
Natural Computing **3:** 323–355, 2004.

Evolvable computing by means of evolvable components

LUKÁŠ SEKANINA
Faculty of Information Technology, Brno University of Technology, Božetěchova 2, 612 66 Brno, Czech Republic (E-mail: sekanina@fit.vutbr.cz)

Abstract. This paper deals with an emerging type of computing – evolvable computing. In evolvable computing solutions to problems dynamically evolve during system's lifespan either as programs for a universal computer or configurations for a physical reconfigurable device. In this paper the roots of evolvable computing are indicated, a method is presented for routine design of evolvable systems by means of evolvable components and some consequences for theoretical computer science are highlighted. In particular it is shown why evolvable computing cannot be simulated on a standard Turing machine. As examples, two evolvable components – for image pre-processing and for evolution of small pipelined combinational circuits – demonstrate implementations in an ordinary field programmable gate array.

Key words: evolvable component, evolvable hardware, genetic programme, reconfigurable device

1. Introduction

Theoretical computer science deals with abstract models of computing and abstract computational machines. On the other hand, in computer engineering, engineers try to build physical computational devices for problem solving, controlling, entertainment etc. In the recent years we could observe a slight change in how the real computers have become to work. A problem is whether contemporary computational systems fit into the scenario assumed by the classical concepts of theoretical computer science.

The existence of the so-called *universal Turing machine* is an important result of theoretical computer science (Gruska, 1997). The universal Turing machine can accomplish any *algorithm* that any computer, with any architecture, can accomplish.

The idea of universal computer has deeply been established in computer engineering. The computer-based problem solving has been transformed to programming; the required behavior is simply obtained by uploading a proper sequence of instructions into memory of a general-purpose (universal) computer and executing the program.

Recent advances in digital circuit technology have allowed designers to construct application-specific digital circuits directly and quickly in *reconfigurable devices* in the same way as programs are constructed. Because program and digital circuit represent two alternative approaches to the implementation of an algorithm, we can choose the most efficient (software and hardware) implementation for a given moment. In reconfigurable computing, digital circuits are dynamically created in reconfigurable devices in order to obtain higher performance in comparison to executing various subprograms in a general-purpose computer (Compton and Hauck, 2002). Novel hybrid high-performance architectures combine general-purpose processor(s) and reconfigurable device(s) on a single chip.

Furthermore, genetic programming and evolvable hardware enabled us to automatically discover novel computer programs and circuits by means of evolutionary algorithms for a variety of difficult engineering problems (Higuchi et al., 1993; Koza et al., 1999). A real challenge is to modify programs and circuits automatically during a run of computer in order to *adapt* the computer (its hardware and software) to a changing environment, i.e. to changing requirements on its behavior.

In this paper, in addition to *programming* of universal computers and *reconfigurable computing* carried out on hybrid platforms, we will deal with an emerging type of computing – *evolvable computing*. We will consider evolvable computing as an approach for designing adaptive and evolvable computational machines. In the scope of this paper, the evolvable computing is primarily carried out on contemporary electronic devices, i.e. it does not deal with the new approaches based on quantum computing, DNA computing or molecular electronics.

In evolvable computing, the solutions to problems evolve over time either as programs for a universal computer or configurations for a reconfigurable device. As far as physical electronic circuits are evolved, they can exploit various not purely digital properties of the reconfigurable platform and environment to solve the task. Moreover, the evolution is considered as endless process. That makes difficulties for theoretical analysis. Note that the use of hardware (i.e. physical implementation) is a crucial requirement in embodied intelligence, agents and cognition (Brooks, 1999; Wiedermann, 2004).

The objective of this paper is to survey the roots of evolvable computing, describe physical platforms that can be utilized for evolvable computing, introduce a method for routine design of digital evolvable systems and present some consequences to theoretical computer science. Our attention will be focused on the computational power of evolvable systems and on modeling of physical evolvable systems. Reusable *evolvable components* realized at the

level of IP cores will be introduced for routine implementation of evolvable computational systems.

The paper is organized as follows. Section 2 describes contemporary reconfigurable hardware. In section 3, evolutionary algorithms are considered as tools for automatic designing of programs and electronic circuits. The component approach to evolvable computing is surveyed in Section 4. Two examples of evolvable components utilizing FPGAs are presented in Section 5. Section 6 deals with consequences of evolvable computing for theoretical computer science. Finally, conclusions are given in Section 7.

2. From universal to reconfigurable computers

2.1. *General purpose computers*

Contemporary computers are usually constructed as universal computers consisting of a controller, an arithmetic logic unit (ALU), a memory and an input/output unit interconnected by a bus system. By changing the content of memory, we can change functionality of the computer. Physical circuits of the computer are not usually modified during computer lifetime. The behavior of computer is determined by instructions and data stored in the memory. When an instruction is executed, the controller activates various digital signals controlling the operations performed by ALU, memories, registers and the controller itself. As a result, data are modified in registers or memory. The main advantage of the approach is just the universality. All the complicated circuits (note that Pentium IV processor consists of 42.10^6 transistors) are controlled by a relatively tiny set of instructions. The required behavior is obtained by composing the right sequence of instructions for a given problem. Problem solving using computers has been transformed to programming. The main problem of the approach is performance, i.e. how much time a single instruction requires to be executed and how many instructions we need to execute in order to solve a given problem. A number of approaches have been discovered to improve performance, including caching, pipelining, parallel processing, vector processing, multithreading, specialized arithmetic, etc. (Hennessy and Patterson, 1996). The obtained performance is sufficient for some problems. On the other hand, many problems cannot effectively be solved on the universal computers.

2.2. *Dedicated computers and FPGAs*

Creating an application specific hardware, i.e. a dedicated computer, can sometimes solve the problem of performance. The dedicated computer

contains application-specific units and interconnecting systems that are not available in common processors. Considering the same microelectronic technology and operational frequency, the dedicated computer can be faster in several orders of magnitude than the universal computer for certain tasks (deHon, 1998). The dedicated computers are produced as application-specific integrated circuits (ASIC). The realization cost is the main disadvantage of ASICs. Millions pieces of these mostly "non-universal" chips have to be manufactured to make the production commercially attractive.

General-purpose *reconfigurable devices* like field programmable gate arrays (FPGAs) offer us other implementation options. The reconfigurable devices operate according to a configuration bit stream that is stored in the configuration memory. The cells of the configuration memory control a set of transistor switches. The switches define the way in which the programmable elements available on the device operate and the way in which are interconnected. By writing to this configuration memory, the user can physically create new (digital) electronic circuits. The advantage of the approach is that new hardware functionality is obtained through a simple reprogramming of the chip, similarly to reprogramming of the universal computer. The first FPGAs were commercially introduced in middle eighties by Xilinx, Inc. They were typically used to facilitate rapid prototyping of new electronic products.

Contemporary FPGAs contain thousands programmable elements. Furthermore, various additional circuits are integrated on the FPGAs, including RAMs, efficient multipliers and interfaces. For instance, the new Xilinx FPGA Virtex II Pro contains four PowerPC processors on the chip (Xilinx 2004). Figure 1 shows the classic architecture of the FPGA, which is a two-dimensional array of reconfigurable resources that include reconfigurable cells (e.g. 16-bit look-up tables), reconfigurable interconnection network and reconfigurable I/O blocks. The FPGA vendors provide collections of tools that are utilized for designing circuits for FPGAs. The designer describes the target-dedicated computer in a hardware description language (HDL), such as VHDL, Verilog, or HandelC. After simulations, a specialized program, a synthesizer, generates a netlist, i.e. an optimized gate-level implementation of the circuit. The next steps, again fully automatic, include mapping, placement, routing and creating the configuration bit stream for a given FPGA. The configuration bits stream is then uploaded into the configuration memory of the reconfigurable device. Although an FPGA-based solution usually requires 100 times more silicon space than an ASIC solution for the implementation of the same behavior, and furthermore, the FPGA-based solution is usually ten times slower than the ASIC, FPGA can offer 10–100 times higher performance than conventional universal processors in a number of problems.

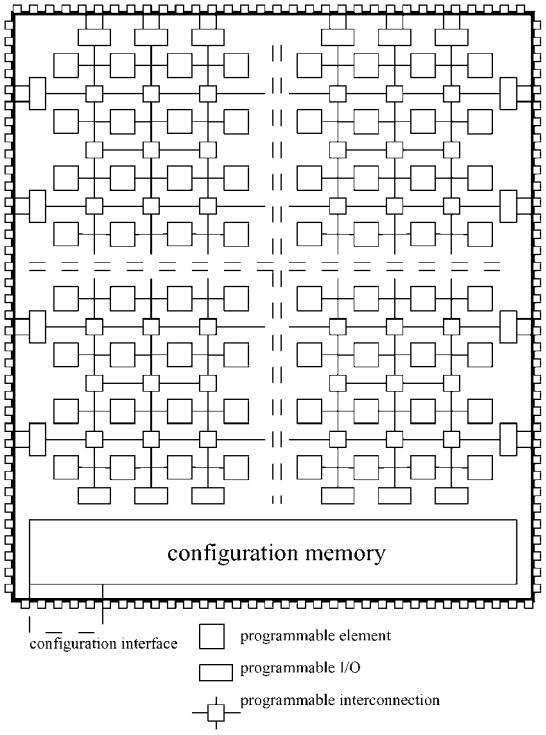

Figure 1. Two-dimensional FPGA composed of programmable units. Its function is defined by uploading configuration bits into the configuration memory.

