Michal Bidlo

Evolutionary Design of Generic Structures Using Instruction-Based Development

Dissertation thesis

Faculty of Information Technology
Brno University of Technology
Brno, Czech Republic

Preface

Computational development rerpesents an extensive subset of the evolutionary design area. In general, the development is intended as an additional mechanism of evolutionary algorithm attempting to overcome the problem of scale that represent a crucial issue during the evolutionary design. Many models and techniques have been introduced so far, including their applications in various fields.

This book introduces a new class of developmental methods called an instruction-based development. The key feature is the evolution of application-specific programs, consisting of simple instructions, which is similar to the linear genetic programming approach. The concept of programs, in fact, enables to establish an universal computational model depending on the instruction set involved, interpretation and way of execution of the instructions. The program, represented as a sequence of instructions, can thus specify an arbitrary algorithm which is understood as a prescription for the development (construction) of a target object.

The objective of this work is to apply the instruction-based development to design generic structures. Combinational circuits have been chosen as suitable domain to demonstrate the capabilities of this approach. Experiments have been conducted, Two different approaches to the instruction-based development have been introduced.

The first approach has been called a continual development. The target circuit can grow from an initial solution theoretically infinitely, preserving a desired function all the time during the development. A case study of the continual development is presented in the domain of the evolutionary design of generic sorting networks. It has been shown that the evolution is able to discover innovative solutions which exhibit better parameters in comparison with a conventional principle. The general properties of the best result have been demonstrated formally. Moreover, evolution of generic polymorphic circuits has been presented using the continual development approach.

The second approach represents a parametric development. In this case the target circuit is developed every time from the start, while the size of its target instance is specified by a parameter. An external information, that we called an environment, has been introduced into the developmental system in order to develop generic structures containing irregular parts. The experiments have been conducted in the area of the evolutionary design of generic combinational multipliers. Two variants of a parametric developmental system have been presented. The first one represents an initial experiment of the evolution of common generic multipliers using the development, whilst the second one is intended to design effective generic carry-save multipliers.

In general, we have introduced an extensive set of experiments demonstrating the capability of the proposed concepts of instruction-based development to design various generic structures, including a discovery of some new general innovative solutions.

IV

Acknowledgement

I would like to thank very much prof. Lukáš Sekanina, the supervisor over a period of my PhD study and the head of the research group which I am a member of at FIT BUT, for his continuous moral and professional support.

Contents

1

Introduction

From the very beginning of life on the Earth, a huge variety of living forms emerged including both vegetation and animal. However, none of them was born in a moment equipped fully with all appropriate capabilities. They are a result of a biological tool called the development (ontogeny), representing a process so complicated that even the modern science utilizing the recent knowledges and front-end devices is not able to provide in some cases a reliable answer to all the question of origins or principles of its functioning. But before the evolution could create such a different range of life, it created the process of development which is liable for creating every species of life on this planet. The development is in fact the process of construction that is based on the genes, proteins, cells and the environment that affect each other in order to create a living organism [S. 03]. In nature the environment influences significantly the developmental process. Therefore, different organisms can arise that are adapted to the specific conditions (e.g. temperature, air pressure, sun shine etc.). The organisms possess different properties living in a desert or in polar areas, possibly in a river or in a deepness of the ocean. There is a big difference of temperature, pressure or light among these environments and the adaptation of the living organisms is crucial for their surviving. The fundamental principles and properties of the biological development represent the main inspiration for a specific biologically inspired technique in the area of the computer science — computational development — which represents the main topic of this book.

1.1 Computational development

In fact, the term computational development, which was originally introduced in [Kum04], represents a set of techniques inspired by the biological development. In the area of the evolutionary algorithms, these techniques are interpreted as the prescriptions transforming the genotypes to the phenotypes. While the genotypes (also called chromosomes or genomes in the parlance of

evolutionary computing) represent directly the candidate solutions, in case of the traditional evolutionary design, the developmental approaches interpret a genotype as a prescription for the construction of the candidate solution (a phenotype). As evident, the process of the construction is crucial for both the biological and computational development. Since the construction of a candidate solution from the genotype using the computational development is performed indirectly by means of the prescription stored in the genotype and the knowledges included into the evolutionary system, the techniques of the computational development is usually denoted as a generative representation or generative mapping from the genotypes to the phenotypes. In the next sections the term development will always denote the computational development, if not explicitly specified. Note that some literature use the terms computational embryology or embryogenesis that pose the same as the development.

The presence of many approaches of the genotype–phenotype mapping arises a question of what can be considered as the developmental mapping and what can be not. To clarify this problem, let us compare the properties of the direct and generative encodings in more detail. The direct representation of the candidate solutions utilizes a complete knowledge of its structure, i.e. the set of the building blocks and their interconnection. If the phenotype contains repeated structures, they are encoded by repeated sequences of code in the genotype. Moreover, one element in the set of the genotypes corresponds to exactly one element in the solution space (the set of the candidate solutions – the phenotypes). For instance, consider a digital circuit represented at the gate level using the cartesian genetic programming (CGP) [MT00]. The circuit is encoded as a sequence of integers that represent the functions of the gates and their interconnection for all the elements of the particular structure used in CGP. It means the genome defines the structure of the circuit by means of the building blocks the circuit is composed of. Another example of the genotype–phenotype mapping may consider the problem of the optimization of a multidimensional mathematical function, i.e. to find its extreme. In this case the chromosome consists of a bit sequence interpreted as a binary representation of the values of the function variables (domain of the function). The mathematical expression of the function maps the elements of the domain onto the appropriate values in the co-domain of the function. In both examples the genome interpretation is determined by the mapping which assigns unambiguously to each genotype an appropriate phenotype. This mapping is based on the exact and complete specification of the candidate solution included in the genotype. Therefore, the genotype–phenotype mapping specifies what to compute (or construct). On the other hand in case of the generative encoding the chromosome represents a prescription (an algorithm) for the construction of the target object, i.e. the genotype specifies how to build the phenotype using the building blocks and knowledges included into the developmental system. The generative representations thus focus on the algorithm for the construction of the candidate solution rather than on representing the

solution itself. Since the developmental process is usually non-trivial, usually including advanced techniques (recursion, external control etc.), it is possible to increase the generative power of the genotype–phenotype mapping and keep the genome length and the evolution time reasonable.

1.2 Why is the development useful?

The increasing demands on the complexity and size of the solutions created by the evolutionary design lead to increasing requirements on the encoding of the solutions for the evolutionary systems. The consequence of this issue is that the genome length (and hence the search space) grows enormously and the evolutionary algorithm is not able to find the solution in a reasonable time using the direct encoding and the common computing resources. This phenomenon is denoted as the problem of scale. Moreover, the time needed for evaluating the candidate solutions plays an important role. In case of the evolutionary design of digital circuits the evaluation time increases exponentially depending on the number of inputs of the target circuit. Therefore, the power of the evolutionary design process is degraded by the evaluation of the candidate solutions. As evident, there are two different levels of the scalability in the evolutionary design which differ in the factor having a crucial impact on the scalability. (1) The length of the chromosome usually relates to the complexity of the solution structure that should be encoded inside the chromosome. Therefore, it is a case of the problem of scale at the level of the structure of the candidate solutions. (2) The problem of scale at the level of the fitness function relates to the computational effort of the evaluation of the candidate solutions.

The development represents one of the possible approaches trying to overcome the problem of scale. Since the genome contains the algorithm — the prescription for the construction of the candidate solution — it is possible to utilize its computational abilities which may, in some cases, reduce the length of the chromosome and make the evolution more efficient. Moreover, advanced features can be described by the algorithm that are out of the scope of the direct encoding. The features, which are inspired by the phenomenons known from the biological development, include:

- Adaptation – the phenotype is able to develop with respect to different condition, requirements etc.
- Compact genotype – in some cases the length of the genotype can be reduced by using the suitable generative representation.
- Search space reduction – in some cases the size of the search space keeps reasonable which is the consequence of shorter genotypes.
- Reusability – if the candidate solution contains repeated substructures, it is possible to utilize only one fragment of code in the genotype to describe all the repeated structures, i.e. to reuse the parts of the phenotype.

- Self-repair – it is possible to restore the function of the malformed part of the phenotype.
- Scalability – the computational abilities of the generative representations enable the phenotype to "grow", for example by means of a parametric approach, recursion and other techniques.
- Emergent behavior and self-organization – simple interaction between the computational elements of the generative representation may lead to a very complex global behavior. Note that this feature is typical especially for cellular automata and genetic regulatory networks.

1.3 Problems of generative representations

Despite the useful properties and abilities of the development summarized in the previous section, there are some problems related to the generative representations which have to be dealt with in the applications of the developmental systems. In general, the design of the generative representation for a particular problem is difficult as the set of building blocks and their interactions may not be known to solve the problem effectively. In consequence, the evolutionary design process using the proposed developmental encoding may be limiting since the design of the generative representation is usually performed by means of the conventional engineering methods that may restrict the evolution in searching the unexplored but promising parts of the solution space.

1.4 Inspiration for this work

This work focuses on a specific class of the developmental systems based on the evolution of computer programs. In fact, the basic idea is very close to the genetic programming (GP) approach that poses one of the most known evolutionary algorithm nowadays. Although the first experiments working with the principles of the genetic programming have already been conducted by Stephen F. Smith [Smi80] and Nichael L. Cramer [Cra85], as the father of this evolutionary technique is usually considered John R. Koza who published a paradigm for genetically breeding populations of computer programs [Koz90] which, in fact, was later used as a draft for the first of Koza's famous series of books [J. 92][J. 94][J. 99][J. 03].

The original Koza's work represents the basic principle of GP applied in the direct genotype–phenotype mapping approach. Later, modifications of GP were introduced in order to exploit the advantages of developmental encoding and the developmental genetic programming (DGP) was introduced (see [J. 99] and [J. 03] for details). DGP was applied for example to design the topology and sizing of electrical circuits, placement and routing of electrical circuits, design of antennas and others. Although the approach to the

genotype–phenotype mapping is different using DGP in comparison with the original GP mapping, both variants use the tree representation of the program code.

In recent years different variants of genetic programming have emerged, following the fundamental idea of Koza's GP, i.e. the automatic evolution of computer programs. Three basic forms of representations may be distinguished which differ in the structure of the programs. In addition to Koza's tree representation these include linear and graph representations [BNKF97]. For the purposes of this work it is especially mentioned the linear representation leading to the linear genetic programming (LGP) that is described and analyzed in detail in [BB07].

In [Sek04] Sekanina presented a novel evolutionary developmental method for the design of arbitrarily large sorting and median networks. His approach utilizes very similar idea based on the LGP representation regarding the evolution of computer programs for the design of digital circuits. The programs to be evolved consist of application-specific instructions intended for manipulating the building blocks (so-called comparators) of the sorting and median networks. Therefore, the developmental model is designed ad hoc – for a particular application and with respect to the properties of the circuits to be developed. Moreover, in comparison with Koza's development of the electric circuits at the transistor level, the design of sorting and median networks is performed at a higher level of abstraction. This work actually represents the main inspiration for the research presented in this book. Advanced modifications will be presented and other developmental models based on the linear representation will be introduced for the evolutionary design of generic structures of combinational circuits.

1.5 Research motivation

Although there are some difficulties regarding the applications of the computational development in the area of the evolutionary design, generative mappings provide many features that are not available in the direct representations. These features may be important for the design and functioning of up-to-date systems in various fields. Therefore, the research of the developmental models and their properties represents a significant area of the present science.

Considering the results obtained by means of traditional genetic programming, and developmental genetic programming in particular, the research in this area indicates that the GP approach (independently on the representation applied) provides a powerful technique to produce interesting and innovative outcomes. Several PhD theses and books have been published dealing with the principles and applications of different developmental models in various fields. A brief overview of the work related to the developmental models is stated in Section 3.2.

6   1 Introduction

The utilization of the LGP concept in this book is based on Sekanina's paper [Sek04], where a developmental method was presented demonstrating the ability of an evolutionary algorithm to design growing structures. In particular, an algorithm for the design of arbitrarily large sorting and median networks was introduced. The evolution discovered a solution not only for a single instance of the problem, but for all the instances. In this case, the concept of the design of generic structures is based on iterative process. The approach introduced in [Sek04] foreshadows that there may be a big potential for a successful research in this area.

Koza has dealt with the evolution of analog circuits using the developmental genetic programming. The circuits are functional in a wide range of values of the circuit components. The values of components are determined by mathematical expressions containing free variables. Koza called this concept as the design of parametrized topologies. For more information, see chapters 9–13 of [J. 99]. Although the evolved circuits may be considered as general from the point of view of the component parameters, they are not general from the point of view of the circuit structure and scalability (i.e. the number of inputs).

In fact, there is a different aspect of considering the generality of the developed solutions if compared Koza's applications of GP and the experiments presented in this book. While Koza considers the development of specific circuit structures in which the generality is achieved by different component values, another kind of generality of the developed solutions can be considered from the point of view of the size of the target instances (at the level of the object structure).

Although plenty of applications utilizing the genetic programming have been presented, the automatic construction of generic solutions at the structural level still represents a rare case in the evolutionary design. Therefore, the evolutionary development of arbitrarily large structures will constitute a crucial issue for the research in this book. Specifically, it will be focused on the adaptation of the developmental system to the design of arbitrarily large combinational circuits of various classes. As Koza and others demonstrated the power of the genetic programming in many applications, it is supposed that this approach may also be successfully applied in this research.

For the purposes of the generic evolutionary design presented in this book, a new concept of development utilizing the principles of genetic programming has been invented. A new term "application-specific instruction-based development" is introduced and two different approaches are identified: (1) continual development and (2) parametric development. The features of this concept are described in Chapter 4. The case studies demonstrating the capabilities of the new approach are presented in Chapter 5 and 6. Note that that these three chapters represent the main contribution of this work.

1.6 Research hypotheses

In the previous sections, some crucial topics were mentioned in which the developmental mappings are contributive. On the basis of the existing successful experiments related to the concept of genetic programming, a promising area of so-called application-specific instruction-based development has been selected as a suitable domain for the research in this book. This concept can be applied in different areas (for different classes of circuits) as demonstrated in the next chapters. Considering the two different approaches of the instruction-based development, the following hypotheses will be formulated for the purposes of this work with respect to the issues stated in the previous sections.

Hypothesis 1

The evolution is able to generate infinitely growing structures if the continual development is applied using a proper encoding.

Hypothesis 2

The evolution is able to generate parametrized generic structures if the parametric development is applied using a proper encoding.

Hypothesis 3

The evolution is able to generate innovative general solutions or rediscover conventional general solutions if an appropriate approach of instruction-based development is applied using a proper encoding.

All the hypotheses deal with an evolutionary algorithm utilized to design the program consisting of application-specific instructions by means of which the generic solution is developed. The successfulness of the methods is strongly dependent on the problem encoding chosen as common in the evolutionary design field. However, plenty of successful experiments have been presented using the genetic programming approach and developmental genetic programming in particular by means of which "general" solutions have been evolved. In addition, as demonstrated in chapters 5 and 6, it is possible to evolve generic combinational circuits (i.e. to design a program for the development of arbitrarily large instance of a given circuit) using the proposed instruction-based approach.

In order to confirm the hypotheses, a domain-specific knowledge is needed to be supplied to the evolutionary developmental system which also influences the representation (encoding) of the candidate solutions – phenotypes. This information is usually inspired by a conventional solution of the given problem. Note that structural level is considered during the design, i.e. a set of building blocks together with the rules of their composition is chosen and the evolution

8   1 Introduction

searches for a program to construct the target system. It is evident that if a general solution is known for the design of a given system (to solve a given problem), it is always possible to choose a set of instructions by means of which a program (algorithm) is created to construct that system (i.e. its arbitrarily large instance). Of course, the fact that a generic solution of a problem can be represented as a program of an instruction-based developmental system does not imply that it can be successfully evolved. The proper encoding is just the crucial issue for the evolutionary algorithm to design that solution. The experiments shows that the design of an efficient developmental encoding for a given problem is not a trivial task. Nevertheless, this is a common obstacle in the evolutionary design and evolutionary developmental design in particular.

However, the approach described in the previous paragraph (i.e. designing an instruction-based evolutionary developmental system on the basis of known general conventional solutions) was successfully applied in several domains from the area of creating generic combinational circuits. For example the general straight-insertion algorithm was considered as conventional construction algorithm on the basis of which the instruction set, sorting networks representation, interpretation and the way of execution of the instructions were chosen as described in Section 5.3. It represents one of the possible approaches that enables to develop arbitrarily large sorting networks which can "grow" continually by increasing the number of inputs maximally by two in each step. In Section 5.5, a different instruction set and a modified construction process is described enabling the sorting networks to "grow" by up to four inputs per a step, exhibiting more complex structures of the resulting networks which demonstrates better properties of this encoding in comparison with the previous one. Similar approach was applied in case of development of generic multipliers presented in Chapter 6. This demonstrates that more generative encodings exist to solve a given problem; different levels of efficiency of evolutionary process as well as resulting solutions may obviously be observed.

1.7 Goals and proposed solutions

The main goal of this work is to apply the concepts of continual development and parametric development to the construction of generic (i.e. arbitrarily large) combinational circuits of different classes.

The problem of scale represent the most significant impediment related to the evolutionary design of complex systems. If the design of generic combinational circuits is considered, the number of inputs represents an important issue that influences the scalability of the solutions to be evolved as the evolution time increases exponentially with the increasing number of inputs. Therefore, it is the case of the problem of scale at the level of the fitness function (i.e. the evaluation of the candidate solutions). Since this work focuses on the evolutionary development of generic circuit structures, the goal is, in particular, to provide the scalability of the circuits evaluation. In order

to do so, the following approach will be applied during the evolutionary process. A genome of the evolutionary algorithm encodes a single prescription for the development (construction) of the solution. If the continual development is considered, then the evaluation involves more solutions resulted from more iterations of the development (i.e. applications of the program being evolved). However, it is not possible to evaluate all the instances. Therefore, only a subset of the solutions are evaluated after each of a reasonable number of iterations. At the end of the evolutionary process, the evolved program is applied for higher number of iterations in order to verify if it is general (i.e. able to construct potentially arbitrarily larger solutions than that were evaluated during the evolution).

Similarly, a reasonable finite number of instances is evaluated during the evolutionary process for several different values of parameters in case of the parametric developmental approach. The resulting programs are then verified in order to determine if they are able to work also for other values of the parameter.

It is not possible to verify the program considering all the instances of the developed solution, therefore, only a subset of instances is evaluated during verification. In general, there are two possibilities how to demonstrate that the development by means of the evolved program is really general. (1) To prove rigorously by means of theoretical apparatus that the evolved program is able to construct arbitrarily large solution. (2) To identify whether the evolved program represents an already known general method, then the comparison of the evolved and conventional method demonstrates the generality of the evolved program.

Suitable problem domains are needed to find out in order to confirm the research hypotheses and to demonstrate the abilities of the proposed developmental approach. Arithmetic circuits whose operands may be represented by arbitrary number of bits were identified as proper objects for the continual and parametric development. In particular, the continual development will be demonstrated on the sorting network problem and the parametric development will be applied to the design of combinational multipliers.

1.8 Contributions

The contributions of this work follow from the features and abilities of the proposed developmental method. This approach is inspired by the linear genetic programming. On the basis of the developmental model two different variants are identified: continual development and parametric development. This concept is applied on the development of generic circuit structures. In order to demonstrate the abilities of the proposed approach, sorting network problem and combinational multiplier design are selected as suitable application domains for the evolutionary design using the development. It is shown

that both the approaches are able to construct arbitrarily large circuits of the given classes.

The continual development approach is applied on solving the generic sorting network problem. It is demonstrated that the evolution is able to discover an innovative solution in comparison with the conventional method of the same type. The best evolved algorithm for the construction of the sorting networks is analyzed, optimized and proved to be general using the mathematical apparatus. The development of the sorting networks represents a rare case in the area of the evolutionary design related to the ability of the evolved solution to grow continually and theoretically infinitely. The scalability obtained by this approach is based on the utilization of the regular structures of the resulting circuits that enable us to increase the number of inputs of the circuit being developed preserving the circuit functionality. In case of the development of the sorting networks, the growth may be performed in steps differing in the number of inputs of the subsequent sorting network instances. The development of sorting networks with the number of inputs increasing by 4 has been evolved so far. The resulting sorting networks exhibit better properties in comparison with the networks of the same size created by means of the conventional method. This feature demonstrates the robustness of the continual development with respect to the specific application.

The parametric development approach is applied to the design of arbitrarily large combinational multipliers. In general, the multipliers are considered as a class of circuits that is difficult to design using the evolutionary algorithms. Therefore a higher-level abstraction has been chosen in order to successfully develop generic multipliers using the proposed representation. It is demonstrated that the evolution is able to discover algorithms for the development of arbitrarily large multipliers. Although the resulting solutions exhibit no innovation in comparison with the conventional design, this method represents the first case if the design of generic combinational multipliers using the evolutionary algorithm combined with a developmental encoding.

1.9 The book structure

This book is divided into eight chapters describing the theoretical background, the developmental method invented during the research, case studies together with experimental results and concluding remarks. The content of the chapters is as follows.

Chapter 2 contains the description of the relevant evolutionary techniques related to the experiments presented in this book. In particular, genetic algorithm is introduced in Section 2.3 which is utilized herein in order to design the developmental programs for the construction of combinational circuits. Genetic programming as the fundamental evolutionary algorithm this work is inspired by in summarized in Section 2.4. A special kind of the genetic

programming — the linear genetic programming — is also mentioned, representing the basis for the proposed developmental system for the design of generic combinational circuits.

The basic principles of the development are summarized in Chapter 3. Section 3.1 briefly introduces the crucial aspects of the biological development that represents the main inspiration for the techniques of the computational development. The most important models related to the computational development are described in Section 3.2. Several works have dealt with the research of the developmental encodings and their applications. A brief survey of the relevant books, PhD theses and important papers is stated in Section 3.2.
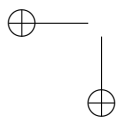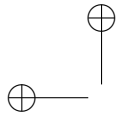
Chapter 4 introduces the principles and features of the application-specific instruction-based developmental method which represents the core topic of this book and its approaches are applied in the description of the case studies.

Chapter 5 describes in detail the first extensive case-study applying the instruction-based development. It is devoted to the evolutionary design of developmental programs for the construction of generic sorting networks using the concept of continual development. Experiments were conducted with the evolution of arbitrarily large sorting networks and then considering only odd-input and even-input networks. The results are presented in Section 5.3. An advanced approach to the continual development of generic sorting networks is introduced in Section 5.5. In the category of even-input sorting networks (see Section 5.4.3) an innovative solution was discovered in comparison with the conventional approach. Finally, a theoretical analysis has been performed together with the proof of generality of the best solution in Section 5.6.

The second case-study, which is presented in Chapter 6, deals with the evolutionary design of generic combinational multipliers using the parametric development. The initial concept of developmental encoding for the design of generic multipliers is introduced in Section 6.2. It represents the first case when generic combinational multipliers were evolved using the development. An advanced approach, focused to design of more effective carry-save multipliers, is presented in Section 6.3 (again, the concept of parametric development is employed).

Chapter 7 presents the continual development of polymorphic circuits. It is the first case when polymorphic circuits were evolved using the developmental encoding. The research if focused on the design of polymorphic parity circuits ans polymorphic median and sorting networks. The results are summarized in Section 7.4.

Chapter 8 summarizes the results and consequences of this work and outlines the potential directions for the future research in the area of instruction-based developmental systems.

2
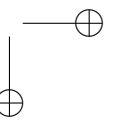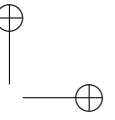
Evolutionary algorithms

This chapter summarizes the basic principles of the evolutionary algorithms (EAs) and focuses in more detail on evolutionary techniques that are relevant for the experiments presented in chapters 5 and 6.

2.1 Concept of evolutionary algorithms

Evolutionary algorithms represent a set of biologically inspired, stochastic search algorithms based on a model of natural, biological evolution, which was formulated for the first time by Charles Darwin. In contrast to the traditional search algorithms, evolutionary algorithms operate over a population of candidate solutions rather than over a single solution. The candidate solutions (also called phenotypes) are encoded into the individuals (also called genotypes, genomes or chromosomes), which constitute a problem representation in the EA.

Every new population is created throughout reproduction. The process of reproduction performs selection of chromosomes (i.e. the encoded form of the candidate solutions) according to their ability to solve the given problem. This ability is called a fitness measure which is calculated in a fitness function. The selected chromosomes are modified by means of so-called genetic operators (sometimes also called variation operators) that are inspired by the similar processes known from nature. Note that the utilization of specific genetic operators and the selection operator depend on the type of the evolutionary algorithm and the problem to be solved. While the genetic operators work over the chromosomes (genotypes), the fitness function evaluates the candidate solutions (phenotypes).

The mapping between genotypes and phenotypes (also called growth function) may be direct or indirect (however, some EAs do not distinguish between genotypes and phenotypes at all). Note that this book deals with the indirect mappings, i.e. the development (see Chapter 3). The concept of the genotype–phenotype mapping and the calculation of the fitness function in the EA is

14   2 Evolutionary algorithms

shown in Figure 2.2. Since the selection operator prefers fitter genomes over the less fitter genomes, there is a selection pressure pushing the evolution of the population to better solutions. The selection pressure implies that the fitter individuals live for a longer while and generate offspring, which inherit their genetic information. The structure of a general evolutionary algorithm is shown in Figure 2.1.

```
set time t = 0
create initial population P(t)
evaluate each individual in P(t)
WHILE acceptable solution not found DO
    t = t + 1
    create P(t) using P(t - 1)
    evaluate P(t)
END WHILE
```

Fig. 2.1. The structure of a general evolutionary algorithm

representation space   solution space   space of fitness values
(genotypes)   (phenotypes)
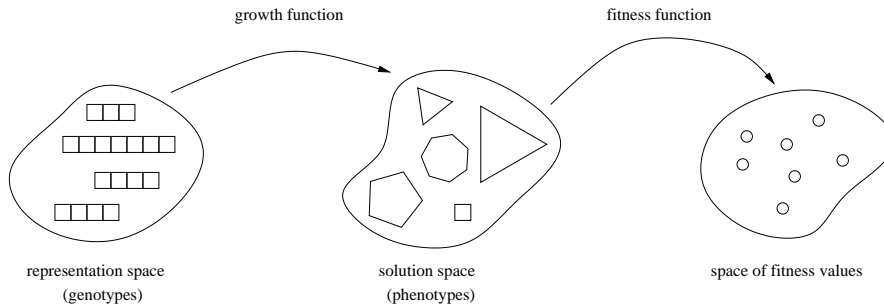
Fig. 2.2. Illustration of the genotype–phenotype mapping and the fitness computation in the evolutionary algorithm. The representation space consists of the encoded forms of the candidate solutions contained in the solution space. The genetic operators are applied on the chromosomes, the chromosomes are translated onto the solutions according to a growth function and the fitness function these solutions.

There are many books providing a survey of different evolutionary techniques covering theoretical as well as practical issues, for example [ES03, P. 99]. Moreover, original literature focuses on the specific variants of evolutionary algorithms. Holland's genetic algorithms [Hol75], Koza's genetic programming [J. 92], Fogel's evolutionary programming [FOW66] and evolutionary strategies [Sch65, Rec73] developed by Bienert, Rechenberg and Schwefel probably represent the most known and widely used variants of evolutionary algorithms.

Although, the operation and representation of the solutions differ in distinct variants of evolutionary algorithms, these algorithms have one in common: inspiration in nature, in particular, evolutionary biology.

The next section contains a brief overview of the natural background of the EAs, considering the fundamental principles stated in [Bä96]. For detailed information see also [Dar06].

2.2 Biological background

Charles Darwin's theory of evolution explains the adaptive change of species by the principle of natural selection, which favors those species for survival and further evolution that are best adapted to their environmental conditions. In addition to selection, another important factor for evolution recognized by Darwin is the occurrence of small, apparently random and undirected variations between the phenotypes, i.e., the manner of response and physical embodiment of parents and their offspring. These mutations prevail through selection, if they prove their worth in light of the current environment; otherwise, they perish. The basic driving force for selection is given by the natural phenomenon of production of offspring. Under advantageous environmental conditions, population size grows exponentially, a process which is generally limited by finite resources. When resources are no longer sufficient to support all the individuals of a population, those organisms are at a selective advantage which exploit resources most effectively.

This point of view is presently generally accepted as the correct macroscopic explanation of evolution. However, modern biochemistry and genetics has extended the Darwinian theory by microscopic findings concerning the mechanisms of heredity. The resulting theory is called synthetic theory of evolution or, sometimes, neodarwinism.

This theory is based on genes as transfer units of heredity and may be described in simplified form as follows. Genes are occasionally changed by mutations. An individual represents the unit of selection. Selection acts on the individuals, which expresses in its phenotype the complex interactions within its genotype, i.e. its total genetic information, as well as the interaction of the genotype with the environment in determining the phenotype. The evolving unit is the population which consists of a common gene pool included in the genotypes of the individuals.

In the evolutionary framework, the fitness of an individual is measured only indirectly by its growth rate in comparison to others. That means the fitness represents the prosperity of the individual in a particular environment. Furthermore, natural selection is no active driving force, but differential survival and reproduction within a population makes up selection. Selection is simply a name for the ability of those individuals that have outlasted the struggle for existence to bring their genetic information to the next generation. This point of view, however, reflects just our missing knowledge about the mapping from

genotype to phenotype, a mapping which — if it were known — would allow us to evaluate fitness in terms of a variety of physical properties. In Section 3.1, the basic principles will be briefly outlined which are presently known about this mapping, representing the crucial part of inspiration for this work, i.e. the biological development.

2.3 Genetic algorithms

The original principle of a genetic algorithm (GA) was introduced by Holland, who proposed a model for studying adaptation in natural and artificial systems [Hol75]. Later, genetic algorithms were popularized by Goldberg [Gol89]. Since that time, this biologically inspired technique underwent a significant development and many different variants of GA emerged. Therefore, there is no single definitive genetic algorithm; rather algorithms have been created from a suite of representations, selection and variation operators to suit our particular applications [ES03].

2.3.1 Simple genetic algorithm

In this section, basic variant of genetic algorithm will be introduced that is commonly referred to as simple GA or canonical GA [ES03, P. 99].

The genetic algorithms are perhaps the most well known variant of evolutionary algorithms. GAs belong to the class of algorithms that explicitly distinguish the search space (i.e. the space of genotypes) and solution space (i.e. the space of phenotypes) – see Figure 2.2. GA's maintain a population of individuals consisting of the genotypes, each of which corresponds to a phenotype from the solution space. The genotypes, which are usually of constant length[1], consist of encoded version of phenotype parameters and are referred to as genes. A value of a gene is called an allele. The collection of genes in one genotype represents a genome (chromosome).

The simple genetic algorithm, whose structure is shown in Figure 2.3, works as follows [P. 99]. The genotype of every individual in the population is initialized with random alleles. The main loop of the algorithm then begins, with the corresponding phenotype of every individual in the population being evaluated and given a fitness value according to how well it fulfils the problem objective or fitness function. These scores are then used to determine how many copies of each individual are placed into a temporary area often termed the "mating pool" (i.e. the higher the fitness, the more copies that are made of an individual). The simple GA uses two genetic operators: crossover (sometime termed as recombination) and mutation.

Two parents are randomly picked from the "mating pool". Offspring are generated by the use of the crossover operator, which randomly allocates genes

---

[1] Advanced versions of GA may utilize variable-length chromosomes.

```
set time t = 0
create initial population P(t)
evaluate each individual in P(t)
WHILE acceptable solution not found DO
    reproduce individuals according to their fitnesses into
    a ''mating pool'' (higher fitness implies more copies of
    an individual in the ''mating pool'')
    t = t + 1
    WHILE P(t) is not filled with new offspring DO
        randomly take two individuals from ''mating pool''
        use probabilistic random crossover to generate two offspring
        apply probabilistic random mutation to the offspring
        place offspring into population P(t)
    END WHILE
    evaluate each individual in P(t)
END WHILE
```

Fig. 2.3. The structure of the simple genetic algorithm

from each parent's genotype to each offspring's genotype. For example, consider two parent chromosomes 'ABCDEF' and 'abcdef'. Consider that a cross point 2 was selected randomly, i.e. the chromosomes will be crossed at the position behind the second gene of each of them. Then two following offspring will be generated by the simple GA: 'ABcdef' and 'abCDEF'. The crossover represents a probabilistic genetic operator, which means that the operation is performed only with a given probability, otherwise the parents are just copied to the next generation. After recombination, mutation operator is occasionally applied (usually with much lower probability than crossover) to the offspring. When it is used to mutate an individual, typically a single gene is selected and its allele is changed randomly. For example, if the fourth gene of a chromosome '110010' ought to be mutated, then the form of the mutated genome will be '110110'.

Using crossover and mutation, offspring are generated until they fill the new population (all parents are discarded). This entire process of evaluation and reproduction then continues until a satisfactory solution emerges or the GA has run for a specified number of generations.

2.3.2 Advanced genetic algorithms

The simple GA is especially favoured for those that try to theoretically analyze and predict the behavior of genetic algorithms rather than for the utilization in practical applications. In reality, typically more advanced GAs are applied. In addition to the common features of the advanced techniques, including for example more realistic natural selection, more genetic operators, ability to detect when evolution ceases, overlapping populations, elitism (where some

fit individuals can survive for more than one generation) etc., some advanced variants of GAs have been utilized [P. 99].

- Parallel GAs – multiple processors are used to run the GA [AC94].
- Distributed GAs – multiple populations are separately evolved with few interactions between them [WS90].
- Messy GAs – a number of 'exotic' techniques are utilized, such as variable-length chromosomes, cut and splice operators in place of fixed-length crossover operations, two-phase evolutionary process (primordial phase and juxtapositional phase), (sometime) competitive templates to emphasize salient building blocks [DG91].
- Hybrid GAs – GAs are combined with local search algorithms, e.g. see [Geo94].
- Structured GAs – allows parts of chromosomes to be switched on and off using evolvable 'control genes' [DM92].
- Steady-state GAs – the entire population is not changed at once, but rather a part of it. In each generation, only a subset of the population is replaced by the offspring created by the genetic operators. Steady-state GA was introduced by Whitley in [WK88].

2.4 Genetic programming

Genetic programming actually represents a modified variant of the genetic algorithm. In particular it differs from the genetic algorithm in the representation of individuals and in the genetic operators. Genetic programming in its original form belongs to the class of EAs, which does not distinguish between genotypes and phenotypes. The individuals (and also solutions to be evolved) represent general computer programs that are evaluated in solving a given problem. In 1990, Koza introduced the concept of genetic programming [Koz90], which represents one of the most known and widely used evolutionary algorithm nowadays. In the process of the GP research, more variants of genetic programming representations have emerged. In addition to the original tree-based representation of computer programs introduced by Koza, there are also graph-based representations (e.g. Miller's cartesian genetic programming [MT00]) or linear representation [BB07]. In this section, the principles of the standard GP will be mentioned, which provided an inspiration for the other GP representations, and the linear GP, which is relevant to this work.

