. p and q represent the arguments of the instruction, i_1 and i_2 denote the indices of the inputs of the comparator and w is the number of inputs of the sorting network being created.

Op. code	Name	Arg1	Arg2	Meaning
0	COPYS	–	–	copy the gates from ep to np; $np = np + 1$
1	COPYN	–	–	copy the gates from ep to np; $np = np + 1, ep = ep + 1$
2	CPMIS	p	q	copy the gates from ep to np and do $i_1 = w - p, i_2 = w - q, cp = cp + 1, np = np + 1$
3	CPMIN	p	q	copy the gate from ep to np and do $i_1 = w - p, i_2 = w - q, cp = cp + 1, np = np + 1, ep = ep + 1$
4	CPMFS	p	q	copy the gates from ep to np and do $f_1 = p, f_2 = q, cp = cp + 1, np = np + 1$
5	CPMFN	p	q	copy the gates from ep to np and do $f_1 = p, f_2 = q, cp = cp + 1, np = np + 1, ep = ep + 1$
6	MODIS	p	q	modify inputs of the gates at ep as follows: $i_1 = w - p, i_2 = w - q; cp = cp + 1$
7	MODIN	p	q	modify inputs of the gates at ep as follows: $i_1 = w - p, i_2 = w - q; cp = cp + 1, ep = ep + 1$
8	MODFS	p	q	modify functions of the gates at ep as follows: $f_1 = p, f_2 = q; cp = cp + 1$
9	MODFN	p	q	modify functions of the gates at ep as follows: $f_1 = p, f_2 = q; cp = cp + 1, ep = ep + 1$

Table 4: The instruction set of the developmental system for the construction of adders at the gate level. p and q represent the arguments of the instruction, i_1 and i_2 denote the indices of inputs of the gates, f_1 and f_2 are the functions of the gates and w is the number of inputs of the circuit being created.

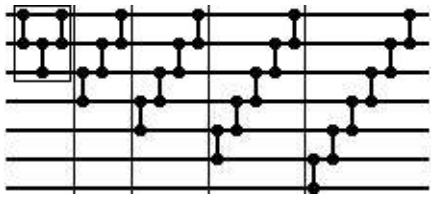

Figure 7: Sorting networks constructed by means of constructor $\mathcal{A}$

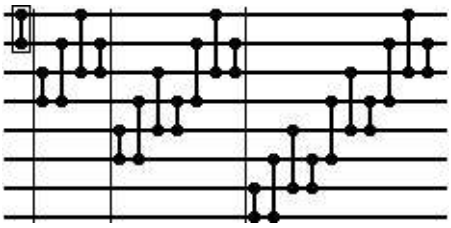

Figure 8: Sorting networks constructed by means of constructor $\mathcal{B}$

5. OBTAINED RESULTS

Some interesting results were obtained in the construction of combinational logic circuits using the development in the evolutionary design. The genetic algorithm was employed to find a constructor representing the developmental rules. In all the cases initial population is seeded randomly using fixed-length chromozomes. A tournament selection mechanism with base 4 is utilized. The length of the chromozome equals the number of instructions in the constructor, i.e. the chromozome consists of a triple number of integers.

5.1 Sorting Networks

Various embryos and various developmental steps were applied for the construction of sorting networks. For a detailed survey of all the results obtained in the sorting network design see [2]. Here we mention some interesting results either rediscovering an already known method or improving it. In this case genetic algorithm works with crossover probability $p_c = 70\%$, mutation probability $p_m = 2\%$ and population size $s_p = 60$. The developmental system performs 4 steps for the fitness calculation. Table 5 shows some of the most interesting constructors and properties of the sorting networks they produce.

The sorting networks produced by constructor $\mathcal{A}$ are the same as the sorting networks for the conventional straight insertion algorithm (see Figure 3a). Thus we have rediscovered the principle of this method. Constructor $\mathcal{B}$ creates sorting networks with even number of inputs, whose properties (the number of comparators and delay) were improved in comparison with the conventional approach (e.g. straight insertion algorithm). Therefore, the implementation cost and the time needed for the sorting are reduced. Figures 7 and 8 show the resulting sorting networks. The bounded elements represent the embryos. Thin vertical lines indicate the developmental steps. Both the sorting networks contain no redundant comparators [2].

5.2 Adders

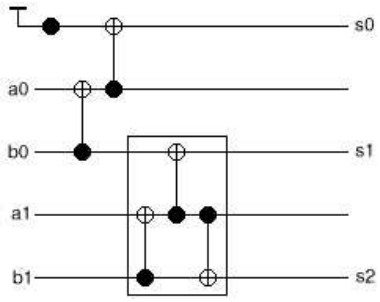

Figure 9: Adder constructed by means of the constructor [MODFS 2 0] [CPMIS 1 0] [CPMIN 2 1] [CPMIS 0 1]. The bounded part is repeated regularly during development.