2.3. *Reconfigurable computing*

Modern (hybrid) computational architectures consist of a universal processor and a reconfigurable device. The reconfigurable device is configured in case that some application requires a specific hardware. In case that the configuration of FPGA is modified dynamically, during the run, we speak about *reconfigurable computing* (Bondalapati and Prasanna, 2002; Compton and Hauck, 2002). Higher performance is then achieved by (dynamically) building custom computational operators, pathways, and pipelines suited to specific properties of the task at hand. Designer has to divide the application to modules and to schedule the optimal sequence of configurations if timing is known beforehand. On the other hand when a request for a given type of operation (i.e. for a module which provides the operation) emerges during execution (the flow of program/reconfiguration is not known in advance), online reconfiguration is done autonomously. Reconfigurable computing in fact provides a sort of *adaptability* at the level of hardware – it is possible

to change physical circuits by means of reconfiguration if a requirement emerges.

Designers have to solve new problems in comparison to classical programmers. They have to design software as well as hardware for a given application. They have to deal with the problem of hardware-software co-design, i.e. the problem of optimal partitioning of the task to software and hardware implementations, and optimal utilization of hardware resources at any moment during a run of the application. Therefore, we can construct the hardware, which is dynamically modified in order to process the input data stream efficiently, and the modification depends on the data processed currently.

The design process of such the application is based on programming; the designer is not practically in a contact with physical electronic circuits. Hardware looks like programs; it is a configuration. Hence programs, hardware (configurations) and data have the same nature – they are sequences of "zeroes" and "ones". In fact the current market offers hardware at the level of configuration bits for a given FPGA or as a HDL source code. These components (i.e. Intellectual Property – IP cores) make hardware design easier. Similarly to the software design process, the hardware design process is based on reusable components.

2.4. *Reconfigurable devices*

The idea of reconfigurable hardware, initially introduced in programmable digital devices, has attracted attention in various other domains in the recent years. Field programmable analog arrays (FPAA) (Flockton and Sheehan, 1998) and field programmable transistor arrays (Stoica et al., 2000) have been developed in the analog circuit domain. In massively parallel reconfigurable processor arrays (such as picoChip PC102 (PicoChip, 2003)), the interconnection network can easily be reconfigured. Linden has invented reconfigurable antennas (Linden 2002). Various reconfigurable implementations of neural networks are popular in the soft computing domain (see a survey in Zhu and Sutton (2003)). Reconfigurable nanodevices (e.g. NanoCell (Tour, 2003)), reconfigurable microelectromechanical systems (MEMS) and reconfigurable optics represent other types of reconfigurable devices. We can see that granularity of reconfigurable elements ranges from molecules to microprocessors.

The concept of the reconfigurable device is probably the most developed in the Cell Matrix architecture. The Cell Matrix (Macias, 1999) is a new type of reconfigurable device that is similar to an FPGA because it consists of (1) a uniform grid of simple cells, (2) cells that are connected in a fixed, nearest-neighbor fashion, and (3) cells that can be configured to perform any desired mapping from inputs to outputs according to a per-cell configuration

memory. The main difference is that the Cell Matrix is *internally* configured, while FPGA is configured *externally*. Thus a cell, in addition to exchanging data with its neighbors, can also exchange the configuration bitstreams with its neighbors. The sequences of zeroes and ones are sometimes considered as data and sometimes as configurations.

2.5. *Reaching the limits of conventional engineering*

Having software as well as hardware under the control of an application designer (a programmer), it could seem that complex problems could be solved routinely. However, the major problem is how to utilize all the resources effectively, i.e. how to design efficient programs, configurations (hardware) and their interactions. Currently, the top down approach to the design allows us to operate at 10 GHz, with 10^8 transistors on a few squared centimeters, with wafers of 300 mm, 90-nanometer features and the delay of a pure wire roughly about 1 ns per 15 cm (Bourianoff, 2003; Tour, 2003). Tour claims that we need 15–20 k electrons to store one bit of information. It is realistic to expect that the bottom-up design in molecular electronics will allow us to utilize 10^{14} computing elements per a squared centimeter. Potentially, 1–100 electrons will be needed to store one bit of information (Tour, 2003). The extremely high number of computing elements, physical laws and other laws known in computer science (such as Amdahl's and Moore's laws (Hennessy and Patterson, 1996)) will probably not enable the use of conventional programming schemes in order to achieve similar utilization of computational resources as it is possible with a relatively small amount of elements nowadays. Simply because the approach is not scalable. It seems that we will need more inspiration from other areas, especially from biology that deals with complex, efficient and intelligent systems.

3. Bio-inspired designs

Living creatures exhibit a number of features that we would like to provide in the engineering products, namely: robust and energy-efficient construction, adaptation, self-healing, self-replication, and many others. In contrary to engineering methods and products, there is no difference between the constructor and the constructed object.

Conventional engineering approaches are inefficient to ensure these features nowadays. All the living creatures are created using the same principles – interpretation of the genetic information inherited form the ancestors (Dawkins, 1991). Multicellular organisms are formed in the process of development, which is based on cellular division and differentiation. Reaching

a certain level of complexity, neural, immune and other systems emerge in the organism. The organisms are able to learn, communicate and cooperate. Any organism is determined by its genetic information and the environment (Wolpert, 2000).

Novel bio-inspired engineering approaches are inspired by the processes which we can observe at the levels of phylogeny, ontogeny, epigenesis and also in social systems. The approaches are applied not only in software but also in hardware domain (Bentley, 2001; Flake, 1998; Sipper, 2002). They can be utilized in the design phase or, which is much more challenging, during the operational time of a computer-based system.

In software domain, the approaches have a longer tradition. As examples, we can mention artificial neural networks, evolutionary algorithms, fuzzy systems, cellular automata, artificial immune systems and many others. Here, we should emphasize *genetic programming*, which nowadays represents an alternative to conventional creative "human" programming in a number of areas of problem solving. For instance, Koza (2003) provides the list of patented inventions duplicated by means of genetic programming. Genetic programming is especially successful in cases in which a programmer does not have an idea how to solve a particular system.

Hardware has been utilized to speed up the bio-inspired techniques. However, with the development of reconfigurable devices, the hardware is used in a quite unique way. Nowadays, bio-inspired engineers might consider the silicon-based electronic circuits in computers similarly to carbon-based molecules in living creatures. Sipper at all have introduced the POE (phylogenetic, ontogenetic, and epigenetic) model for classification of bio-inspired approaches in hardware domain (Sipper et al., 1997). New paradigms have been identified within this scope, for instance, *embryonic electronics* inspired by the process of development (Mange et al., 2000) or *immunological electronics* inspired by immune systems (Bradley et al., 2000).

Combining evolutionary algorithms with reconfigurable circuits in the area of *evolvable hardware* has probably attracted the greatest attention of designers in recent years. The benefits of evolvable hardware are particularly suited to a number of applications, including the design of low-cost hardware, creation of adaptive systems, fault tolerant systems and innovations. Starting in 1992 (Higuchi et al., 1993), evolvable hardware is a quite young research field that offers us promising technology for the future, especially in connection with smart materials, evolvability, adaptation, survivability, nanotechnology and evolvable computing. Since this paper primarily deals with evolutionary techniques used to create programs for universal computers and configurations for reconfigurable devices, we will provide more details about that.

3.1. *Evolutionary algorithms*

Evolutionary algorithms (EAs) are stochastic search algorithms inspired by Darwin's theory of evolution (Bäck, 1996; Holland, 1975). Instead of working with one solution at a time, these algorithms operate with the *population* of candidate solutions (individuals). Every new population is formed using genetically inspired operators (like crossover and mutation) and through a selection pressure, which guides the evolution towards better areas of the search space. The evolutionary algorithms receive this guidance by evaluating every candidate solution to define its *fitness*. The fitness, calculated by the *fitness function*, indicates how well the solution fulfills the problem objective (specification).

The majority of the research in the field of evolutionary algorithms is devoted to problems with a single (static) fitness function. However, real-world applications (like robotics, evolvable hardware, scheduling, etc.) operate in a *dynamic* environment, i.e. with a time-varying specification of the fitness function.

Rather than evolutionary optimization, we are interested in *creative evolutionary design* in this paper (Bentley and Corne, 2001). We use evolutionary techniques because they contain an element of randomness. The randomness leads to innovation – the emergence of unexpected ingenious solution to a complex problem.