2.4.1 Standard genetic programming

In standard genetic programming [J. 92], the structures undergoing adaptation (i.e. the chromosomes to be evolved like in the genetic algorithms) are general, hierarchical computer programs of dynamically varying size and shape. The programs are composed of functions and terminals appropriate

to the problem domain. The functions may be standard arithmetic operations, standard programming operations, standard mathematical functions, logical functions, or domain-specific functions. Depending on the particular problem, the programs may be Boolean-valued, integer-valued, real-valued, complex-valued, vector-valued, symbolic-valued or multiple-valued. The initial population is usually generated randomly.

Each individual computer program in the population is evaluated by mens of the fitness function in terms of how well it performs in the particular problem environment, i.e. how well it can solve the particular task.

For example, in a problem involving finding the strategy for playing a game, the fitness measure would be the score (payoff) received by a player in the game (i.e. the ability of the program being evolved to play the game with the score to be optimized). For many problems, fitness is naturally measured by the error produced by the computer program.

Typically, each computer program in the population is run over a number of different fitness cases so that its fitness is measured as a sum or an average over a variety of representative different situations. These fitness cases sometimes represent a sampling of different values of an independent variable or a sampling of different initial conditions of a system. For example, the fitness of an individual computer program in the population may be measured in terms of the sum of the absolute values of differences between the output produced by the program and the correct answer to the problem. This sum may be taken over a sampling of 100 different inputs to the program, which may be chosen at random or may be structured in some way.

Primary genetic operators

The Darwinian principle of reproduction and survival of the fittest and the genetic operation of recombination (crossover) represent the primary operators utilized in genetic programming. These genetic operators are used to create a new offspring population of individual computer programs from the current population of programs.

The reproduction operation is the basic engine of Darwinian natural selection and survival of the fittest. The operation of reproduction consists of two steps. First, a single individual computer program is selected from the population according to some selection method based on fitness. Second, the selected individual is copied, without alteration, from the current population into the new population (i.e. the new generation).

The crossover operator creates variation in the population by producing new offspring that consist of parts taken from each parent. The crossover operation starts with two parental individuals selected by means of a selection operation and produces two offspring individuals (programs). The parental programs are typically of different sizes and shapes. The operation begins by independently selecting, using a uniform probability distribution, one random point in each parent to be the crossover point for that parent. The crossover

fragment for a particular parent is the rotated subtree which has as its root
the crossover point and which consists of the entire subtree lying below the
crossover point (i.e. more distant from the root of the original tree). The
first offspring individual (computer program) is produced by replacing the
crossover fragment of the first parent by the crossover fragment of the second
parent and the second one is created by replacing the crossover fragment of the
second parent by the crossover fragment of the first parent (i.e. the crossover
fragments of the parental individuals are simply swapped in the offspring). For
example, consider the two parental expressions in Figure 2.4a. The functions
appearing in these expressions are the Boolean AND, OR and NOT functions
the terminals are the Boolean arguments D0 and D1. Assume that the points
of both trees are numbered in a depth-first, left-to-right way. Suppose that
the second point of the first parent (the NOT function) is randomly selected
as the crossover point. Suppose also that the sixth point of the second par-
ent (the AND function) is selected as the crossover point. The portions of
the two parental expressions marked by dashed lines in Figure 2.4a are the
crossover fragments. The crossover operation is performed by swapping the
crossover fragments between the two parental individuals in order to produce
two offspring individuals. The resulting offspring are depicted in Figure 2.4b.
The offspring programs are typically of different sizes and shapes than their
parents.
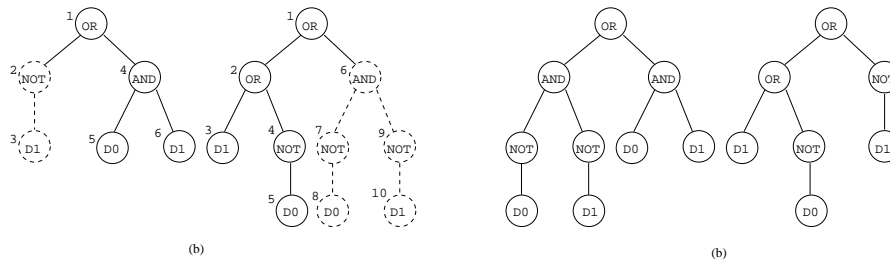
Fig. 2.4. The operation of crossover in genetic programming

Secondary genetic operators

In addition to the two primary genetic operations of reproduction and
crossover in genetic programming, there are five optional secondary opera-
tions, for example [J. 92]: mutation, permutation, editing, or encapsulation.

The mutation operation begins by selecting a point at random within the
individual computer program. The mutation point can be an internal (i.e.
function) or an external (i.e. terminal) point of the tree. The mutation oper-
ation then removes whatever is currently at the selected point and whatever
is below the selected point and inserts a randomly generated fragment of
computer program (i.e. subtree) to that point.

The permutation operator selects a function (internal) point of the tree at random. If the function at the selected point has k arguments, a permutation is selected at random from the set of $k!$ possible permutations. Then the arguments of the function at the selected point are permuted in accordance with the random permutation.

The editing operation provides a means to edit and simplify the individual programs. This operation recursively applies a pre-established set of domain-independent and domain-specific editing rules to each individual in the population. The universal domain-independent editing rule is the following: If any function that has no side effect and is not context dependent has only constant atoms as arguments, the editing operation will evaluate that function and replace it with the value obtained from the evaluation. For example, the fragment $1+2$ will be replaced with 3 and the fragment of Boolean expression $1AND1$ will be replaced with 1. In addition, the editing operation applies a pre-established set of domain-specific editing rules. For example, for numeric problem domains, there might be an editing rule that inserts 0 whenever a subexpression is subtracted from itself. In Boolean domain, for instance, one might use an editing rule to apply one of De Morgan's laws to an expression or other expression-simplifying rules.

The encapsulation is a means for automatically identifying a potentially useful subtree and giving it a name so that it can be references and used later. The operation of encapsulation begins by selecting a function (internal) point at random. A new function is defined corresponding to the fragment represented by the subtree at the selected point. Then the subtree is removed from the original tree and the reference to the newly defined function in inserted at that point.

Algorithm of genetic programming

The genetic programming starts with an initial population of randomly generated computer programs. In each generation, the primary operations of reproduction and crossover are applied to the selected individuals. Occasionally, secondary genetic operators may be applied to the offspring generated by the primary genetic operations. After this process, the population of offspring (i.e. the new generation) replaces the old population (i.e. the old generation). Each individual in the new population of computer programs is then measured for fitness, and the process is repeated over many generations. The basic algorithm of the genetic programming is in fact identical with the genetic algorithm shown in Figure 2.3 without mutation [J. 92].

2.4.2 Linear genetic programming

Linear genetic programming (LGP) represents another variant of encoding of the computer programs in the genetic programming. This section summarizes the basic features of LGP, gathering from [BB07].

The tree programs used in Koza's standard GP correspond to expressions (syntax trees) from a functional programming language. This approach is also referred to as tree-based genetic programming (TGP). Functions are located at inner nodes, while leafs of the tree hold input values and constants. In contrast, linear genetic programming is a GP variant that evolves sequences of instructions from an imperative programming language or from a machine language.

The term linear refers to the structure of the (imperative) program representation. It does not stand for functional genetic programs that are restricted to a linear list of nodes only. Moreover, it does not mean that the methods itself is linear, i.e. may solve linearly separable problems only. In many cases, genetic programs represent highly non-linear solutions due to their inherent power of expression.

The use of linear bit sequences in GP goes back to Cramer and his JB language [Cra85]. A more general approach was introduced in [Ban93]. Nordin's work [Nor94] of subjecting machine code to evolution was the first GP approach that operated directly on an imperative representation. It was subsequently expanded and developed into the AIMGP (Automatic Induction of Machine Code by Genetic Programming) approach [Nor97, BNKF97]. In AIMGP individuals are manipulated as binary machine code in memory and are executed directly without passing an interpreter during the fitness calculation, which poses a significant speedup compared to interpreting systems. In [Nor97] machine code GP and the applications of this linear GP approach to different problem domains have been studied.

There are two basic differences between a linear program and a tree program [BB07]:

(1) A linear genetic program can be seen as a data flow graph produced by multiple usage of register content. That is, on the functional level the evolved imperative structure represents a special directed graph. In traditional tree-based GP the data flow is more rigidly determined by the tree structure of the program.

The higher variability of linear program graphs allows the result of subprograms (subgraphs) to be reused multiple times during calculation. This permits linear solutions to be more compact in size than tree solutions and to express more complex calculations with less instructions.

(2) Special noneffective code segments coexist with effective code in linear genetic programs. They result from the imperative program structure — not from program execution — and can be detected efficiently and completely. Such structurally noneffective code manipulates registers not having an impact on the program output at the current position. It is thus not connected to the data flow generated by the effective code. In a tree program, by definition, all program components are connected to the root. As a result, the existence of noneffective code necessarily depends on program semantics.

Noneffective code in genetic programs is also referred to as introns. In general, it denotes instructions without any influence in the program behavior.

In standard genetic programming, the noneffective code causes a "bloat" (expansion in size) of the evolved programs. In linear representation, it is easy to avoid this problem by limiting the program length (i.e. specifying a maximal number of instruction) which, in effect, leads to the constant-length chromosomes. In fact, the amount of introns in the evolving program can vary the effective length of the code (which will never be larger than the specified limit) and hence more effective algorithms can potentially be evolved. Note that explicit introns may be specified by means of a "no operation" (NOP) instruction. Moreover, the noneffective code is considered to be beneficial during evolution for two major reasons. First, it may act as a protection that reduces the effect of variation on the effective code. Second, noneffective code allows variations to remain neutral in terms of fitness change. In linear programs introns may be created easily at each position with almost the same probability.

According to the above notion, there is a distinction between an absolute program and an effective program in linear GP. While the former includes all instructions, the latter contains only the structurally effective instructions. The effective length of a program is measured in the number of effective instructions it holds. Each program position or line is supposed to hold exactly one instruction.

In [BB07], the general concept of LGP is described in detail, including coding of instructions, instruction set, branching, iteration and modularization concepts, execution of programs and the issues related to the evolution of programs in the linear representation, i.e. initialization, reproduction and variation operators. However, only the fundamental concept — the linear representation of the genetic programs — is utilized in its original form with respect to the specific requirements of the domains in which the experiments were conducted. Therefore, the features and setup of the application-specific evolutionary design systems applied in this work will be provided in chapters 5 and 6, where experimental results are presented.

2.4.3 Developmental genetic programming

Koza introduced a variant of genetic programming — developmental genetic programming (DGP) — for the construction of topology and sizing of electrical circuits (see [J. 03] and Part V of [J. 99]). For this purpose, program architecture operations and architecture-altering operations were introduced. The circuit is developed progressively from an embryo (usually supplied to the system by the designer) according to the program being evolved in genetic programming. The operations of the genetic programs are interpreted as a prescription for manipulating the parameters and structure of the embryo (e.g. electric values of the components, adding new components into the circuit, modifying their interconnection etc.).

24   2 Evolutionary algorithms

Similar principle was utilized to the automatic design of the geometry and sizing of antennas [J. 03]. In this case the principle of a flying turtle was utilized that is known from LOGO programming language.

2.4.4 Summary of genetic programming

As evident from the sections 2.4.1, 2.4.2 and 2.4.3, genetic programming provides a general concept and a wide range of representations for the evolution of computer programs in various programming languages. Although genetic programming in its original form represents a direct mapping that does not distinguish between genotypes and phenotypes during evolution, it can be enhanced easily into the developmental form as demonstrated in [J. 03]. Moreover, application-specific changes may be introduced in order to utilize the GP concept in different domains.

In this book, a developmental approach based on the principles of LGP for the design of generic structures of digital circuits will be introduced that has been called instruction-based development. Experiments will be presented from the domains of the evolutionary design of arbitrarily large sorting and median networks and combinational multipliers.

3

Development

The issue of development represents the crucial part of this work. Some important general features of development was introduced in Chapter 1. This chapter summarizes the fundamental principles of biological and computational development, proposes a survey of the fundamental developmental models and an overview of the recent work in this area.

3.1 Biological background

The basic principles of the computational development are inspired by the natural (biological) development. Therefore, it is important to summarize the fundamental principles of the biological development before the artificial developmental models will be introduced. Since the field of natural development is very complex, only a brief introduction to its principles will be stated.

Biological development concerns the life cycle of multicellular organisms. DNA (deoxyribonucleic acid) acts as a long-term information storage of how to develop the organism. DNA encodes genes according to which proteins are synthesized through the process of transcription and translation. Proteins may specify the behavior of cells. This process has been understood as the "central dogma" of biology [Coh07].

The following subsections summarize the basic principles of functioning the development, based on that "central dogma". Detailed information of this field from the both biological and computational point of view can be found, for example, in [Wol07, S. 03, ZB08].

3.1.1 DNA

DNA forms the genetic material for almost all living organisms (with the exception of some viruses that replace DNA with RNA (ribonucleic acid) as the genetic material) [Lew99].

Nucleic acids consist of polynucleotide chains. They contain four types of bases, two purines: adenine (A) and guanine (G), and two pyrimidines: cytosine (C) and thymine (T). In RNA, thymine is replaced by uracil (U). The DNA molecule is built up as follows: the bases extend inwards from each nucleotide chain, with purines pairing with pyrimidines, resulting in the following base pairings: guanine pairs with cytosine (G-C) and adenine pairs with thymine (A-T) or (A-U in RNA). These base pairings are termed complementary. A consequence of complementary base pairing is that one strand of DNA or RNA can act as a template for the synthesis of a complementary strand. Nucleic acids are therefore uniquely capable of directing their own self-replication.

The sequence of nucleotides in DNA is important as it codes for amino acids that constitute the building blocks of the corresponding polypeptide or protein. Each amino acid is represented by a sequence of three nucleotides. The nucleotide triplet is termed as a codon and the rules according to which the codons are translated to amino acids are referred to as genetic code. This code is unambiguous, i.e. one codon does not specify more than one amino acid. However, each amino acid can be specified by more different codons. Considering the four bases in DNA (or RNA) strand, there are sixty-four codons. Sixty-one of them define amino acids (although only 20 different amino acids exist). The remaining three represent stop signals for protein synthesis.

3.1.2 Genes and proteins

A gene represents a region of DNA (or RNA in some viruses) that codes for one or more molecular products (mRNA or protein) [ZB08]. Genes control development mainly by specifying which proteins are made in which cells and when. In this sense genes are passive participants in development, compared with the proteins for which they code.

A protein is a sequence of amino acids that directly determine cell behavior, including which genes are expressed. To produce a particular protein its gene must be switched on (expressed) and transcribed into messenger RNA that is then translated into protein [Wol07].

3.1.3 Proteins synthesis

The sequence of bases in the DNA (representing a gene) specifies the sequence of amino acids in a protein chain. However, DNA does not directly control protein synthesis. It occurs in the cell cytoplasm under the control of RNA synthesized from the DNA template in a process called transcription. Ribosomes, macromolecular structures in the cells, involve RNA produced by transcription in order to transfer it into protein in the process termed as translation.

Transcription

After a signal to switch on a gene is received, a single-stranded RNA copy of the gene is first made in a process called transcription. This RNA synthesis is catalyzed by enzymes called RNA polymerases. Only the relevant region of DNA is transcribed into RNA, so while the DNA carries information about many proteins, the RNA carries information from just one part of the DNA, usually information for a single protein [ZB08]. When a gene is being transcribed into RNA, which is in turn directing protein synthesis, the gene is said to be expressed.

Translation

There are three main classes of RNA in all cells: messenger RNA (mRNA – it is the RNA produced by transcription), ribosomal RNA (rRNA) and transfer RNA (tRNA) as well as numerous smaller RNAs with a variety of roles [ZB08]. rRNA and tRNA are involved in the process of mRNA translation and protein synthesis.

The mRNA is translated into protein by ribosomes, multimolecular complexes formed of rRNA and ribosomal proteins. Amino acids do not recognize the codons in mRNA directly and their addition in the correct order to a new protein chain is mediated by the tRNA molecules, which transfer the amino acid to the growing protein chain when bound to the ribosome. The tRNA molecules have a three-base anticodon at one end that recognizes a codon in mRNA, and at the other end a site to which the corresponding amino acid becomes attached by a specific enzyme. This process results in the synthesis of a protein chain.

3.1.4 Cells

Cells are the atomic constructional units of organisms. They come in two specific classes: prokaryotic (bacteria and blue-green algae) and eukaryotic (plants and animals). The DNA in a prokaryotic cell is not encased within a nuclear envelope. Eukaryotic cells enclose their DNA within a membrane, conferring an additional opportunity to control gene regulation (they also contain organeless such as the chloroplast and mitochondria).

Generally, in the terminology of computer science, cells can be likened to autonomous agents in that they have:

- sensors (in the form of protein-based receptors that bind signals, or ion-channels that permit signals through the cell membrane that receive information from the environment),
- internal logic that integrates this information (the genome) and
- effectors (synthesized proteins) able to perturb the environment.

28   3 Development

As evident, cells are fundamentally protein-processing machines, sensing protein signals, being controlled by proteins and outputting new proteins for other cells to sense.

Cell membrane

The cell takes great paints to separate itself from its immediate environment. That is to say the cell has a very well defined boundary – the cell membrane. The immediate purpose of the membrane is to prevent proteins from seeping away. At the same time, however, proteins need to be able to enter and leave the cell in order to affect it. This passage of proteins is not random. Cell exercise specificity: they can select which proteins enter and leave them. To this end, the cell membrane is semipermeable (or selectively permeable), allowing only certain protein molecules through.

Cell signaling

Groups of cells can influence the development of another group of cells by emitting signals. This process is termed induction. Inductive signals provide instructions to cells on how to behave. In essence, the inductive signal 'selects' a single cellular response from an already limited number of responses [Wol07].

Cell signaling enables cells to detect and respond to conditions within the extracellular environment. In the case of multicellular organisms, cells need to be able to communicate over short and long distances in order to shape the developing organism.

Cell division

Cells multiply by duplicating their content and then dividing into two cells – parent and daughter. This forms a cell division cycle, which is a complicated process that consists of a number of stages [AJL$^+$94]: interphase (DNA replicates and proteins are synthesized before and after mitosis), mitosis (nuclear division, which itself consists of several stages) and cytokinesis (the division of the cytoplasm of a cell following the division of the nucleus).

Cell division is a crucial aspect of development. Two types of cell division occur: symmetric and asymmetric. The symmetry or asymmetry is in relation to cytoplasmic factors sequestered within the cell. Symmetric division occurs when the plane of cleavage divides the cell into equal sizes with equal proportions of cytoplasmic proteins. Asymmetric division occurs when the plane of cleavage divides the cell into unequal sizes with daughter and parent cell containing different cytoplasmic factors. The orientation or direction of the cell division is not random, it is controlled and directed. However, exactly what determines the direction in which a cell is to divide is still under investigation in cell biology [S. 03].

3.1.5 Developmental processes

Central to development is construction and self-organization. During the developmental process, a single cell (the zygote) "grows up" — develops — into a complex multicellular organism.

Five developmental phases can be observed during the cellular development: cleavage division, pattern formation, morphogenesis, cell differentiation and growth. These processes do not necessarily occur sequentially, but overlap. Let us now briefly describe each of them.

Cleavage division

Cleavage division involves the zygote (fertilized cell) undergoing a series of rapid divisions to create more cells. Unlike cellular proliferation where cells grow after dividing, during cleavage division there is no increase in cellular mass between each division. The result of this process is a hollow ball of cells, known as blastula.

Pattern formation

Pattern formation is the process by which 'a spatial and temporal pattern of cell activities is organized within the embryo so that a well-ordered structure develops' [Wol07]. Pattern formation comprises two main stages: (1) the process by which the initial body plan is laid down and (2) the allocation of cells to different germ layers (primary cell layers in an embryo). The first stage results in the setting up of a coordinate system, which is achieved through two axes defining the anterior and posterior ends and the dorsal and ventral sides of the body (both axes are at right angle to each other). The second stage of pattern formation is also responsible for creating the different germ layers, namely the ectoderm (external layer of cells), mesoderm (middle layer) and endoderm (inner layer of cells).

Morphogenesis

Morphogenesis involves incredible change in the three-dimensional form of the developing embryo as a result of cell movement and conformational changes that generate forces. Extensive cell migration (movement of cells) can also occur. The most dramatic change during morphogenesis is gastrulation [Wol07]. Typically, all animal embryos undergo the dramatic changes of the gastrulation process after which a gastrula is created from the blastula.

Cell differentiation

Cell differentiation is a gradual process by which cells acquire different structure and function from one another, resulting in the emergence of distinct

30   3 Development

cell types, for example, neurons or skin cells. Differentiation is fundamentally about the different proteins cells contain. If a cell has become terminally differentiated, it continues to produce these proteins due to a change in a gene expression that causes a stable pattern of gene activity, else the cell may continue differentiating over successive cell divisions. Therefore, differentiation is influenced by at least the following two processes.

- Cell signaling – a mechanism by means of which intercellular communication is performed.
- Asymmetric division – division that results in the asymmetric apportioning of factors (proteins) in the parent cell, causing parent and daughter cell to acquire different developmental fates. It also acts as a symmetry-breaking mechanism [Tur52, Wol07].

Growth

The final process, growth, involves an increase in size due to one of a number of methods: cell proliferation, in which cells multiply; a general increase in cell size and the accretion of extracellular materials, such as bone.

3.2 Models in computational development area

Computational development is a recently established field in which many artificial models have been devised and investigated. The applications of these models range from testing hypotheses and predicting new issues in biology to exploring artificial developmental systems in order to investigate the properties of specific developmental models or constructing complex adaptive systems in computer science. A wide diversity of topics has been covered in such research. This section provides a brief survey of the most significant developmental models and their applications.

3.2.1 Lindenmayer systems

Lindenmayer systems (or L-systems) are parallel rewriting systems (grammars) which were originally introduced by Aristid Lindenmayer to model the development of multicellular organisms [Lin68]. L-systems work on a finite set of symbols called an alphabet, where each symbol represents a cell. The initial organism is represented by a finite string of symbols that is called axiom. Each cell of that string may be rewritten according to a rewriting rule from a finite set of rules. Since the development of organisms in nature happens in parallel, L-systems are inherently parallel rewriting systems, i.e. in one developmental step each cell of the organism is rewritten.

The advantage of L-systems is the simple form of the developing objects consisting of simple symbols. Moreover, the developmental process performed

by L-systems may be adapted to specific requirements of a given application. For example, parametric L-systems, stochastic L-systems or environmentally-sensitive L-systems exist [RA97]. Considering these features, theoretically arbitrary problem may be encoded using the strings of symbols that can be generated by L-systems. If the number of rewriting steps is not limited, it is possible to develop the strings of arbitrary length which may potentially enable L-systems to perform generic design by means of the rewriting process. However, the selection of symbols and their interpretation to solve a specific problem as well as the design of a suitable rewriting system represent a difficult task.

L-systems fundamentally represent a mathematical model to investigate the development of multicellular organisms in biology (the first experiments conducted by Lindenmayer were focused to plant development). Later, L-systems became a popular parallel grammar concept in the area of formal languages. In addition, typical applications include the utilization of L-systems in the field of computer graphics. The strings generated by L-systems are interpreted as geometric "commands" to draw graphical primitives. This of drawing images is known as turtle graphics. For example, L-systems have been utilized to produce biologically realistic images [PL90]. Fractals represent another well-known domain of this kind of rewriting systems.

Koch's curve represents one of the most known fractal that is possible to generate by L-systems. It is a case of a mathematical curve described for the first time by the Swedish mathematician Helge von Koch in 1904 [vK93]. Figure 3.1 shows the development of Koch's curve together with the rewriting rules of the L-system applied. The L-system works with the alphabet containing symbols F, $+$ and $-$ whose interpretation is as follows:

- F: draw a line,
- $+$: change the direction of drawing the next line by 60° counter-clockwise,
- $-$: change the direction of drawing the next line by 60° clockwise.

The axion consists of a single symbol F that in graphical interpretation represents a single line denoted by (0) in Figure 3.1a. After the first developmental step, the axion is rewritten according to the rule (1) from Figure 3.1b to the string $F + F - -F + F$. The graphical interpretation corresponds to four segments I.-IV. (drawn respectively by the appropriate Fs of the string) shown in the instance denoted by (1) of Figure 3.1a. During the second step, all Fs are rewritten in parallel according to the rule (1) and the symbol $+$, respective $-$, is rewritten to itself according to the rule (2), respective (3), from Figure 3.1b. The development continues in the same way. The graphical interpretations of the developed strings after the second and third step are represented by the instances denoted (2) and (3) in Figure 3.1a.

$$(1)\ F \to F + F - -F + F$$

$$(2)\ + \to +$$

$$(3)\ - \to -$$

(a)   (b)
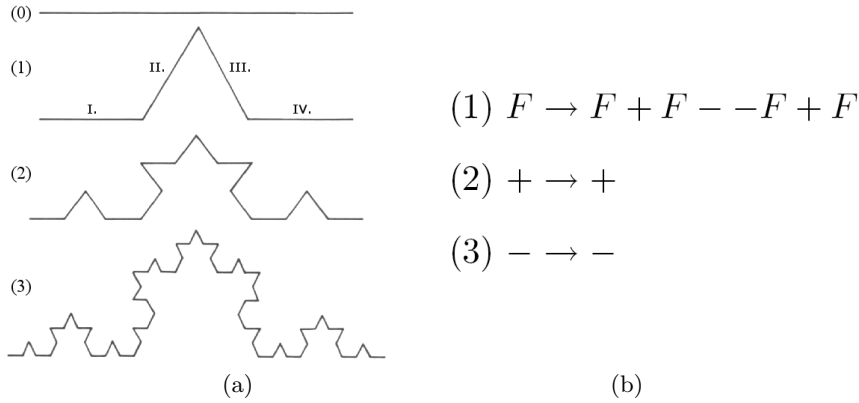
Fig. 3.1. (a) Three steps of development of Koch's curve, (b) Rewriting rules of the L-system applied for the development

3.2.2 Cellular automata

The concept of cellular automata (CA) was originally invented by Ulam and von Neumann as a mathematical model for investigating the behavior of complex systems and self-replication [vN66]. CAs are discrete dynamical systems composed of a regular array of cells. Each cell may be in a state from a finite set of states. The form of the cellular array is specified by the dimension of the CA. In most cases, one-dimensional (1D) and two-dimensional (2D) automata are considered. CAs of higher dimensions are also possible.

States of the cells are updated synchronously in discrete time steps according to a local transition function. The local transition function determines the next state of a cell in dependence on the combination of states of cells included into a cellular neighborhood (for example, typical cellular neighborhood of a cell in a 1D CA consists of the cell and its two immediate neighbors). If the local transition function is identical for all the cells of a CA, the CA is referred to as uniform, otherwise the CA is called non-uniform or hybrid. In case of a finite cellular structure, which is typical in real applications, additional neighborhood has to be specified for the cells lying at the boundary of the cellular array. This is called as boundary conditions. Constant or cyclic boundary conditions have usually been considered. The constant boundary conditions assign constant states to the "missing" neighbors of the boundary cells; these states are invariable during the development of the cellular automaton. In case of the cyclic boundary conditions, the boundary cells are considered to be adjacent to the appropriate boundary cells at the opposite of the cellular array.

The behavior of the CA is determined by the local transition function. The development (i.e. the computation) of the CA is represented by a sequence of configurations of the after each developmental step (i.e. the synchronous

update of states of all the cells according to the local transition function), where the configuration is understood as a sequence of cell states of the CA at a given time. Although the processing elements (i.e. the cells) and the local transition function are usually simple, the CA may exhibit very complex global behavior. This phenomenon is referred to as emergent behavior.

The cellular automata represent universal computational model which means that arbitrary algorithm can be realized. Since there are only local interactions between the cells in the basic variant of the CA concept, the cellular arrays exhibit highly regular structures. Therefore, CAs are suitable for hardware implementations. Massive parallelism represents another advantage of this model. However, the design of a CA (i.e. the initial configuration and the local transition function) in order to achieve a specific behavior is often very difficult which represents a substantial disadvantage of the CAs. Another limitation is that the complex problems (for example, to process many input data) require high number of cells of the CA which gets expensive if implemented in hardware in comparison with an optimized solution that does not utilize cellular automaton.

The Game of Life invented by John Conway probably represents one of the most known applications of cellular automata [Gar70]. It simulates the life of cells using binary (i.e. each cell may possess either state 0 – a dead cell or 1 – a living cell) 2-dimensional CA. Each cell survives or dies in dependence on its surrounding cells according to the following rules (local transition function):

- Each living cell with one or no neighbors dies, as if by loneliness.
- Each living cell with four or more neighbors dies, as if by overpopulation.
- Each living cell with two or three neighbors survives.
- Each unpopulated place with three living neighbors becomes populated (i.e. a new cell arises).

A popular application of CAs is the study of self-replicating structures. The goal is to design a CA which is able to make a copy of an object next to the original one specified in the CA initial state. Langton proposed a so-called self-replicating loop which represents a 2D cellular automaton whose cells can possess one of eight states. The replicating structure consists of a loop (a suitable arrangement of states in the initial configuration) and a replicator (an extension of the loop) which ensures "copying" of the loop structure next to the original one. After a sufficient number of steps (depending on the size of the loop) the original structure is replicated. If the development of the CA continues, then more copies of the loop arise [Lan84]. Other typical applications include the utilization of CAs to solve the density task (a decision if the initial binary configurations contain more 1s than 0s) or synchronization task (every arbitrary binary initial configuration will produce switching configurations consisting of all 1s and all 0s after a sufficient number of steps).

34   3 Development

3.2.3 Genetic regulatory networks

In biology, genetic regulatory networks (GRNs) describe interactions of genomes and environment which leads to protein synthesis. On the basis of gene regulation and protein synthesis in biology, Dellaert and Beer introduced a general computational model called identically – genetic regulatory networks [DB94, Del95]. A gene model represents the basic part of a GRN. This model consists of a regulatory region and a coding region. The regulatory region contains a condition that must be satisfied in order to execute the coding region. The coding region contains proteins produced by the gene activation. The GRN is represented by an oriented graph whose vertices represent genes and edges represent the interaction of the genes, i.e. proteins produced by a gene depend on other genes (specified in the coding region of the gene).

For example, consider the genetic regulatory network shown in Figure 3.2. A simple network showing how different gene products of some genes regulate other genes is shown inside a cell, depicted as a rounded rectangle in Figure 3.2. The network describes a system that produces a number of products and then turns off when enough of product B is produced. The GRN shows that the entire network becomes activated when product B enters the cell from an external source. B causes A to be produced, which in turn causes C to be produced by another gene. A and C, without D in the cell cause more B to be created, which in turn feeds back into the production of A, further strengthening the cycle. Eventually, when a great deal of B is present, D is finally produced, stopping the generation of B and ending the feedback cycle. The GRN shows that interesting dynamics can result from the regulatory interactions of different genes [SM03].
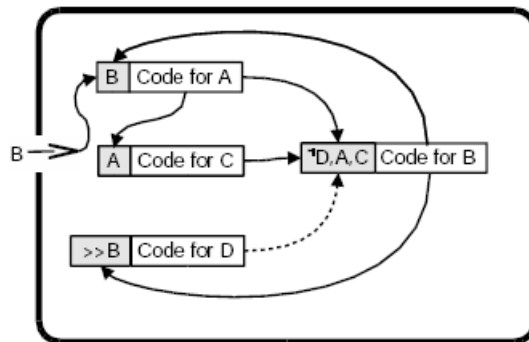
Fig. 3.2. An example of genetic regulatory network [SM03]. A, B, C and D denote products produced by the genes (vertices of the graph representing the GRN). The symbol >> means a large amount.

3.2.4 Random boolean networks

Random boolean networks (RBNs) were introduced by Stuart Kaufmann [Kau69, Kau93] for simulating the protein expression patterns in cells of a developing embryo. Later, this model was investigated by Dealler and Beer [DB96]. The concept of RBNs is similar to genetic regulatory networks described in the previous Section. In contrast with the GRNs, the basic element of a RBN (a vertex of the oriented graph describing the network) is represented by an arbitrary Boolean expression whose result effect other Boolean expressions — "genes" — of the RBN. The edges of the graph represent interactions between the vertices (i.e. input values to be processed by the vertices and output values produced by the boolean expressions and processed by other vertices). RBNs are computationally less expensive than GRN implementations, while exhibiting similar computational dynamics.

For example, consider the random boolean network shown in Figure 3.3. The state of the network is given by the boolean values of A, B, and C. At each time step, the state values are updated based on their values in the previous time step according to the boolean expressions specified in the vertices. The current state of a cell is given by the current state of the RBN by means of which the cell is modeled [DB96].
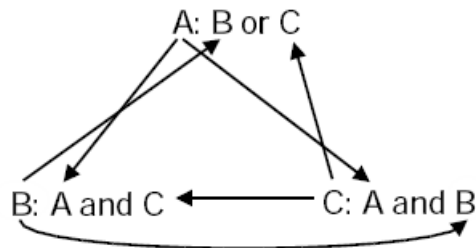
Fig. 3.3. An example of random boolean network [SM03]. In general, the expression $A : B\ op\ C$ specifies a boolean variable A whose value is determined by the boolean expression on the right of the colon. *op* denotes an arbitrary boolean operation.

3.2.5 Cellular encoding

Frédéric Gruau introduced a specific grammar-based developmental model for the construction of neural networks in an efficient modular manner. This method was termed cellular encoding [Gru94].

Cellular encoding takes the form of grammar tree used to encode a developmental process. The language upon which the grammar is defined uses instructions that correspond to local graph transformations controlling cell division (the genotype). Each cell retains a copy of the entire grammar-tree,

but has a pointer pointing to different subtrees of the main tree. Development begins with a single cell, which is subjected to cell division by duplication, according to the instructions in the genome. The result of this process is the phenotype, a neural network.

Cellular encoding is an example of context free system, in which neighbor state is not taken into consideration (it does not use cell signaling). Additionally, the model does not make use of diffusion chemicals, such as proteins. Symbolic knowledge is used in the grammar to permit hierarchies, symmetry and problem decomposition into sub-problems.

3.2.6 Graph-generation grammar

Hiroaki Kitano and his group were among the first to conduct research into evolving computational development (i.e. the utilization of an evolutionary algorithm to find a suitable behavior of a given developmental model, in particular, to evolve rewriting rules of a 2D parallel rewriting system). Kitano's model, that he called as graph-generation grammar, was successfully applied in a variety of areas, e.g. neural networks [Kit90], logic circuits or complex chemical circuits [Kit96].

The graph-grammar generation system is based on a rewriting system that develops a solution from a single starting symbol. In contrast with a standard grammar (for example an L-system described in Section 3.2.1), each rewriting rule substitutes a 2×2 matrix for a single symbol. The symbols are rewritten in parallel (similarly to L-systems) in each developmental step. After a specified number of developmental steps a square matrix is developed that is interpreted as a graph connection matrix. The resulting graph represents the target object (e.g. a neural network, a digital circuit etc.).

3.2.7 Some other developmental models

In the previous section, some of the most popular developmental models were summarized that have been applied in the evolutionary design fields. There are much more models which are based on those approaches (possibly their modifications and combinations) presented in the previous paragraphs. However, a detailed description of each of them is out of the scope of this work. Let us summarize some important references dealing with other developmental systems which were applied in various areas in recent years. An important resource related to development represents the book [S. 03], where a detailed description of some developmental models are stated from both the biological and computational point of view. The research of other developmental techniques may be found, for example, in the following publications:

- Design, optimization and analysis of cellular automata and their applications to solve some typical difficult tasks is described in Sipper's book [Sip97]. The author also introduces an original evolutionary method for the design of cellular automata called cellular programming.