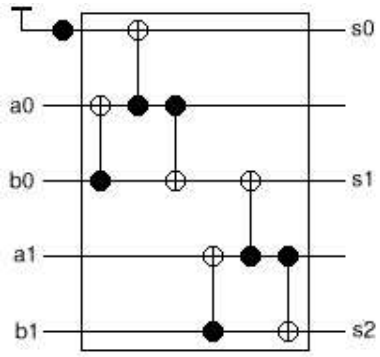

Figure 10: Adder constructed by means of the constructor [MODFS 2 0] [CPMIS 3 2] [CPMIS 4 3] [CPMIS 2 3] [CPMIS 1 0] [CPMIS 2 1] [CPMIS 0 1]. The bounded part is repeated regularly during development.

We were interested in the construction of arbitrarily large binary adders. The most crucial part of the process was to find a suitable representation for increasing the evolvability of the circuit. The following schema showed to be a good solution for the evolutionary design. The developmental step takes only even values (2 and 4 are reasonable). In each step the number of inputs increases of a given value. In this case genetic algorithm works with crossover probability $p_c = 2/1000$, mutation probability $p_m = 16/1000$ and population size $s_p = 400$. Such the low GA-parameters (p_c and p_m) may be curious in comparison with the other evolutionary design techniques utilizing genetic algorithms. We assume that the reason for these settings is a rugged fitness landscape of the problem. The developmental system performs 2 or 3 steps for the fitness calculation depending on the size of the developmental step.

Figure 9 shows the resulting adder constructed by means of the constructor involving the development with the step of size 2, i.e. adders produced in each developmental step are able to add up two 2–, 3–, 4–bit numbers, etc. An adder constructed by means of the development of step 4 is shown in Figure 10. As obvious, many parts of the presented circuit correspond to the structure of the carry-propagate adder, though it is not rediscovering of the principle. It is probably due to the fact that we are not interested in values

N	16	17	18	19	20	21	22	23	24	25	26
Constructor $\mathcal{A}$	(ModifyS 2 2) (ModifyS 1 1) (CopyN 3 2)										
Num. of comp.	120	136	153	171	190	210	231	253	276	300	325
Delay	29	31	33	35	37	39	41	43	45	47	49
Constructor $\mathcal{B}$	(0 2 2) (0 1 2) (0 0 1) (1 1 1) (3 1 2) (3 1 1)										
Num. of comp.	92	-	117	-	145	-	176	-	210	-	247
Delay	27	-	31	-	35	-	39	-	43	-	47

Table 5: The examples of constructors for the construction of sorting networks: Constructor $\mathcal{A}$ builds arbitrarily large sorting networks, constructor $\mathcal{B}$ generates the sorting networks with even number of inputs. The properties of some chosen sorting networks are given under each constructor.

of initial *carry-in* as shown in Figures 9 and 10.

5.3 Discusion

The result presented for the sorting networks of even size is efficient in view of both the number of comparators and delay in comparison with a conventional method (e.g. straight insertion sort). The larger sorting network, the larger variation of the number of comparisons in favour of the sorting network built by the evolved constructor. Even if the sorting networks presented are not competitive to the best known structures (see Table 1), nowadays it is extremely difficult to design very large and efficient sorting networks directly. Sorting networks can be utilized as a basis for calculating of medians, so these results may contribute to median circuits too.

In the case of binary adders, we have not been able to discover a novel principle yet. The main problem might be in relatively restricting representation, which on the other hand offers a good level of evolvability. However, it is very interesting to observe that genetic algorithm is able to find a program for the construction of arbitrarily large adders at the gate level completely from scratch, i.e. no particular information about the embryo is known a priori. Moreover, the instruction set (see Table 4) and the set of the logic gates (see Table 3) have shown to be redundant. The evolutionary process have selected only 3 instructions of 10 and 2 logic functions of 4.

Although all the produced results can construct very large circuits, which are fully functional, the question of usability in real-world applications remains unanswered. A number of developmental systems have been proposed to make the evolutionary design scalable but only a few of them have been applied to design objects more complex than we can do without development. Besides that, our approach offers a substantial improvement in the case of sorting networks.

6. CONCLUSIONS

A non-traditional method for automatic design of arbitrarily large combinational logic circuits was presented. We have focused on the sorting networks and adders, which have shown to be suitable for the developmental process. A simple instruction set was utilized to represent the rules of the development. The experimental results showed the ability of the genetic algorihm to (1) rediscover the principle of an already known method and (2) improve the construction algorithm, especially for the large sorting networks. Moreover, this is the first case when the development was applied for the design of fully functional arbitrarily large logic circuits at the gate level.