3.2. *Evolvable hardware*

Nowadays, it is not difficult to modify electronic circuits, even of a running computer, since these circuits are often implemented using reconfigurable devices. In case of evolvable hardware, configurations are designed automatically by an evolutionary algorithm (Gordon and Bentley, 2001; Higuchi et al., 1993).

Figure 2 shows the difference between genetic programming and evolvable hardware. Genetic programming is usually performed on a single computer or a cluster of workstations in case of parallel implementations (Koza et al., 1999). There are several implementation options in case of evolvable hardware: (1) the evolutionary algorithm is executed in a common computer and the produced configurations are sent to a reconfigurable device available at a connected board or card (e.g. Thompson et al. (1999)), (2) evolutionary algorithm is implemented as ASIC on the same chip as evolving circuits (e.g. Higuchi et al. (1999)), or (3) evolutionary algorithm is implemented in FPGA on the same chip as evolving circuits (e.g. Sekanina and Friedl (2004)). If the chromosome exactly corresponds to the structure of

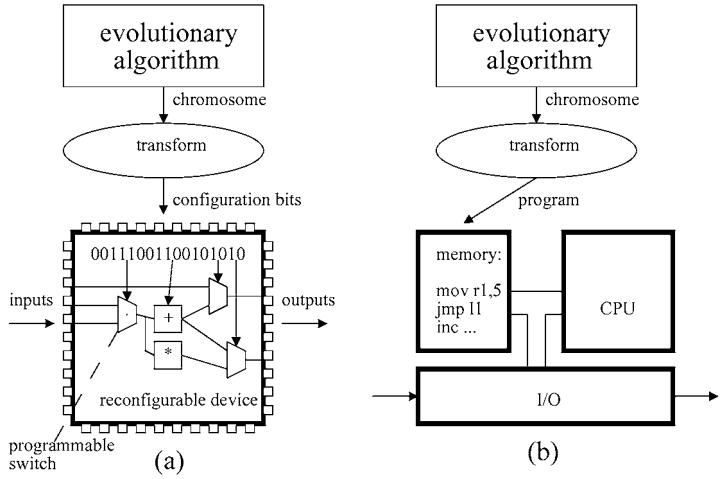

Figure 2. Comparison of (a) evolvable hardware and (b) genetic programming.

configuration data of the reconfigurable device then "transform" process can be omitted in Figure 2.

In is important to distinguish between two approaches: evolutionary circuit design and evolvable hardware. In case of the evolutionary circuit design, the objective is to evolve (i.e. to design) a single circuit. The circuit is usually designed in a design lab. The aim is typically to find an innovative implementation of a required behavior, for instance, to reduce the number of utilized gates or to ensure testability of a circuit. Up to now, the technique was successfully applied to design a number of unique digital as well as analog circuits. The evolved circuits exhibit properties that we have never reached by means of traditional engineering methods. It was demonstrated on small-scale-circuits that creative designers can effectively be replaced by machines (Koza et al., 2003; Miller et al., 2000; Sekanina, 2003a; Thompson et al., 1999).

Evolvable hardware deals with continuous designing of electronic circuits. Evolutionary algorithm is responsible for adaptation. Online evolution has enabled the origin of high-performance and adaptive systems for the applications in which the problem specification is unknown beforehand and can vary in time (Higuchi et al., 1999; Tan et al., 2004). Its main objective is the development of a new generation of hardware, self-configurable and evolvable, environment-aware, which can adaptively reconfigure to achieve optimal signal processing, survive and recover from faults and degradation, and improve its performance over lifetime of operation. Evolutionary algorithm is an integral part of the target system.

Due to the existence of redundancy in reconfigurable devices, the evolvable hardware is inherently fault tolerant. It means that in case of a failure of a circuit element, the evolutionary algorithm is usually able to recover the functionality using the remaining elements. If a critical number of elements are damaged, the functionality cannot be recovered and the chip "dies". Hence evolvable hardware is a method for automatic designing of adaptive as well as fault-tolerant (e.g. self-repairing) systems (Thompson et al., 1999).

The circuit evolution can be carried out using software simulators (then we speak about *extrinsic* approach) or directly in a reconfigurable circuit (so-called *intrinsic* evolution). Adrian Thomson has discovered that even if only digital behavior is required, the evolution (conducted directly in hardware) can in some cases exploit physical properties of a given chip in order to satisfy the requirements defined by fitness function. For instance, the circuit behavior might depend on temperature. Hence analog nature of the circuits must be taken into account (Thompson, 1998).

Recently, Miller has used the more general term *evolvable matter* to address the usage of evolutionary algorithms for the design on any physical reconfigurable platform (e.g. chemical). The idea behind the concept is that applied voltages may induce physical changes that interact in unexpected ways with other distant voltage-inducted configurations in a rich physical substrate. In other words, it should theoretically be possible to perform the evolution directly *in materio* if the platform is configurable in some way. Miller and Downing reviewed this promising technology in (Miller and Downing, 2002). Tour's group has presented a concrete working example – the Nanocell (Tour, 2003).

Evolvable hardware does not deal with digital hardware only; a lot of research was done in evolution of sensors, configurations for reconfigurable analog arrays, reconfigurable antenna, reconfigurable optics and evolutionary design of various objects.

3.3. *Scalability of evolvable hardware*

Figure 3 illustrates the basic idea of the creative evolutionary design (e.g. of a computational system). All the genetic operations are carried out over genotypes. Phenotypes are created using a genotype-phenotype mapping. It is important to understand that phenotypes (not genotypes) are evaluated using the fitness function. In case of evolvable hardware, the fitness function can be changed dynamically. In some cases the genotype-phenotype mapping is also influenced by environment, which is typical for the development in biological systems (Wagner, 1996; Wolpert, 2000).

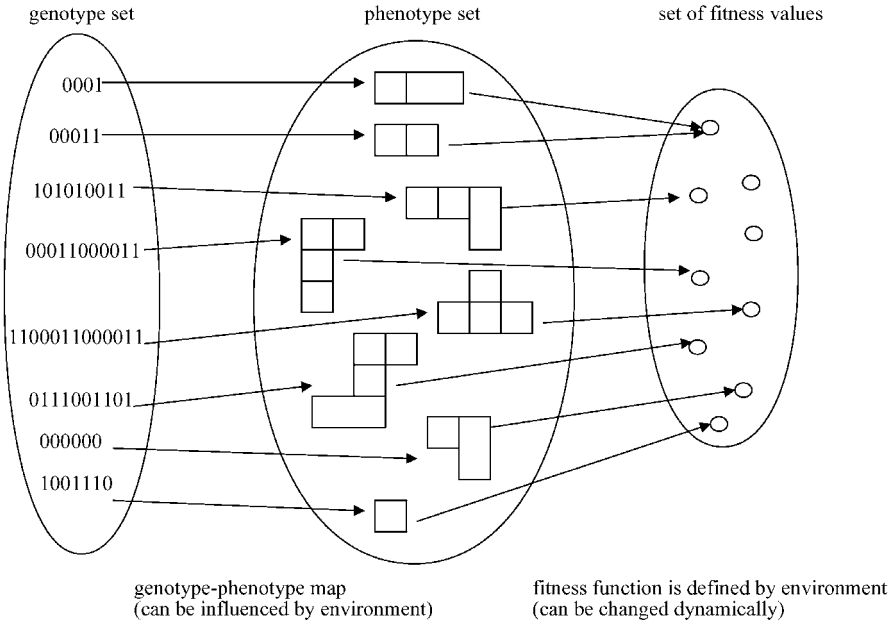

Figure 3. Genotype-phenotype mapping and fitness function in evolutionary design.

Promises and challenges of evolvable hardware have been summarized in Gordon and Bentley (2001), Sekanina (2003a) and Yao and Higuchi (1999). Scalability of the evolutionary approach is usually considered as the main problem of the evolutionary design. As Torresen commented on classical paradigm of evolutionary circuit design, large objects require longer chromosomes, i.e. the search space is also larger and so difficult to be effectively explored by evolutionary algorithm (Torresen, 2002). A great effort has been invested to allow the evolutionary design to produces larger objects. Let us note that it is not a problem to evolve a large object at all. The problem is that the evolved large object is not usually interesting (innovative, etc.) because a lot of domain knowledge has been included into the design method. Hence we are looking for a balance between the inserted domain knowledge and the size and level of innovation of the resulting object. Three basic methods have been introduced to overcome the problem of scale, namely functional level evolution (Murakawa et al., 1996), incremental evolution (Torresen, 2002) and developmental approaches (Haddow and Tufte, 2001).