- Haddow and Tufte investigated the development of digital circuits by means of L-systems and 2D cellular automata through FPGA reconfiguration, i.e. the design of circuits is performed directly in hardware [HTvR01, Tuf04, TH05].
- Utilization of L-systems as a generative encoding was investigated by Gordon. Development of mechanical structures, neural networks, controllers or robot simulators and the comparison of these methods with parametric encoding is presented in [HP01, Hor03].
- Gordon conducted the research of scalability of the evolutionary design in the area of digital circuits using principles of biologically inspired development. A concept of abstraction, modularity and reusability in the development was introduced. Possibilities for overcoming the problem of scale were demonstrated on various applications in comparison with direct mapping [GB02, Gor03, GB05b, GB05a, Gor05].
- A computationally complete developmental model based on the principles of gene expression and cell differentiation was introduced by Roggen in [RFF07]. In fact, it is a special case of cellular automata (a regular cell structure possessing only local cell interactions). The abilities of this model were demonstrated on the development of 2D patterns (e.g. the Norwegian flag) and neural networks for image recognition.
- Adaptation and self-repair during the developmental process were investigated by Miller in [Mil03] by means of a cellular automaton-based model. Experiments were conducted considering development of French flag and other graphical patterns. A detailed description of Miller's approach may also be found in Chapter 15 of [S. 03].

3.2.8 Summary of developmental models

As evident from the previous sections, most of the basic developmental models have been inspired by the appropriate processes observed in biology, for example cell division, gene regulation etc. However, their application in practical evolutionary systems may often be difficult. Therefore, more general methods have been introduced in order to be easily applied in technical domains rather than simulating purely biological principles. A typical representative of such approach is, for instance, developmental genetic programming.

In the next chapter, another variant of developmental concept will be introduced whose objective is to demonstrate the ability to design generic structures. The main inspiration is also taken from the general genetic programming approach – the evolution of computer programs.
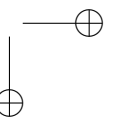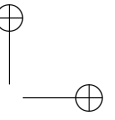
4

Instruction-based development

This work is focused on the research of generative encodings for the purposes of the evolutionary design of generic structures of digital circuits. This chapter briefly summarizes the key concepts invented for and applied in the experiments presented in the next chapters. First, problems of the existing developmental models will be discussed. Then a new approach will be introduced whose features have been considered with respect to the problems of the existing models. This approach will be referred to as instruction-based development. The issue of introducing a new developmental system is not to propose a biologically plausible model or to study any particular aspects of the biological development. The objective is to propose a more general concept that will be convenient for engineers (as designers of the evolutionary and developmental systems for specific applications), focusing on scalable structural evolutionary design (in terms of generic solutions as the products of the evolutionary process) that represents one of the challenging tasks in the common field of evolutionary computing. The concept of the instruction-based development represents the main contribution of this book. In the last part of this chapter, different variants of the instruction-based model will be described which are crucial for the case studies presented in the next chapters.

4.1 Problems of existing models

Before introducing the instruction-based developmental approach, let us briefly summarize the features of the existing developmental models and discuss their suitability to solve different kinds of problems. In particular, let us focus on the problem of the design of generic structures (or more generally, on solving arbitrarily large instances of a problem) which represents the crucial issue of this work. The discussion will be devoted to cellular automata, Lindenmayer systems (or grammars, in general), genetic regulatory networks and random boolean networks. These models can be considered as basic approaches in the

area of the computational development. Although there are much more models invented to solve specific problems, they utilize various modifications of the basic concepts, possibly their combinations. For instance, Kitano's graph generation grammar is a 2D extension of L-systems, Gruau's cellular encoding is a variant of graph grammar (a specific form of a rewriting system) and so on.

Cellular automata

Probably the most limiting issue of the CAs is the way of their design that is less intuitive for human designer. Therefore, CAs have often been evolved. However, the state space of cellular automata grows with increasing number of possible states, especially in the case of non-uniform CAs with increasing number of cells. Hence the evolutionary design of cellular automata is not scalable. If the CAs are applied as a developmental model, it is possible to achieve very complex (emergent) behavior. However, the generic developmental design using CAs may potentially be possible if a uniform theoretically infinite cellular model is applied.

Grammar models

Considering the typical grammar-based rewriting system, the crucial issue for evolutionary developmental design is the representation of candidate solutions by means of symbols. The size of the search space depends on the number of symbols and the number and complexity of the rewriting rules. Although there are some applications involving L-systems as developmental model in the evolutionary design, the generic development based on this model is still a rare case. In theory, however, strings of arbitrary size can be developed using rewriting systems if recursive and generative (i.e. non-reducing) rules are allowed which could make the generic design possible. The crucial feature of such developmental system is the interpretation of the developed strings because no systematic approach exists.

Genetic regulatory networks

GRNs represent a computational model inspired by specific biological process – the gene regulation in living organisms. The concept of GRNs enables to design extensive networks that may exhibit very complex behavior. However, the design of a suitable GRN for a specific application is a non-trivial task especially due to the difficult encoding of the problem using the basic elements of this model (no systematic approach exists to design GRNs in order to perform computations). Therefore, the utilization of GRNs as developmental model in an engineering area is still a rare case. Because of irregular structure of GRNs in general, in which the computations occur only in the existing vertices of the network, this model seems to be inadvisable for generic design.

Random boolean networks

In fact, RBNs can be understood as a generalized concept of binary cellular automata in which the cellular neighborhood is non-local and irregular (different cells may possess different number of neighbors from different places of the cellular structure which leads to graph representation of the RBNs). Therefore, similar features can be observed as in the case of cellular automata. Because of more complex structure of RBNs and interaction between the cells (i.e. nodes of the graph representing the RBN), more complex behavior is achievable in comparison with cellular automata of the same size. In addition, RBNs are more difficult to design due to larger state (search) space in comparison with CAs. Because of similar features regarding the structure of RBNs in comparison with genetic regulatory networks, the utilization of this model for the generic development is unusual.

In the previous paragraphs, the suitability of different developmental models to solve the generic design problem was discussed. The main problem regarding this task is that the model is either less intuitive for investigation "by hand" (which makes the solutions difficult to understand and hard to modify to adjust them to a specific requirements) or the concept of the model (structure, the way of computation etc.) is unsuitable to solve generic (scalable) tasks. Considering these issues, it is evident that research of new concepts and developmental models is needed in order to advance in solving the common challenges where complexity and scalability represent the main features. Therefore, a new approach has been devised and studied with the aim to improve the features that are critical in the developmental models discussed. A new developmental model — an instruction-based approach — was invented whose concept is inspired by the assembly languages in which an instruction represents the basic program construct. The programs can be represented by simple linear sequences of instructions and hence easily be encoded in the chromosomes of the evolutionary algorithms. Moreover, the instructions are comprehensible to human designer (programmer), hence the instruction-based developmental systems are easier to design and verify before the evolutionary process is applied and the evolved solutions are more suitable for the subsequent analysis and modification/optimization.

4.2 Introducing the instruction-based approach

Two variants of Koza's genetic programming were mentioned in sections 2.4.2 and 2.4.3: developmental genetic programming and linear genetic programming. Whilst, the linear GP refers to the representation of genotypes in genetic programming, the developmental GP represents an enhancement of the standard GP (the genotype is interpreted as a developmental encoding determining how to create the phenotype). These two aspects represent a basis

for introducing the new developmental approach whose main principle will be described in the following paragraphs.

Considering the capabilities of the genetic programming in different areas, its concept (regardless of what representation of the programs is utilized in the encoding) can be viewed as a universal technique for the evolution of computer programs (i.e. for the automatic design of algorithms). The genetic programming approach considering the linear representation of the programs will be used to evolve algorithms that are designed specifically for a given application (problem to be solved). These programs will be interpreted as a prescriptions for the development of the target objects (circuits) at the structural level. This approach will be termed an **application-specific development**. Because only simple instructions will be considered (similar to machine-language instructions) rather than more complex program constructs, our approach will be referred to as **instruction-based development**. Note that the single term "development" related to the experiments described in the next chapters will refer to "application-specific instruction-based development".

On the basis of the IBD concept, it is possible to define an **evolutionary instruction-based developmental system** to perform the automatic design process. This design system consists of an application-specific assembly language (let us call it by its abbreviation – ASAL), an ASAL interpreter, a set of domain-specific building blocks and an evolutionary algorithm in which a genome is interpreted as an ASAL program – a prescription for the construction of a target object using the building blocks. The ASAL represents a suitable set of instructions chosen with respect to the specific application and its programming rules. The programming rules are implemented by the ASAL interpreter. These rules typically include the encoding of the instructions, representation of their arguments, way of execution of the instructions and so on. In fact, the ASAL interpreter represent a runtime environment for executing the evolved programs (i.e. to perform the developmental process). Moreover, it implements additional resources which are provided to the evolutionary algorithm in order to make the automatic programming easy (analogous to programming languages provide similar means to the programmer). For example, the interpreter may implement variables that the programs use or a specific rules of interconnecting the building blocks during the development. The evolutionary algorithm is typically a GP-based approach. However, arbitrary encoding of the instructions in the chromosomes, operations and algorithm to evolve the programs may be applied.

As evident, the concept of an IBD design system is very similar to Koza's developmental genetic programming. For the purposes of this book, an enhancement of this approach will be introduced in order to enable the design of generic structures which is the main goal of this work. The following requirements will be applied on the developmental process and the object being developed to reach the goal:

1. The evolved program (or its part) should be possible to apply repeatedly or iteratively.
2. Additional parameters may be introduced into the ASAL interpreter specifying, in particular, the "size" of the target design.
3. The target object is able to "grow", respective the size of the instance to be developed is parametrized, according to the item (1), respective (2).

4.3 Application of instruction-based development

Considering the concept of the instruction-based developmental system described in the previous section, the problem of finding a solution using an IBD design system may be defined as discovering a program (by means of an evolutionary algorithm) for the design of generic structures. In fact, the key feature of the instruction-based development is the linear genetic programming approach transfered into the area of computational development. In this work, the IBD approach is intended to be applied on the structural level of the target objects.

In general, the goal of this work is the research of the computational development for the purposes of construction of generic circuit structures. Two different variants of the instruction-based development have been investigated:

- **Continual development**. The construction process is performed iteratively. The size of the target object is determined according to the number of iterations (developmental steps). It means that the object "grows up" from a simple initial instance (or possibly from scratch). If the initial instance of a given problem is specified at the beginning of the developmental process, it is usually called an embryo. Since the size of the object depends on the number of developmental steps, the solution is usually required to be functional after each iteration.
- **Parametric development**. A crucial feature of this approach is a parameter (or a set of parameters) as the input of the developmental system supplied by the designer that determines the size of the target structure. For different values of parameters the target object is developed from the start and the last instance is usually taken as the result. Therefore, the objects before the last step need necessarily not to be fully functional.

5
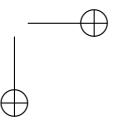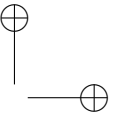
Continual development of arbitrarily large sorting networks

In this chapter, an evolutionary design system is presented concerning an application of instruction-based development for the automatic design of generic (i.e. arbitrarily large) sorting networks. The continual development approach is utilized in combination with the genetic algorithm. The key feature of this method is that the target object is required to be fully functional after each step of the development. It means that the circuit can "grow" continually and theoretically infinitely.

The domain of sorting networks was chosen because (1) conventional solutions of designing arbitrarily large sorting networks exist and, therefore, the results can be compared and (2) evolutionary techniques have already been utilized to design sorting networks for a predefined number of inputs; the evolution of arbitrarily large sorting networks, however, represents a rare case.

In overview, the proposed developmental system works as follows. First, a small sorting network (that is called an embryo) has to be prepared to solve the trivial instance of a problem. Then the evolved program is applied on the embryo to create a larger sorting network (solving a larger instance of the problem). In the next step, the same program is used to create a new instance of the sorting network from the created larger sorting network and so on. Every new instance of the sorting network is able to perform the function of all its previous instances. It will be demonstrate that such program can be designed automatically by means of evolutionary techniques.

It will be shown that the proposed approach is able to rediscover the common conventional principle of insertion which is traditionally used for constructing larger sorting networks from smaller instances. Furthermore, it will be shown that some of the evolved programs are able to produce much more efficient sorting networks (in terms of the comparison count and delay) than the conventional solution can offer. Finally, a formal proof will be presented demonstrating the generality of the best evolved algorithm for the construction of the sorting networks. The proposed method improves Sekanina's initial approach, described in [Sek04], which did not yield better solutions than the

conventional method. His method also did not deal with delay of resulting circuits.

5.1 Research motivation in this domain

Many studies have still been published dealing with the evolutionary design of sorting networks. Most of them are focused on the design of sorting networks with a given (constant) number of inputs. However, design of general approaches for the sorting networks represents a rare case in the field of evolutionary computation. Inspired by the existing conventional principles for the design of generic sorting networks and by Sekanina's work [Sek04] dealing with the evolutionary development of sorting networks, the first case study of this work will be focused on the evolutionary design of this kind of circuits in order to demonstrate the abilities of instruction-based development. Sorting networks exhibit a simple structure consisting of homogeneous building blocks – comparators. Therefore, the structure can be easily represented in the evolutionary algorithms and manipulated by the development.

Although there are efficient sorting networks known for up to 16 inputs, the direct design of larger networks is difficult. Therefore, an effective algorithm is needed for creating larger sorting networks from existing smaller instances. For example, an $N+2$ sorting network may be needed to be created from an N-input network. The issue of what arrangement of comparators should be added to the existing network in order to obtain larger efficient network represents the main problem of this approach. Possible solutions will be proposed in this chapter.

Effective implementation and sorting in hardware represents the main advantage of the sorting networks. Sorting networks can also be utilized as circuits for calculating medians. For example, median circuits represent a crucial components of image filters. Therefore, the results of this research may be utilized in this field.

5.2 Sorting networks and their design

The concept of sorting networks (SNs) was introduced in 1954; Knuth traced the history of this problem in his book [Knu98]. A sorting network is defined as a sequence of compare–swap operations (comparators) that depends only on the number of elements to be sorted, not on the values of the elements. A compare–swap of two elements (a, b) compares and exchanges a and b so that $a \leq b$ is obtained after the operation.

The main advantage of the sorting network is that the sequence of comparisons is fixed. Thus it is suitable for parallel processing and hardware implementation, especially if the number of sorted elements is small. Figure 5.1 shows an example of a 3-input sorting network.
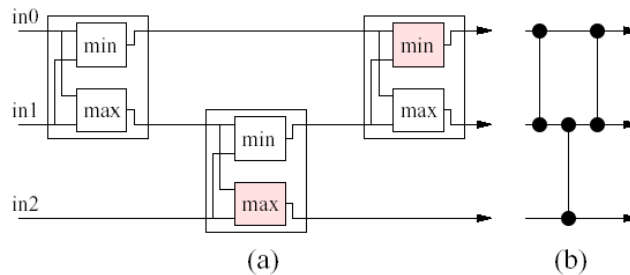
Fig. 5.1. (a) a three-input sorting network consists of three comparators, (b) alternative symbol. This network can be described using the string (0,1)(1,2)(0,1).

Inputs (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
Comparators	0	1	3	5	9	12	16	19	25	29	35	39	45	51	56	60

Table 5.1. The number of comparators and delay of some of the best currently known SNs

The number of compare–swap components and delay are two crucial parameters of any sorting network. By delay it is meant the minimal number of groups of compare–swap components that must be executed sequentially. The inputs of the comparators inside a single group are independent of each other and therefore these comparators can be executed in parallel. Designers try to minimize the number of comparators, delay or both parameters. Table 5.1 shows the number of comparators and delay of some of the best currently known sorting networks. Some of these networks were designed (or rediscovered) using evolutionary techniques [CM01, CM02b, CM02a, Hil90, Jui95, J. 99]. In most cases the evolutionary approach was based on the encoding given in Figure 5.1 (in which comparator inputs are encoded using two integers). Evolutionary techniques were also utilized to discover fault-tolerant SNs [HF04, MCFF00].

In order to find out whether an N-input SN operates correctly, it should be tested $N!$ input combinations. Thanks to the zero–one principle this number can be reduced. This principle states that if an N-input sorting network sorts all 2^N input sequences of zeros and ones into nondecreasing order, it will sort any arbitrary sequence of N numbers into nondecreasing order [Knu98]. Furthermore, if a proper encoding is used, on say 32 bits, and binary operators AND instead of minimum and OR instead of maximum, 32 test vectors can be evaluated in parallel and thus reduce the testing process 32 times. Unfortunately, it is usually impossible to obtain the general solution if only a subset of input vectors is utilized during the evolutionary design [IFK00].

SNs are usually designed for a fixed number of inputs. It is also valid for the mentioned evolutionary approaches. As the evolutionary approach which

utilizes direct encoding is not scalable, there exists a limit on the size of
the evolved SNs. Some conventional approaches exist for designing arbitrarily
large sorting networks. Figure 5.2 shows two principles for constructing a SN
for $N + 1$ inputs when an N-input network is given [Knu98].

- Insertion – the $(N+1)$st input is inserted into a proper place after the first
  N elements have been sorted.
- Selection – the largest input value can be selected before the remaining
  ones are sorted.
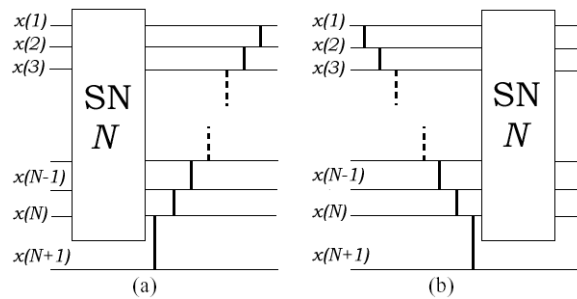
Fig. 5.2. Making $(N+1)$-sorters from N-sorters: (a) insertion principle, (b) selection
principle

It is evident that the insertion principle corresponds to the straight inser-
tion algorithm known from the theory of sorting. The selection principle is
related to the bubble sort algorithm. Examples of sorting networks created
using the two principles are shown in Figure 5.3. Observe that while physical
positions of comparators are different, their logical positions are equivalent.
Hence it is possible to re-arrange these comparators in order to obtain a sin-
gle SN (see Figure 5.4). The network contains the comparators that can be
executed in parallel. Therefore, its delay can be reduced substantially. These
comparators form the so-called parallel layers.

It is obvious that the sorting networks created using insertion or selection
principle are much larger than those networks designed for a particular N.
However, the method can be treated as a general design principle for building
arbitrarily large SNs. In the next sections, the principle will be rediscovered
firstly and then improved by means of evolutionary techniques.

5.3 Development for sorting networks

The objective of this section is to propose an application-specific development
for evolutionary algorithms, which, consequently, will be able to produce in-
novative arbitrarily large sorting networks.
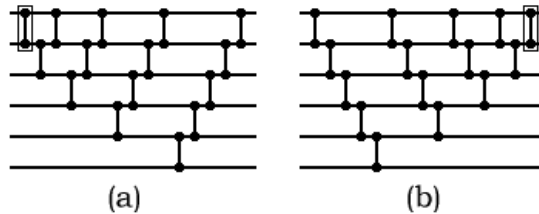
Fig. 5.3. Examples of sorting networks created using (a) insertion principle, (b) selection principle
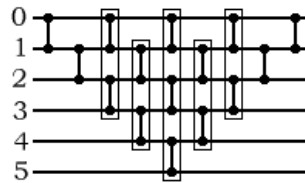
Fig. 5.4. A sorting network with parallel layers (in rectangles)

5.3.1 Basic concept

The proposed algorithm is based on Sekanina's approach described in [Sek04]. Unlike in [Sek04] this work deals with the delay of the sorting networks. A genetic algorithm is used to design a program, consisting of application-specific instructions, that is able to create a larger sorting network from a smaller SN (the smallest one is called an embryo). Then the program is applied on its result in order to create a larger sorting network and so on. Algorithm 1 and Figure 5.5 demonstrate this idea.

Algorithm 1:
Set time t = 0;
Create initial population of programs P(t);
Create SNs using programs from P(t);
Evaluate the sorting networks;
while (termination condition is false) **do**
{
  t = t+1;
  P(t) = create new population using P(t–1);
  Create SNs using programs from P(t);
  Evaluate sorting networks;
}

The development is realized as follows. Consider that we have a 2-input SN (i.e. $N = 2$ as seen in Figure 5.5) and we are going to evolve a program

that will create a 3-input sorting network from the 2-input SN. The same
program has to be able to create a 4-input sorting network from the 3-input
SN and so on. In general, the aim of the developmental system is to create
a larger sorting network from a smaller one. For the purposes of description
of this design process, let us introduce the following terms. A developmental
step is defined as a single application of the evolved program. A finite num-
ber of the developmental steps is denoted as a developmental sequence. Let
us define the size of a developmental step as the difference of the number of
inputs of two resulting sorting networks following immediately in the devel-
opmental sequence. For example, if a 4-input sorting network is created from
a 2-input SN network after one developmental step, the size of developmental
step possesses the value 2 that is determined as a difference of these numbers
of inputs.
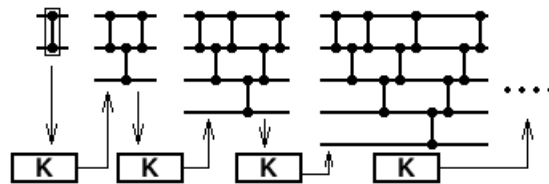
Fig. 5.5. Designing larger sorting networks from smaller sorting networks by means
of a construction program K

5.3.2 Representation and the proposed developmental method

Sorting networks are encoded as sequences of pairs of integers. For instance,
as Figure 5.1 shows, the 3-input SN is represented by the sequence of pairs
$(0, 1)(1, 2)(0, 1)$ indicating the ordering of compare–swap operations over the
inputs 0, 1 and 2. The program is a sequence of instructions, each of which
is encoded as three integers – the operation code, the first argument and the
second argument. The representation is very similar to the concept of lin-
ear genetic programming [BB07]. Only two types of instructions are utilized:
copy and modify. Table 5.2 introduces their semantics, variants, operation
codes and parameters. Note that the instructions were designed to solve the
specific task – the construction of arbitrarily large sorting networks. The Mod-
ify instructions read the indices of inputs of a comparator and add the values
of their arguments to them. Modulo-operation ensures that the created com-
parator remains inside the sorting network of a given number of inputs. This
type of instructions may be considered as a "shift" of a comparator to another
position preserving the ordering of comparators. The Copy instructions copy
some comparators (beginning from the actual one) to the next instance. The

number of comparators to be copied depends on the instruction argument and the number of inputs of the sorting network being created.

Instruction	arg1	arg2	description
0: ModifyS	a	b	$c1 = (c1 + a) \bmod w, c2 = (c2 + b) \bmod w, cp = cp + 1,$ $np = np + 1$
1: ModifyM	a	b	$c1 = (c1 + a) \bmod w, c2 = (c2 + b) \bmod w, cp = cp + 1,$ $ep = ep + 1, np = np + 1$
2: CopyS	k	$-$	copy $w - k$ comparators, $cp = cp + 1, np = np + w - k$
3: CopyM	k	$-$	copy $w - k$ comparators, $cp = cp + 1, ep = ep + w - k,$ $np = np + w - k$

Table 5.2. Instruction set utilized in development. "*mod*" denotes the modulo operation.

Let $c1$ and $c2$ (i.e. the pair $(c1, c2)$) denote indices of inputs of a comparator in the embryo that is processed by an instruction from Table 5.2.

The Instructions utilize three pieces of information: (1) operation codes and (2) argument values given by GA, and (3) w, which is the number of inputs (width) of the currently constructed SN. This value must be inserted into the developmental process externally. Three pointers are utilized in order to indicate the current position in sequences:

- ep – pointer to the source sorting network (embryo pointer),
- np – pointer to the next comparator in a constructed network (next-position pointer), and
- cp – instruction pointer.

Note that there are two different variants of the instructions. Let us denote them as S-variant (i.e. the instructions ModifyS and CopyS) and M-variant (i.e. the instructions ModifyM and CopyM). The instructions of M-variant update the embryo pointer (ep) after the execution of the instruction while the S-variant of the instructions does not influence the embryo pointer.

As Figure 5.6 shows, the instructions of the program are executed sequentially; each of the instructions process the comparator pointed by the embryo pointer. The comparators of the embryo are also processed sequentially. If a group of comparators are to be processed (concerning the Copy instructions), the comparators are processed in sequence starting from the comparator pointed by the embryo pointer. Before execution of the first instruction, an auxiliary variable (e_end) is initialized by the value of np. This auxiliary value marks the end of embryo and is invariable during actual application of the program. The process of construction terminates when either all instructions of the program are executed or the end of embryo is reached (i.e. $ep = e\_end$). After a single application of the program the obtained sorting network is evaluated. If the program is applied again, a larger sorting network

is created and so on. In such case, the pointers ep and np possess their values resulted from the previous application; only cp and e_end are updated. Note that the sorting network obtained by repeated application of the program possesses all the comparators of its precursors.
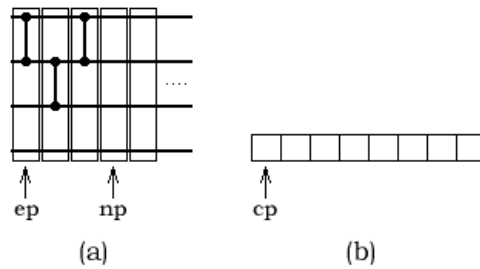
Fig. 5.6. Initialization of the development: (a) growing sorting network, (b) chromosome (i.e. instructions in the construction program)

The goal is to find such a program that will create valid sorting networks with the minimal number of comparators and/or delay. Because the delay of constructed SNs should be minimized, the following special condition has to be satisfied in order to acknowledge the result of execution of the Modify instruction: the result of the Modify instruction is valid only in case that $c1 < c2$ holds for the newly created comparator. Otherwise, the new comparator is not included in the sorting network and the instruction only updates the embryo pointer. Figure 5.7 shows an example of the invalid result of Modify instruction. The pointer ep determines a comparator that will be used to create a comparator at position specified by np. However, the comparator created at the np position does not satisfy the condition $c1 < c2$ because $c1 = 3$ and $c2 = 0$. Therefore, it will not be included into the sorting network being developed. The experiments showed that such comparators are often redundant, i.e. they do not swap the input values for any arbitrary input vector of the sorting network. If these comparators were accepted, the redundancy would propagate to the larger sorting networks, which would be ineffective too.

5.3.3 An example of two steps of development

Figure 5.8 shows an example of two steps of the construction process. The horizontal sequence of numbers denotes the comparator positions. The vertical sequence of numbers denotes the indices of inputs of sorting network. The thin rectangle surrounds the embryo. The vertical thin line separates the comparators created in the first and second application of the program. $ep1 = 0$ denotes the comparator pointed by embryo pointer, $np1 = 3$ denotes
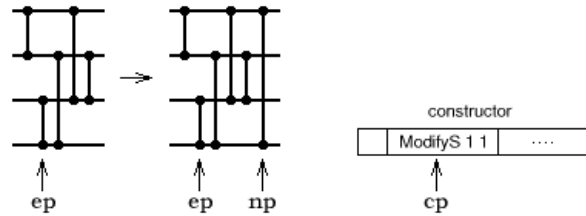
Fig. 5.7. Example of invalid result of *Modify* instruction

next-position pointer and $end1 = 3$ denotes the end of embryo before the first application of the program. Similarly, $ep2 = 3$ denotes the comparator pointed by embryo pointer, $np2 = 8$ denotes next-position pointer and $end2 = 8$ denotes the end of embryo before the second application of the program. Before any application of the program the pointers ep and np are initialized to the values of $ep1$ and $np1$ respectively.

After execution of instructions [ModifyS 2 2] and [ModifyS 1 2], comparators (2,3) and (1,3) are created in positions 3 and 4 (using the comparator (0,1) at the position 0). The embryo pointer (ep) remains unchanged and $np = 5$. Execution of the [ModifyM 0 1] results in creating comparator (0,2) at the position 5. Now, $ep = 1$ and $np = 6$. By applying the [ModifyS 2 1] on comparator (1,2) a new comparator (3,3) is created. However, such the comparator does not satisfy the $c1 < c2$ condition and hence it will not be included in the sorting network. ep and np remain unchanged. The [CopyM 3 1] instruction copies one comparator from the position 1 to the position pointed by $np = 6$ (since a 4-input SN is being created and the first argument of the CopyM instruction is 3, the 4–3 results in 1 comparator to be copied – see Table 5.2). The instruction updates the pointers, so now $ep = 2$ and $np = 7$. The [CopyM 2 4] should copy two comparators. Since there is only one comparator before the end of embryo, only one comparator will be copied and the pointers will be updated to $ep = 3$ and $np = 8$. Because the end of embryo was reached and all the instructions of the program were executed, the first application is finished.

The ep and np pointers now possess the values of $ep2$ and $np2$ and this is the starting configuration for the second application of the program. Execution of instructions proceeds in the same manner. Comparators will be created in positions 8–15. Note that during the second application of the program the result of the [ModifyS 2 1] instruction, that was applied on the comparator (1,3) at the position 4, is valid and the comparator (3,4) will be created at the position 11. Since we are now creating a 6-input SN, the [CopyM 3 1] copies three comparators from the positions 4, 5 and 6. The last [CopyM 2 4] instruction copies one comparator from the position 7 before the end of the second embryo and the second application of the program is finished. The next applications would construct the 8-, 10-, 12-input SNs and so on.

54   5 Continual development of arbitrarily large sorting networks
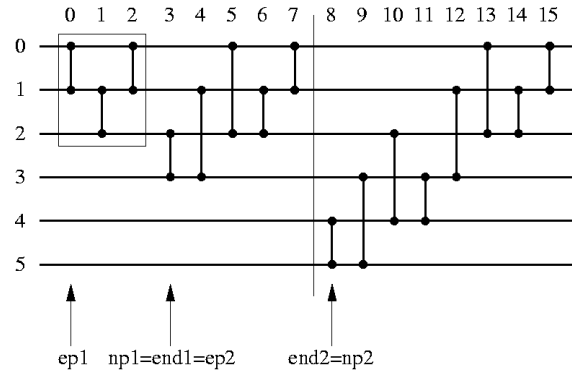
Fig. 5.8. Example of the construction of sorting networks by means of the program [ModifyS 2 2][ModifyS 1 2][ModifyM 0 1][ModifyS 2 1][CopyM 3 1][CopyM 2 4]

5.3.4 Evolutionary design system

Two variants of genetic algorithm were applied during the experiments: simple GA and steady-state GA. In total three sets of experiments have been performed: evolution of arbitrarily large sorting networks, evolution of odd-input sorting networks and evolution of even-input networks. The utilization of a particular variant and its parameters for a given set of experiments was determined experimentally. For the first two sets the steady-state GA was utilized. For the last set of experiments the simple GA was applied.

The setup of a genetic algorithm is specified by the crossover probability, mutation probability, the number of individuals in the population (i.e. the population size) and the maximal number of generations. In addition, steady-state GA requires the number of overlapping individuals between the generations to be specified. Each GA applied during the experiments works with the chromosomes of the constant length (i.e. the constant number of instructions of the program to be evolved). Each instruction in the chromosome is encoded as a triplet of integers that are interpreted as the operation code and two arguments of the instruction. The uniform crossover and the mutation operator to mutate exactly one instruction per chromosome was applied. The mutation operator is applied on all offspring generated by the crossover. The initial population is generated randomly. Note that the parameters of a genetic algorithm are stated in the sections presenting the results of the specific experiments.

The proposed developmental scheme can fully be defined using the following parameters which will be utilized to characterize the results in Section 5.4. Let $w1$ denote the number of inputs of the smallest sorting network that is constructed from ew-input embryo in the fitness calculation process (i.e. the SN created by the first application of the program). Similarly, w_max denotes the largest sorting network constructed during the fitness evaluation. Let dw

be a difference between the number of inputs of neighboring networks created
by the program. In this section, dw is 1 or 2. Finally, it is useful to define
one more parameter, de, $de = w1 - ew$. The following parameters summarize
the mentioned example: $w1 = 4$, $w\_max = 7$, $dw = 1$, and $ew = 3$. The
specific values of these parameters are given in the sections describing the
experimental results.

The goal is to evolve arbitrarily large sorting networks. However, because
of problems with the scalability of fitness evaluation, only several instances of
the growing SN can be evaluated in the fitness calculation process. Assume
that we will start with a 3-input sorting network and we are going to develop
larger sorting networks considering the developmental steps of size 1. Then
the candidate program is used to build a developmental sequence consisting
of a 4-input, 5-input, 6-input and 7-input SN. The fitness value is calculated
using a sorting network simulator that evaluates that developmental sequence
as follows:

$$fitness = f(4) + f(5) + f(6) + f(7),$$

where $f(j)$ is the fitness value for a j-input SN. This value is calculated using
the zero–one principle. Hence the value $2^4 + 2^5 + 2^6 + 2^7 = 240$ represents
the best possible value that could be obtained. At the end of evolution it has
to be tested whether the evolved program is general, i.e. whether it generates
infinitely large sorting networks which sort all possible input sequences. If
a program is able to create a sorting network for a sufficiently high N ($N = 28$
in our case) then the program is considered as general.

The experiments were conducted on common PCs running RedHat-based
Linux operating system. The hardware configuration consisted of a 2.0 GHz
processor and 512 MB RAM. Sun Grid Engine (SGE) system was utilized
so that several independent experiments could be run on different PCs in
parallel.

5.4 Experimental results

In this section the evolved programs and resulting sorting networks developed
by means of that programs are summarized. A general success rate will be
measured that is determined as the number of general programs (i.e. programs
that are able to develop theoretically infinitely large sorting network) obtained
out of 100 independent runs of the evolutionary process.

The produced sorting networks will be characterized in terms of compara-
tors count and delay. Each program will be labeled by its length (the number
of instructions), size of utilized embryo and identification. For instance, g4-
3all_2 denotes a general program consisting of four instructions that develops
from a three-input embryo sorting networks of all (i.e. odd as well as even)
inputs. The last digit 2 denotes the order of the program in the category of
presented four-instruction programs. Moreover, it was recognized that very

interesting sorting networks are produced in the case that only even-input (or odd-input) networks are required. Hence, in addition to the arbitrary number of inputs of resulting sorting networks, the programs were evolved also for growing sorting networks possessing only the odd or even number of inputs which is identified in the program labels. Table 5.3 shows a general description of how the evolved programs will be labeled in the experiments description.