Still, the open questions are, for instance, whether it is possible to evolve general constructors for more efficient adders (e.g. rediscovering the principe of carry-look-ahead adder) or other arithmetic circuits. Is it necessary the constructed circuit to be fully functional during development? How does consideration of a suitable form of environment impact the evolution? All these questions form a direction for our future research. We do believe that application-specific evolutionary algorithms endowed with application-specific developmental system will allow designers to discover novel design principles for constructing other complex systems in near future.

7. ACKNOWLEDGMENTS

The research was performed with the support of the Grant Agency of the Czech Republic under No. 102/04/0737 *Modern Methods of Digital Systems Design*.

8. REFERENCES

[1] A. P. Shanthi et al. Development based evolution for scalable, fault tolerant digital circuits. In *Workshop on Soft Computing (WoSCo 03), International Conference on High Performance Computing (HiPC 03)*, pages 165–176, 2003.

[2] Anonymous. Technical report.

[3] P. J. Bentley. Fractal proteins. *Genetic Programming and Evolvable Machines*, 5(1):71–101, March 2004.

[4] E. J. W. Boers and H. Kuiper. Biological metaphors and the design of artificial neural networks, master thesis. Technical report, Departments of Computer Science and Experimental and Theoretical Psychology, Leiden University, 1992.

[5] 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.

[6] G. Tempesti et al. Ontogenetic development and fault tolerance in the poetic tissue. In *Proc. of the 5th Conf. on Evolvable Systems: From Biology to Hardware (ICES 2003), Lecture Notes in Computer Science, vol. 2606*, pages 141–152, Berlin, DE, 2003. Springer–Verlag.

[7] T. G. W. Gordon. Exploring models of development for evolutionary circuit design. In *2003 Congress on Evolutionary Computation*, pages 2050–2057. IEEE Press, 2003.

[8] 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 Computer Society Press.

[9] H. de Garis et al. Atr's artificial brain (cam-brain) project: A sample of what individual "codi-1 bit" model evolved neural net modules can do with digital and analog i/o. In *Proc. of the 1st NASA/DoD Workshop on Evolvable Hardware*, pages 102–110. IEEE Computer Society Press, 1999.

[10] P. Haddow and G. Tufte. Bridging the genotype–phenotype mapping for digital fpgas. In *Proc. of the 3rd NASA/DoD Workshop on Evolvable Hardware*, pages 109–115, Los Alamitos, CA, US, 2001. IEEE Computer Society.

[11] P. Haddow, G. Tufte, and P. van Remortel. Shrinking the genotype: L-systems for ehw? In *Proc. of the 4th International Conference on Evolvable Systems: From Biology to Hardware, Lecture Notes in Computer Science, vol. 2210*, pages 128–139. Springer–Verlag, 2001.

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

[13] 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 Computer Society Press, 2001.

[14] L. Huelsbergen. Finding general solutions to the parity problem by evolving machine-language representations. In *Proc. of the Genetic Programming 1998 Conference*, pages 158–166, San Francisco, CA, 1998. Morgan Kaufmann.

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

[16] 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.

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

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

[19] 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.

[20] M. Murakawa et al. Evolvable hardware at function level. In *Proc. of the Parallel Problem Solving from Nature IV, Lecture Notes in Computer Science, vol. 1141*, pages 62–71, Berlin Heidelberg New York, 1996. Springer.

[21] J. F. Miller and D. Job. Principles in the evolutionary design of digital circuits – part i. *Genetic Programming and Evolvable Machines*, 1(1):8–35, April 2000.

[22] J. F. Miller and D. Job. Principles in the evolutionary design of digital circuits – part i. *Genetic Programming and Evolvable Machines*, 3(2):259–288, July 2000.

[23] J. F. Miller and P. Thomson. Cartesian genetic programming. In *Proc. of the 3rd European Conference on Genetic Porgramming, Lecture Notes in Computer Science, vol 1802*, pages 121–132, Berlin Heidelberg New York, 2002. Springer.

[24] 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.

[25] P. J. Bentley (ed.). *Evolutionary Design by Computers*. Morgan Kaufmann, San Francisco, 1999.

[26] S. Kumar (ed.) and P. J. Bentley (ed.). *On Growth, Form and Computers*. Elsevier Academic Press, 2003.

[27] M. J. Streeter, M. A. Keane, and J. R. Koza. Routine duplication of post-2000 patented inventions by means of genetic programming. In *Proc. of the 5th European Conference on Genetic Programming, Lecture Notes in Computer Science, vol. 2278*, pages 26–36, Kinsale, Ireland, 2002. Springer-Verlag.

[28] J. Toressen. A scalable approach to evolvable hardware. *Genetic Programming and Evolvable Machines*, 3(3):259–282, September 2002.

[29] L. Wolpert. *The Principles of Development*. Oxford University Press, Oxford, UK, 1998.

[30] X. Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447, September 1999.