Wagner and Altenberg noted that in evolutionary algorithms it was found that the Darwinian process of mutation, recombination and selection is not universally effective in improving complex systems like computer programs or circuits (Wagner and Altenberg, 1996). For adaptation to occur, these systems must possess *evolvability*, i.e. the ability of random variations to

sometimes produce improvement. It was found that evolvability critically depends on the way how genetic variation maps onto phenotypic variation. Hence various approaches to implementing the genotype-phenotype map have been introduced in the recent years, especially in the field of evolvable hardware.

4. Evolvable components

Routine combining of reconfigurable devices (e.g. as co-processors) with common processors in order to serve as a general-purpose hybrid computational platform has been investigated only for a relatively short time in comparison to the use of universal computers (Hartenstein, 2002). Software design tools are under development for designing on such the platforms. The tools contain various methods of soft computing utilized for various tasks, for instance, in optimization of scheduling, placement and routing (Bondalapati and Prasanna, 2002; Compton and Hauck, 2002).

A much more complicated issue is how to integrate evolvable hardware (i.e. continually evolving circuits) to the applications of reconfigurable computing. One of possibilities, elaborated in detail in next sections, is to develop evolvable IP cores, i.e. evolvable components (modules) for reconfigurable devices (Sekanina, 2003a; Sekanina, 2003b). The basic idea is that the evolvable component can be uploaded into the reconfigurable device; when executed, evolution of a physical circuit is started. The evolution is controlled by other components placed on the same reconfigurable device – these components represent the environment for the evolvable component. When adaptive behavior is not required, the evolvable component is removed from the reconfigurable device. The evolvable components have been developed to introduce reusability to evolvable hardware.

4.1. *Reusability of evolvable hardware*

It is practically impossible to routinely reuse evolvable hardware in its traditional form. If reusability should be achieved, it is necessary to identify *what* has to be reused.

In classical system theory we define a *system* as a collection of components and a *relation* that binds the components together. The system operates within a *context* (environment) which produces stimuli for the system and which receives the reactions of the system to those stimuli. By a *subsystem* we mean a single component or a subset of coupled components of a system. Contemporary computational systems benefit from *component* technology because they are composed of software and/or hardware components that

provide required behaviors via their interfaces. Designers have the access to various software and hardware components available in the market.

We can identify several reasons for the usage of the component approach. Primarily it is reduction of *time to market* (because of reusability) and increase in *reliability* (because target systems are built from verified and tested elements). Generally, the component approach leads to effective problem solving, especially in the case of complex systems. The component approach in fact determines the *business model* for producers.

Stoica et al. mentioned four years ago that "the path leads to the IP (Intellectual Property) level and evolvable hardware solutions will become an integrated component in a variety of systems that will thus have an evolvable feature" (Stoica et al., 2000). Subsequently, evolvable components were introduced as the components that can autonomously change their functionality according to requirements of an environment (Sekanina, 2003a).

It was recognized that reusability plays two different roles in evolvable hardware (Sekanina, 2003a): (1) Some application parts can be reused in all future designs independently of the application. We should identify these parts, prepare them beforehand and reuse them in the future. As typical example of such the part, we can mention the controller, which controls the evolvable component and provides the interface to the environment. (2) As soon as an evolvable hardware-based application operates in a dynamic environment, the same reconfigurable circuit and genetic operators are reused for different fitness functions in the system. Hence the reconfigurable circuit and genetic operators have to be designed in such a way that they are able to produce efficient solutions for a wide range of different problem specifications (fitness functions).

4.2. *Structure of evolvable components*

Any evolvable component consists of a reconfigurable device and a genetic unit. The genetic unit implements evolutionary algorithm; however, fitness calculation is performed outside the component, by environment, because only the environment "knows" the specification and thus only the environment can assign the fitness value to any circuit/program realized in the programmable device. Note that we can easily define evolvable system as a system that contains at least one evolvable component.

Figure 4 shows a general architecture of an evolvable system. Communication between the environment (that is represented by components 1–5 in our example) and the evolvable component is as follows: First the evolvable component is initialized (initial population is generated). Then the following sequence of operations is repeated endlessly. The environment requires a new circuit to be uploaded into the reconfigurable circuit. The evolvable

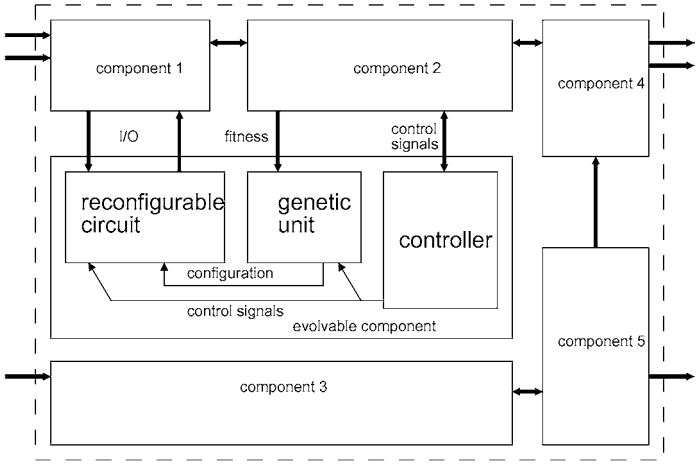

Figure 4. Evolvable computing using an evolvable component. Components 1–5 represent an environment for the evolvable component in our example.

component is to generate a new configuration and to configure the reconfigurable circuit. If there is no configuration (chromosome) in the chromosome memory available then a new population has to be generated autonomously. When the evaluation of the circuit behavior is finished the fitness value is sent to the evolvable component. The component is to store the fitness value and to wait for another request from the environment. The environment can also require uploading of the best circuit that has been evolved so far. Hence the component has to continually store the best configuration that has been evolved so far and to provide it when requested. In all cases the evolvable component encapsulates the reconfiguration process that is *invisible* from the external environment. We see that the evolvable component is controlled from the environment. As an example of evolvable system, we can mention an adaptive filter (realized as evolvable component) which dynamically modifies its circuits according to changing requirements.

The idea of the evolvable component was introduced for the digital electronic domain. The same approach is easily applicable in the software domain: genetic programming will produce programs for a universal computer.

4.3. *A design approach using evolvable components*

The proposed component approach also provides a certain kind of methodology for the design of evolvable hardware-based systems, i.e. for evolvable computing. Evolvable hardware has traditionally been understood as a combination of reconfigurable circuits and evolutionary algorithms. We argue that

the separation of the fitness function (which is then considered as an environment for the evolvable component) from the evolutionary algorithm is a crucial idea for evolvable computing and the efficient design of evolvable systems. We prefer to understand the evolvable computing as a combination of an evolvable component and an environment (possibly created of other evolvable components). Therefore the designer can utilize a suitable evolvable component as a design pattern and specify only the fitness calculation according to a given problem domain.

5. Examples of evolvable components in FPGAs

This section gives two examples of evolvable components that are completely implemented as digital circuits in FPGAs. The components can dynamically be uploaded and removed to/from the FPGA making the computational system not only reconfigurable, but also evolvable. Since both components utilize the so-called *virtual reconfigurable circuit*, we have to introduce the idea firstly.

5.1. *Virtual reconfigurable circuits*

Virtual reconfigurable circuits (VRCs) were introduced for digital evolvable hardware as a new kind of reconfigurable platform utilizing conventional FPGAs (Sekanina and Ruzicka, 2000, 2003a). Similarly to Cell Matrix, they exhibit the so-called internal reconfiguration mode that is crucial for evolvable systems completely implemented on a single FPGA. As Figure 5 shows, VRC is a new reconfigurable device realized on top of an ordinary FPGA, consisting of sixteen programmable elements (PE), sixteen inputs and eight outputs in our example. When the VRC is uploaded into the FPGA then its configuration bit stream has to cause that the following units will be created at specified positions: array of programmable elements, programmable interconnection network, configuration memory and configuration port. All these units are created from the resources available in the FPGA.

For example, "virtual" PE2 depicted in Figure 5 is controlled using 6 bits determining selection of its operands (2 + 2 bits) and its internal function Fn1–Fn2 (2 bits). This architecture is very similar to the representation employed in Cartesian Genetic Programming (CGP) that has been developed for circuit evolution (Miller et al., 2000). The routing circuits are implemented using multiplexers. The (virtual) configuration memory is realized as a register array. All bits of the configuration memory are connected to multiplexers that control routing and selection of functions in PEs.