Symbols	Description
X	program length (the number of instructions)
Y	embryo width (the number of inputs)
zzz	odd/even/all (possible inputs)
ID	additional identification

Table 5.3. Definition of labels for the programs in the form *gX-Yzzz_ID*

Three tables will summarize each experiment. The first table lists the best programs. The second table gives the number of compare–swap components and the number of redundant comparators (in parentheses). Delay and the number of parallel layers in parentheses (that are available after removal of redundant comparators) are given in the third table. The best solution is typed *italic*. Note that due to the space reasons the longer programs will be shown with only the operation codes of their instructions (see Table 5.2) instead of the instruction names. Various types of embryos have been utilized in the experiments. Figure 5.9 shows the embryos that were utilized.
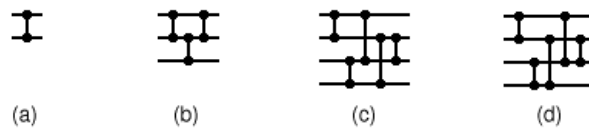
Fig. 5.9. Embryos utilized: (a) 2-input, (b) 3-input, (c) 4-input, (d) 4-input – another type

5.4.1 Evolving arbitrarily large sorting networks

In the first set of experiments, the sorting networks with the even as well as odd number of inputs were evolved from a three-input embryo. It corresponds to setting: $ew = 3$, $de = 1$ and $dw = 1$. A simple GA was used, operating with 60 individuals, with the probability of crossover $p_c = 0.75$ and the probability of mutation $p_m = 0.08$. Results are summarized in Tables 5.4, 5.5, and 5.6.

The evolved programs are very simple and of the same quality as the conventional approach produces. In fact the conventional straight insertion

Program	Instructions	#general
g3-3all	*[ModifyS 2 2] [ModifyS 1 1] [CopyM 3 2]*	
g3-3all_2	[ModifyS 1 1] [ModifyM 2 2] [CopyM 0 2]	100
g3-3all_3	[ModifyS 2 2] [ModifyS 1 1] [CopyM 0 3]	
g4-3all	[ModifyS 3 2] [ModifyS 2 2] [ModifyS 1 1] [CopyM 3 3]	100
g4-3all_2	[ModifyM 0 0] [ModifyS 1 1] [ModifyS 1 0] [CopyM 0 2]	

Table 5.4. Examples of general programs evolved for a 3-input embryo considering the following setup of parameters: $ew = 3, de = 1, dw = 1$

N	4	5	6	7	8	9	10	11	12	13	14	15
conv.	6	10	15	21	28	36	45	55	66	78	91	105
g3-3all	*6*	*10*	*15*	*21*	*28*	*36*	*45*	*55*	*66*	*78*	*91*	*105*
g3-3all_2	7	12	18	25	33	42	52	63	75	88	102	117
	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)
g3-3all_3	8	14	21	29	38	48	59	71	84	98	113	129
	(2)	(4)	(6)	(8)	(10)	(12)	(14)	(16)	(18)	(20)	(22)	(24)
g4-3all	6	10	15	21	28	36	45	55	66	78	91	105
g4-3all_2	7	12	18	25	33	42	52	63	75	88	102	117
	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)
N	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**
conv.	120	136	153	171	190	210	231	253	276	300	325	351
g3-3all	*120*	*136*	*153*	*171*	*190*	*210*	*231*	*253*	*276*	*300*	*325*	*351*
g3-3all_2	133	150	168	187	207	228	250	273	297	322	348	375
	(13)	(14)	(15)	(16)	(17)	(18)	(19)	(20)	(21)	(22)	(23)	(24)
g3-3all_3	146	164	183	203	224	246	269	293	318	344	371	399
	(26)	(28)	(30)	(32)	(34)	(36)	(38)	(40)	(42)	(44)	(46)	(48)
g4-3all	120	136	153	171	190	210	231	253	276	300	325	351
g4-3all_2	133	150	168	187	207	228	250	273	297	322	348	375
	(13)	(14)	(15)	(16)	(17)	(18)	(19)	(20)	(21)	(22)	(23)	(24)

Table 5.5. The number of comparators of sorting networks for programs from Table 5.4. The number of redundant comparators is given in parentheses.

algorithm has been rediscovered (see Figure 5.10). Some other examples are given in Figure 5.11. We were not able to improve the principle of construction in this way. Hence the parameters of the development and GA have been changed as the next section illustrates.

5.4.2 Evolving odd-input sorting networks

The constructed sorting networks were restricted to the odd number of inputs. Surprisingly, the most interesting odd-input sorting networks were generated by using an even-input embryo. A 4-input embryo was chosen, $ew = 4$, and parameters $de = 1$ and $dw = 2$. After some experiments, the best results were produced by a steady-state genetic algorithm with $p_c = 0.74$ and $p_m = 0.1$.

N	4	5	6	7	8	9	10	11	12	13	14	15
conv.	5	7	9	11	13	15	17	19	21	23	25	27
g3-3all	*5*	*7*	*9*	*11*	*13*	*15*	*17*	*19*	*21*	*23*	*25*	*27*
g3-3all_2	7	11	15	19	23	27	31	35	39	43	47	51
	(5)	(7)	(9)	(11)	(13)	(15)	(17)	(19)	(21)	(23)	(25)	(27)
g3-3all_3	7	11	15	19	23	27	31	35	39	43	47	51
	(5)	(7)	(9)	(11)	(13)	(15)	(17)	(19)	(21)	(23)	(25)	(27)
g4-3all	5	7	9	11	13	15	17	19	21	23	25	27
g4-3all_2	6	9	12	15	18	21	24	27	30	33	36	39
	(5)	(7)	(9)	(11)	(13)	(15)	(17)	(19)	(21)	(23)	(25)	(27)
N	16	17	18	19	20	21	22	23	24	25	26	27
konv.	29	31	33	35	37	39	41	43	45	47	49	51
g3-3all	*29*	*31*	*33*	*35*	*37*	*39*	*41*	*43*	*45*	*47*	*49*	*51*
g3-3all_2	55	59	63	67	71	75	79	83	87	91	95	99
	(29)	(31)	(33)	(35)	(37)	(39)	(41)	(43)	(45)	(47)	(49)	(51)
g3-3all_3	55	59	63	67	71	75	79	83	87	91	95	99
	(29)	(31)	(33)	(35)	(37)	(39)	(41)	(43)	(45)	(47)	(49)	(51)
g4-3all	29	31	33	35	37	39	41	43	45	47	49	51
g4-3all_2	42	45	48	51	54	57	60	63	66	69	72	75
	(29)	(31)	(33)	(35)	(37)	(39)	(41)	(43)	(45)	(47)	(49)	(51)

Table 5.6. Delay of sorting networks from Table 5.4. Parentheses show delay after removal of redundant comparators.
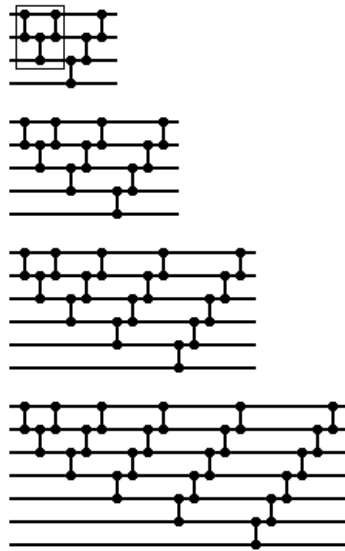
Fig. 5.10. The insertion principle rediscovered using instructions: [ModifyS 2 2][ModifyS 1 1][CopyM 3 2] or [ModifyS 3 2][ModifyS 2 2][ModifyS 1 1][CopyM 3 3]
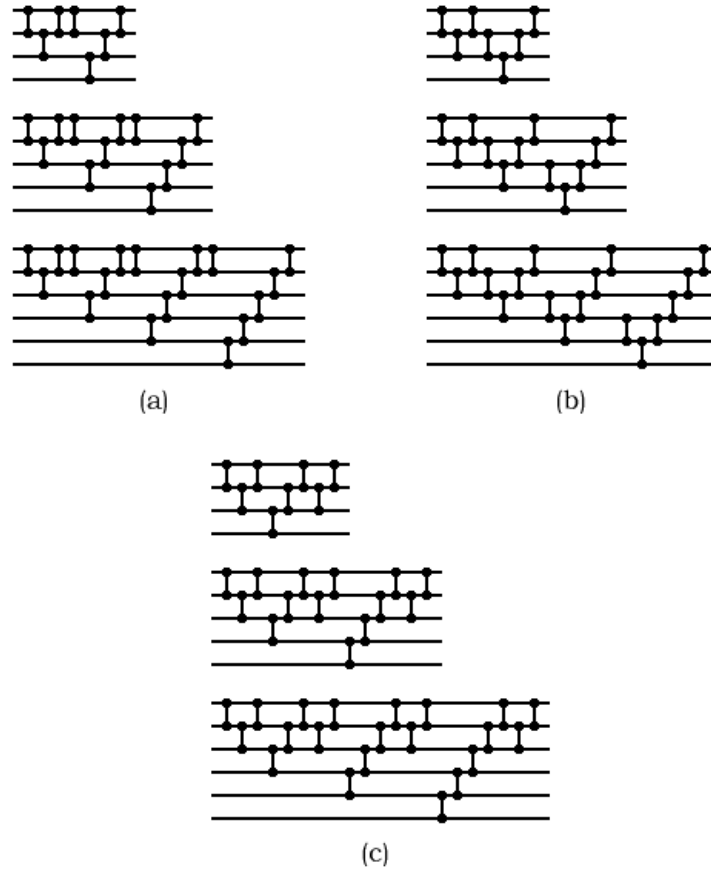
Fig. 5.11. Examples of growing sorting networks created using the programs (a) g4-3all_2, (b) g3-3all_2, (c) g3-3all_3

Population consists of 400 individuals with overlapping 12 individuals. Table 5.7 shows chromosomes of some evolved programs. As Table 5.8 indicates, it is possible to reduce the number of comparators substantially in this set of experiments. Delays are given in Table 5.9.

If the number of comparators is measured then the best-evolved sorting network is given in Figure 5.12. In case of minimizing the delay, the best solution is shown in Figure5.13. However, all the SNs contain redundant comparators which make their delay unnecessarily long. After their removal the quality (delay) of the conventional solution can be obtained.

Program	Instructions	#general
g8-4odd	[0 2 2] [0 2 3] [1 3 3] [0 1 1] [0 4 0] [3 2 3] [3 0 4] [3 1 3]	
g8-4odd_2	[0 2 2] [0 2 3] [1 3 3] [0 1 1] [3 4 2] [3 2 2] [3 2 2] [3 4 4]	41
g8-4odd_3	[0 2 2] [0 2 3] [0 3 3] [1 2 0] [0 1 1] [3 0 4] [0 3 3] [3 3 3]	
g8-4odd_4	[0 2 2] [0 3 3] [0 2 2] [0 1 1] [0 2 2] [3 0 0] [3 3 2] [3 0 0]	
g7-4odd	[0 2 2] [0 3 2] [1 2 3] [0 3 2] [0 1 1] [3 0 1] [3 0 3]	
g7-4odd_2	[0 2 2] [1 2 3] [0 3 2] [0 1 1] [0 2 1] [3 0 2] [3 4 0]	62
g7-4odd_3	[0 2 2] [1 2 3] [0 1 1] [0 3 2] [0 1 1] [3 3 2] [3 2 3]	
g6-4odd	[0 2 2] [0 2 3] [0 3 3] [1 1 2] [3 2 1] [3 3 3]	80
g6-4odd_2	[0 2 2] [1 2 3] [0 3 2] [0 1 1] [3 1 3] [3 3 4]	

Table 5.7. Evolved programs for the construction of odd-input sorting networks using a four-input embryo. The parameters of the developmental system are: $ew = 4, de = 1, dw = 2$.

N	5	7	9	11	13	15	17	19	21	23	25	27
conv.	10	21	36	55	78	105	136	171	210	253	300	351
g8-4odd	14	26	41	59	80	104	131	161	194	230	269	311
	(5)	(8)	(11)	(14)	(17)	(20)	(23)	(26)	(29)	(32)	(35)	(38)
g8-4odd_2	13	24	38	55	75	98	124	153	185	220	258	299
	(4)	(6)	(8)	(10)	(12)	(14)	(16)	(18)	(20)	(22)	(24)	(26)
g8-4odd_3	13	24	39	58	81	108	139	174	213	256	303	354
	(4)	(6)	(9)	(13)	(18)	(24)	(31)	(39)	(48)	(58)	(69)	(81)
g8-4odd_4	15	30	50	75	105	140	180	225	275	330	390	455
	(6)	(10)	(15)	(21)	(28)	(36)	(45)	(55)	(66)	(78)	(91)	(105)
g7-4odd	12	22	35	51	70	92	117	145	176	210	247	287
	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)
g7-4odd_2	12	23	38	57	80	107	138	173	212	255	302	353
	(3)	(5)	(8)	(12)	(17)	(23)	(30)	(38)	(47)	(57)	(68)	(80)
g7-4odd_3	13	25	41	61	85	113	145	181	221	265	313	365
	(4)	(7)	(11)	(16)	(22)	(29)	(37)	(46)	(56)	(67)	(79)	(92)
g6-4odd	13	24	38	55	75	98	124	153	185	220	258	299
	(4)	(6)	(8)	(10)	(12)	(14)	(16)	(18)	(20)	(22)	(24)	(26)
g6-4odd_2	12	22	35	51	70	92	117	145	176	210	247	287
	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)

Table 5.8. The number of comparators for odd-input sorting networks created using programs from Table 5.7

N	5	7	9	11	13	15	17	19	21	23	25	27
conv.	7	11	15	19	23	27	31	35	39	43	47	51
g8-4odd	11	18	25	32	39	46	53	60	67	74	81	88
	(6)	(12)	(17)	(22)	(27)	(32)	(37)	(42)	(47)	(52)	(57)	(62)
g8-4odd_2	10	16	22	28	34	40	46	52	58	64	70	76
	(6)	(12)	(17)	(22)	(27)	(32)	(37)	(42)	(47)	(52)	(57)	(62)
g8-4odd_3	10	16	22	28	37	45	53	61	70	80	90	100
	(6)	(12)	(17)	(22)	(27)	(32)	(37)	(42)	(47)	(52)	(57)	(62)
g8-4odd_4	*11*	*19*	*27*	*35*	*43*	*51*	*59*	*67*	*75*	*83*	*91*	*99*
	(6)	*(11)*	*(15)*	*(19)*	*(23)*	*(27)*	*(31)*	*(35)*	*(39)*	*(43)*	*(47)*	*(51)*
g7-4odd	9	14	19	24	29	34	39	44	49	54	59	64
	(6)	(12)	(17)	(22)	(27)	(32)	(37)	(42)	(47)	(52)	(57)	(62)
g7-4odd_2	9	16	23	30	37	44	51	58	65	72	79	86
	(6)	(12)	(17)	(22)	(27)	(32)	(37)	(42)	(47)	(52)	(57)	(62)
g7-4odd_3	10	17	24	31	38	45	52	59	66	73	80	87
	(6)	(12)	(16)	(20)	(24)	(28)	(32)	(36)	(40)	(44)	(48)	(52)
g6-4odd	10	16	22	28	34	40	46	52	58	65	70	76
	(6)	(12)	(17)	(22)	(27)	(32)	(37)	(42)	(47)	(52)	(57)	(62)
g6-4odd_2	9	14	19	24	29	34	39	44	49	54	59	64
	(6)	(12)	(17)	(22)	(27)	(32)	(37)	(42)	(47)	(52)	(57)	(62)

Table 5.9. Delay of odd-input sorting networks created using programs from Table 5.7
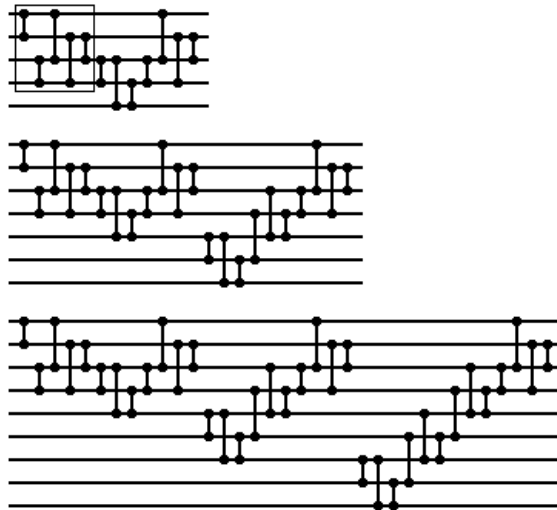
Fig. 5.12. Comparator-efficient odd-input sorting networks created by means of the program *g6-4odd_2*. The embryo is marked.
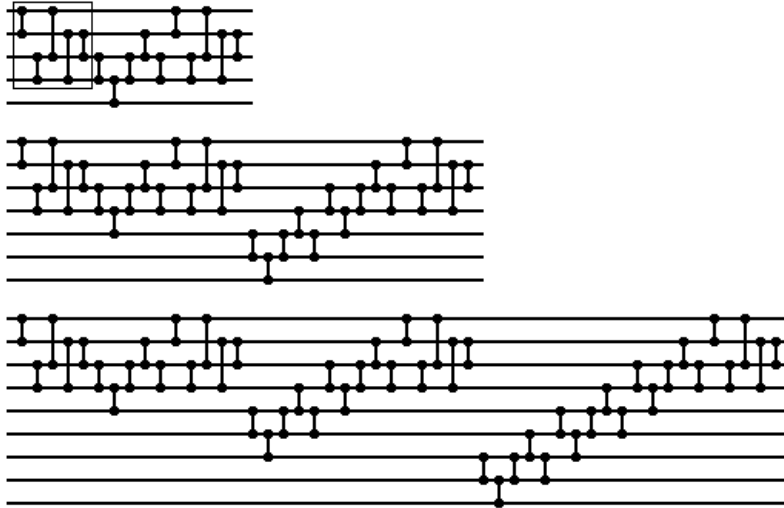
Fig. 5.13. Delay-efficient odd-input sorting networks created by means of the program *g8-4odd_4*

5.4.3 Evolving even-input sorting networks

In the previous section better programs were discovered than the conventional approach offers for the odd-input sorting networks. This section deals with discovered even-input SNs that are better than conventional ones.

Program	Instructions	#gen.
g9-2even	[0 2 2] [0 1 2] [0 0 1] [1 1 1] [0 4 4] [3 3 2] [3 1 1] [1 1 2] [2 1 0]	14
g8-2even	[0 2 2] [0 0 1] [0 1 2] [1 1 1] [3 0 2] [0 1 3] [3 0 0] [3 2 3]	25
g8-2even_2	[0 2 2] [0 1 2] [0 0 1] [1 1 1] [0 4 4] [3 0 1] [3 4 1] [1 2 3]	
g6-2even	[0 2 2] [0 1 1] [0 0 2] [0 2 2] [3 0 4] [3 0 0]	73
g6-2even_2	*[0 2 2] [0 1 2] [0 0 1] [1 1 1] [3 1 2] [3 1 1]*	

Table 5.10. Evolved programs for the construction of even-input sorting networks utilizing a two-input embryo. The parameters of the developmental systems are: $ew = 2, de = 2, dw = 2$. **#gen.** denotes the number of general programs.

In contrast to the previous section, various types of embryos have been confirmed as useful for constructing novel sorting networks. The simple genetic algorithm that was applied in this set of experiments was specified by the following parameters: $p_c = 0.7$, $p_m = 0.023$ and the population size counts 60 individuals. Tables 5.10, 5.11 and 5.12 summarize the results for the two-input embryo.

N	4	6	8	10	12	14	16	18	20	22	24	26	28
conv.	6	15	28	45	66	91	120	153	190	231	276	325	378
g9-2even g8-2even_2 g6-2even_2	5	12	22	35	51	70	92	117	145	176	210	247	287
g8-2even	5	12	22	35	51	71	95	123	155	191	231	275	323
	(0)	(0)	(0)	(0)	(0)	(1)	(3)	(6)	(10)	(15)	(21)	(28)	(36)
g6-2even	6	15	28	45	66	91	120	153	190	231	276	325	378

Table 5.11. The number of comparators of even-input sorting networks created from a two-input embryo using programs given in Table 5.10

N	4	6	8	10	12	14	16	18	20	22	24	26	28
conv.	5	9	13	17	21	25	29	33	37	41	45	49	53
g9-2even g8-2even_2 g6-2even_2	3	7	11	15	19	23	27	31	35	39	43	47	51
g8-2even	3	7	11	15	19	23	28	34	39	45	51	57	63
	(3)	(7)	(11)	(15)	(19)	(23)	(27)	(31)	(35)	(39)	(43)	(47)	(51)
g6-2even	3	7	11	15	19	23	27	31	35	39	43	47	51

Table 5.12. Delay of even-input sorting networks created from a two-input embryo using programs given in Table 5.10

As Figure 5.14 shows, the optimal 4-input sorting network was created from a 2-input embryo after the first developmental step (it is the best possible 4-input sorting network from both the point of view of the number of comparators and delay).

The *g8-4even_2* represents one of the best programs that has ever been evolved in this work. This program uses a four-input embryo and produces sorting networks with a better comparator count and delay than the conventional solution (the insertion principle). However, it contains redundant comparators that have to be removed. Examples of programs evolved from the 4-input embryo (including *g8-4even_2*) are given in Table 5.13. Other parameters are summarized in Tables 5.14 and 5.15. Sorting networks created using the best programs are shown in figures 5.15 and 5.16.

Two interesting programs were evolved by using a three-input embryo. They are not as good as the programs utilizing a four-input embryo. However, they still produce better results than the conventional approach (see Tables 5.16, 5.17 and 5.18). Examples of sorting networks are given in Figure 5.17.
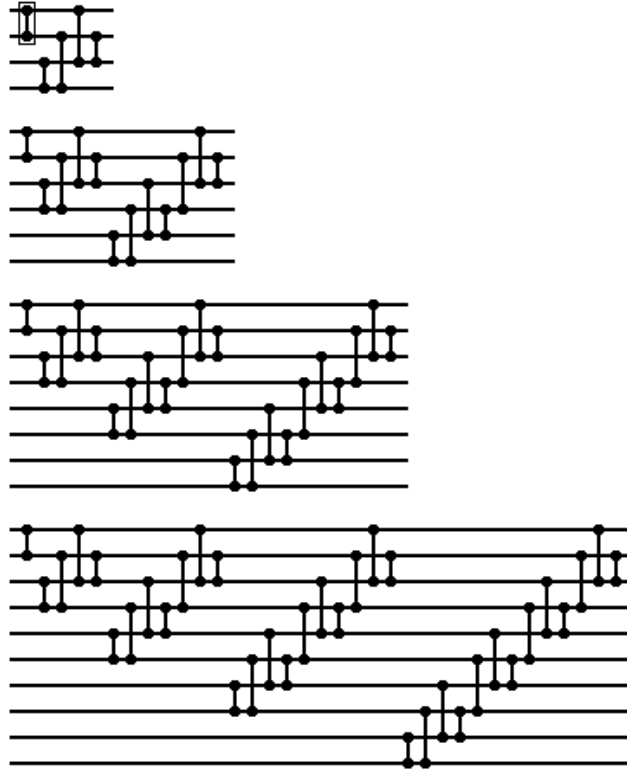
Fig. 5.14. The most comparator-efficient as well as delay-efficient even-input sorting networks created from a two-input embryo using programs g9-2even, g8-2even_2 or g6-2even_2

Program	Instructions	#general
g8-4even	[1 4 1] [0 0 1] [0 2 2] [0 0 1] [1 1 2] [0 3 2] [3 3 0] [3 3 2]	
g8-4even_2	*[1 4 4] [1 2 1] [0 4 3] [0 2 2] [0 3 3] [3 4 1] [0 2 2] [3 1 3]*	41
g8-4even_3	[0 2 2] [0 4 4] [0 3 4] [1 2 3] [1 2 0] [3 4 2] [0 2 2] [3 4 4]	
g7-4even	[1 4 4] [1 2 2] [0 2 2] [0 3 3] [0 3 2] [3 2 0] [3 3 3]	46

Table 5.13. Evolved programs for the construction of even-input sorting networks utilizing a four-input embryo. The parameters of the developmental system are: $ew = 4, de = 2, dw = 2$.

N	6	8	10	12	14	16	18	20	22	24	26	28
conventional	15	28	45	66	91	120	153	190	231	276	325	378
g8-4even	13	24	38	55	75	98	124	153	185	220	258	299
g8-4even_2	*13*	*24*	*38*	*55*	*75*	*98*	*124*	*153*	*185*	*220*	*258*	*299*
	(1)	*(2)*	*(3)*	*(4)*	*(5)*	*(6)*	*(7)*	*(8)*	*(9)*	*(10)*	*(11)*	*(12)*
g8-4even_3	13	24	38	55	75	98	124	153	185	220	258	299
	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)
g7-4even	13	24	38	55	75	98	124	153	185	220	258	299
	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)

Table 5.14. The number of comparators of even-input sorting networks created using a four-input embryo by means of programs given in Table 5.13

N	6	8	10	12	14	16	18	20	22	24	26	28
conventional	9	13	17	21	25	29	33	37	41	45	49	53
g8-4even	9	15	21	27	33	39	45	51	57	63	69	75
g8-4even_2	*6*	*9*	*14*	*19*	*23*	*26*	*31*	*36*	*41*	*46*	*51*	*56*
	(6)	*(9)*	*(12)*	*(15)*	*(18)*	*(21)*	*(24)*	*(27)*	*(30)*	*(33)*	*(36)*	*(39)*
g8-4even_3	7	12	17	22	27	32	37	42	47	52	57	62
	(6)	(9)	(12)	(15)	(18)	(21)	(24)	(27)	(30)	(33)	(36)	(39)
g7-4even	7	11	16	20	24	28	33	37	41	45	49	53
	(7)	(11)	(15)	(19)	(23)	(27)	(31)	(35)	(39)	(43)	(47)	(51)

Table 5.15. Delay of even-input sorting networks created using a four-input embryo by means of programs given in Table 5.13

Program	Instructions	#general
g6-3even	*[0 2 2] [0 1 2] [1 0 1] [0 2 1] [3 3 1] [3 2 4]*	59
g6-3even_2	[0 2 2] [0 1 2] [0 0 1] [1 1 1] [3 4 4] [3 0 1]	

Table 5.16. Evolved programs for the construction of even-input sorting networks utilizing a three-input embryo. The parameters of the developmental system are: $ew = 3, de = 1, dw = 2$.

N	4	6	8	10	12	14	16	18	20	22	24	26	28
Conventional	6	15	28	45	66	91	120	153	190	231	276	325	378
g6-3even	*8*	*16*	*27*	*41*	*58*	*78*	*101*	*127*	*156*	*188*	*223*	*261*	*302*
	(2)	*(3)*	*(4)*	*(5)*	*(6)*	*(7)*	*(8)*	*(9)*	*(10)*	*(11)*	*(12)*	*(13)*	*(14)*
g6-3even_2	9	18	30	45	63	84	108	135	165	198	234	273	315
	(3)	(5)	(7)	(9)	(11)	(13)	(15)	(17)	(19)	(21)	(23)	(25)	(27)

Table 5.17. The number of comparators of even-input sorting networks created using a three-input embryo by means of programs given in Table 5.16
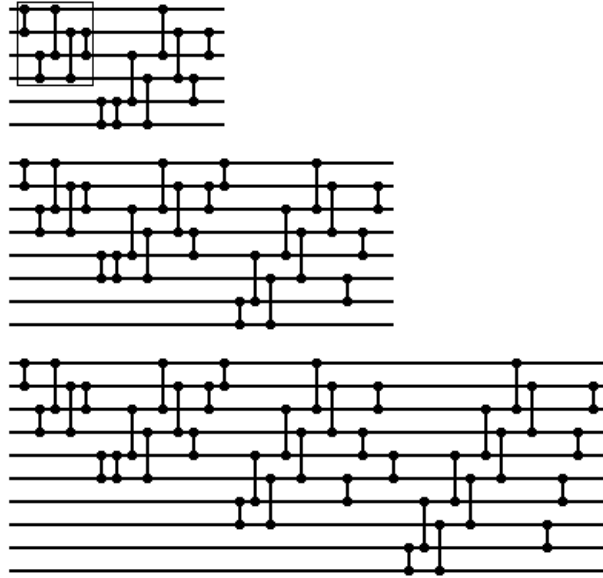
Fig. 5.15. Efficient even-input sorting networks created using the program g8-4even_2
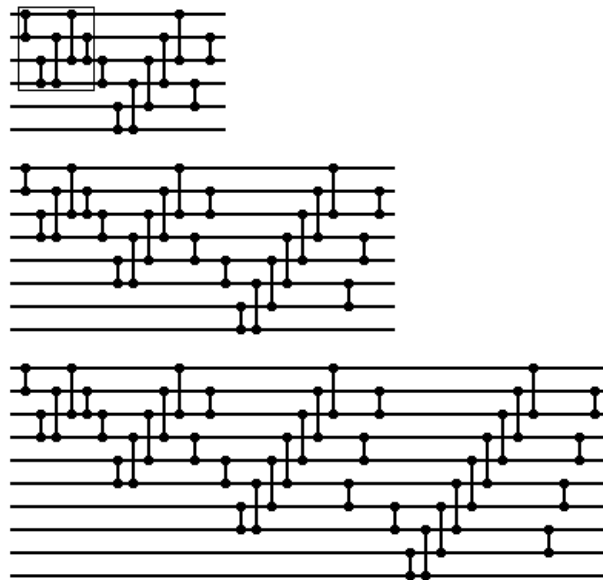
Fig. 5.16. Efficient even-input sorting networks created using the program g8-4even_3

N	**4**	**6**	**8**	**10**	**12**	**14**	**16**	**18**	**20**	**22**	**24**	**26**	**28**
Conventional	5	9	13	17	21	25	29	33	37	41	45	49	53
g6-3even	*6*	*10*	*14*	*18*	*22*	*26*	*30*	*34*	*38*	*42*	*46*	*50*	*54*
	(5)	*(9)*	*(13)*	*(17)*	*(21)*	*(25)*	*(29)*	*(33)*	*(37)*	*(41)*	*(45)*	*(49)*	*(53)*
g6-3even_2	7	12	17	22	27	32	37	42	47	52	57	62	67
	(5)	(9)	(13)	(17)	(21)	(25)	(29)	(33)	(37)	(41)	(45)	(49)	(53)

Table 5.18. Delay of even-input sorting networks created using a three-input embryo by means of programs given in Table 5.16
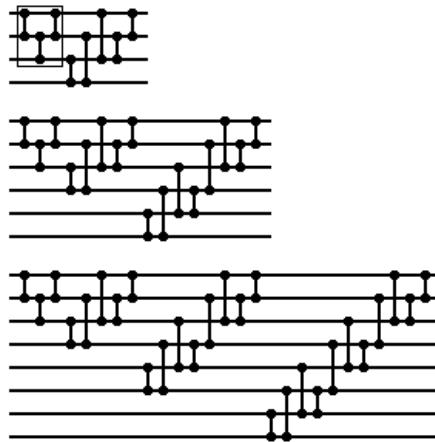
Fig. 5.17. Even-input sorting networks created using the program g6-3even

5.4.4 Improving odd-input sorting networks

The presented evolutionary approach produced sorting networks with better implementation cost (the number of comparators) than the conventional approach for even-input as well as odd-input SNs. Delay of even-input sorting networks was also improved. However, in case of odd-input SNs, none of the presented programs is better than a conventional one in terms of delay.

It is possible to show that the best-known program for even-input sorting networks (*g8-4even_2*) can be utilized to improve delay in case of odd-input networks. Figure 5.18 shows that by removing the bottom line together with "connected" comparators, the odd-input sorting network is established. The improvement of created sorting networks was verified for $N \leq 29$.

5.4.5 Computational effort

More than 10,000 independent runs of evolutionary algorithm were performed. The number of generations needed for gaining a solution varies from about 150 to many thousands. The limit of 10,000 generations showed to be sufficient to

Fig. 5.18. Creating delay efficient odd-input sorting networks from even-input SNs by removing the bottom line of comparators. The original six-input sorting network: (0,1) (2,3) (0,2) (1,3) (1,2) (4,5) (4,5) (2,4) (3,5) (0,2) (1,3) (3,4) (1,2). The new five-input SN: (0,1) (2,3) (0,2) (1,3) (1,2) (2,4) (0,2) (1,3) (3,4) (1,2).

get some solutions in a reasonable time. If the evolution does not terminate successfully within this limit, the evolutionary process is stopped.

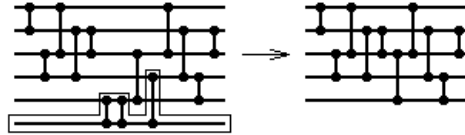For instance, consider even-input sorting networks constructed from a 2-input embryo. In this case, 58% of independent runs of evolutionary process terminated successfully. The average number of generations is 2053. Figure 5.19 shows a typical example of the progress of average fitness of the population along with the rise of fitness value of the best individual during evolution. This experiment worked with a simple genetic algorithm, the crossover probability 0.7, the mutation probability 0.023 and for population size of 60 individuals. The fitness function considered four developmental steps, i.e. the maximum fitness value was $f_{max} = f(4) + f(6) + f(8) + f(10) = 2^4 + 2^6 + 2^8 + 2^{10} = 1376$, where $f(N)$ is the number of all possible binary sequences of ze of N-input SN. As evident from Figure 5.19, there is a good level of evolvability related to this experiment using the proposed approach. Although the success rate was only 58%, more experiments had to be conducted in order to obtain statistically credible results. This was not a problem because of lower time requirements of the evolutionary runs. Similar behavior was observed also in other sets of experiments.

5.4.6 Summary and discussion

Let us summarize the results for each category and discuss advantages and potential problems of the proposed approach.

Sorting networks with arbitrary number of inputs

It is easy to evolve a general program in this category. The principle of the straight insertion algorithm was rediscovered. However, sorting networks constructed by means of these programs are not efficient because many comparators are required. The results obtained in this category confirm Hypothesis 1 because a general solution of a given problem has been evolved using the continual development approach and partially also Hypothesis 3 because the conventional general principle of straight-insertion sort has been rediscovered.
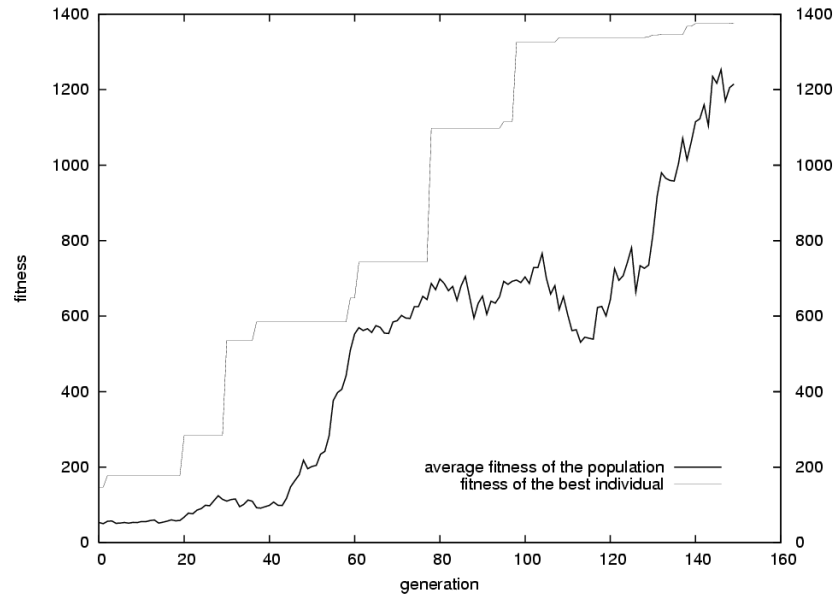
Fig. 5.19. The best and average fitness value in a typical run of a simple GA for the following settings: $ew = 2, de = 2, dw = 2, p_c = 0.7, p_m = 0.023$, 60 individuals in population, 4 developmental steps for fitness calculation ($f_{max} = 1376$)

Odd-input sorting networks

Some programs were evolved that produce smaller sorting networks (in terms of the comparator count) than the conventional insertion and selection method can offer. However, the method works only using a four-input embryonic network. The next improvement can be done by removing redundant comparators that are often generated by the evolved programs. We were not able to improve delay in this category – the best program has reached the quality of the conventional methods. Surprisingly, it is possible to modify the best even-input sorting networks in order to obtain odd-input sorting networks whose delay is shorter than delay of conventional networks.

Even-input sorting networks

In this category various types of embryos have generated interesting results. The usage of the two-input embryo has led to a substantial reduction of the number of comparators and a small reduction of delay. The programs evolved from a two-input embryo did not produce redundant comparators. On the

other hand, the programs g8-4even_2 and g8-4even_3, evolved using a four-input embryo, minimize the number of comparators as well as delay substantially. However, first, it is necessary to remove redundant comparators from the created networks. These programs represent the best solutions obtained for this problem.

It was clearly demonstrated that the proposed evolutionary method combined with the instruction-based development is able to design generic sorting networks and in some cased it is even possible to find an improved solution in comparison with the conventional design. Therefore, the results from the categories of odd-input and even-input sorting networks development confirm Hypothesis 1 and also Hypothesis 3 because, in addition to having already rediscovered a general conventional solution, an innovative general solution has been invented.

All candidate programs were evaluated using the zero–one principle; however, only for a limited number of inputs. We found this approach very efficient because about 50% of the candidate programs are considered as "general" (see the #general columns in the previous tables). Although the word "general" is used, it is obvious that the evolved programs may not be really general – the verification method that was applied (i.e. the evaluation of a program up to a sufficiently high N) is not a proof. Therefore, the best evolved solution will be analysed theoretically together with a demonstration of its generality by the mathematical induction in Section 5.6.

The main feature of the proposed developmental approach is that a lot of problem-domain knowledge (such as the definition and the way of execution of the instructions, comparator definition and its utilization as a basic building block) has been supplied to the evolutionary system by the designer. However, the innovative solutions represent a very interesting outcome. Although they do not reach the qualities of the best known solutions for a given N, the best evolved program gave rise a new general sorting network construction method that exhibits substantially better properties in comparison with the conventional approaches of the same type (i.e. the insertion or selection principles).

Except the instructions that were designed for this particular application manually and that GA had to put them together to make a program, the developmental scheme has utilized another information – the size of the currently constructed network (i.e. the number of its inputs N). This information is not a part of our artificial genetic code; it is controlled externally by the algorithm controlling the development. Therefore, it can be understood as a property of environment, which surrounds the growing sorting network.

The approach presented in the previous sections belongs to the first experiments involving the development for the design of generic circuit structures. It is a case of an application-specific model that is mostly inspired by the conventional generic design. Though several innovations were discovered. The results shows that the concept of instruction-based development applied at the

level of circuit structure is worth of future investigation not only in the sorting networks domain. Specifically, a research was conducted in the development of generic circuits consisting of polymorphic gates and generic combinational multipliers which have not been investigated so far in the evolutionary design field (see the next chapters).

5.5 Another developmental approach to sorting networks

During the next research in the area of developmental systems for the purposes of this work, a novel instruction-based developmental model was introduced for the evolutionary design of generic structures of combinational multipliers (see Chapter 6). The continuation of this kind of research led to the experimentation with that concept also in other areas, including the field of sorting networks.

This phase of research is especially motivated by the aim of advanced optimization of the sorting networks developed in Section 5.3. Considering the innovative solutions obtained for size 2 of the developmental step in comparison with the conventional approach, where the developmental step possesses size 1, it suggests that by increasing the size of the developmental step more efficient sorting networks might be developed. However, no other values of this parameter have led to successful designs in Section 5.4 using the developmental model from Section 5.3. So that is the reason for investigation of the new instruction-based developmental approach in the area of sorting networks.

5.5.1 Concept of an advanced developmental system

Although the developmental encoding utilized in these experiments is based on the approach presented in Section 6.2, which is very similar, it will be described in detail herein, adjusted for purposes of the design of generic sorting networks. The reason is that the sorting networks possess totally different structure which influence some key features of the developmental model. In addition, the detailed description will make this section self-explanatory.

The proposed approach differs from the previous developmental model introduced in Section 5.3 in the following aspects: (1) The form of the embryo does not influence the form of the developed SNs. (2) An advanced concept of instruction-based development is applied. More complex instruction set is considered. The program to be evolved may use internal variables integrated in the system interpreter. Loops are specified explicitly by an appropriate instruction. (3) The building blocks (comparators) are generated by means of a special instruction instead of copying the existing comparators. The placement of the newly created comparators is specified internally using internal variables of the developmental system.

The construction of the circuit is performed using a single developmental program, which is the subject of evolution. The instructions of the program

are executed sequentially according to the program pointer (pp). In addition to the approach introduced in Section 5.3, the instructions make use of numeric literals $0, 1, \ldots, max\_value$, where max_value is specified by the designer at the beginning of the evolutionary process. In addition to the numeric literals, a set of variables integrated into the developmental system may be utilized. The variables are denoted symbolically $v_0, v_1, v_2, \ldots$ and their values are altered by the appropriate instructions during the execution of the program.

Table 5.19 describes the instruction set utilized for the development. The SET instruction assigns a value determined by a numeric literal or another variable to a specified variable. Instructions INC/DEC are intended for increasing/decreasing the value of a given variable (specified in first argument) by a given numeric literal (specified in second argument). Simple loops inside the developmental program are provided by the REP instruction whose first argument determines the repetition count and the second argument states the number of instructions after the REP instruction to be repeated. Inner loops are not allowed, i.e. REP instructions inside the repeated code are interpreted as NOP (no operation) instructions. The GEN instruction generates a comparator of the type specified in its argument.

Instruction	Arguments	Description
0: SET	$variable, value$	Assign a $value$ to a $variable$.
1: INC	$variable, value$	Increase the value of a $variable$ by a given $value$ specified as a numeric literal.
2: DEC	$variable, value$	If $variable \geq value$, then decrease the value of $variable$ by the $value$ specified as a numeric literal.
3: REP	$count, number$	Repeat $count$-times $number$ following instructions. $count$ is a variable and $number$ is a numeric literal. REP instructions in the repeated code are interpreted as NOP instructions (no inner loops allowed).
4: GEN	$block$	Generate block of type $block$ on the actual position (row, col); increase col by 1.
5: NOP		An empty operation.

Table 5.19. Instructions utilized for the advanced development of generic sorting metworks

The following application-specific setup was utilized for the advanced design of the sorting networks. Again, the sorting networks are intended to develop continually (i.e. to "grow") in developmental steps realized as iterative applications of the evolved program. During each developmental step, some comparators are generated next to the existing ones in order to create the larger SN. There are six variables inside the developmental system which may be utilized by the instructions during the program execution. Four types of comparators are utilized as basic building blocks in the evolutionary process. These comparators differ in their "width", i.e. the number of wires of the

sorting network they are connected over - see Figure 5.20a. The structure for
the construction of the sorting networks consists of a one-dimensional array
in which each element can contain one comparator (see Figure 5.20b). A 2-
input, 3-input or 4-input embryo will be utilized for the development of the
sorting networks as shown in Figure 5.20c). The embryo is stored in the first
e elements of the array depicted in gray. The embryo is invariable during the
evolutionary and developmental process. The position of the comparators in
the sorting network (i.e. the connection to the particular wires) is specified
by the value a given variable. For example, consider the comparator of width
2 from Figure 5.20a. Let its position be determined by the variable $v_0 = 1$.
Then the first input of this comparator is connected to the wire 1 and the
second input to the wire 3. During the development of a sorting network the
comparators are generated sequentially into the free positions of the array
pointed by the index pointer p according to the program, which is the subject
of evolution. Note that the comparator is generated only if p does not exceed
the array boundary of L elements and the connection of the comparator does
not exceed the width (the number of inputs) of the sorting network being
developed. Before proceeding a developmental step, the values of variables
are initialized to the following values which were determined with respect to
the trait of the sorting networks design and showed to be suitable for the
development: $v_0 = 0, v_1 = w - 2, v_2 = w - 2, v_3 = w - 3, v_4 = w - 4, v_5 = w$,
where w denotes the number of inputs (width) of the sorting network to be
developed in the forthcoming developmental step. Note that the initial values
of the variables may be changed during the development (execution of the
program) by the appropriate instructions.
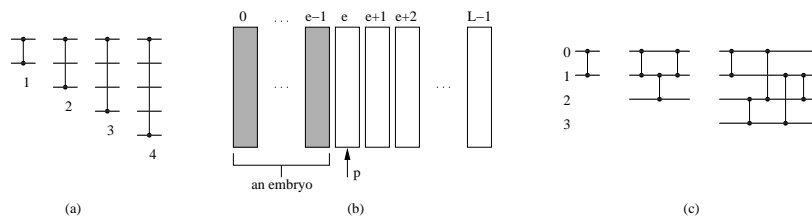
Fig. 5.20. The concept of development of generic sorting networks: (a) The building
blocks are represented by the comparators of the width 1, 2, 3 and 4 respectively.
(b) The array where the building blocks are generated to during the development.
The gray elements denote the building blocks of the embryo. (c) The embryos of the
width 2, 3 and 4 respectively, utilized for the development.

5.5.2 Evolutionary system setup

A simple genetic algorithm was utilized in combination with the developmen-
tal model described in Section 5.5.1. The population of the GA consists of

32 constant-length chromosomes, each of which represents a single program possessing eight instructions. The chromosomes are initialized randomly before the evolution starts. The genetic operators of selection, mutation and crossover are the same as described in Section 6.2.2 and illustrated by Figure 6.5.

The candidate solutions are evaluated in a simulator typically for three developmental steps, i.e. for each chromosome in the population a finite developmental sequence is created consisting of three sorting networks of different sizes depending on the width of the embryo and the size of the developmental step. The principle of evaluation is the same as stated in Section 5.3.4.

5.5.3 Experimental Results and Discussion

The experiments were carried out using the same hardware configuration as described in Section 5.3.4. Three types of experiments will be presented which differ in the size of the developmental step and the embryo utilized.

The first type of experiments was focused on the development of arbitrary even-input SNs from a two-input embryo when the size of the developmental step was set to 2. The 1000 of independent experiments were conducted from which 90 % finished successfully in 40000 generations of the evolutionary algorithm and 98 % of the evolved programs were classified as general.

Figure 5.21a shows a sequence of sorting networks developed by the three steps of the best evolved program which is shown in Figure 5.21b. The sorting networks have been constructed from a two-input embryo by the following process. Let $w_e = 2$ denote the number of inputs of the embryo, $s = 2$ denote the size of the developmental step and $w = v_e + i \cdot s$ denote the width of the sorting network to be developed in the i-th developmental step. Recall that the values of appropriate variables involved in the evolved program are initialized as $v_1 = w-2, v_2 = w-2, v_3 = w-3$. For the first developmental step $(i = 1)$, $w = 2 + 1 \cdot 2 = 4$, therefore, $v_1 = 2, v_2 = 2$ and $v_3 = 1$. Considering these initial values the first instruction 1 from Figure 5.21b generates the comparator of width 1 labeled as a in Figure 5.21a which is connected to the wires denoted by indices 2 and 3. The instruction 2 initiates a loop repeating 2 times (since $v_1 = 2$) the two following instructions. During the first pass of this loop, the instruction 3 generates comparator b whose connection to the SN is determined by $v_3 = 1$ and the instruction 4 decreases v_3 by one, i.e. $v_3 = 0$ at the moment. Similarly, comparator c is generated in the second pass of the loop considering the actual value of v_3. Note that negative values are not allowed, therefore, the execution of the instruction 4 during the second pass of the loop has no effect in this step. Note, however, that the instruction 4 does not represent a true intron from the point of view of LGP because this instruction takes effect in other phases of the loop. Instruction 5 generates comparator d with respect to the value of $v_2 = 2$. Instruction 6 initiates a loop to be repeated 2 times (since $v_1 = 2$) and the two following instructions 7 and 8 result in generating comparators e, f in each pass of this loop. The first developmental

step is now finished. At the beginning of the second developmental step the
variables are initialized to the new values with respect to actual w. During the
second developmental step comparator g is generated by the instruction 1 and
comparators from h to k are generated by the loop initiated by instruction
2. Then comparator l is generated by instruction 5 and comparators $m - p$
are generated by the loop initiated by instruction 6. The next developmental
steps proceed in the same way the consequence of which is the "growth" of
the sorting network. Note that this program was successfully verified for the
construction of up to 28-input sorting network, i.e. it is considered as general.
However, the analysis of the developed sorting networks indicates that there
are redundant comparators in these networks which can be removed without
the loss of its functionality. Therefore, these sorting networks are optimized
both from the point of view of the number of comparators and delay. Note
that the redundant comparators are crossed in Figure 5.21.

Fig. 5.21. (a) Even-input sorting networks developed from a two-input embryo
using the developmental step of size 2. The crossed comparators are redundant in
the sorting network, therefore, they can be removed from the network without any
loss of its functionality. (b) The evolved general program for the development of
these SNs.

In the second type of experiments sorting networks were developed from
a three-input embryo considering the size of the developmental step $s = 3$.
Therefore, 6-input, 9-input, 12-input etc. sorting networks could be designed
by means of the evolved programs. From 1000 independent experiments con-
ducted, 88 % of working programs were evolved from which 99 % were classified
as general. Figure 5.22 shows the best and most interesting result obtained in
this set of experiments. The evolved program, which was classified as general,
produces sorting networks without any redundant comparators. Moreover,
there are both even-input and odd-input sorting networks in a single devel-
opmental sequence (because of the size of the developmental step $s = 3$).
This result represents the first case of observing such a behavior that was
not achieved in the developmental system introduced in [SB05]. Note that the
structure of the sorting networks and the evolved program is very similar in

76   5 Continual development of arbitrarily large sorting networks

comparison with that shown in Figure 5.21. In addition, the algorithm from
Figure 5.21 (without any modifications) showed the ability to construct sort-
ing networks with the size of the developmental step $s = 3$. The only difference
is the dashed line drawn comparator shifted before its predecessor in each de-
velopmental step (caused by different variable in instruction 5 determining
the connection of the comparator to be generated, see Figure 5.22) which,
however, results in better delay in comparison with the solution constructed
by means of the program from Figure 5.21b.

1:    GEN C1 v2
2:    REP v2 2
3:    GEN C2 v3
4:    DEC v3 1
5:    GEN C1 v0
6:    REP v2 4
7:    GEN C1 v2
8:    DEC v2 1

(a)   (b)

Fig. 5.22. (a) Sorting networks developed from a three-input embryo using the de-
velopmental step of size 3. (b) The evolved general program. Note that this solution
constructs sorting networks without any redundant comparators.

The goal of the third type of experiments was to develop arbitrary even-
input sorting networks considering the size of the developmental step $s = 4$
and a four-input embryo. Since there are 8-input, 12-input etc. sorting net-
works in the developmental sequence considering the four-input embryo and
the developmental step of size 4, only two developmental steps were performed
for the fitness calculation because of very time-consuming evaluation of such
large sorting networks. The 500 if independent experiments were conducted
from which 34 % evolved a working general solution, i.e. 100 % successfulness
of the evolved programs. Figure 5.23 shows one of the best evolved program
together with the optimized sorting networks developed by means of it.

Tables 5.20 and 5.21 summarize the number of comparators and delay of
the developed sorting networks and their optimized variants for selected num-
bers of inputs of the sorting networks. It is evident that all the SNs presented
in this section (Figs. 5.21, 5.22 and 5.23) exhibit better properties from the
both point of view of the number of comparators and delay in comparison with
the general conventional principle of the same type (straight-insertion sort).
Note that the optimized sorting networks created using the developmental
step of size 2 corresponds to the best results developed in Section 5.4.3. More-
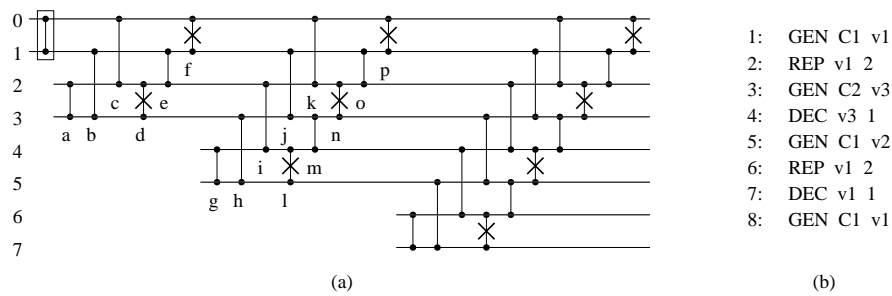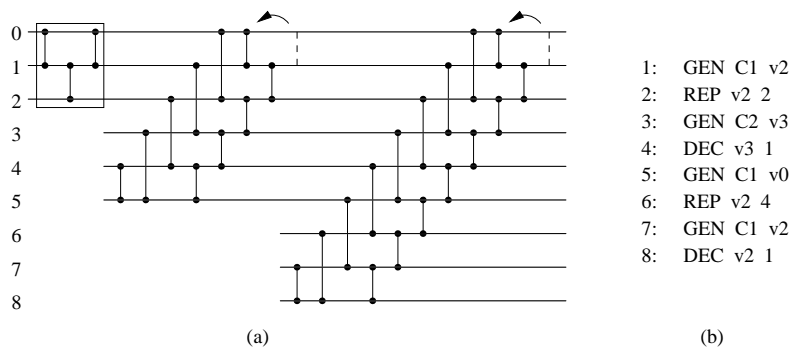over, general programs were evolved herein for the developmental step of sizes
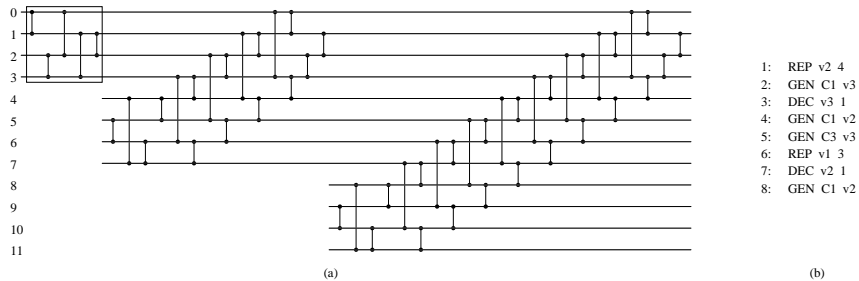
Fig. 5.23. (a) Even-input sorting networks developed from a four-input embryo using the developmental step of size 4. Note that only the effective (non-redundant) comparators are shown. (b) The evolved general program.

3 and 4 that we were not able to achieved in our original approach. These networks also exhibit better properties in comparison with the conventional solution. In case of the step of size 3 a general program was evolved which even constructs sorting networks without any redundant comparators. The results presented in this section suggest that this instruction-based developmental model is more robust and flexible in comparison to the system introduced in the previous section.

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

Table 5.20. The number of comparators of the evolved sorting networks for different sizes of the developmental step in comparison with the conventional straight-insertion sorting networks. The *italic* values represent the number of comparators of the optimized sorting networks (after removing the redundant comparators). Note that the sorting networks created using the developmental step of size 3 do not contain any redundant comparators.

Despite the successful development considering the size of the developmental steps of values 3 and 4, no better solution was discovered in comparison with the best networks obtained in Section 5.3. The best results obtained by means of the advanced developmental model exhibit the same properties as observed before. "Wider" comparators were included in the set of building blocks, which were not applied very often by the developmental programs. It is surprising, one would rather expect that these comparators will make

78   5 Continual development of arbitrarily large sorting networks

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

Table 5.21. Delay of the evolved sorting networks for different sizes of the developmental step in comparison with the conventional straight-insertion sorting networks. The *italic* values represent the delay of the optimized SNs (after removing the redundant comparators). Note that the sorting networks created using the developmental step of size 3 do not contain any redundant comparators.

the development easier and will optimize the final design. However, comparators arrangements similar to the sorting networks known from Section 5.3 were generated instead in many cases. Most of the developed networks contains a high number of redundant comparators which is probably caused by the generative trait of the proposed model (i.e. the comparators have been created explicitly from the set of building blocks and placed into the circuit by a special instruction). The amount of redundant comparators was often several tens even for sorting networks containing less than 20 inputs.

In general, however, a progress has been observed against the previous method. We have been able to develop sorting networks which can grow "faster". This has not been accomplished before. The "wider" comparators and larger size of developmental steps lead to more combinations of the comparators arrangement for potentially more effective solutions (such the comparators are common in the best sorting networks designed for a particular number of inputs). It is evident that further research is needed in this domain.

5.6 Theoretical understanding of the best evolved solution

In Section 5.4.3 an evolved construction algorithm for the design of arbitrary even-input sorting networks was presented. This results represents a rare case in the area of evolutionary design with development when the EA has been able to discover an innovative solution in comparison with the conventional approach.

In this section theoretical aspects of the best evolved solution are presented. The SNs created by means of that evolved algorithm will be analysed and their properties compared to the sorting networks constructed using the conventional insertion/selection principle. The sorting networks will be described mathematically and the generality of the new construction algorithm
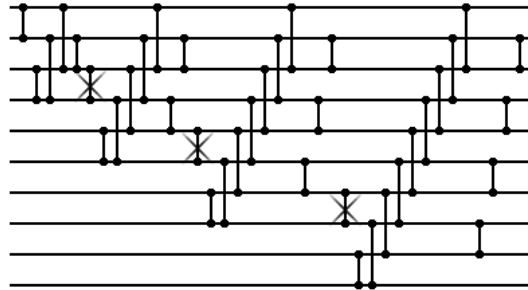
Fig. 5.24. 10-input instance of sorting network developed by means of the best evolved program

will be proven formally, similarly to proving the properties of human inventions in the area of theoretical computer science.

5.6.1 Invention of a new construction method for SNs

Figure 5.24 shows 10-input SN created by means of the best evolved program from a 4-input embryo for three developmental steps. Specifically, a 6-input sorting network was developed from a valid 4-input (embryonal) network, an 8-input SN was developed from the previously created 6-input network and so on. However, the resulting sorting network contains redundant comparators which are crossed in Figure 5.24. They can be removed from the sorting network without any loss of its functionality. After their removal, the delay of the sorting network decreases substantially in comparison with its original form and, in addition, in comparison with the conventional insertion/selection principle too. Moreover, the resulting networks possess lower number of comparators against the conventional solution. Since the sorting network has been required to be fully functional after each developmental step during the evolution, the developmental process can be generalized for arbitrary even number of inputs N. Then, each larger $N + 2$-input SN created by a single application of the evolved program actually accepts an N-input sequence and two newly added inputs. The N-input sequence has already been sorted by the N-input network created in the previous developmental step. The values of the new inputs have to be inserted at proper positions of that sorted sequence which is ensured by the suitable arrangement of comparators appended to the $N + 2$-input sorting network being developed.

The step of removing the redundant comparators and the concept of development taken into account, as summarized in the previous paragraph, constitute crucial issues of inventing a new construction method for arbitrarily large sorting networks as illustrated in Figure 5.25b. For comparison, variant (a) of this picture shows the conventional principle of the straight-insertion sort.
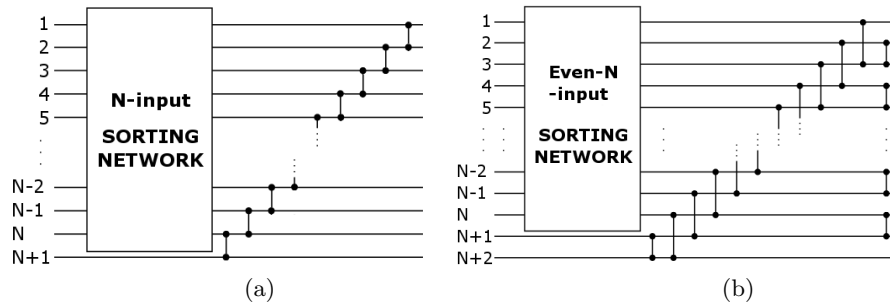
(a)   (b)

Fig. 5.25. General methods of building larger sorting networks by adding some comparators into smaller sorting networks: (a) straight-insertion method, (b) new method based on the best solution evolved in this book

5.6.2 Analysis of the best evolved sorting networks

The evolved method is based on the same idea as the conventional insertion or selection principle – creating a larger sorting network from a smaller SN by appending a suitable arrangement of comparators. Unlike the insertion or selection algorithm, the evolved program constructs $(N + 2)$-input sorting network from an even-N-input SN. Thus there are some restrictions in the design process in comparison with the conventional approaches. However, the sorting networks obtained by means of the evolved approach exhibit better properties in terms of both the number of comparators and delay than the conventional sorting networks. Moreover, the evolved method allows to create more efficient networks from existing arbitrary even-N-input network in comparison with the case of twofold application of the conventional approach. For example, it is possible to construct an 18-input SN from the best known 16-input SN and the resulting network will exhibit better properties than the network created by means of the insertion or selection principle. Specifically, consider the number comparators 60 and delay 10 of the best currently known 16-input sorting network (as stated in Table 5.1 in Section 5.2). Then the 18-input SN constructed from this 16-input SN by means of double application of the conventional principle will possess 93 comparators and the delay 28. However, the 18-input SN created from the same 16-input SN using the evolved method will possess only 85 comparators and the delay 20. Therefore, the difference of both the number of comparators and delay of these 18-input SNs is 8.

The developed sorting networks were analysed and compared to the appropriate instances obtained by the conventional algorithm of the same type – the straight insertion principle. The overview of basic parameters (the number of comparators and delay) for up to 28 inputs of the networks are summarized in Table 5.22. On the basis of the data from Table 5.22, the following equations were derived which determine the number of comparators, respective delay of sorting networks for general $N \geq 4$ – see equation (5.1), respective (5.3).

Similar formulas are known for the conventional solution – see equations (5.2), respective (5.4)[Knu98].

$$C(N)_{\text{evol}} = \frac{3}{8}N^2 - \frac{1}{4}N = \mathcal{O}(N^2) \tag{5.1}$$

$$C(N)_{\text{conv}} = \frac{1}{2}N^2 + N = \mathcal{O}(N^2) \tag{5.2}$$

$$D(N)_{\text{evol}} = \frac{3}{2}N - 3 = \mathcal{O}(N) \tag{5.3}$$

$$D(N)_{\text{conv}} = 2N - 3 = \mathcal{O}(N) \tag{5.4}$$

The number of comparators (i.e. area complexity) and delay (i.e. time complexity) of the sorting networks are shown in Figure 5.26. Although the asymptotic complexities are identical in case of both evolved and conventional solution, the evolved algorithm constructs sorting networks with better properties (the number of comparators and delay) than the conventional insertion or selection principle.

#inputs of the SN	6	8	10	12	14	16	18	20	22	24	26	28
Evolved SN: #comparators	13	24	38	55	75	98	124	153	185	220	258	299
Evol. SN: #redund. comp.	1	2	3	4	5	6	7	8	9	10	11	12
Evol. SN: #effective comp.	12	22	35	51	70	92	117	145	176	210	247	287
Conventional SN: #comp.	15	28	45	66	91	120	153	190	231	276	325	378
Evolved SN: delay	7	12	17	22	27	32	37	42	47	52	57	62
Evol. SN: effective delay	6	9	12	15	18	21	24	27	30	33	36	39
Conventional SN: delay	9	13	17	21	25	29	33	37	41	45	49	53

Table 5.22. Analysis of the parameters of the best evolved general solution in comparison with the conventional insertion or selection sorting networks. Note that the effective delay denotes delay of the SN after removing the redundant comparators.

5.6.3 Is the evolved method general?

In order to demonstrate formally that the evolved solution is really general (i.e. it is able to create theoretically infinitely large sorting network), some definitions of basic terms will be introduced which in fact constitute a mathematical model of a sorting network. These definitions are inspired by the formal model of the sorting network stated in [Lan06]. After that a formal proof is proposed demonstrating the generality of the new construction method for the sorting networks.

Definition 1 *Let $I = \{1, 2, \ldots, N\}$ be an index set and let A be a set with an order relation $\leq$. A data sequence is a mapping $a : I \to A$. The set of all data sequences of length N over A is denoted by A^N.*
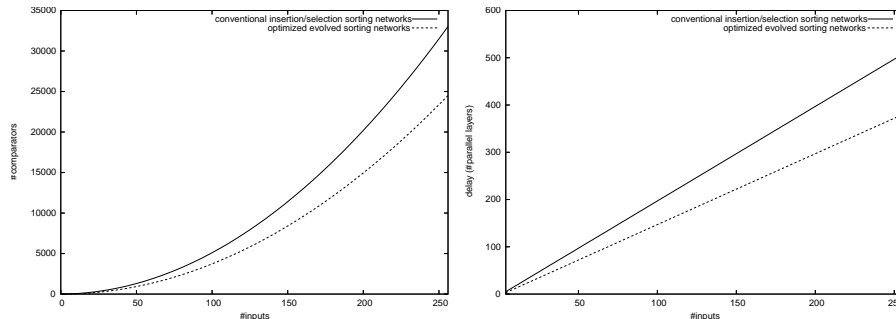
Fig. 5.26. Area complexity and delay complexity of the evolved sorting networks in comparison with the conventional insertion/selection principle

Definition 2 *The sorting problem consists of reordering an arbitrary data sequence $a_1, a_2, \ldots, a_N$, $a_i \in A$ for $i = 1, 2, \ldots, N$, to a data sequence $a_{\Phi(1)}, a_{\Phi(2)}, \ldots a_{\Phi(N)}$ such that $a_{\Phi(i)} \leq a_{\Phi(j)}$ for $i < j$, where Φ is a permutation of the index set $I = \{1, 2, \ldots, N\}$.*

The definition of a sorting network is based on comparator networks introduced in [Knu98]. Herein a comparator $[i; j]$ is considered as a circuit element that sorts the i-th and the j-th element of a data sequence into the nondecreasing order.

Definition 3 *A comparator is a mapping $[i; j] : A^N \to A^N$, $i, j \in \{1, 2, \ldots, N\}$, where $[i; j](a)_i = min(a_i, a_j)$, $[i; j](a)_j = max(a_i, a_j)$, $[i; j](a)_k = a_k$ for all $k \neq i$, $k \neq j$, $i < j$ and for all $a \in A^N$.*

The formal notation of the comparator can be simplified as follows. Let $[i; j]$ be a comparator applied to a data sequence $a \in A^N$. If $[i; j](a)_i = a_j$ and $[i; j](a)_j = a_i$, then it is said the comparator $[i; j]$ swaps a_i with a_j in a.

Definition 4 *A parallel layer, S, is a composition of comparators $S = [i_1; j_1] \cdot [i_2; j_2] \cdot \ldots \cdot [i_k; j_k]$, $k \geq 0$ such that i_r and j_s are distinct for all $i_r = 1, \ldots, N-1, j_s = 2, \ldots, N, i_r \neq j_s$, for all $r = 1, \ldots, k, s = 1, \ldots, k, r \neq s$. Comparators within a parallel layer are executed in parallel.*

Definition 5 *A comparator network is a composition of parallel layers.*

Note that the orientation of a comparator in a comparator network is important. We assume that every $[i; j]$ satisfies $i < j$. That being supposed, if $a_i > a_j$, then $[i; j]$ swaps a_i with a_j in $a \in A^N$. Moreover, the order of parallel layers in a comparator network is important since it defines the reordering algorithm. However, the order of the comparators within a parallel layer is not important because they are independent of each other.

Definition 6 *A sorting network is a comparator network that sorts all the data sequences correctly.*

Because of uniformity in terminology with respect to the previous sections, the term sorting network will be used in the next paragraphs even before proving the correctness of the appropriate comparator network.

Definition 7 *Let S be a sorting network with even number of inputs N, $N = 2k$, $k \geq 0$. Let us define k to be the degree of S.*

Since the zero–one principle is applied for testing the correctness of the sorting networks, the elements of a data sequence can contain only binary values, i.e. $a_i \in \{0, 1\}$ for all $i = 1, \ldots, N$.

Considering the generalized approach to the development of sorting networks introduced in Section 5.6.1, the following theorem can be formulated.

Theorem 1

Every arbitrary k-degree sorting network can be used as a base for constructing $(k + 1)$-degree sorting network by appending $3k + 1$ comparators, specifically $[2k+1; 2k+2]$, $[2k; 2k+2]$, $[2k-1; 2k+1]$, ..., $[1; 3]$ and $[2; 3]$, $[4; 5]$, ..., $[2k; 2k + 1]$ (the arrangement of the comparators corresponds to the evolved structure that is shown in Figure 5.25b).

Proof

Theorem 1 will be proven by induction on the degree of sorting network, i. Recall that zero–one principle is applied in this proof.

Basis

Let the degree $i = 0$. Increasing i by one, a 2-input SN is obtained containing just one comparator, specifically $[1; 2]$. The proof of its correctness follows directly from Definition 3.

Induction hypothesis

Assume that Theorem 1 holds for all $i \leq k$, where k is a positive integer.

Induction step

Consider an arbitrary k-degree sorting network and use Theorem 1 to create $(k + 1)$-degree sorting network. According to the induction hypothesis, the obtained sorting network is correct. Let z denote the number of 0's contained in the data sequence $a_1 a_2 \ldots a_{2k}$. Since the k-degree sorting network is correct, it produces non-decreasing data sequence of z 0's followed by $2k - z$ 1's. It has to be proven that the comparators $[2k+1; 2k+2]$, $[2k; 2k+2]$, $[2k-1; 2k+1]$,

84   5 Continual development of arbitrarily large sorting networks

..., $[1; 3]$ and $[2; 3]$, $[4; 5]$, ..., $[2k; 2k + 1]$ appended by the program are able to put all the possible binary combinations of elements a_{2k+1} and a_{2k+2} of the data sequence into their proper positions. The situation is illustrated in Figure 5.27.
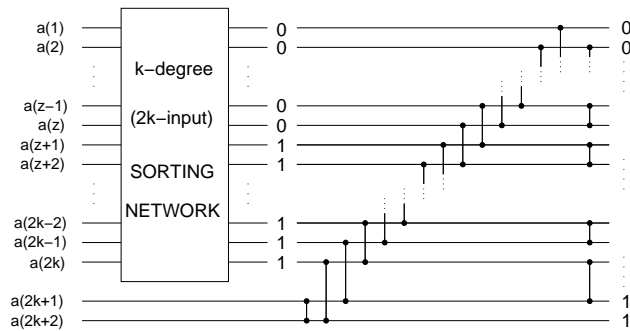

Fig. 5.27. The principle of creating $(k+1)$-degree sorting network from a k-degree SN

a) $a_{2k+1} = 0$ and $a_{2k+2} = 0$
   The situation is illustrated in Figure 5.28a. Consider $0 \leq z < 2k$. Observe that comparators $[2k; 2k+2]$, $[2k-1; 2k+1]$, ..., $[z+1; z+3]$ successively swap the input values processing zero-elements a_{2k+1} and a_{2k+2} of the data sequence. There are only zero-inputs in comparators $[z; z+2]$, $[z-1; z+1]$, ..., $[1; 3]$ so their execution has no effect. Similarly, none of comparators $[2; 3]$, $[4, 5]$, ..., $[2k; 2k+1]$ needs to swap its inputs. If $z = 2k$, then the data sequence has been already sorted.

b) $a_{2k+1} = 0$ and $a_{2k+2} = 1$
   The situation is illustrated in Figure 5.28b. Let $0 \leq z < 2k$. Comparators $[2k-1; 2k+1]$, $[2k-3; 2k-1]$, ..., $[z + (z \bmod 2) + 1; z + (z \bmod 2) + 3]$ successively swap the input values processing zero-element a_{2k+1} of the data sequence. If z is odd and $0 < z < 2k$, then the comparator $[z+1; z+2]$ swap its inputs to finish the resulting order of the data sequence. If $z = 2k$, then the data sequence has been already sorted.

c) $a_{2k+1} = 1$ and $a_{2k+2} = 0$
   Consider $0 \leq z < 2k$. Observe that after swapping a_{2k+1} with a_{2k+2} by the comparator $[2k + 1; 2k + 2]$, the sorting proceeds in the same way as in the case (b); consider Figure 5.28b with $a_{2k+1} = 1$ and $a_{2k+2} = 0$.

d) $a_{2k+1} = 1$ and $a_{2k+2} = 1$
   In this case, the data sequence has already been sorted.