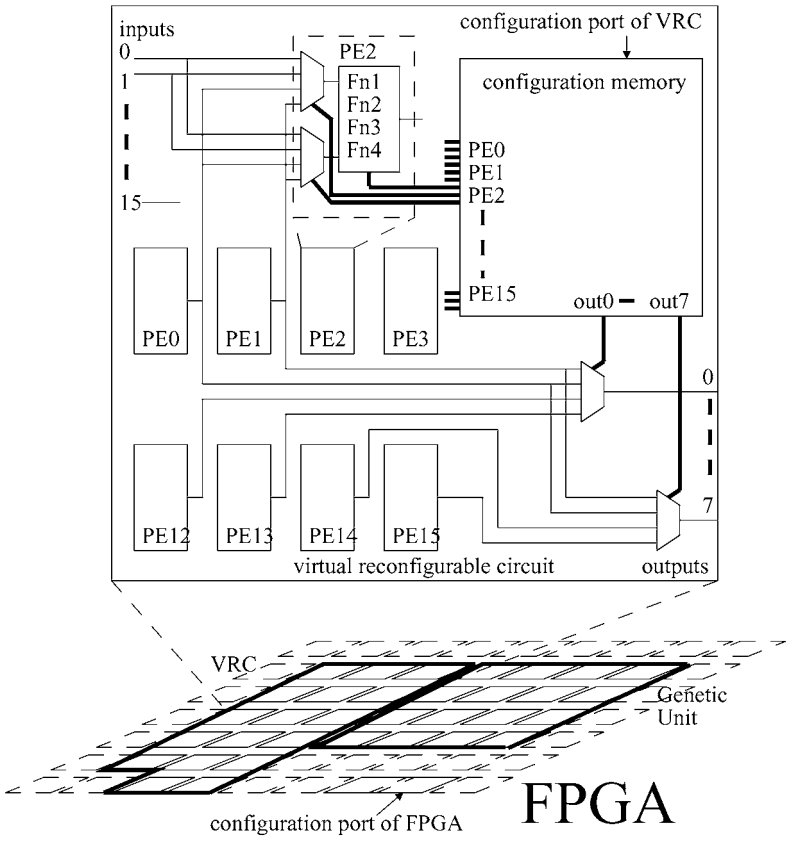

Figure 5. An example of virtual reconfigurable circuit and its structure.

The main advantage of the proposed method is that the array of PEs, the routing circuits and the configuration memory can be designed exactly according to the requirements of a given application. Furthermore, the style of reconfiguration and granularity of the new VRC can exactly fit the needs of a given application. Because VRC can be described in a HDL, it can be synthesized with various constraints and for various target platforms.

The VRC can directly be connected to hardware implementation of the evolutionary algorithm placed on the same FPGA. If the structure of chromosome corresponds to the configuration interface of VRC, then a very fast reconfiguration (e.g. consuming one clock only) can be achieved – which is impossible to reach by direct using the configuration subsystem of an ordinary FPGA.

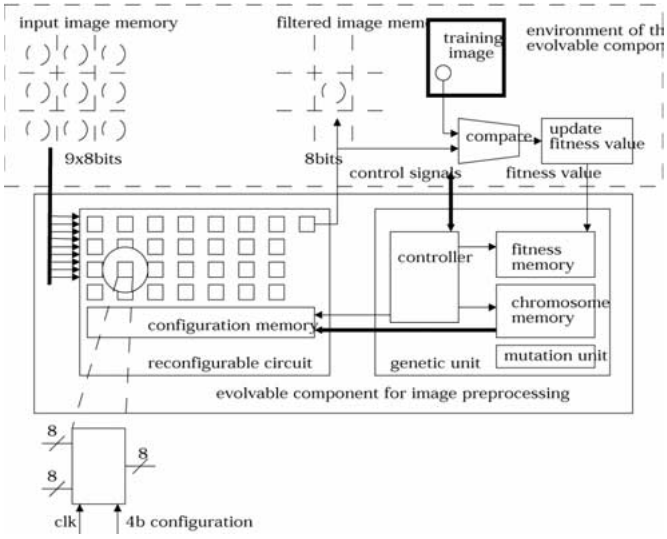

Figure 6. Evolvable component for image pre-processing and the environment calculating fitness values.

5.2. *Evolvable component for image pre-processing*

This evolvable component consists of evolutionary algorithm and reconfigurable device serving as an adaptive image filter. Figure 6 shows that all inputs as well as outputs of programmable elements of the reconfigurable device utilize exactly 8 bits and perform simple functions, for example: binary and, or, xor, addition, addition with saturation, average, minimum and maximum. Every image operator is considered as a digital circuit of nine 8bit inputs and a single 8bit output (the so-called 3×3 filter) which processes gray-scaled (8bits/pixel) images. Every pixel value of the filtered image is calculated using a corresponding pixel and its eight neighbors in the processed image.

The design objective is to minimize the mean difference between the filtered image and the original (training) image. We chose to measure the *mean difference per pixel* (*mdpp*), since it is easy for hardware implementation. The 256×256 pixel Lena image was utilized in the fitness function; the evolved filters were tested using other images. We have tried to evolve innovative 3×3 image filters which can compete in terms of implementation costs and quality (*mdpp*) with conventional filters such as median filter, averaging using various coefficients and Sobel edge detector. First, we described these conventional filters in VHDL and synthesized them into Xilinx FPGA XC4028XLA. For instance, the median filter costs 4740 gates, averaging filters cost about 1400 gates and the standard Sobel detector requires 1988 gates.

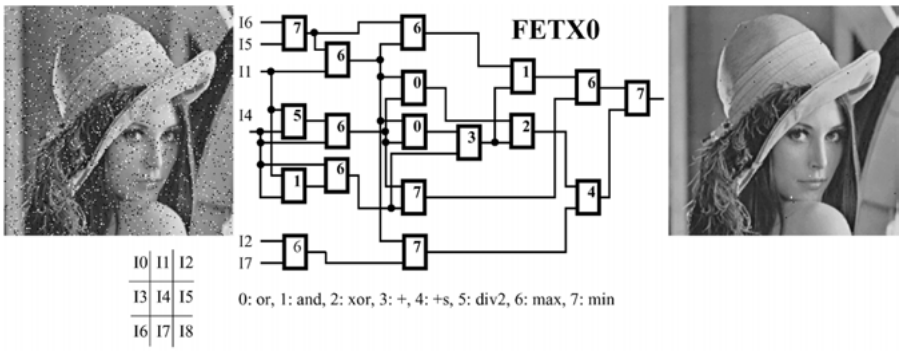

Figure 7. FETX0 evolved to supress 5% "salt and pepper" noise.

In particular we evolved "salt and pepper" noise filters, random shot noise filters, Gaussian noise filters, uniform random noise filters, edge detectors and various application-specific image filters. The evolved circuits exhibit better quality than conventional circuits if *mdpp* is measured. In some cases the evolved circuits require less of equivalent gates than the conventional circuits. It is mainly evident if an evolved circuit is compared with the median filter. For instance, the "salt and pepper" noise is traditionally suppressed by means of the median filter. However, we evolved RA3P5 and FETX0 filters that cost only 1702 and 2075 gates respectively. The FS3 and FS7 are edge detectors evolved (which cost 1350 and 2079 gates respectively) but they provide the similar quality to the Sobel operator. More than 500 image filters were evolved; the most interesting of them are presented and analyzed in detail in (Sekanina, 2003a). As an example, FETX0 is depicted in Figure 7.

Complete hardware implementation of the evolvable filter has been proposed in (Sekanina, 2003b). In order to find out how many Virtex slices are needed for the implementation of the virtual reconfigurable circuit we modeled the circuit by means of VHDL and synthesized the circuit into FPGA XC2V1000. This design costs 4879 slices (74357 equivalent gates) and it can operate at 134.8 MHz. Therefore, a single filter can be evaluated in 0.48 ms and full reconfiguration takes 7.4 ns during the evolutionary design. We obtained a speeding up more than 70 in comparison with the software approach. Analysis performed in (Sekanina, 2003a) have shown that the adaptation time could be a few seconds for some applications.

5.3. *Evolvable component for high-performance evolution of small pipelined combinational circuits*

This evolvable component is considered as a high-performance system for evolving small pipelined combinational circuits, such as 3 × 3-bit

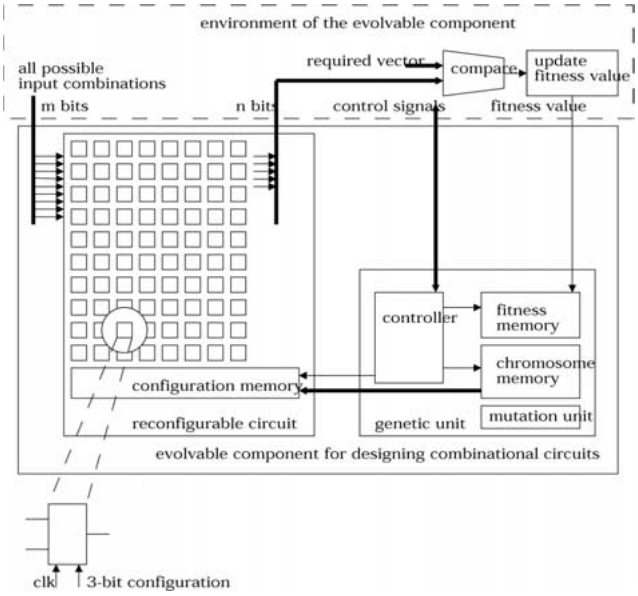

Figure 8. Evolvable component for the evolutionary design of small combinational circuits and the environment calculating fitness values.

multipliers, in a few seconds (Sekanina and Friedl, 2004). As the target device the COMBO6 card developed in the Liberouter project is employed (www.liberouter.org).