Observe that for every combination of values a_{2k+1} and a_{2k+2}, the $(k+1)$-degree sorting network has processed all the data sequences correctly, i.e. the evolved approach is general.
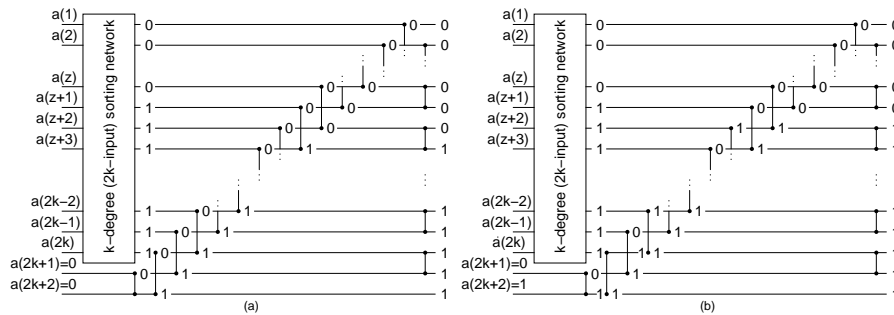
Fig. 5.28. Processing data sequences: (a) $a_{2k+1} = 0, a_{2k+2} = 0$, (b) $a_{2k+1} = 0, a_{2k+2} = 1$

QED

As it has been proven, the evolved approach forms an improved general method which is similar to the conventional insertion or selection principle. Although it is focused primarily on the construction of even-input SNs, the resulting circuits can be reduced to odd-input networks, which retain better properties in comparison with conventional sorting networks as demonstrated in Figure 5.18. The proof proposed in this section represents a formal confirmation of Hypothesis 3 because the innovative solution invented by the evolution has been demonstrated to be general. Note that the same principle (mathematical induction) can be applied to the rest of the innovative results presented in this chapter. However, no more proofs will be stated herein because their construction is straightforward. Note that the generality of the rediscovered conventional solutions follows from their known properties published in the appropriate literature (e.g. [Knu98]).

5.7 Summary of the chapter

In this chapter a concept of continual development will be introduced which allows the target object to "grow" while keeping its full functionality all the time during the development. In order to demonstrate abilities of this approach, sorting networks will be chosen as a suitable domain for the evolutionary design combined with the continual development.

A developmental method was described that enables us to create larger sorting networks from smaller ones by means of a developmental program which is a subject of evolution. The sorting network can grow from a trivial initial solution (an embryo) to theoretically arbitrary size. The approach was inspired by the conventional construction algorithms for the arbitrarily large sorting networks – insertion and selection principle known from the theory of sorting. A set of experiments was performed concerning the development of generic sorting networks based on this method. Moreover, the aim was to

optimize the parameters of the resulting solutions (the number of comparators and delay of the SNs).

First the conventional principle of straight insertion algorithm was rediscovered by means of genetic algorithm endowed with the continual development, utilizing the size of developmental step 1. Later, developmental steps of size 2 were applied and the experiments were focused on the development of either only even-input or odd-input sorting networks, applying different embryos in each category. More efficient principles (programs) were discovered for the development of arbitrarily large even-input and odd-input SNs from the point of view of the number of comparators and delay in comparison with the conventional principles of the same type (i.e. insertion or selection sort). The best evolved solution was analyzed and a new general sorting network construction principle was introduced. Its generality was proved formally by means of mathematical induction.

The reported research represents the rare case in which a new scalable principle is discovered by an evolutionary algorithm. In most cases, evolutionary algorithms are being used to find a single suitable solution. A method was introduced for discovering an innovative solutions for all instances of the problem.

Note that Hypotheses 1 and 3 have been confirmed because general solutions have been generated using the continual development approach, a general conventional solution has been rediscovered and a new innovative general solution has been invented.

6

Parametric development of generic combinational multipliers

The objective of this chapter is to present an application-specific instruction-based developmental model for evolutionary design of generic combinational multipliers. While in case of the sorting networks the generic structure was growing continually and the size of the target network was determined by the number of developmental steps, the method of development of generic multipliers is based on parametrization of the design, i.e. a parameter is specified at the beginning of the developmental process which determines the size of the target circuit instance that will be created after appropriate number of developmental steps. Therefore, this approach has been called as parametric development. Although the developmental system presented in this chapter is also based on the instruction approach, it differs substantially from the previous one, especially in the way of generating the building blocks in order to build the target circuit (as described in Section 6.2). Two different developmental setups are presented, focusing of various aspects of the developmental process.

In the first part of this study an initial experiment of the development of generic multipliers will be described. In order to design irregular structures (inspired by the irregularity observed in conventional multipliers), an artificial environment is introduced into the system that is interpreted as an external control of the developmental process. Moreover, the environment is intended to demonstrate the ability of adaptation of the evolutionary process to different development controls resulting in the circuit of the same functionality. The results presented in this chapter represent the first case in the field of the evolutionary design when generic multipliers have been evolved by means of the development.

The goal of the second part of experiments (as described in Section 6.3) is to present a modified model for the development of multipliers at a lower level of abstraction. In particular, simplified building blocks are utilized in the circuit development. The crucial result of this research is the finding of the possibility to design generic multiplier structures which exhibit better delay in comparison with the first part of this study.

6.1 Motivation and related work

Combinational multipliers represent a class of circuits that is usually considered hard to be designed by means of the evolutionary techniques. Their construction has been often concerned as a non-trivial task for demonstrating the capabilities of the evolutionary design systems. In case of applying a direct encoding (a non-developmental genotype–phenotype mapping) it is difficult to achieve scalability of the evolved solutions, i.e. to obtain larger instances of the circuits, for example, when the traditional Cartesian Genetic Programming (CGP) is utilized [MT00]. Therefore, more effective representations have been investigated in order to improve the scalability and evolvability of digital circuits in general. The developmental encoding may represent a promising approach. However, most of the research deals with direct representations as summarized in the following overview.

Miller et al. outlined the principles in the evolutionary design of digital circuits and showed some results of evolved combinational arithmetic circuits, including multipliers, in [MJ00a]. A detailed study of the fitness landscape in case of the evolutionary design of combinational circuits using Cartesian Genetic Programming is proposed in [MJ00b]. 3×3-bit multipliers constitute the largest and most complex circuits designed by means of traditional CGP in these papers. Vassilev et al. utilized a method based on CGP which exploits redundancy contained in the genotypes. Larger (up to 4×4 bits) and more efficient multipliers were evolved by means of this approach in comparison with the conventional designs [VJM00]. Vassilev and Miller studied the evolutionary design of 3×3-bit multipliers by means of evolved functional modules rather than only two-input gates [VM00]. Their approach is based on Murakawa's method of evolving sub-circuits as the building blocks of the target design in order to speed up and improve the scalability of the design process [MYK+96]. Torresen applied the partitioning of the training vectors and the partitioning of the training set approach (so-called increased complexity evolution or incremental evolution) for the design of multiplier circuits. His approach was focused on improving the evolution time and evolvability rather than optimizing the target circuit. The 5×5-bit multipliers were evolved using this method [Tor03]. Stomeo et al. devised a decomposition strategy for evolvable hardware which allows to design large circuits [SKL06]. Among others, the 6×6-bit multipliers were evolved by means of this approach. Aoki et al. introduced an effective graph-based evolutionary optimization technique called Evolutionary Graph Generation [AHH03]. The potential capability of this method was demonstrated through experimental synthesis of arithmetic circuits at different levels of abstraction. 16×16 multipliers were evolved using word-level arithmetic components (such as one-bit full adders or one-bit registers).

The approaches for the evolutionary design of multipliers that have been investigated so far usually dealt with a certain level of abstraction, e.g. Miller's CGP-based approach involved a gate-level design, Aoki et al. utilized more

complex arithmetic components. The lower levels of abstraction usually allow to perform optimizations of the target design (e.g. area, cost or delay of the circuit) but the scalability of the design is limited. On the other hand, it is difficult to perform optimizations in case of the design at a higher level of abstraction when more complex building blocks are utilized (e.g. full adders). However, such building blocks may easily be assembled in order to create larger (scalable) circuits. This approach will be considered during the experiments presented in the next sections because the objective of this work is to design generic solutions rather than optimize the target circuit.

6.2 Initial concept for development of generic multipliers

This section introduces an artificial developmental model based on application-specific instructions in connection with the genetic algorithm in order to (1) reduce the length of the chromosome needed for encoding the target circuits, (2) introduce an external control of the developmental process in form of a string of values interpreted as a biologically inspired environment for the construction of irregular structures, (3) enable to design generic combinational multipliers and (4) demonstrate the ability of the evolutionary process to adapt to different environments retaining the capability to design generic structures of the given functionality. A concept of environment, which is relevant for this work, was proposed in [SZK01], where the environment was utilized to affect the function of the circuit (co-called polymorphic electronics). In our method, the environment is intended to influence the development of the circuit structure. The approach presented herein differs from the former one presented in Section 5.3 particularly in the following aspects: (1) parametrization of the design rather than continual development, (2) evaluation of only one instance of the circuit in the fitness calculation during evolution and (3) arrangement of the building blocks into the grid instead of linear array. The system also introduces some new features, in particular: involving the environment, utilization of more complex building blocks (e.g. adders) or connecting the inputs of the components to the primary inputs of the circuit via built-in variables of the developmental system.

6.2.1 Instruction-based developmental system

The method of the development is inspired by the construction of conventional combinational multipliers for which generic design algorithms exist. Figure 6.1 shows a typical 4×4-bit combinational multiplier designed by means of the conventional approach [Wak01]. It is evident that the first level of AND gates and the following sequence of adders are specific in comparison with the rest of the circuit, which poses a kind of irregularity. However, the rest of the circuit exhibits regular sequences which can be expressed by means of iterative algorithm utilizing variables. Moreover, the whole design can be easily

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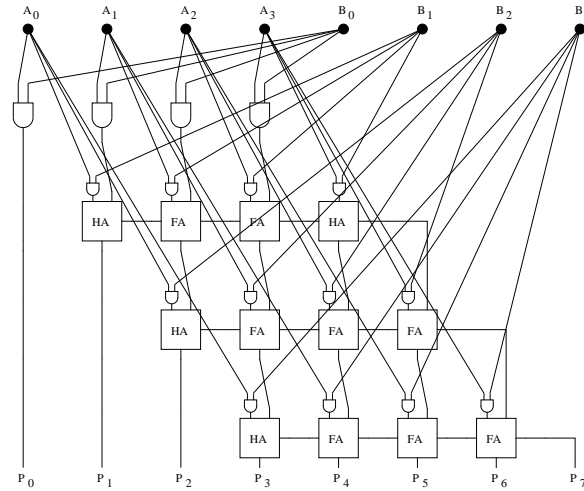

Fig. 6.1. A 4×4-bit conventional combinational multiplier. $A_0 \ldots A_3, B_0 \ldots B_3$ represent the bits of the operands, $P_0 \ldots P_7$ denote the bits of the product.

A system is introduced for the development of generic multipliers which operates with building blocks inspired by the conventional combinational multipliers. A simple two-dimensional grid consisting of a given number of rows and columns was chosen as a suitable structure for the development of the target circuits. The building blocks are placed into this grid by means of a developmental program.

A building block represents the basic component of the circuit to be developed. The general structure of the block is shown in Figure 6.2a. Each building block contains three inputs from which one or two may be unused depending on the type of the block. There are two outputs at each building block from which one may be meaningless, i.e. permanently set to logic 0, depending on the block type. The outputs are denoted symbolically as *out*0 and *out*1. In case of the block containing only one output, *out*0 represents the effective output and *out*1 is permanently set to logic 0. The circuit is developed inside a grid (rectangular array) which proved to be a suitable structure for the design of combinational multipliers (see Figure 6.2b). Figure 6.3 shows the set of building blocks utilized for the experiments presented in this section. For the interconnection of the blocks the position (row, col) in the grid is utilized. The inputs of the blocks are connected to the outputs of the neighboring blocks by referencing the symbolic names of the outputs or via indices to the primary inputs of the circuit, depending on the block type. Feedback is

not allowed. For example, $out1(row, col-1)$ means that the input of the block at the position (row, col) in the grid is connected to the output denoted $out1$ of the block on its left-hand side. The connections to the primary inputs are determined by the indices v_0 and v_1. Let $A = a_0a_1a_2, B = b_0b_1b_2$ represent the primary inputs (operands A and B) of a 3×3-bit multiplier. For instance, the AND gate with $v_0 = 1$ and $v_1 = 2$ has its inputs connected to the second bit (a_1) of operand A and the third bit (b_2) of operand B. In case of the building blocks at the borders of the grid (when $row = 0$ or $col = 0$), where no blocks with valid outputs occur (for $row - 1$ or $col - 1$), the appropriate inputs of the blocks at (row, col) are set to 0. In this way, for example, full adder (Figure 6.3f) at $(0, 0)$ is degraded to AND gate, the buffer (Figure 6.3b) at $(1, 0)$ becomes the source of logic 0 etc.

(a)   (b)
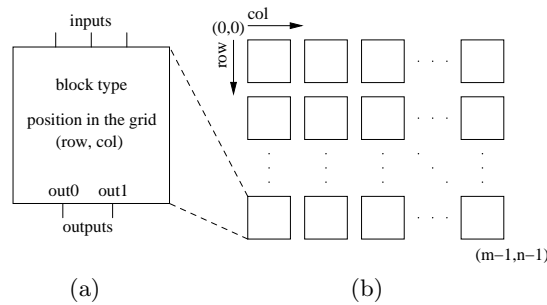
Fig. 6.2. (a) Structure of a building block. (row, col) determines the position of the block in the grid – see part (b). The connection of the inputs depends on the type and position of the block. (b) Grid of the building blocks with m rows and n columns for the development of generic multipliers.

(a)   (b)   (c)   (d)   (e)   (f)
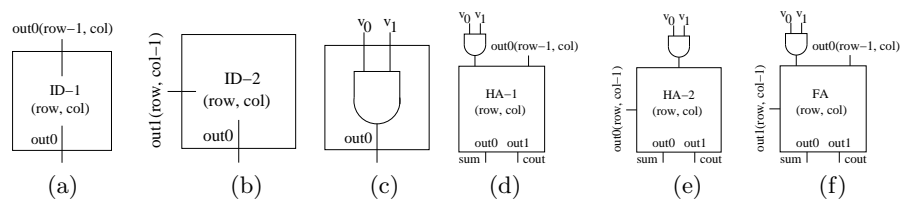
Fig. 6.3. Building blocks for the development of combinational multipliers. (a, b) buffers, (c) AND gate, (d, e) half adders, (f) full adder. (row, col) denotes position in the grid. v_0 and v_1 determine indices of primary input bits. Connection of different inputs of the blocks are shown. Unused inputs and outputs are not depicted (they are considered as logic 0).

The development of the circuit is performed by means of a developmental program. This program, which is the subject of evolution, consists of simple

application-specific instructions. The instructions make use of numeric literals $0, 1, \ldots, max\_value$, where max_value is specified by the designer at the beginning of evolution. In addition to the numeric literals, a parameter and some variables of the developmental system can be utilized. The parameter represents the width (the number of bits) of the operands – inputs of the multiplier. The parameter is referenced by its symbolic name w, whose value is specified by the designer at the beginning of the evolutionary process. For example, in case of designing a 4×4-bit multiplier, the parameter possesses this value, i.e. $w = 4$. The values of parameter is invariable during the evolutionary process. There are four variables integrated into the developmental system that will be denoted v_0, v_1, v_2 and v_3. These variables can be utilized in the evolving programs as a simple data structure — storage elements for integer values — and involved during the program execution (developmental process). For instance, loops can utilize variables to count a sequence of values. The values of variables can be altered by the appropriate instructions. Table 6.1 describes the instruction set utilized for the development. The SET instruction assigns a value determined by a numeric literal, parameter or another variable to a specified variable. Instructions INC, respective DEC are intended for increasing, respective decreasing the value of a given variable. The difference can be specified only by a numeric literal. Simple loops inside the developmental program are provided with the REP instruction whose first argument determines the repetition count and the second argument states the number of instructions after the REP instruction to be repeated. Inner loops are not allowed, i.e. REP instructions inside the repeated code are interpreted as NOP (no operation) instructions. The GEN instruction generates a building block of the type specified in the argument. Note that, if a block containing the AND gate is generated (e.g. the AND gate itself or FA), the inputs of the AND gate are connected to the primary inputs indexed by the values of variables v_0, v_1 as shown in Figure 6.3. In case when v_0 or v_1 exceeds the correct values, the appropriate input of AND gate is connected to logic 0. If (row, col) do not exceed the grid boundaries, the block is generated at that position, otherwise no block is generated. After executing GEN, col is increased by one.

In fact, the developmental program may consist of several parts (or subprograms), which may consist of different number of instructions. Let us define the length of a program (or a part of a program) as the number of instructions it is composed of. These parts are executed on demand with respect to an external information that is called an environment. A single execution of a part of program is referred to as a developmental step. The meaning of the environment is to enable the system to develop more complex structures which may not be fully regular. The environment is represented by a finite sequence of values specified by the designer at the beginning of the evolution, e.g. $env = (0, 1, 2, 2)$. The number of different values in the environment usually equals the number of parts of the developmental program. In addition, there is an environment pointer (let us denote it enp) determining a particular

Instruction	Arguments	Description
0: SET	$variable, value$	Assign $value$ to $variable$. $variable \in \{v_0, v_1, v_2, v_3\}$, $value \in \{0, 1, \ldots, max\_value, w, v_0, v_1, v_2, v_3\}$.
1: INC	$variable, value$	Increase $variable$ by $value$. $variable \in \{v_0, v_1, v_2, v_3\}$, $value \in \{0, 1, \ldots, max\_value\}$.
2: DEC	$variable, value$	If $variable \geq value$, then decrease $variable$ by $value$. $variable \in \{v_0, v_1, v_2, v_3\}$, $value \in \{0, 1, \ldots, max\_v.\}$.
3: REP	$count, number$	Repeat $count$-times $number$ following instructions. All REP instructions in the repeated code are considered as NOP instructions (no inner loops allowed).
4: GEN	$block$	Generate $block$ on the actual position (row, col) and then increase col by 1.
5: NOP		An empty operation.

Table 6.1. Instructions utilized for the development

value in the environment during the development time. Each subprogram is executed deterministically, sequentially and independently on the others according to the environment values. However, the parameter and the variables of the developmental system are shared by all the parts of the program. The concept of development described in this paragraph is illustrated in Figure 6.4.
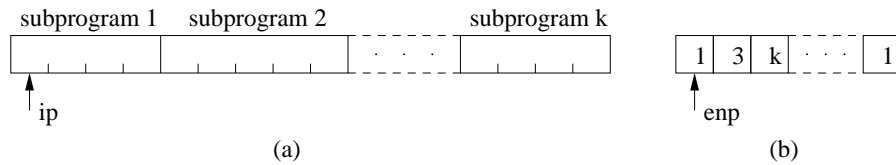
Fig. 6.4. The concept of instruction-based developmental encoding controlled by the environment: (a) a set of subprograms, each of which is identified by a unique index. The instructions of a subprogram are executed sequentially according to the instruction pointer ip. (b) The environmental vector of subprogram indices for the additional control of the development. The environment is scanned sequentially according to the environment pointer enp.

At the beginning of the evolutionary process the value of the parameter w and the form of the environment env are defined by the designer. By the inspiration from conventional multipliers the number of developmental steps needed for creating a working multiplier and the length of the environment will correspond to w. The developmental program, whose number of parts and their lengths are also specified a priori by the designer, is intended to operate over these data in order to develop multiplier of a given size. As evident, the different sizes of multipliers are created by setting the parameter and adjusting the environment. Hence the circuit of a given size is always developed

from scratch; it is a case of parametric developmental design. The following algorithm will be defined in order to handle the developmental process.

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

6.2.2 Evolutionary system setup

For the experiments presented in this section a simple genetic algorithm was utilized in combination with the developmental system described in Section 6.2.1.

A chromosome consists of a linear array of the instructions, each of which is represented by the operation code and two arguments (the utilization of the arguments depends on the type of the instruction). The array contains n parts of the developmental program stored in sequence, whose lengths (the number of instructions) correspond to $l_0, l_1, \ldots, l_{n-1}$. The number of the parts and their lengths are determined by the designer. In general, the structure of a chromosome can be expressed as $i_{0,0}i_{0,1} \ldots i_{0,l_0-1}; \ldots; i_{n-1,0}i_{n-1,1} \ldots i_{n-1,l_{n-1}-1}$, where $i_{j,k}$ denotes the k-th instruction of j-th part of program for $k = 0, 1, \ldots, l_j - 1$ and $j = 0, 1, \ldots, n - 1$. During the application of the genetic operators the parts of the program are not distinguished, i.e. the chromosome is handled as a single sequence of instructions. The chromosomes possess constant length during the evolution. The population consists of 32 chromosomes which are generated randomly at the beginning of evolution. Tournament selection operator of base 2 is utilized.

Mutation of a chromosome is performed by a random selection of an instruction followed by a random choosing a part of the instruction (operation code or one of its arguments). If the operation code is mutated, entire new instruction will replace the original one, otherwise one of its arguments is mutated. The mutation algorithm ensures that proper values of arguments will be created depending on the instruction type. The mutation is performed with the probability 0.03, only one instruction per chromosome is mutated.

A special crossover operator was applied during the experiments (see Figure 6.5). Two parent chromosomes are selected and an instruction is selected randomly in each of them (i_1, i_2). A position (index) is chosen randomly in each of the two offspring (c_1, c_2). After the crossover, the first, respective the second offspring contains the original instructions from the first, respective the second parent with the exception of i_1, respective i_2, which is put at the position c_2 in the second offspring, respective c_1 in the first offspring. The crossover occurs with the probability 0.9.

The fitness function is calculated by means of a circuit simulator as the number of output bits calculated correctly by the multiplier developed by the
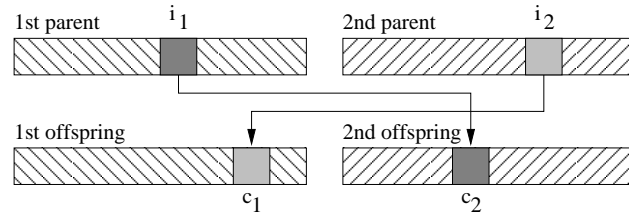
Fig. 6.5. Crossover of two chromosomes. i_1, i_2 denote the instructions to be crossed, c_1, c_2 pose the offspring positions the instructions will be placed to.

program stored in the chromosome. The experiments were conducted with the evolution of programs for the construction of 4×4 multipliers, i.e. the parameter $w = 4$. There are $2^{4+4} = 256$ possible test vectors and the multipliers produce 8-bit results. Therefore, the maximum fitness representing a working solution equals $256 \cdot 8 = 2048$. If a working solution is not evolved in 2 millions of generations in case of the first set of experiments, possibly in 1 million of generations in the second set of experiments (see the next section), the evolution is restarted with the new (randomly generated) population. After the evolution the resulting program is verified in order to determine whether it is able to create larger multipliers, typically up to the size 14×14 bits. This size of circuit was determined experimentally, allowing to perform a sufficient number of developmental steps for demonstrating the correctness of the evolved program and keeping a reasonable verification time. If a program shows this ability, it will be considered as general.

Two different sets of experiments were performed for demonstrating the adaptation.

1. Evolution of 3-part developmental programs with the lengths of the parts 6, 12 and 12 instructions that are executed according to the environment $env = (0, 1, 2, 2)$. This form of environment is inspired by the structure of the conventional multipliers – see Figure 6.1. In fact, the multiplier is composed of several "levels" (or rows if the grid structure is considered the building blocks are generated into), each of which is able to generate separately using a specific program. If the 4x4-bit multiplier from Figure 6.1 is considered, the first row consists of only basic AND gates whose inputs are connected to the primary inputs of the circuit. These AND gates can be generated by means of the program denoted as 0 in the environment. The second row consists of half adders and full adders combined with basic AND gates. Similarly, the inputs of the AND gates are connected to the primary inputs and the interconnection of the adders is well defined. This row can be generated using another program denoted by 1 in the environment. The structure of the third and fourth row differs from the structure of the first and second row. However, the third and fourth row (if compared to each other) exhibit a regular structure. Therefore, they can be generated using the same program that is denoted by 2 in the en-

vironment. The number of the AND gates, the number of adders in each
rows, the "shift" of the rows containing adders against the first column
of the grid as well as the number of rows of the multiplier can be deter-
mined by the number of bits (width) of the multiplicands as a parameter.
Moreover, the index of the row is also considered during the circuit devel-
opment. Starting by the third row of the circuit, the structure is regular
for all instances (widths of the operands) of the multiplier. Therefore, the
first two rows can be generated by the programs 0 and 1 and the rest of
the circuit can be generated by repeated application of the program 2.
Note that the first row represents an irregularity of the circuit in compar-
ison with the rest of the multiplier structure. The number of applications
of the program 2 is determined by the width of the multiplicands. There-
fore, the environment can be specified for arbitrarily large instance of the
multiplier; the environment will possess the form $(0, 1, 2, \ldots, 2)$, where the
number of 2s is determined as the width of the operand minus 2. The to-
tal number of rows, which equals to the total length of the environment
and the total number of applications of the programs needed to develop
a working multiplier, corresponds to the width of the operand. Note that
only the operands of the same size are considered in this work.

2. Experimental design of 1-part developmental programs consisting of 10
   instructions using the environment $env = (0, 0, 0, 0)$. In general the length
   of the environment corresponds to the width of the multiplicand (sim-
   ilarly to the principle described in the item 1). It is expected that the
   multipliers developed by means of this approach will exhibit a regular
   structure. Because of the reduced search space in comparison with the
   item (1), the maximal number of generations was decreased to one mil-
   lion. In both cases the system is to perform four developmental steps to
   design a solution which is evaluated by the fitness function. The goal is
   to evolve a program that is able to create generic multipliers.

The experiments were conducted on common PCs running RedHat-based
Linux operating system. The hardware configuration consists of a 2.0 GHz
processor and 512 MB RAM. The SGE system was utilized so that several
independent experiments could be performed on different PCs in parallel. The
evolution of a single solution required 15–20 minutes in average.

6.2.3 Experimental Results and Discussion

In the first set of experiments 3-part programs (6+12+12 instructions) were
evolved utilizing the environment $env = (0, 1, 2, 2)$ for controlling the develop-
ment. 1000 independent experiments were conducted from which 67% working
solutions (i.e. the programs for constructing 4×4-bit multipliers) were evolved
and 18% of them were classified as general programs.

Figure 6.6 shows one of the multipliers designed by the evolution together
with detailed logic schemes of the building blocks utilized (half adder from

Figure 6.3e and full adder from Figure 6.3f). This multiplier was constructed by means of one of the most efficient programs that were evolved in this set of experiments. The program, that demonstrated the ability to construct generic multipliers, is shown in Table 6.2. Let us go through the program in order to understand the developmental process.
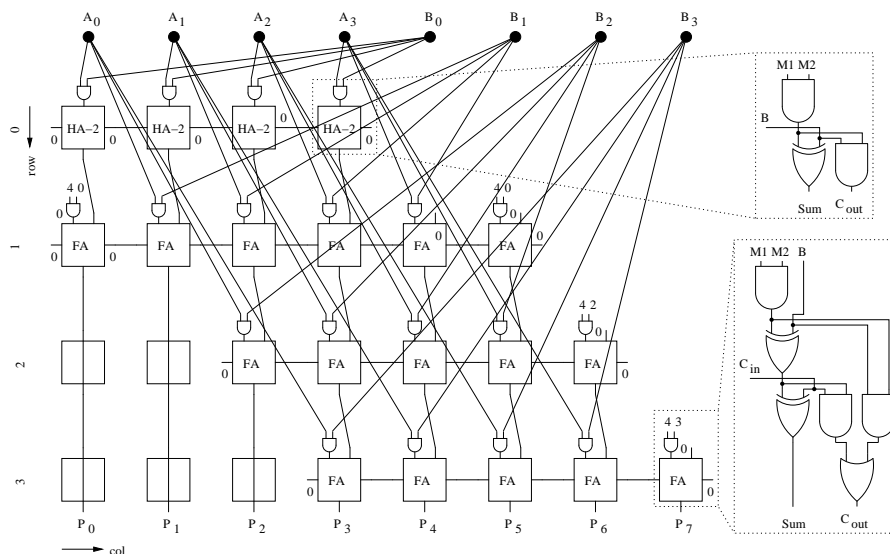

Fig. 6.6. A 4×4-bit multiplier created by means of evolved program shown in Table 6.2 using the environment $env = (0, 1, 2, 2)$. $A_0 \ldots A_3, B_0 \ldots B_3$ represent the bits of the operands, $P_0 \ldots P_7$ denote the bits of the product. Logic schemes of the half adder- and full adder-based building blocks utilized by the evolved program are shown. M1 and M2 denote the multiplicands whose partial product represents the first operand of the full adder, B denotes the second operand, C_{in} poses the input carry, Sum and C_{out} represent the resulting sum and output carry.

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

At this point $env(e) = 0$, therefore, Part 0 of the program will be executed. The instruction 0 should repeat zero times instructions 1 and 2 (because $v_1 = 0$), therefore, this code has no effect. Since part 0 of the program is executed only once at the beginning of the development according to the environment and the value of v_1 is always 0 during the execution, the instructions 0, 1 and 2 constitute an intron in this part of the program. Therefore, they might be removed without any loss of its functionality. The instruction 3 will repeat the instructions 4 and 5 four times (because $w = 4$). These instructions create row 0 of the multiplier (blocks HA-2) with the inputs of AND gates of these blocks connected to the primary inputs (operands of the multiplier) specified

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

Table 6.2. Evolved general program by means of which the multiplier from Figure 6.6 was created. In this case ($w = 4$), the program consists of 3 parts executed according to the environment $env = (0, 1, 2, 2)$.

by the actual values of v_0 and v_1. While v_1 retains 0, v_0 is increased by 1 by instruction 5 and *col* is increased by 1 automatically by the system in each pass (in general, after executing a GEN instruction). Since there are no more instructions to be executed in Part 0, the system increases *row* and e by 1 and the construction of row 0 of the circuit is finished. Note that the variables hold their current values after finishing the execution of the program 0, i.e. $v_0 = 4$ and the others equal 0.

Now, $env(e) = 1$ for $e = 1$, therefore, Part 1 of the program will be executed in order to develop row 1 of the multiplier. The instruction 0 of Part 1 generates the full adder (FA block), where the inputs of AND gate of this block should be connected to bits 4 and 0 of the operands (according to the variables $v_0 = 4, v_1 = 0$). Note, since v_0 exceeds the operand width, the first input of AND gate of this FA block will be considered as logic 0 causing permanent logic 0 at the output of the AND gate, i.e. the AND gate of this block is meaningless (see Figure 6.6). Instructions 1 and 2 actually set v_0 to 0. Then, v_1 is increased by 1 by instruction 3. Instructions 4, 5 and 6 generate four FA blocks with the inputs of AND gates of these blocks connected to the appropriate operand bits. Note that instruction 7 sets v_1 to 0 which, in fact, voids the result of instruction 3 (i.e. instruction 3 can be considered as intron). An FA block is generated by instruction 8 (again, its AND gate is meaningless). Instruction 9, decreasing v_1 by 0, has no effect, v_1 is increased by 1 by instruction 10 and instruction 11 represents an intron since there is no code to repeat. Row 1 is completed with the actual values of $v_0 = 4, v_1 = 1$ and other variables possessing zeros.

The row 2 of the circuit will be constructed using Part 2 of the program according to the next environment value $env(e) = 2$ for $e = 2$. Instruction 0 initiates a loop repeating once instructions 1 and 2. Instruction 1 is interpreted as no operation because inner loops are not allowed and instruction 1 generates

an ID-1 block. In addition, instruction 3 creates one more ID-1 block in the next column. Value of v_0, respective v_1 is increased by one by instruction 4, respective 5. In fact, the only effect of the loop initiated by instruction 6, repeating instruction 7, is to set v_0 to 0 (according to v_2 which equals 0). This operation actually voids the result of instruction 4. Four FA blocks are generated by instruction 9 inside the loop started by instruction 8. Instruction 9, which is also a part of the loop body, determines the connection of the inputs of AND gates generated inside these blocks. The last instruction 11 generates an FA block with a redundant AND gate. Now row 2 is finished. The variables $v_0 = 4, v_1 = 2$ and the other ones equal 0.

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

It is evident that the multiplier shown in Figure 6.6 could be optimized considering the inputs of the building blocks. For instance, half adders in row 0 of the circuit can be replaced by simple AND gates since the first input of each of these adders is permanently connected to logic 0. Similarly, full adders at positions $(1, 1), (1, 4), (2, 2)$ and $(3, 3)$ actually represents half adders and full adders at positions $(1, 0), (1, 5), (2, 6)$ and $(3, 7)$ can be replaced by identity functions. In fact, the circuit corresponds to the conventional multiplier after performing this optimization (compare with Figure 6.1).

The second set of experiments was devoted to the evolutionary design of single developmental programs consisting of 10 instructions. A new form of the environment was specified in order to demonstrate the adaptation of the program being evolved to the new conditions of creating generic multipliers. Again, 1000 independent experiments were conducted from which 97% working solutions were obtained. 85% of the evolved programs were classified as general. An evolved 4×4-bit multiplier adapted to the new environment $env = (0, 0, 0, 0)$ is shown in Figure 6.7. Table 6.3 shows the appropriate developmental program. This program showed the ability to construct generic multipliers. Note that, in general, for w-bit operands the environment possess the form $env = (0, \ldots, 0)$ whose length corresponds to w, i.e. w developemntal steps is needed in order to develop a working multiplier.

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

Table 6.3. Evolved general program by means of which the multiplier from Figure 6.7 was created. In this case ($w = 4$), there is only one program part operating in the environment $env = (0, 0, 0, 0)$.

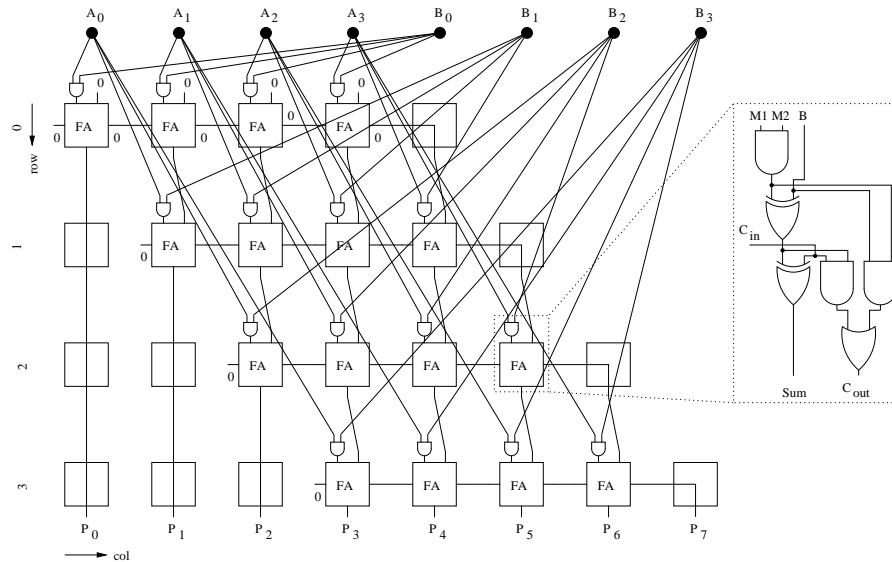**Fig. 6.7.** A 4×4-bit multiplier created by means of evolved program shown in Table 6.3 adapted to the environment $env = (0, 0, 0, 0)$. $A_0 \ldots A_3, B_0 \ldots B_3$ represent the bits of the operands, $P_0 \ldots P_7$ denote the bits of the product. Logic scheme of the fundamental full adder-based building block (see Figure 6.3f) utilized by the evolved program is shown. M1 and M2 denote the multiplicands whose partial product represents the first operand of the full adder, B denotes the second operand, Sum and C_{out} represent the sum and output carry of the full adder.

The results presented in this section confirms Hypothesis 2 because general solutions have been generated using the parametric development and partially also Hypothesis 3 because the conventional general principle of constructing common combinational multipliers has been rediscovered. The results are presented without proofs because of known properties of the conventional solutions (e.g. see [Wak01]).

Experiments for the evolution of 3×3 multipliers were conducted, however, no general solution was obtained. Although basic AND gates and ID functions were available in the set of building blocks, they were rarely used in the design and adders were generated instead. This behavior could be explained by predominating occurrence of adders which pushes the evolution to design regular structures, utilizing the properties of the building blocks and their interconnection. The evolved programs exhibit certain degree of redundancy, which is caused by the determination of the program length based on the conventional design. Therefore, there is an additional possibility for reducing the search space. Despite the worse level of evolvability as seen in the progress of the average population fitness shown in Figure 6.8, a very good success rate was observed both in the case of the evolution of initial solutions and the occurrence of general programs among these solutions after verification especially in

the second set of experiments, which indicates the suitability of the proposed representation to evolve generic structures. However, the selection of building blocks represent a crucial issue for successful evolution of this kind of circuits. Both the programs presented herein showed the ability to construct generic multipliers, which has never been seen before in the field of the evolutionary design.
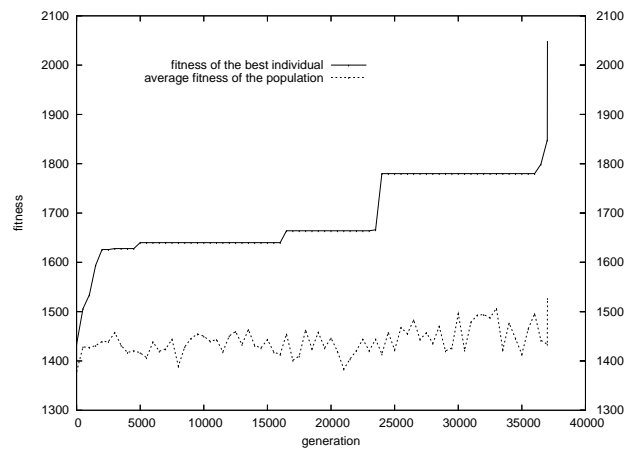

Fig. 6.8. A typical progress of the fitness during the evolution of multipliers using the proposed developmental system

An environment was integrated into the developmental model in order to allow the system to construct irregular structures (inspired by the conventional multipliers). Several general programs were evolved that construct the multipliers that correspond to the structure of the conventional combinational multipliers. Moreover, a single program was evolved that is executed repeatedly in order to develop a working circuit (the evolutionary design process has been adapted to another environment). The multipliers developed by means of this program exhibit fully regular structure.

Note that the feature of adaptation observed in the evolutionary developmental system is significantly influenced by the general properties of the building block and the grid structure the building blocks are generated into. In particular, the facility of degradation of more complex blocks (e.g. full adders) to simpler blocks (e.g. AND gates, ID functions etc.) according to the input values of the blocks represents the key feature that enables to develop fully regular multiplier structure considering the grid representation of the circuit and the interconnection of the building blocks in the regular manner. However, the results of adaptation represent significant findings with respect to the future research. For example, the development of generic combinational multipliers possessing exactly that structure shown in Figure 6.1 would not be

possible without applying the environment. A variety of building blocks exist which could be involved in the design process in order to develop more complex generic circuits exhibiting irregularities. Therefore, the approach utilizing a form of environment suggests an extensive area deserving of subsequent investigation.

6.3 Advanced development of generic multipliers

The experiments conducted under the initial research demonstrated (1) a suitability of the instruction-based developmental model to design generic multiplier structures using a parametric approach, (2) a possibility of the development of irregular structures by introducing an environment which is considered as an external control of the developmental process – inspired by the structures of conventional multipliers and (3) an adaptation of the developing structures to the different environments by utilizing the properties of the building blocks. These results indicate that the area of evolutionary development of multipliers is worth of future investigation. Therefore, an improved developmental system was proposed that is based on the approach introduced in Section 6.2.1.

The original developmental model utilized complex building blocks which were inspired by the elements present in the structure of the common combinational multipliers. Specifically, the basic component of adder (half adder and full adder) was combined with an AND gate in order to create a functional building block. However, the complexity of those building blocks may restrict the evolution in finding effective solutions. As the results showed, there is no innovation in comparison with the conventional design. Therefore, the objective of the next research was to introduce simplified building blocks and to modify the developmental system in order to develop more effective generic multipliers.

6.3.1 Concept of improved developmental system

In order to reach the objective described in the previous paragraph, the following aspects were introduced to the developmental system. The inspiration for the development of more efficient multipliers was taken from the principle of carry-save multipliers which exhibit shorter delay in comparison with the common combinational multipliers as described, for instance, in [Wak01]. The experiments presented herein are devoted to design this kind of circuits.

To clarify the developmental process and to decrease the restriction of evolutionary process caused by the complexity of the building blocks in the initial approach, the building blocks are split to pure half and full adders and basic AND gates. These components represent elementary building blocks of the circuit in the modified developmental model. Sample instances of common and carry-save 4×4-bit multipliers composed of these building blocks are

shown in Figure 6.9. Considering this circuit representation, the first level ("row") of AND gates occurring in both multipliers and the last level of adders occurring in the carry-save multiplier (Figure 6.9b) constitute irregularities that have to be taken into account during the developmental process (similarly to the initial experiments presented in Section 6.2.3). The rest of the circuits exhibits regular structure; the number of bits of the operands can be used to determine the number of AND gates and adders in the appropriate circuit level and also the number of levels of the multiplier. Therefore, it may be assumed that the representation utilizing the new concept will be suitable for development by means of similar instruction-based approach. Again, several forms of environment will be considered in the evolution of both types of multipliers.
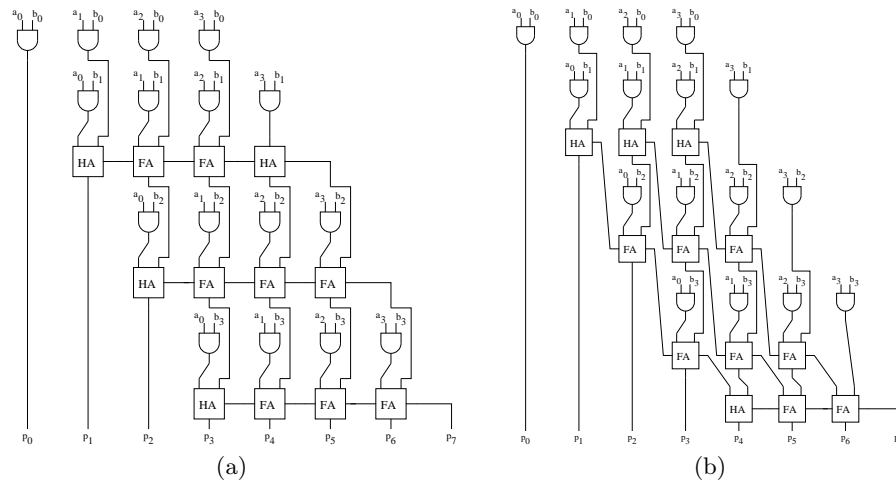
Fig. 6.9. 4×4-bit conventional multipliers: (a) common combinational multiplier, (b) more efficient carry-save multiplier possessing shorter delay in comparison with the common one. $a_0, \ldots, a_3$, respective $b_0, \ldots, b_3$ denote the bits of the first, respective the second operand, $p_0, \ldots, p_7$ represent the bits of the product.

6.3.2 Differences in the developmental models

The model of development utilized for the experiments presented in this section is based on the principles introduced in Section 6.2.1. However, several differences have to be considered in order to enable the system to deal with the issues stated in Section 6.3.1. The differences are summarized in the following paragraphs.

For the development of more efficient generic multipliers, simplified building blocks (in comparison with those shown in Figure 6.3) have been introduced. Figure 6.10 shows the new set of building blocks. As evident, the adder

constitutes the crucial element of the circuit (similarly to the design of conventional multipliers). Moreover, the basic AND gate was separated from the adders, i.e. only pure half and full adders and a standalone AND gate constitute the new set of building blocks utilized for this kind of development. The consequence of this modification is that all the AND gates are represented in separate positions of the grid during the circuit development (see Figure 6.9). Therefore, it is needed to introduce a more general interconnection pattern of the elements in order to be able to create a working multiplier. To satisfy this requirement, new types of half and full adders have been included in the set of building blocks whose appropriate inputs can be connected to the second neighbouring row in the grid relatively to the actual position of the adder. These new adders are shown in Figure 6.10 under the variants (d), (f) and (g). Moreover, a new type of full adder (Figure 6.9h) has been introduced possessing another prescription for the connection of its inputs. The modifications performed in the new developmental system is primarily inspired by the aim to design carry-save multipliers (a sample instance is shown in Figure 6.9b). The grid utilized for the circuit development and the new building blocks keep their original features (i.e. the principle of their interconnection and the rules according to which the multipliers are developed) as described in Section 6.2.1.
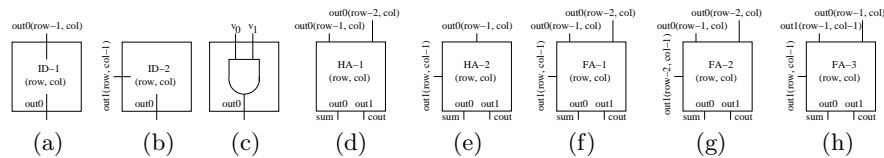
Fig. 6.10. Building blocks for the development of generic multipliers. (a, b) buffers – identity functions, (c) AND gate, (d, e) half adders, (f, g, h) full adders. (row, col) denotes the position in the grid. v_0 and v_1 determine indices of primary input bits. Connection of different inputs of the blocks are shown. Unused inputs and outputs are not depicted (set to logic 0). Note that the full adders (g, h) are new to the advanced developmental model that has been introduced (inspired by the conventional approach) in order to design carry-save multipliers.

The instruction set utilized in the modified developmental model is the same as summarized in Table 6.1 with the only difference related to the GEN instruction. In the new setup, the GEN instruction may generate one or two building blocks at a time. Note that only one building block could be generated by means of the original developmental model. The types of the building blocks to be generated are specified by two arguments of the GEN instruction. In case of generating two blocks, the second one is placed to the position $(row+1, col)$ in the grid. This variant of circuit development has been chosen in order to reduce the complexity of the design process when the simplified building blocks have been utilized. Note that this approach does nowise restrict the

capabilities of the construction algorithm because the number and types of
the blocks to be generated are determined independently by means of the
evolutionary algorithm. Note that the evolutionary system setup is identical
as described in Section 6.2.2

The following developmental algorithm has been defined in order to handle
the design process by means of the modified model.

1. Initialize $row, col, v_0, v_1, v_2, v_3$ and e to 0.
2. Execute $env(e)$-th part of program.
3. If a GEN instruction was executed, increase row by 2 in case of generat-
   ing two building blocks simultaneously or by 1 if only single blocks were
   generated. Increase e by one and set col to 0.
4. If neither e, nor row exceed, go to step 2.
5. Evaluate the resulting circuit.

6.3.3 Experimental Results and Discussion

The evolutionary design process was devoted especially to the design of carry-
save multipliers which exhibit better properties in comparison with the com-
mon multipliers. In addition, the adaptation was involved by considering dif-
ferent environments in order to investigate the ability of the design system to
develop the common multipliers as well by means of the modified representa-
tion. The selection of the evolved programs and resulting circuits presented
in this section is based on their generality (i.e. the ability to construct generic
multipliers) and the resemblance to the carry-save multiplier structure with
respect to the circuit delay and the number of building blocks the developed
multipliers are composed of.

In the first set of experiments, a subset of the building blocks from Figure
6.10 was chosen for the design of the carry-save multipliers (see Figure 6.9b).
Therefore, only the blocks (a, b, c, d, g, h) were involved in the design process.
Considering the irregular structure of the conventional carry-save multiplier,
a program consisting of four parts is to be evolved. The parts of the program
are executed according to the environment $env = (0, 1, 2, 2, 3)$, which is spec-
ified a priori with respect to the structure of carry-save multiplier. Therefore,
the construction of the circuit is performed as follows. Considering Figure
6.9b, the first level of the AND gates is created using part 0. The second
level of AND gates together with the following level of adders are constructed
by means of part 1. According to the environment, the next levels of AND
gates and adders are created by means of double application of part 2. Fi-
nally, part 3 is utilized to design the last level of adders. Two hundreds of
independent runs of the evolutionary algorithm were conducted from which
18% evolved a correct program for the construction of 4×4-bit multipliers.
60% of the evolved programs were classified as general, i.e. able to create arbi-
trarily large multiplier. Figure 6.11 shows (a) one of the best evolved general
program and (b) a 4×4-bit multiplier constructed by means of that program.

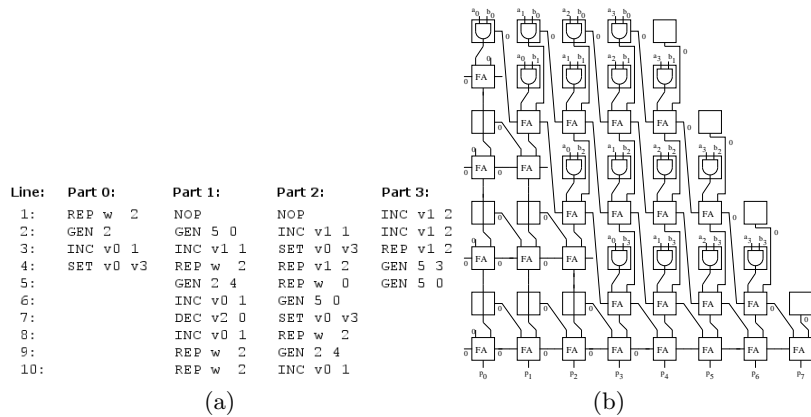106   6 Parametric development of generic combinational multipliers

Line:	Part 0:	Part 1:	Part 2:	Part 3:
1:	REP w 2	NOP	NOP	INC v1 2
2:	GEN 2	GEN 5 0	INC v1 1	INC v1 2
3:	INC v0 1	INC v1 1	SET v0 v3	REP v1 2
4:	SET v0 v3	REP w 2	REP v1 2	GEN 5 3
5:		GEN 2 4	REP w 0	GEN 5 0
6:		INC v0 1	GEN 5 0	
7:		DEC v2 0	SET v0 v3	
8:		INC v0 1	REP w 2	
9:		REP w 2	GEN 2 4	
10:		REP w 2	INC v0 1	

(a)   (b)

Fig. 6.11. (a) An evolved general program, (b) 4×4 multiplier exhibiting the carry-save structure created by means of this program. Note that blank rectangles represent empty blocks (not generated by any instruction) whose outputs are considered as logic 0.

At the beginning of the development, the system is initialized: the variables are set to 0, the parameter is set to 4, row and column positions are initialized to 0 and the number of rows and columns is limited to 8 – no gate may be generated behind the grid boundaries. According to the first element of the environment (0), part 0 of the evolved program is executed (see Figure 6.11a). The first REP instruction initiates a loop repeating 4 times (because $w = 4$ for designing a 4×4 multiplier) two instructions after the REP instruction. In each pass, an AND gate (code 2 in the argument of GEN instructions at line 2) is generated with its inputs connected to the primary inputs of the circuit indexed by the values of variables v_0, v_1. Moreover, v_0 is increased by 1 (line 3) so that the AND gates generated in different passes possess the different first input. After executing a GEN instruction, the column position is increased by 1. After finishing part 0, the row position is increased by 1 and the column position is set to 0. According to the next element of the environment (1), part 1 will be executed. Note, however, that the GEN instruction at line 2 of part 1 generates two building blocks into the actual column, the second block "under" the first one: full adder is generated into the second row of the first column (code 5 in the first GEN argument) and the identity function is generated into the third row of the first column (code 0 in the second argument). Since there have been building blocks generated into two rows, the row position is increased by 2 after finishing part 1. In case of executing part 3, only the full adders are generated (code 5 of the GEN instructions at lines 4 and 5) as there is no space left in the grid for the second level of blocks specified by the second argument of the GEN instructions – the number of rows of the grid was limited to 8 for this experiment.

It is evident that the multiplier shown in Figure 6.11 could be optimized with respect to the inputs of some building blocks (e.g. adders possessing only one non-zero input could be replaced by the identity functions as demonstrated in Section 6.2.3). After this optimization the circuit corresponds to the carry-save multiplier shown in Figure 6.9b.

The second set of experiments was devoted to the design of multipliers using the full set of building blocks shown in Figure 6.10 and the same form of environment like in the previous experiment. Therefore, this setup corresponds to both variants of the multipliers from Fig 6.9. The prefix (0, 1, 2, 2) of the environment may be utilized for the evolution of common multiplier structures shown in Figure 6.9a. Again, 200 independent experiments were conducted from which 37% of working programs were obtained and 54% of them were classified as general. However, the experiments showed that the evolution of efficient carry-save multipliers is difficult using this setup. Although there are all the resources available as in the first set of experiments, no valid carry-save structure was obtained. The evolution generated the carry-save components very rarely and not to the positions at which they could be usefully utilized during the circuit operation. The common structures (Figure 6.9a) were evolved instead. An example of a general program together with a 4×4-bit multiplier is shown in Figure 6.12 which represents the same type of the common multiplier structure that was evolved in Section 6.2.3.

Part 0:	Part 1:	Part 2:	Part 3:
SET v1 v2	INC v0 1	INC v0 1	INC v1 1
REP w 2	SET v1 v2	REP v0 1	INC v2 0
GEN 2 5	GEN 1 6	GEN 0 7	GEN 7 6
INC v1 1	REP w 2	SET v1 v2	INC v2 1
	GEN 2 5	GEN 2 3	GEN 2 5
	INC v1 1	INC v1 1	
	GEN 0 5	REP w 2	
	DEC v1 0	GEN 2 5	
	SET v0 1	INC v1 1	
	REP w 0	NOP	

(a)   (b)
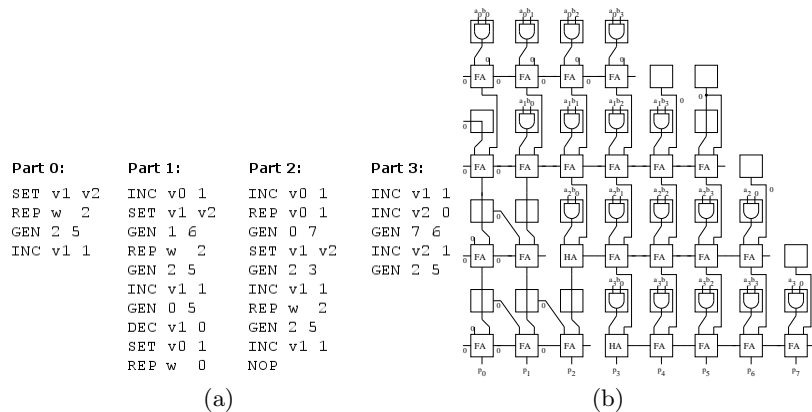
Fig. 6.12. (a) An evolved general program, (b) a 4×4-bit multiplier based on the structure of the common combinational multiplier. Blank rectangles represent empty blocks with the outputs possessing logic 0.

The experiments presented in this section represent a continuation of the successful research in the field of the evolutionary design of generic multipliers using development introduced in Section 6.2.

The phenomenon of adaptation of the developmental process to different environments during the evolution enabled us to design various multiplier

structures. In particular, the carry-save structure was rediscovered in these experiments, exhibiting shorter delay in comparison with the common multipliers, which was the main goal of experiments using the modified developmental model. Note that these results confirm Hypothesis 2 (generating general solutions using parametric development) and partially also Hypothesis 3 (rediscovering a general conventional solution). Again, the results are presented without proofs because of known properties of the conventional carry-save multipliers (e.g. see [Wak01]).

Although the carry-save multipliers showed to be very hard to evolve, the evolutionary developmental system demonstrated the ability to design this class of multipliers using a reduced set of building blocks. Moreover, simplified building blocks were introduced in this section together with an improved developmental model in comparison with the initial experiments presented in Section 6.2. Therefore, there is a smaller limitation of the evolutionary process which, however, leads to more difficult evolution because of lower level of abstraction in the circuit representation. The success of the evolution of carry-save multipliers demonstrates an ability of the experimental system to design different circuit structures with more complex interconnection of their components which represents a promising area for the next research.

6.4 Summary

In this chapter, an original developmental approach to the automatic evolutionary design of arbitrarily large combinational multipliers was introduced. Since the size of the target multiplier to be developed is specified explicitly by the number of bits of the operands as a parameter and the working circuit is every time constructed from scratch by means of the evolved program, it is a case of parametric developmental design. In total, two systems of development were presented, each of which works with a different representation of the circuits. Both models demonstrated the ability to design generic multipliers.

In the first developmental model that was used to conduct the initial experiments of this kind, a specific form of an external information, that was called the environment, was integrated into the design system. The environment, representing an additional control of the developmental process, is intended as a tool enabling to design irregular structures that are observable in the conventional structures of combinational multipliers. Moreover, the environment was utilized in order to demonstrate adaptation of the design process, retaining its ability to design generic multipliers. The experiments confirmed the capability of adaptation in connection with the proposed circuit representation. General programs were evolved for the construction of multipliers which exhibit a high degree of regularity in the circuit structure. This approach represents the first case in the area of the evolutionary design when arbitrarily large combinational multipliers were constructed by means of the development.
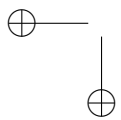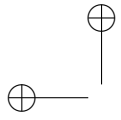
The interesting results obtained from the initial experiments indicated that this area of evolutionary design may be worth of future research. Therefore, an advanced developmental system was designed that has been primarily intended to design more effective multipliers. The inspiration was taken from the principle of carry-save multipliers which exhibit shorter delay in comparison with the common ones. Simplified building blocks were introduced in order to clarify the design process and to enable the evolutionary algorithm to explore a wider design space. The new set of building blocks led to different circuit representation for which the developmental system had to be adjusted. The phenomenon of adaptation to the different environments was utilized for the design of common as well as carry-save multipliers and their potential variants using the modified representation of the circuit structures. Although the experiments showed a substantial difficulty of designing the efficient carry-save multipliers, several general programs were evolved that construct this class of circuits using a reduced set of building blocks which represents a contribution of this dissertation.

In view of the experiments conducted in this domain, it is evident that there may be a big potential for the application of this model to other classes of well-scalable circuits, in particular, sorting networks. In general, the rectangular structure introduced for the circuit representation can be easily adjusted to the requirements of other applications. For instance, a linear array was utilized for an advanced development of generic sorting networks as described in Section 5.5.

Despite the successful results presented in this chapter, the evolved solutions are expectable because the circuit representation was inspired by the conventional design and the concept of the developmental process was mostly adjusted to the known features of the conventional approach. However, the resulting programs were discovered completely automatically and exhibits more complexity in comparison with the case of continual development of the sorting networks. Although there are less results presented in this chapter, it is influenced by the fact that most of them was monotonous, inefficient and therefore not so interesting for future investigation.

Note that Hypotheses 2 and 3 have been confirmed because general solutions have been evolved using the parametric development approach and the general conventional solution of carry-save principle has been rediscovered.

The instruction-based development approach is also applicable to the evolutionary design of generic adders. However, it is a trivial task because the basic (regular) adder structures (e.g. ripple carry adder or carry skip adder) constitute only simple chain of gates or half/full adder modules. Therefore, no such experiments will be presented herein.

7
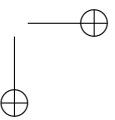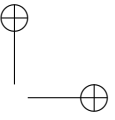
Development of generic polymorphic circuits

In Chapter 6 an instruction-based developmental model was introduced that utilized an external information (which was called the environment) allowing us to control the developmental process. It was shown that the evolution can adapt to different environments while preserving the ability to design programs for the construction of the circuit with a given functionality. In fact, this kind of environment (more precisely, its interpretation) is aimed to influence the physical structure of the circuit to be developed. For instance, for some environments, a functional unit, say X, is created, for other environments functional unit X is not created. Therefore, the evolution must adjust the program to be designed in order to develop the desired function of the target circuit for the different environments.

In this chapter it will be shown that there is an other option for providing external information for growing digital circuits. In general, this information need not to be provided in the digital form; rather it may influence the system in an analogue (and perhaps non-electrical) form, for example, as temperature, light, radiation, specific voltage etc. The motivation for this approach is that every electronic circuit operates in a real physical environment and "good" physical environment is crucial for the correct behavior of the resulting system (similarly, cells survive and divide only in a "good" environment).

In this approach the growing circuits should inherently be able to interact with the physical environment. Technology, which, in principle, allows engineers to build such systems, is called polymorphic electronics [SZG$^+$04, SZKL02, SZK01]. It was shown that it is possible to create digital gates whose functionally can be controlled in a non-traditional way: by temperature, power supply voltage (Vdd), some external signals etc. For example, the polymorphic (i.e. multifunctional) NAND/NOR gate operates as NOR in the case that Vdd=1.8V and as NAND in the case that Vdd=3.3V. Polymorphic gates do not influence the physical structure of the growing circuit, i.e. the circuit topology is independent of the environment. However, because the circuit contains polymorphic gates, its behavior depends on the environment, i.e. the circuit

is inherently multifunctional. For instance, it can operate as the adder in low temperatures and as the sorter in high temperatures.

If some gates of a growing circuit are polymorphic then the function of the circuit to be developed can be controlled by the mentioned external signals (in addition to controls derived from expressed genetic information). The objective of this chapter is to propose a simple example of a developmental electronic system which consists of polymorphic gates and so which can be influenced by an external control signal.

The proposed research is also based on our previous work presented in Chapter 5 in which arbitrarily large sorting networks were evolved using an application specific instruction-based developmental model. In particular, programs were evolved which are able to construct an $N+1$-input or $N+2$-input sorting networks from an existing N-input SN. Here a modified developmental system is presented that is able to work with polymorphic gates which are considered as basic building blocks of the growing circuits. Therefore, the target circuit can specialize its functionality according to the environment which is sensed through polymorphic gates. In this stage of research it is assumed that arbitrary polymorphic gates exist; however, only bi-functional (polymorphic) gates are considered in the current phase of our research. The results are presented on growing even/odd parity circuits and sorting/median networks. Since the circuits are able to “grow” and preserve their required function during the development, it is a case of continual approach similarly to that in Chapter 5.

7.1 Polymorphic electronics

Polymorphic circuits, introduced by Stoica's team at JPL, are in fact multifunctional circuits. The change of their behavior comes from modifications in the characteristics of components (e.g. in the transistor's operation point) involved in the circuit in response to controls such as temperature, power supply voltage, light, etc. [SZKL02]. Polymorphic circuits are able to work in several modes of operation corresponding to different operational conditions. Table 7.1 gives examples of the polymorphic gates reported in literature. Most of them have been designed by means of evolutionary techniques. The mentioned NAND/NOR gate is the most famous example [SZG$^+$04]. The circuit consists of 6 transistors and was fabricated in a 0.5-micron CMOS technology. The circuit is stable for $\pm 10\%$ variations of Vdd and for temperatures in the range $20°C – 200°C$.

Potential applications are discussed in [SZKL02]. Polymorphic electronics should allow engineers to build inherently adaptable digital circuits. By changing the temperature, Vdd or some other conditions a circuit can change its functionality immediately, with no reconfiguration overhead. The potential applications include: special circuits that are able to decrease resolution of digital/analog converters or speed/resolution of a data transmission when

Gate	Control values	Control method	Reference
AND/OR	27/125°C	temperature	[SZK01]
AND/OR/XOR	3.3/0.0/1.5V	external voltage	[SZK01]
AND/OR	3.3/0.0V	external voltage	[SZK01]
AND/OR	1.2/3.3V	Vdd	[SZKL02]
NAND/NOR	3.3/1.8V	Vdd	[SZG$^+$04]
NAND/NOR	5.0/3.3V	Vdd	[RSP08]

Table 7.1. Examples of existing polymorphic gates

a battery voltage decreases, circuits with a hidden/secret function that can be used to ensure security, intelligent sensors and novel solutions for reconfigurable cells and function generators in reconfigurable devices (such as FPGA and CPLD) [SZKL02].

The design of polymorphic circuits is considered as main problem because these circuits typically utilize normally unused characteristics of electronic devices and working environment; conventional design techniques are usually not able to deal with that. Therefore, the current research of the polymorphic circuits design methods often deals with the simulated polymorphic gates (that may even not have to exist physically) rather than working only with the existing polymorphic gates directly in hardware. For example, Sekanina has proposed a method for the evolutionary design of gate-level digital polymorphic circuits by means of cartesian genetic programming. Polymorphic gates have been considered as basic building blocks [Sek05]. For instance, a circuit was evolved operating as 2-bit adder in environment E1. This circuit can also work as 2-bit multiplier in environment E2. A typical feature of polymorphic gate-level circuits is that their topology (i.e. connection of components) is fixed; however, the components can change the functionality.

Code	Gate	Code	Gate	Code	Gate	Code	Gate
0	AND/I	5	OR/I	10	XOR/I	15	NOT/I
1	AND/AND	6	OR/AND	11	XOR/AND	16	NOT/NOT
2	AND/OR	7	OR/OR	12	XOR/OR	17	I/NOT
3	AND/XOR	8	OR/XOR	13	XOR/XOR	18	I/I
4	I/AND	9	I/OR	14	I/XOR		

Table 7.2. List of gates. Some of them are polymorphic. "I" denotes the identity function (a buffer).

7.2 Development of polymorphic circuits

All the approaches to development of digital circuits, mentioned in the previous chapters, can be enriched by using polymorphic gates. Again, the aim is to develop arbitrarily large circuits.

Since the polymorphic gates are to be present in the circuits, causing multifunctional (bi-functional) trait of their operation, the development is much more difficult because it must design a proper circuit topology with respect to the polymorphic gates. Moreover, the requirement of generality (i.e. ability of the circuit to develop to theoretically arbitrary size) makes this process a challenging task.

In this section, a genetic algorithm is combined with the continual development using the instruction-based approach to design arbitrarily large polymorphic circuits, in particular, polymorphic odd/even parity circuits and polymorphic sorting networks (with increasing/decreasing order of the sorted sequences and median networks).

Similarly to the approach described in Section 5.3, the GA is utilized to design a program by means of which arbitrarily large polymorphic circuit can be developed. Genetic algorithm is employed to evolve a sequence of instructions (a program), by means of which an initial simple instance of the problem (an embryo) will *grow* to form a more complex circuit.

Again, each instruction of the program consists of three integers (*opcode*, $arg1, arg2$), where *opcode* represents the operation code of the instruction and $arg1$ and $arg2$ denote its arguments. The meaning of arguments depends on the type of the instruction (instructions with no operands are allowed; then $arg1$ and $arg2$ are not interpreted). A developmental step is understood as a single application of the program on a circuit created in the previous developmental step to construct more complex circuit. After each developmental step the number of circuit inputs increases according to the size of the developmental step that is determined as the difference of the number of inputs of two subsequent circuits. The size of developmental step is specified by the designer before the evolution is executed.

Figure 7.1 shows the structure of a basic building block utilized in the developmental system. Each building block consists of two gates, $g1$ and $g2$, representing logic functions performed in the gates and indices of inputs of the gates $i1$ and $i2$. Note, however, that each gate may perform two functions because it is a case of polymorphic gates. Each building block is then encoded using a 4-tuple $(i1, i2, g1, g2)$. The target circuit is represented as a sequence of building blocks. Table 7.2 shows the list of gates that have been utilized for the development and their integer codes. Standard (non-polymorphic gates) are represented by a pair of identical functions (e.g. 1 – AND/AND for a common AND gate). Polymorphic gates consist of two different functions (e.g. 2 – AND/OR). A model of wire was also included representing the identity function (e.g. 18 – I/I). The wire can also be present in polymorphic gates as it is suppose that arbitrary polymorphic gate exists. All the polymorphic

gates considered for the development are bi-functional. For instance, the gate AND/OR operates as a common AND in environment (value of the control signal) E1 and function OR is performed in environment E2.
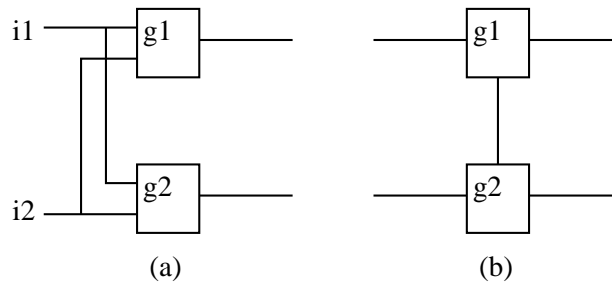

Fig. 7.1. The basic building block: (a) logic structure, (b) symbolic notation. $g1$ and $g2$ represent logic functions performed in the gates. $i1$ and $i2$ denote the input indices.
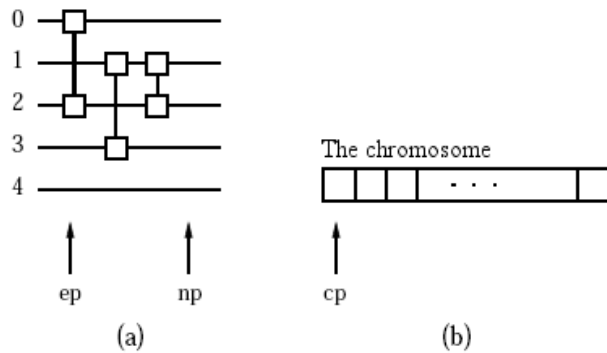
Fig. 7.2. Configuration of developmental system for a polymorphic circuit: (a) the growing circuit, (b) the program which is represented by a chromosome

A sample configuration of the proposed developmental system is shown in Figure 7.2. The embryo pointer (ep) indicates the building block that is actually processed by the instruction selected through the instruction pointer (cp). As the result of application of the instruction new gates will be placed on the position denoted by the next-position pointer (np). This pointer denotes the first empty position where the circuit will grow to. The instructions of the program are processed sequentially. The process of construction terminates when either all the instructions of the program are executed or the end of embryo is reached. After executing an instruction the pointers ep, cp and np are updated. An "empty" embryo is used at the beginning of the develop-

mental process, i.e. no particular information is known about the structure of the initial circuit, which is, therefore, considered as a 4-tuple $(0, 0, I/I, I/I)$. Considering that, there must be instructions in the instruction set that are able to set the input signals of the polymorphic gates and their functions (e.g. instruction MODIS and MODFS). These so-called *modify*-instructions only modify the gates denoted by the embryo pointer (ep) without copying them and hence the next-position pointer (np) remains unchanged after their execution. The instructions which are responsible for growing the circuit are of the types *copy* or *copy-and-modify* (e.g. instructions CPOS or CPMIS). Every instruction is used in two variants (e.g. CPMIS and CPMIN), whose difference lies in updating strategy of the embryo pointer (ep) after execution of the instruction. Table 7.3 lists all the utilized instructions. For example, CPOS copies a pair of gates from position given by ep to the position given by np; the embryo pointer remains unchanged in this case. The position (indices of inputs) of the newly created or modified gates depends on the position of the gates being processed and the number of inputs of currently constructed circuit (w).