Figure 8 shows the evolvable component and its environment. The VRC consists of 80 PEs (10 rows $\times$ 8 columns) equipped with flip-flops allowing pipelined processing. Each of them can be programmed to perform one of eight functions that are evident from Figure 9. In order to define behavior of the VRC, 880 configuration bits have to be uploaded from the chromosome memory. Similarly to the previous example, the genetic unit (placed on the same FPGA) generates the configuration bits. All possible input combinations are applied at the primary inputs and the outputs are compared with the required values in the process of fitness calculation performed by the environment.

The entire evolvable system was described in VHDL. After simulations, the design was synthesized using Leonardo Spectrum to Virtex FPGA XC2V3000bf957, which is available at COMBO6 card. The complete design requires 403,372 equivalent gates, including the implementation of fitness calculation and PCI interface.

Figure 9 shows an example of the evolved 3 $\times$ 3-bit multiplier. The circuit utilizes 62 PEs. Twelve of them are pure connection wires (functions $c = a$ and $c = b$ in Figure 9). Thus the multiplier consists of 50 active

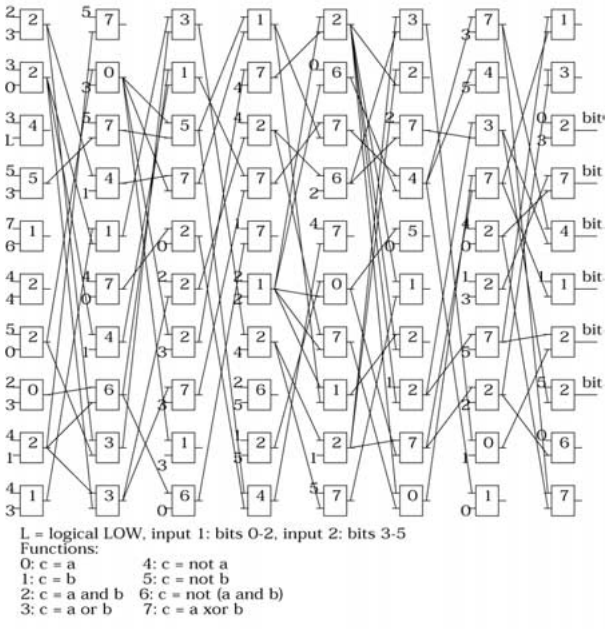

Figure 9. Pipelined 3 × 3-bit multiplier evolved using the evolvable component.

elements. We performed 1600 runs and obtained the fully correct solutions in all cases and in generation 5,377,900 on average. If operational frequence $f_m = 100$ ~MHz is considered then we will obtain the resulting circuit after 57.5 sec. on average. The approach was also utilized to evolve other small combinational circuits, e.g. 4 × 3-bit multiplier (Sekanina and Friedl, 2004).

The presented evolvable components are available as configuration bit streams or at the level of VHDL code. It means that they can be utilized for any reconfigurable digital device of sufficient capacity. Therefore evolvable computing performed in a physical digital circuit has become independent of a concrete reconfigurable device.

6. Consequences for theoretical computer science

We introduced evolvable computational systems that can completely be realized in the reconfigurable devices (as evolvable IP cores) or by means of combining a reconfigurable device and a universal computer (which is used to execute the evolutionary algorithm). In some applications, the evolution can be considered as an *endless* process in order to achieve continuous (real-time) adaptation to a changing environment. Adaptive filtering in a real-world industrial application is a typical case. The second feature of these systems is

that the algorithm (realized as a program or as a digital circuit) is dynamically changed in time. The change is in fact controlled by data processed by the evolvable system. The time of the change of the algorithm is *unpredictable*, because it can depend on various factors, including the weather, human interactions, etc. Moreover, these systems are interactive, as evolutionary robotics illustrates (Nolfi and Floreano, 2000). As far as evolvable systems are physical objects and sometimes with the behavior more analog than digital, we can ask whether these features have some interesting consequences for theoretical computer science that is traditionally approached in a purely mathematical way. Let us start with a brief look at theory of evolutionary computation.

6.1. *Lessons from theory of evolutionary computation*

Various theoretical approaches have been formulated to evolutionary computation in the recent years, including: the schema theorem, the Vose model, fitness landscape analysis, the coarse graining theory, etc. (Stephens and Zamora, 2003). The most interesting question that arises from the applications of an evolutionary algorithm is if it will succeed in finding the global optimum. Second, we are interested in the convergence velocity.

Although the contemporary theory of evolutionary computation does not provide too many practical results for application designers, there are some important theoretical achievements. The two of them are relevant for this paper: computational power of evolvable systems and the so-called No Free Lunch theorem.

(1) E. Eberbach has performed theoretical analysis of evolutionary computation from the point of view of theoretical computer science (Eberbach, 2002). He recognized that under some conditions evolutionary algorithms are guaranteed (in infinity) to find an optimal solution. He defined the Evolutionary Turing Machine and showed that it is more powerful than a conventional Turing machine. The higher expressiveness is obtained either evolving infinite populations in infinite number of generations or using nonrecursive variants of genetic operators.

(2) The No Free Lunch (NFL) theorem states: If some algorithm a_1's performance is superior to that of another algorithm a_2 over some set of optimization problems, then the reverse must be true over the set of all other optimization problems. It means that a random search is as good as any other method (like evolutionary computation) in average over all optimization problems (Wolpert and Macready, 1997). The result sounds much more pessimistically if we consider that evolvable systems deal with dynamically changing optimization problems.

6.2. *Evolvable computational machines*

An abstract model of an *evolvable computational machine* has been formulated in (Sekanina, 2003a). The aim was to investigate the computational power of evolvable computational machines in real-world applications. The definition comes out from Figure 3 and reflects the idea of evolvable component.

The definition is constructed as follows. Recall that classical evolutionary algorithms utilize fitness function of the form $\Phi : C \to R$ where C denotes a set of chromosomes (representation space) and R is the set of real numbers (Bäck, 1996). As we said, chromosomes are not evaluated. Only machines (more precisely, behaviors of these machines) are and can be evaluated. Hence it is reasonable to define a set of machines M which can be constructed from chromosomes for a given problem domain using genotype-phenotype mapping $g : C \to M$. It is supposed that g is surjective. The machines are then evaluated using "machine" fitness function $f : M \to R$. Finally, Φ is expressed as composition $\Phi = f \circ g$.

Therefore, we can summarize that any evolvable computational machine is fully defined in terms:

E – evolutionary algorithm employed (with fitness function $\Phi : C \to R$);
M – a set of possible machines which can be created;
g – a surjective genotype-phenotype mapping of the form $g : C \to M$;
f – a "machine" fitness function of the form $f : M \to R$;
$\Phi = f \circ g$

In order to illustrate the proposed formal approach, assume that a definition of non-uniform cellular automaton A consists of four components $A = (c_1, c_2, c_3, r)$, where $r = \{0, 1\}^n$ denotes cellular automaton rules encoded as n-bit string. These rules represent a genotype. Then a set of all machines that can be evolved corresponds to a 2^n-element set of cellular automata of the form

$$M = \{(c_1, c_2, c_3, r) \mid c_1 = k_1, c_2 = k_2, c_3 = k_3\}$$

where k_1, k_2, and k_3 are some invariable objects. Genotype-phenotype mapping is constructed as $g : \{0, 1\}^n \to M$. Cellular automata are then evaluated using f.

We can see that only four mathematical components (E, M, g, f) are needed in the definition. Note that f is a problem specific function that can be changed over time; g can cover any type of constructional process and M is defined implicitly or explicitly before the evolution is executed. Looking via the proposed definition, all evolvable machines operate in the *same way*. That is clearly illustrated on very similar Figures 6 and 8. We consider the

evolvable computational machine as a basic theoretical model of evolvable computing.

6.3. *The computational power and evolvable computing*

In the case of evolvable computational machines, we try in fact to find an algorithm (machine) in a given set of algorithms (e.g. in the set of all configurations of an FPGA) that satisfies the objectives (formulated via the fitness function) by means of an evolutionary algorithm in which the problem encoding, genotype–phenotype mapping and genetic operators remain unchanged during the run (see Figures 6 and 8). Even if the (machine) fitness function is being changed, we do not have any chance to *improve* the encoding, genotype–phenotype mapping and genetic operators (which is unpleasant from the NFL theory perspective). Furthermore, we have to assume infinite and interactive computation.

It was proven that evolvable computational machines operating in a dynamic environment exhibit a super-Turing computational power (Sekanina, 2003a). If the evolutionary algorithm employed is efficient, then their computational power is equivalent to the so-called *interactive Turing machine with advice*. If the lifespan of the physical implementations of these machines (e.g. evolvable hardware) can be considered as infinite, then these computers cannot be simulated by means of a standard Turing machine.

This viewpoint corresponds with the recent results in an emerging field – hypercomputation (or super-Turing computation) (Copeland and Sylvan, 1999; van Leeuwen and Wiedermann, 2001a, 2001b) – that some theoretical models and modern computational systems do not share the computational scenario of a standard Turing machine and hence they can not be simulated on Turing machines. Paper (Eberbach et al., 2004) clearly surveys the field.

Let us only note that a standard Turing machine supposes that input data are available before a computation is started (no interaction is enabled later) and that a uniform algorithm (which is invariable and never changed during execution) processes them. Nevertheless van Leeuwen and Wiedermann have shown that such computations may be realized by an interactive Turing machines with advice (van Leeuwen and Wiedermann, 2001b) – a classical Turing machine endowed with three important features: advice function (it is a weaker type of oracle), interaction and infinity of operation. For example, a model of Internet (van Leeuwen and Wiedermann, 2001b) and some embedded systems (van Leeuwen and Wiedermann, 2001a) possess the same computational power as an interactive Turing machine with advice. However, only in the case that their life-span is *infinite*. Otherwise, the computation is finite and remains in the scope of standard Turing machine and the standard Church-Turing thesis.

We can observe that evolvable computational machines operating in a dynamic environment show simultaneous non-uniformity of computation, interaction with an environment, and infinity of operations. Furthermore, the point at time in which the fitness function (specification) is changed is in general *uncomputable* (see the *Driving Home from Work* problem presented in (Eberbach et al., 2004) that is uncomputable on a standard Turing machine, but computable in reality).

At each time point these evolvable devices have a *finite* description. However, when one observes their computation in time, they represent *infinite* sequences of reactive devices computing non-uniformly. The "evolution" of machine's behavior is supposed to be endless. In fact it means that they offer an example of real devices (physical implementations) that can perform computation that no single Turing machine (without oracle) can.

6.4. *Beyond the digital world*

Theoretical computer science has always distinguished between abstract models of computers and their physical implementations. The limits of computation are formulated using mathematical concepts (e.g. the Church-Turing thesis). D. Deutsch claims in his book (Deutsch, 1997) that computation is a physical process. Hence it is impossible to determine by mathematical reasoning what can or cannot be calculated mathematically. That depends entirely on the laws of physics. This idea is mainly discussed in the area of quantum computing (Gruska, 1999).

There are some interesting issues coming with evolvable hardware realized in *physical* reconfigurable devices. Assume that we have a reconfigurable digital circuit consisting of K programmable elements whose function is defined using the configuration bit stream stored in the B-bit configuration memory. There are 2^B different configurations, i.e. up to 2^B algorithms that can be realized in the reconfigurable device. In fact the number of different algorithms is usually much smaller. Although every two different configurations determine two physically different electronic circuits, the two configurations can produce the same (digital) behavior because of inherent redundancy of reconfigurable circuits.

It was shown that under some conditions, some configurations could exhibit *useful* digital behavior that *none* of 2^B configurations can do under normal conditions. Note that the fitness function has remained formally unchanged. For instance, Thompson has evolved a tone discriminator, which returns logical `H` for a 1 kHz input signal and logical `L` for a 10 kHz input signal utilizing only a few programmable elements without using any of the counter/timers or RC networks that conventional design would require for this task. The evolved circuit contained several continuous-time recurrent loops

and the timing mechanism relied on a subtle analogue property, possibly parasitic capacitance, which effected delays in the internal signal paths. All the loops and timing mechanisms would have been forbidden under conventional design procedure. The evolutionary algorithm exploited physical properties of a given FPGA and some other conditions during the evolution (such as temperature) to create the required behavior. Thompson noted: "This circuit is discriminating between inputs of period 1 ms and 0.1 ms using only 32 cells, each with a propagation delay of less than 5 ns, and with no off-chip components whatsover: a surprising feat. Evolution has been free to explore the full repertoire of behaviors available from the silicon resources provided, even being able to exploit the subtle interactions between adjacent components that are not connected directly (Thompson et al., 1999)." When the circuit is created from discrete components according to the evolved configuration, it did not work at all. Furthermore, the evolved configuration has not produced the same behavior when it was uploaded into another FPGA of the same type.

The presented example shows that a purely finite digital computational device can exhibit useful computation that is beyond the space defined by its abstract mathematical model. The behavior did not emerge accidentally; the evolutionary algorithm was able to repeat the behavior.

Of course, we could include various features into the model of the reconfigurable device and to consider various variable aspects of the environment where the computation is carried out. Nevertheless, nobody can guarantee that the evolutionary assemble-and-test approach to problem solving will utilize something, which is not included in the model of the computational system (consider the evolved radio utilizing the radio waves emanating from nearby PCs (Bird and Layzell, 2002)). Hence the evolutionary design conducted directly in physical reconfigurable devices can always surprise us. On the other hand, the evolutionary approach can be utilized to explore materio directly at a very low level, for instance, atomic. NanoCell is a working example (Tour, 2003).

Traditional approaches of theoretical computer science such as computability and complexity do not give us sufficient tools for investigation of evolvable computing performed in physical devices. The problem is that it is difficult to create a realistic model of such a device and its working environment. We do not know the entities that will be used by evolution to compose the required behavior. Although the required behavior is digital, it is created from non-digital entities.

We do believe that the mappings g and f are crucial for theoretical understanding the evolvable computing. We offer two explanations of Thompson's experiments:

In the first viewpoint, f is important. In fact, f assigns two different fitness values to a single machine $m \in M$. Note again that the fitness function has remained formally unchanged. In reality, m is represented by an electronic circuit (configuration) whose behavior depends on something missing in the fitness calculation. In this case, f can not be seen as a function, but f is a *relation*. In theoretical computer science the situation usually leads to defining the non-deterministic variant of a machine.

In the second viewpoint, g is important. Fitness function has never assigned different fitness values to a single machine (assuming invariable fitness function). All phenotypes are seen as different (but they can obtain the same fitness value). However, a single genotype can generate two or more different phenotypes (circuits), because it is assumed that environment influences the genotype-phenotype mapping. The situation is known from nature in which phenotypes depend on the environment where they are constructed (developed). For instance, the development depends on gradient concentrations of some substance nearby a dividing cell (Wolpert, 2000).

Note that there is not practically any problem in the area of genetic programming in case that a program is evolved directly in the computer memory. The evolution works at the level of well-defined discrete instructions. Every, even if randomly generated, sequence of instructions is legal and will be interpreted in the same way in all reasonable environments.

7. Conclusions

In this paper we understood evolvable computing as a dynamically executed evolutionary design process producing efficient hardware and/or software for a given problem in a given environment and at a given moment. Typical applications of evolvable computing exhibit the following features: software and hardware is dynamically changed and evolved during execution, the system interacts within a physical environment and it is assumed that the computation will not be terminated.

The proposed evolvable components have allowed us to introduce reusability concepts to evolvable computing. We introduced evolvable components that are available as configuration bit streams or at the level of VHDL code. It means that they can be utilized for any reconfigurable digital device of a sufficient capacity. Thus evolvable computing has become platform independent. The two examples demonstrate complete hardware implementations of evolvable systems in ordinary FPGAs.

An open issue is whether evolvable computing will be able to *routinely* solve some problems more effectively than traditional computers determined by abstract computational models and their limitations. In fact, some inter-

esting digital circuits have already been discovered in the evolvable hardware field. We do not know why they work, but they work.

Evolvable computing is computing beyond the scope of an ordinary Turing machine. It does not violate the Church-Turing thesis because the thesis deals with a slightly different class of computations, which corresponds to the concept of algorithm. However, this class of computations is not typical for contemporary computational systems.

Acknowledgment

The research was performed with the Grant Agency of the Czech Republic under No. 102/03/P004 *Evolvable hardware based application design methods* and No. 102/04/0737 *Modern methods of digital system synthesis* and the Research intention No. MSM 262200012 – *Research in information and control systems*.

References

Bäck T (1996) Evolutionary Algorithms in Theory and Practice. Oxford University Press, New York Oxford

Bentley P and Corne DW eds (2001) Creative Evolutionary Systems. Morgan Kaufmann

Bentley P (2002) Digital Biology. Simon and Schuster

Bird J and Layzell P (2002) The evolved radio and its implications for modelling the evolution of novel sensors. In: Proceedings of Congress on Evolutionary Computation (CEC 2002), pp. 1836–1841

Bondalapati K and Prasanna VK (2002) Reconfigurable computing systems. Proc. of the IEEE 90(7): 1201–1217

Bourianoff G (2003) The future of nanocomputing. IEEE Computer August: 44–53

Bradley D, Ortega-Sanchez C and Tyrrell A (2000) Embryonics + immunotronics: A bio-inspired approach to fault tolerance. In: Proc. of the 2nd NASA/DoD Workshop on Evolvable Hardware, Palo Alto, CA, USA, 2000, pp. 215–222. IEEE Computer Society, Los Alamitos

Brooks R (1999) Cambrian Intelligence. The MIT Press, Cambridge, MA

Compton K and Hauck S (2002) Reconfigurable computing: A survey of systems and software. ACM Comput. Surv. 34(2): 171–210

Copeland BJ and Sylvan R (1999) Beyond the universal Turing machine. Australasian J. of Philosophy 77(1): 46–66

Dawkins R (1991) The Blind Watchmaker. Penguin Books, London

deHon A (1998) Comparing computing machines. In: Configurable Computing: Technology and Applications, pp 124–133. Bellingham, WA, Proc. SPIE 3526

Deutsch D (1997) The Fabric of Reality. Penguin Books, New York

Eberbach E (2002) On expressiveness of evolutionary computation: Is EC algorithmic? In: Proc. of Congress on Evolutionary Computation 2002, pp. 564–569. IEEE Press

Eberbach E, Goldin D, Wegner P (2004) Turing's ideas and models of computation. In: Teuscher Ch (ed.), Alang Turing: Life and Legacy of a Great Thinker, pp. 159–194. Springer-Verlag, Berlin

Flake GW (1998) The Computational Beauty of Nature. The MIT Press, Cambridge, MA

Flockton SJ and Sheehan K. Intrinsic circuit evolution using programmable analogue arrays. In: Proc. of the Conf. on Evolvable Systems: From Biology to Hardware ICES'98, pp 144–153. Springer-Verlag, Berlin

Gordon T and Bentley P (2001) On evolvable hardware. In: Ovaska S and Sztandera L (eds) Soft Computing in Industrial Electronics, pp. 279–323. Physica-Verlag, Heidelberg

Gruska J (1997) Foundations of Computing. Int. Thomson Publishing Computer Press

Gruska J (1999) Quantum Computing. McGraw Hill, New York

Haddow P and Tufte G (2001) Bridging the genotype–phenotype mapping for digital FPGAs. In: Proc. of the 3rd NASA/DoD Workshop on Evolvable Hardware, Long Beach, CA, USA, 2001, pp. 109–115. IEEE Computer Society, Los Alamitos

Hartenstein R (2002) Configware/Software co-design: Be prepared for the next revolution. In: Proc. of the 5th IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop, Brno, Czech Republic, 2002, pp. 19–34. Brno University of Technology, Brno

Hennessy JL and Patterson DA (1996) Computer Architecture – A Quantitative Approach. Morgan Kaufman Publishers, San Francisco

Higuchi T et al. (1993) Evolving hardware with genetic learning: A first step towards building a Darwin machine. In: Proc. of the 2nd International Conference on Simulated Adaptive Behaviour, pp. 417–424. MIT Press, Cambridge MA

Higuchi T et al. (1999) Real-world applications of analog and digital evolvable hardware. IEEE Trans. on Evolutionary Computation 3(3): 220–235

Holland J (1975) Adaptation in Natural and Artificial Systems. University of Michigan Press, Ann Arbor

Koza JR (1992) Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge MA

Koza JR et al. (1999) Genetic Programming III: Darwinian Invention and Problem Solving. Morgan Kaufmann Publishers, San Francisco CA

Koza JR, Keane MA and Streeter MJ (2003) What's AI done for me lately? Genetic programming's human-competitive results. IEEE Intelligent Systems May/June: 25–31

www.liberouter.org

Linden DS (2002) Optimizing signal strength *in-situ* using an evolvable antenna system. In: Proc. of the 4th NASA/DoD Conference on Evolvable Hardware, Alexandria, Virginia, USA, 2002, pp. 147–151. IEEE Computer Society, Los Alamitos

Macias N (1999) The PIG paradigm: The design and use of a massively parallel fine grained self-reconfigurable infinitely scalable architecture. In: Proc. of the 1st NASA/DoD Workshop on Evolvable Hardware, Pasadena, CA, USA, 1999, pp. 175–180. IEEE Computer Society, Los Alamitos

Mange D et al (2000) Towards robust integrated circuits: The embryonics approach. Proc. of IEEE. 88(4): 516–541

Miller J, Job D and Vassilev V (2000) Principles in the evolutionary design of digital circuits – Part I. Genetic Programming and Evolvable Machines, Vol. 1(1), pp. 8–35

Miller J and Downing K (2002) Evolution in materio: Looking beyond the silicon box. In: Proc. of the 4th NASA/DoD Conference on Evolvable Hardware, Alexandria, Virginia, USA, 2002, pp. 167–176. IEEE Computer Society, Los Alamitos

Murakawa M et al (1996) Evolvable hardware at function level. In: Proc. of the Parallel Problem Solving from Nature Conference, LNCS 1141, pp 62–71. Springer, Berlin

Nolfi S and Floreano D (2000) Evolutionary Robotics: The Biology, Intelligence, and Technology of Self-Organizing Machines. MIT Press, Cambridge MA

PicoChip home page, http://www.picochip.com

Sekanina L and Ruzicka R (2000) Design of the special fast reconfigurable chip using common FPGA. In: Proc. of the IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop, Bratislava, Smolenice, 2000, pp. 161–168. Polygrafia SAF, Bratislava

Sekanina L (2003a) Evolvable Components: From Theory to Hardware Implementations. Natural Computing Series, Springer Verlag, Berlin

Sekanina L (2003b) Towards evolvable IP cores for FPGAs. In: Proc. of the 2003 NASA/DoD Conference on Evolvable Hardware, Chicago, USA, pp. 145–154. IEEE Computer Society Press

Sekanina L and Friedl S (2004) On routine implementation of virtual evolvable devices using COMBO6. In: Proc. of the 2004 NASA/DoD Conference on Evolvable Hardware, Seattle, USA. IEEE Computer Society Press

Sipper M et al. (1997) A phylogenetic, ontogenetic, and epigenetic view of bio-inspired hardware systems. IEEE Trans. on Evolutionary Computation 1(1): 83–93

Sipper M (2002) Machine Nature: The Coming Age of Bio-Inspired Computing. McGraw Hill, New York

Stephens CR and Zamora A (2003) EC theory: A unified viewpoint. In Proc. of GECCO 2003, LNCS 2724, pp. 1394–1405. Springer Verlag

Stoica A et al (2000) Evolution of analog circuits on field programmable transistor arrays. In: Proc. of the 2nd NASA/DoD Workshop on Evolvable Hardware, Palo Alto, CA, USA, 2000, pp. 99–108. IEEE Computer Society, Los Alamitos

Tan KC, Wang LF, Lee TH and Vadakkepat P (2004) Evolvable Hardware in Evolutionary Robotics. Autonomous Robotics. 16(1): 5–21

Thompson A (1998) Hardware Evolution: Automatic Design of Electronic Circuits in Reconfigurable Hardware by Artificial Evolution. Distinguished Dissertation Series, Springer, London

Thompson A, Layzell P and Zebulum RS (1999) Explorations in design space: unconventional electronics design through artificial evolution. IEEE Trans. on Evolutionary Computation 3(3): 167–196

Torresen J (2002) A scalable approach to evolvable hardware. Genetic Programming and Evolvable Machines 3(3): 259–282

Tour JM (2003) Molecular Electronics. World Scientific

van Leeuwen J and Wiedermann J (2001a) A Computational Model of Interaction in Embedded Systems. Technical Report UU-CS-2001-02, Utrecht University, The Netherlands

van Leeuwen J and Wiedermann J (2001b) The Turing machine paradigm in contemporary computing. In: Mathematics Unlimited – 2001 and Beyond, pp. 1139–1155. Springer, Berlin

Wagner G and Altenberg L (1996) Complex adaptations and the evolution of evolvability. Evolution 50(3): 967–976

Wiedermann J (2004) Building a bridge between mirror neurons and theory of embodied cognition. In: SOFSEM 2004: Theory and Practice of Computer Science, 30th Conference on Current Trends in Theory and Practice of Computer Science, LNCS 2932, pp. 361–372. Springer-Verlag, Berlin

Wolpert DH and Macready WG (1997) No free lunch theorems for optimization. IEEE Transactions on Evolutionary Computation 1(1): 67–82

Wolpert L (2000) The Triumph of Embryo. Oxford University Press

Xilinx, Inc. (2004) WWW home page: http://www.xilinx.com

Yao X and Higuchi T (1999) Promises and challenges of evolvable hardware. IEEE Transactions on Systems, Man, and Cybernetics 29(1): 87–97

Zhu J and Sutton P (2003) FPGA implementations of neural networks – a survey of a decade of progress. In: Proc. of the 13th International Conference on Field-Programmable Logic and Applications, LNCS 2778, Springer Verlag, Berlin, pp. 1062–1066