7.3 Evolutionary system setup

A simple genetic algorithm is utilized to find a program whose repeated application on an existing circuit will create a larger circuit. All the experiments were performed with the following settings: standard one-point crossover with the probability 0.55, the probability of mutation 0.04, population size 40, tournament selection mechanism with the base 3, the maximal number of generations 20,000.

The genetic algorithm works with chromosomes of constant length. The number of instructions in a chromosome was determined experimentally during our previous research. Note, that all the evolved programs work only with either even or odd number of inputs, i.e. the developmental steps of size 2 is considered.

The objective is to develop arbitrarily large circuits; however, only four developmental steps are considered in the fitness calculation in order to make the time of evolution reasonable. For a single environment the fitness value is determined as $f = f(2) + f(4) + f(6) + f(8)$ for even-input circuits and $f = f(3) + f(5) + f(7) + f(9)$ for odd-input circuits, where $f(i)$ is the number of correctly processed testing sequences by the circuit with i inputs. Therefore, the maximum fitness value that is possible to reach is $f_{max} = 2^2 + 2^4 + 2^6 + 2^8 = 340$ for even-input circuits and $f_{max} = 2^3 + 2^5 + 2^7 + 2^9 = 680$ for odd-input circuits. For the second environment the fitness value is calculated in the same way. A software simulator is utilized for the circuits evaluation. At the end of evolution it has to be verified whether the resulting programs are general, i.e. able to produce arbitrarily large circuits (typically the verification is performed for up to 28 inputs).

Op. code	Name	Arg1	Arg2	Meaning
0	CPOS	–	–	copy the pair of gates from ep to np; $cp = cp + 1, np = np + 1$
1	CPON	–	–	copy the pair of gates from ep to np; $cp = cp + 1, np = np + 1, ep = ep + 1$
2	CPNS	p	–	copy $w - p$ pairs of gates; $cp = cp + 1$, $np = np + w - p$
3	CPNN	p	–	copy $w - p$ pairs of gates; $cp = cp + 1$, $np = np + w - p, ep = ep + w - p$
4	CPMIS	p	q	copy the pair of gates from ep to np and do $i_1 = (i_1 + p) \bmod w, i_2 = (i_2 + q) \bmod w$, $cp = cp + 1, np = np + 1$
5	CPMIN	p	q	copy the pair of gates from ep to np and do $i_1 = (i_1 + p) \bmod w, i_2 = (i_2 + q) \bmod w$, $cp = cp + 1, np = np + 1, ep = ep + 1$
6	CPMFS	p	q	copy the gates from ep to np and do $f_1 = p, f_2 = q, cp = cp + 1, np = np + 1$
7	CPMFN	p	q	copy the gates from ep to np and do $f_1 = p$, $f_2 = q, cp = cp + 1, np = np + 1, ep = ep + 1$
8	MODIS	p	q	modify inputs of the gates at ep as follows: $i_1 = (i_1 + p) \bmod w, i_2 = (i_2 + q) \bmod w$, $cp = cp + 1$
9	MODIN	p	q	modify inputs of the gates at ep as follows: $i_1 = (i_1 + p) \bmod w, i_2 = (i_2 + q) \bmod w$, $cp = cp + 1, ep = ep + 1$
10	MODFS	p	q	modify functions of the gates at ep as follows: $f_1 = p, f_2 = q; cp = cp + 1$
11	MODFN	p	q	modify functions of the gates at ep as follows: $f_1 = p, f_2 = q; cp = cp + 1, ep = ep + 1$
12	NOP	–	–	an empty instruction: $cp = cp + 1$

Table 7.3. The instruction set for development of polymorphic circuits. p and q represent the arguments of the instruction; i_1 and i_2 denote the indices of inputs of the polymorphic gates; f_1 and f_2 are functions of the gates and w is the number of inputs of the circuit being created.

The experiments were conducted on common PCs running RedHat-based Linux operating system. The hardware configuration consists of a 2.0 GHz processor and 512 MB RAM. The SGE system was utilized so that several independent experiments can be executed on different PCs in parallel.

7.4 Experimental results

Two sets of experiments were conducted: (1) the development of polymorphic even/odd parity circuits and (2) the development of polymorphic sorting networks and medians. In each experiment the objective was to evolve a program

for the construction of arbitrarily large polymorphic circuit of the given class. The circuits are intended to grow continually and theoretically infinitely. The experimental results of each category are summarized in the next sections.

7.4.1 Polymorphic parity circuits

The parity problem (i.e. the decision if the number of boolean variables with assignment true from a set of boolean variables is even or odd) has emerged from the genetic programming framework as a difficult problem for program induction [J. 92]. For instance, the following works were focused on investigating general solutions for the parity problem. Wong et al. evolved recursive functions for the even-n parity problem from noisy training examples [WL96]. They utilized logic grammars specifically devised to solve the parity problem. Huelsbergen presented general solutions to the parity problem by evolving machine-language representations [Hue98]. His approach considered a set of boolean variables represented as a string interpreted by means of an integer of appropriate size. Although the approaches to the parity problem solving often do not consider XOR gate (that represents a typical function for the straightforward effective parity computation), the goal is usually to design non-polymorphic parity circuits.

In this section, a general approach is introduced using an evolved program for the construction of generic even/odd parity circuits at the gate level employing polymorphic gates. Since the development of polymorphic circuits is difficult due to the limited circuit topology, the entire set of building blocks shown in Table 7.2 will be utilized for the development.

Some general programs were successfully evolved for this class of circuits. According to the environment the circuits calculate either even or odd parity. The solution is usually based on the conventional XOR gate (no. 13 in Table 7.2) and a polymorphic NOT switch (no. 17 in Table 7.2). Its structure is not surprising; however, the structure was designed fully automatically without any supporting domain knowledge (the design started with the empty embryo $(0, 0, I/I, I/I)$). Figure 7.3 shows two instances of polymorphic circuits calculating even/odd parity functions. 35 general programs have been gained out of 200 independent runs of the evolutionary design process for a five-instruction chromosome. 42 programs consisting of six instructions were evolved out of 200 independent runs of the evolutionary process from which 36 were recognized as general.

7.4.2 Median and sorting networks

The evolutionary process has succeeded in the design of structurally variable median circuits. These circuit structures, considered as polymorphic circuits, do NOT compute different functions in different environments. However, the way in which the median is calculated depends on the environment. Therefore,
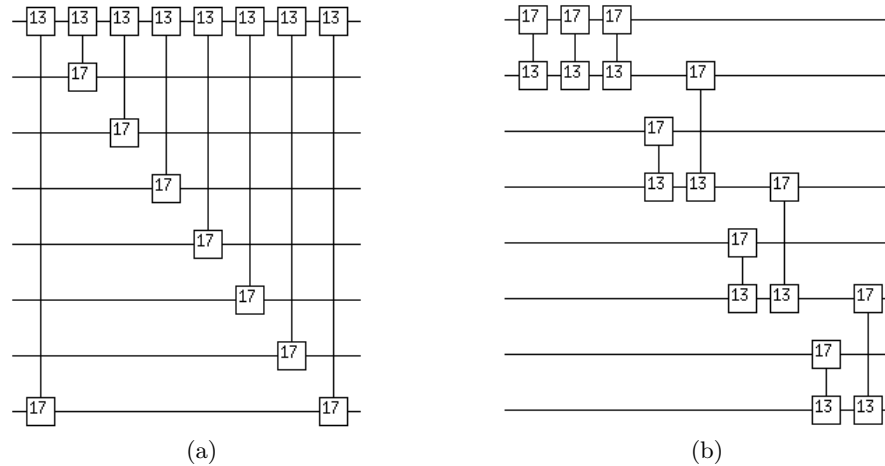
Fig. 7.3. Examples of evolved generic polymorphic odd/even parity circuits. Only even-input circuits have been considered. (a) A circuit developed by means of 5-instruction program [MODIS 4 3] [MODFS 13 17] [CPMIS 0 1] [MODIS 0 2] [CPOS 3 1]. (b) A circuit developed using 6-instruction program [MODFS 17 13] [MODIS 2 3] [CPMIS 2 2] [CPMIS 2 0] [CPMIN 1 2] [MODIN 0 3].

the environment determines, through polymorphic gates, physical implementation of median circuit.

In addition, the mentioned polymorphic circuits can also work as sorting networks. It is interesting that they sort the input sequences in increasing or decreasing order according to the environment. Figure 7.4 shows evolved polymorphic median and sorting network circuit built by means of a seven-instruction program. Unfortunately, this program is not general. Median networks can be constructed only up to 13 inputs and sorting networks only up to 11 inputs. However, these circuits exhibit better properties (the number of comparators) than the conventionally constructed circuits (e.g. bubble-sort network) [Knu98]. The evolutionary process has succeeded 68 times in 200 independent runs, from which eight programs are able to build fully functional median circuits up to 13 inputs.

Figures 7.5 and 7.6 show polymorphic median and sorting networks created by means of a seven- and eight-instruction program. These programs are general. 45 programs were obtained out of 200 independent runs of the evolutionary process, from which 5 programs are general.

The evolved circuits use gate 2 (AND/OR) and 6 (OR/AND) which means that they sort the input sequences in increasing order in the first environment and in decreasing order in the second environment. As the median value is taken from the middle output, it does not depend on signals coming from the environment.

Fig. 7.4. Evolved polymorphic median and sorting network created by means of the program [MODFS 2 6] [CPMIS 2 2] [CPMIS 1 2] [CPMIS 3 2] [CPMIS 0 1] [CPMIN 1 1] [CPNN 0 2] (a 7-instruction program, odd number of inputs)

Fig. 7.5. Evolved polymorphic median and sorting network created by means of the program [CPMIS 2 2] [MODFS 2 6] [CPMIS 1 2] [CPMIS 3 2] [CPMIS 0 1] [CPMIS 1 1] [CPNN 2 4] [CPNN 4 4] (a 8-instruction general program, odd number of inputs)
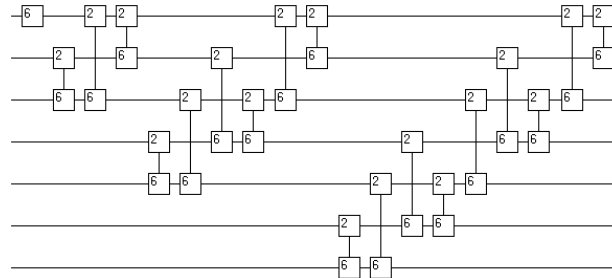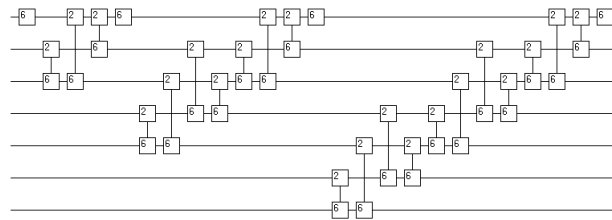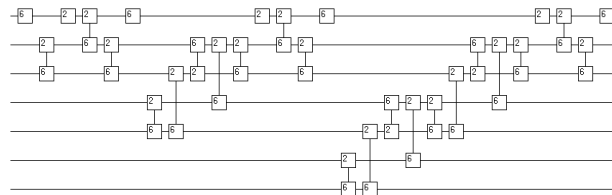

Fig. 7.6. Evolved polymorphic median and sorting network created by means of the program [MODFS 2 6] [CPMIS 2 2] [CPMIS 1 2] [CPMFS 6 2] [CPMIS 0 1] [CPNN 3 0] [CPMIS 4 2] [CPNN 1 2] (a 8-instruction general program, odd number of inputs)

7.5 Discussion

General programs for the continual development of polymorphic parity circuits were evolved. Considering the conventional design of this class of circuits a tree-based topology of XOR gates is typically implemented. The results obtained from our system are area inefficient compared to the conventional solutions. This can be explained by the encoding that was used. The design of generic parity circuits using the conventional topology would evidently

be very difficult with the proposed developmental encoding. Though the key component of the evolved design is a common (non-polymorphic) XOR gate. Therefore, the evolution utilized the same elements as the conventional design uses. Some of them are meaningless regarding the topology caused by the developmental encoding and the structure of the building blocks. Finally, the effective output is switched by the polymorphic I/NOT gate which is probably the simplest solution – there is no innovation in this case against the conventional principle.

In case of sorting networks, evolution has discovered that by exchanging AND–OR gates for OR–AND gates, the ordering of sorted sequence can be changed. There is no innovation; human designer would construct the circuit in the same way. Although the evolution could use many types of gates, it has utilized the same gates as a human designer uses. The implementation of AND as well as OR gate costs 6 transistors in the standard CMOS technology. Surprisingly, the cost of polymorphic AND/OR gate controlled by temperature is also 6 transistors [SZK01]. If one were able to build OR/AND gate with the same cost, the resulting polymorphic sorting network would consist of the same number of transistors as the original one whose behavior cannot be changed.

The results presented in this chapter confirm Hypothesis 1 because generic solutions are possible to be generated by means of continual development and partially also Hypothesis 3 because more effective general polymorphic sorting networks have been developed in comparison with the case if the conventional insertion or selection sort would be utilized for their construction using the polymorphic gates applied by the evolution (note that only even-input circuit have been investigated).

Despite a big effort was put into the development of other types of polymorphic circuits (such as adder/sorting network and parity/Boolean symmetry circuits etc.), no functional result has been obtained yet. The explanation could be as follows. The number of correct suitable topologies which perform the required behavior is very limited with the proposed encoding. Therefore, the probability is very low that a single topology can represent two different behaviors (e.g. n-bit adder and n-bit multiplier) in two different environments.

Of course, because of the utilized representation, it is always possible to manually merge two different circuits into a single working polymorphic circuit. The method is as follows: Construct the resulting circuit from left to right. If the first circuit requires logic function L_1, create polymorphic gate L_1/I. If the second circuit requires logic function L_2, create polymorphic gate I/L_2. Then the resulting circuit (consisting of polymorphic gates) will perform the first function in the first environment and the second function in the second environment. However, we are not interested in this type of solution, since there is no innovation visible.

In real biological systems as well as in some artificial developmental systems (e.g. in [Gor03]) the interplay between a growing solution and its environment is very complex. In the proposed developmental system the interplay

122   7 Development of generic polymorphic circuits

practically does not exist; this kind of interaction with the environment was not the goal in this work. This chapter was especially to demonstrate another possibility of introducing an external information to the growing digital circuit that can alter its behavior. Some of the experiments were successful and the results represent the first case of automatic evolutionary design of polymorphic circuits by means of the development. The emerging concept of polymorphic electronics was involved.

8

Conclusions

Computational development represents a new area of computational intelligence, serving as a tool for evolutionary techniques to overcome the problem of scale. This book have proposed a contribution related to a limited subset of computational development. The research was focused to the developmental encodings based on simple application-specific instructions. Therefore, our approach was termed as instruction-based development.

Although the concept of instruction-based development is based on a very simple principle — applying evolving computer programs to construct structure — we believe that it can be used generally. In this research the application of the instruction-based development was presented demonstrating the possibilities of the design of generic circuit structures. The development was applied at the structural level. The instructions have been intended to create or possibly manipulate the building blocks of the structure under development. It represents the significant difference from the traditional genetic programming approach. Though the developmental genetic programming works in the similar way (i.e. building electronic circuits using building blocks), the construction of arbitrarily large, scalable objects was not its main goal. Moreover the DGP approach was performed at a lower level of abstraction. Our approach, utilizing the instruction-based development, have worked with more complex building blocks, therefore more complex and extensive (generic) structures could be designed.

Two different approaches to performing the instruction-based development were identified (1) the continual development and (2) the parametric development. The experimental results have demonstrated capabilities of these approaches to develop generic (i.e. arbitrarily large and scalable) structures of digital circuits of different classes. Both the concepts utilize application-specific programs consisting of the simple instructions, which are the subject of evolution, to realize the development of generic structures.

8.1 Continual development

The continual development approach has been applied to the evolutionary design of generic sorting networks. This class of circuits has shown to be suitable for demonstrating abilities of continual instruction-based development because of the structure of the sorting networks. If a comparator is chosen as a building block, the development of sorting networks can be performed by means of the iterative process (in this case on the basis of repeated application of the evolved program). This is possible as the comparators do not alter the result of a sorting network if inserted into (or appended to) an existing working circuit. This feature was exploited in order to perform continual development of this class of circuits in the same way as some conventional general approaches do. For instance, the straight-insertion principle known from the theory of sorting allows to create larger sorting network from a working smaller sorting network by appending a suitable arrangement of comparators to the existing network. This approach has been utilized in our developmental systems.

In contrast to the straight-insertion algorithm, which can develop sorting networks possessing one more input in comparison with the preceding instance in each developmental step, this work was focused on the design of either odd-input or even-input sorting networks (i.e. the number of inputs has been increased by two in each developmental step starting from an embryo of suitable size). Moreover, the developmental steps of sizes three and four were involved during advanced experiments conducted in this domain. It has been supposed that this modification in the development will enable us to develop more efficient generic sorting networks in comparison with the conventional principle.

This assumption has been confirmed. Several general programs were evolved that can develop generic sorting networks applying the mentioned restrictions to the number of inputs. The best solution has been discovered for even-input sorting networks when the developmental step of size two has been applied. On the basis of this result a new general construction algorithm for the sorting networks was invented. The generality of the new principle has been proved by mathematical induction. The properties of the sorting networks created by means of the new approach (i.e. the number of comparators and delay) are substantially better if compared with the conventional principle. However, no better solution (with respect to the evolved best one) has been obtained for larger developmental steps. Therefore, our next presumption emerged from the ongoing research — the larger developmental step the better sorting networks could be developed — has not been able to confirm yet.

It is true that the innovative solutions have been obtained using the developmental system that relatively much domain-specific information has been supplied to. However, the new approaches to the design of generic sorting networks have not been known before. The best evolved solutions represent

a significant outcome as the innovation discovered automatically by means of evolutionary techniques is a rare case in the evolutionary design field especially in the situation when a solution to all instances of the given problem has been found instead of discovering a single solution.

Next, the concept of the continual development was applied on the design of generic polymorphic circuits. Solutions to polymorphic even/odd parity circuits and polymorphic sorting and median networks, that differ in the algorithm for obtaining the results (either increasing or decreasing sorting process depending on the control signal) have been evolved. If one assumes that arbitrary two-function polymorphic gate could exist, the evolution, in principle, discovered multifunctional behavior of the resulting polymorphic sorting and median networks totally for free in comparison with the case when such the circuits would be created conventionally using common logic gates. Only either even or odd numbers of inputs were considered during these experiments. In case of the polymorphic sorting networks, the evolution has found programs that can construct solutions with reduced number of comparators and delay if compared to the generic conventional approach (e.g. straight insertion sort), similarly to the development of non-polymorphic sorting networks. Although no other significant innovation has been discovered, our approach represent the first case when generic polymorphic circuits are designed using evolutionary techniques combined with the development.

8.2 Parametric development

Parametric development represents an alternative approach to the continual approach discussed in the previous section. Instead of developing the target object by iterative applications of the evolved program, the construction of a specific instance (size) of the circuit is always developed from the start. A parameter is involved to specify the size of the circuit; it influences the developmental program and determines the number of developmental steps needed to be performed for creating a working circuit. An external information was introduced (that was called the environment) for an additional control of the developmental process. In the experiments conducted the environment is primarily intended to enable the design system to develop generic structures which may contain irregularities.

The parametric development approach was applied to the design of arbitrarily large combinational multipliers. In general the evolution of multipliers is considered as hard problem because of typically rugged fitness landscape in case of the gate-level representations. The method for constructing generic multipliers was inspired by the structures of the conventional circuits of this class for which general construction algorithms exist. More complex building blocks have been involved in our system. In addition, an advances instruction-based developmental system has been introduced that is able to tackle the

126   8 Conclusions

increased complexity of the target structures in comparison with the previous research related to sorting networks.

The initial experiments have shown that the new developmental system is able to design generic combinational multipliers by means of developmental programs that are the subject of evolution. The best resulting circuit are identical to the common combinational multipliers composed of AND gates and adders as basic building blocks. The ability of adaptation of the evolutionary process to different environments was demonstrated. It means that different external control of the developmental process can lead to the construction of the circuits of the same functionality while retaining the ability of the system to design arbitrarily large instances of the target object. Although no innovation has been discovered in this case, these experiments represent the first case when generic multipliers were designed automatically by means of genetic algorithm combined with the development.

In order to improve the efficiency of the initial results, the developmental system has been modified taking inspiration from the concept of carry-save multipliers which exhibit shorter delay in comparison with the common multipliers. Simplified building blocks have been considered in comparison with the approach discussed in the previous paragraph and the developmental system has been modified in order it could work with that building blocks. Although some limitations have been introduced into the developmental process related to the requirement of developing generic multiplier structures, the developmental encoding has been kept at the level which restrict the evolution as little as possible against the initial approach.

The evolution has succeeded in the automatic design of generic multipliers using the improved developmental model. However, this design process has been much more difficult. Although some general programs have been evolved that are able to construct effective carry-save multipliers, most of the results have tended to the development of common multipliers structures. Similarly to the approach discussed in the initial experiments, no innovation has been discovered in this set of experiments. However, the objective of developing effective generic carry-save multipliers has been fulfilled.

The crucial issue related to the development of combinational multipliers is the circuit representation with respect to the requirement of producing generic solutions. In fact, the obtained results are expectable because the developmental model has been devised with the high inspiration by the conventional solutions. For the successful development of generic multipliers relatively complex building blocks had to be involved. This evidently limits the evolution in finding more effective results that would be, for example, achievable at the gate-level representation. However, the design of a generative encoding that would be evolvable for the development of generic multipliers at that level of abstraction.

8.3 Findings from experimental results

The principle of instruction-based development can be viewed as the linear genetic programming approach applied at the structural level. It is possible to introduce a general computational model into the developmental system with respect to the instructions chosen. Therefore, countless numbers of applications might be considered in various domains, where the instruction-based development could be utilized, regardless of whether generic solutions are required or not.

Two different concepts of the instruction-based development were proposed: the continual development and the parametric development. In both categories it was demonstrated that many solutions (programs) can be evolved which are able to construct generic structures. Three hypotheses have been formulated in Introduction with respect to the proposed developmental schemes. All of them have been confirmed because (1) generic solutions have been generated using continual development, (2) generic solutions have been generated using parametric development and (3) generic conventional solutions have been rediscovered and innovative generic solution have been invented.

It was shown that the instruction-based development, in general, is possible to use to generate scalable solutions. In the applications presented, a solution was found not to a single instance of the given problem but for all the instances considering only a finite number of evaluations during the evolutionary process. Note that the generic developmental design was, in fact, the crucial issue of our research. However, what really does the term "scalable" mean in context of this work? Let us go back to this question in the next section.

The automatic discovery of an innovative solution by means of the evolutionary algorithm represents a difficult task. To make the evolution to discover innovative result in the generic scope is much more difficult. However, what really does the term "generic innovative solution" mean in context of this work? Again, let us leave this question unanswered before the next section. Meanwhile, the obtained results have shown that the instruction-based development is (at least partially) capable of discovering generic innovations (or developing a structure that has not been seen so far). In this case, the innovative solution evolved by the genetic algorithm provided an inspiration of inventing a new general construction algorithm that was, in addition, proved formally.

Finally, the concept of both continual and parametric development has been successfully applied to the design of totally different structures (which is dependent on the circuit representation). For instance, the sorting networks were developed in a one-dimensional array while the multipliers were developed inside a rectangular grid. These results suggest that the proposed approach and the representation of the given problem in particular may be adjusted to potentially arbitrary structure that is possible to describe algorithmically and subsequently to evolve effectively. However, what actually means

128   8 Conclusions

"to adjust to" or, more precisely, how could one adjust a developmental system to satisfy the needs of a specific problem for effective evolution?

8.4 Common problems and open questions

In the previous section, a discussion was proposed considering the findings obtained from the outcomes of the research performed in this work. Several questions have arisen which we will try to answer in this section together with other crucial issues related to the developmental evolutionary design.

The scalability of the evolved solutions has been mentioned. In general, a specific form of developmental mappings was investigated in this book representing a potential solution to overcome the problem of scale in the evolutionary design. As stated in Introduction, two different views of the scalability problem have to be distinguished: (1) the problem of scale at the structural level and (2) scalability of the level of fitness function calculation. In this case, mainly the second variant had to be taken into account.

Since the time complexity of the fitness evaluation grows exponentially as the number of inputs of the developing circuit increases, a limitation was needed to introduce in order to fulfil the objective of designing generic circuit structures. This issue has been tackled by evaluating only a finite number of solutions created by the development during the evolutionary process. Then the generality of the resulting programs were verified. Although the verification process is also time-consuming, it represents a reasonable part of the design process because only a limited subset of the evolved solutions have been verified. From this point of view it may be evident that that the scalability of the fitness calculation has been overcome in this case. However, most of the evolved structures do not allow the resulting circuits to operate effectively enough.

This issue can be explained as follows. If the building blocks utilized for the development are complex (i.e. a large amount of domain-specific information is supplied to the system), it may lead to evolvable representation which, however, limit the evolution in exploring promising parts (i.e. parts containing effective solutions) of the search space. In fact, the effective solutions may not even be achievable using the complex building blocks. On the other hand, if simple building blocks are considered (providing small amount of domain-specific information), a complex structure needs to be developed which is difficult to evolve (the problem of scale at the level of structure). Unfortunately, there is no general prescription for the problem encoding in order to evolve effective solutions.

As it was mentioned in the previous paragraph, an "innovative generic solutions" have been discovered during the experiments. However, the problem is the rate of innovation in comparison with the best conventional solutions. Of course, a direct encoding can evolve much more effective solutions but only for limited sizes. Several representations were proposed that are able to generate

continually innovative solutions in comparison with the conventional generic solution of the same type. Therefore, the innovation obtained is needed to consider with respect to the generic design. The reduction in the genome size, search space and the time needed to evolve a solution are the main advantages of the proposed approaches. As will be discussed in the next paragraph, the effective problem encoding represents the common issue in the evolutionary design and in the developmental evolutionary design in particular. However, any result may suggest ideas for future research that may potentially move the know edge in a given area.

Different generative encodings were introduced for the developmental evolutionary design of generic circuit structures. All of them have led to successful design with respect to the main objective of this work. However, it is in no case possible to claim that they are the best for the given problems. Moreover, one can ask how to design a developmental system for other problems if there ever are any that would be easily evolvable using the common technology. A related question is the selection of building blocks and the amount of domain knowledge needed to successful evolution. As the experiments showed all these aspects of a design system may be crucial. However, it seems that designing an efficient developmental system and the representation of candidate solutions is at least as difficult as to design an evolutionary algorithm for a given problem working with direct encoding. Unfortunately, there is no definite instruction of how to do this. Therefore, the area of developmental systems will rather exhibit features of experimental work and will require research specialists for various domains also in the future.

8.5 Possibilities of future research

A limited subset of the computational development area has been covered by the experiments. Though the results have shown a promising potential of applying the instruction-based developmental design in combination with the artificial evolution. Advantages and disadvantages of the evolved solutions were discussed and some problems related to the applied approaches were mentioned which represent a basis for possible directions of the future research.

The experiments have shown that more efficient generic sorting networks in comparison with the appropriate conventional approach can be developed if larger developmental step is considered. This capability was demonstrated for developmental steps of sizes two, three and four. However, in the last two cases no optimization has been observed in comparison with size two of the developmental step. Since much more effective sorting networks are known than that were obtained by our approach, it could be interesting to investigate more methods of evolutionary development of this class of circuits. The present experiments have not provided better solutions.

130   8 Conclusions

The outcomes in the combinational multipliers area represent rather initial results in this field. The developmental model which was utilized has shown the potential of designing more complex generic structures in comparison with the sorting networks. However, more complex building blocks had to be utilized together with relatively large amount of domain-knowledge in order to evolve a successful generic solution. Moreover, an external information for an additional control of the developmental process was introduced that enabled do develop generic structures possessing irregular parts. The next research in this field could be focused on the applications of the developmental system to other problems with simpler building blocks (e.g. adders) and to investigate advanced methods and possibilities of applying external control of the development.

In general, the proposed concept of instruction-based continual and parametric development has provided an overall insight on the possibilities of generic evolutionary design. We do believe that the idea of evolving programs for infinitely growing objects is generally applicable. The key issues of the problem representation and design of the developmental encoding will constitute the main directions of our future research.

References

[AC94]   H. Adeli and N. Cheng. Concurrent genetic algorithms for optimization of large structures. *ASCE Journal of Aerospace Engineering*, 7(3):276–296, 1994.

[AHH03]   T. Aoki, N. Homma, and T. Higuchi. Evolutionary synthesis of arithmetic circuit structures. *Artificial Intelligence Review*, 20:199–232, 2003.

[AJL⁺94]   B. Alberts, A. Johnson, J. Lewis, M. Raff, K. Roberts, and P. Walter. *Molecular Biology of the Cell*. Gerland Publishing, 1994.

[B96]   T. Bäck. *Evolutionary Algorithms in Theory and Practice*. Oxford University Press, 1996.

[Ban93]   W. Banzhaf. Genetic programming for pedestrians. In *Proc. of the Fifth International Conference on Genetic Algorithms (ICGA'93)*, pages 628–638, San Francisco, CA, 1993. Morgan Kaufmann.

[BB07]   M. F. Brameier and W. Banzhaf. *Linear Genetic Programming*. Springer, 2007.

[BNKF97]   W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic Programming – An Introduction. On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, 1997.

[CM01]   S. S. Choi and B. R. Moon. A hybrid genetic search for the sorting network problem with evolving parallel layers. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 258–265, San Francisco, California, USA, 2001. Morgan Kaufmann.

[CM02a]   S. S. Choi and B. R. Moon. Isomorphism, normalization, and a genetic algorithm for sorting network optimization. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 327–334, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc.

[CM02b]   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.

[Coh07]   W. W. Cohen. *A Computer Scientist's Guide to Cell Biology*. Springer, 2007.

[Cra85]   N. L. Cramer. A representation for the adaptive generation of simple sequential programs. In *Proc. of an International Conference on Ge-*

132   References

        netic Algorithms and the Applications, pages 183–187, Grefenstette, 1985.
        Carnegie Mellon University.

[Dar06] C. Darwin. *On the Origin of Species By Means of Natural Selection.*
         Dover Publications, 2006.

[DB94]   F. Dellaert and R. Beer. Toward an evolvable model of development for
         autonomous agent synthesis. In *Proc. of the 4th International Workshop
         on the Synthesis and Simulation of Living Systems (Artificial Life IV)*,
         pages 246–257. MIT Press, 1994.

[DB96]   F. Dellaert and R. Beer. A developmental model for the evolution of
         complete autonomous agents. In *Proc. of the 4th International Confer-
         ence on Simulation of Adaptive Behavior*, pages 393–401, Cambridge,
         MA, 1996. MIT Press-Bradford Books.

[Del95] F. Dellaert. Toward a biologically defensible model of development, Mas-
         ter thesis. Case Western Reserve University, Cleveland, 1995.

[DG91]   K. Deb and D. E. Goldberg. Messy genetic algorithm in C, report no.
         91004. Technical report, Illinois Genetic Algorithms Laboratory (Illi-
         GAL), 1991.

[DM92]   D. Dasgupta and D. R. McGregor. Nonstationary function optimization
         using the structured genetic algorithm. In *Parallel Problem Solving from
         Nature 2*, pages 145–154, Brussels, 1992. Elsevier Science Pub.

[ES03]   A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computation.*
         Springer, Berlin Heidelberg, 2003.

[FOW66] L. J. Fogel, A. J. Owens, and M. J. Walsh. *Artificial Intelligence through
         Simulated Evolution.* Wiley, New York, 1966.

[Gar70] M. Gardner. Mathematical games: The fantastic combinations of john
         conway's new solitaire game "life". *Scientific American*, 223(October
         1970):120–123, 1970.

[GB02]   T. G. W. Gordon and P. J. Bentley. Towards development in evolvable
         hardware. In *Proc. of the 2002 NASA/DoD Conference on Evolvable
         Hardware*, pages 241–250, Washington D.C., US, 2002. IEEE Press.

[GB05a] T. G. W. Gordon and P. J. Bentley. Bias and scalability in evolution-
         ary development. In *Proceedings of the 2005 Genetic and Evolutionary
         Computation Conference, GECCO 2005*, pages 83–90. ACM Press, 2005.

[GB05b] T. G. W. Gordon and P. J. Bentley. Development brings scalability to
         hardware evolution. In *Proc. of the 2005 NASA/DoD Conference on
         Evolvable Hardware*, pages 272–279. IEEE Computer Society, 2005.

[Geo94] F. George. Hybrid genetic algorithms with immunisation to optimise
         networks of car dealerships. Technical report, Edinburgh Parallel Com-
         puting Centre, EPCC-PAR-GMAP, 1994.

[Gol89] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Ma-
         chine Learning.* Addison-Wesley, 1989.

[Gor03] T. G. W. Gordon. Exploring models of development for evolutionary cir-
         cuit design. In *Proc. of the 2003 Congress on Evolutionary Computation,
         CEC 2003*, pages 2050–2057. IEEE Press, 2003.

[Gor05] T. G. W. Gordon. Exploiting development to enhance the scalability
         of hardware evolution, PhD thesis. Department of Computer Science,
         University College London, 2005.

[Gru94] F. Gruau. Neural network synthesis using cellular encoding and the
         genetic algorithm, PhD thesis. Laboratoire de l'Informatique du Paral-
         lelisme, Ecole Normale Superieure de Lyon, France, 1994.

[HF04]   M. L. Harrison and J. A. Foster. Co-evolving faults to improve the fault tolerance of sorting networks. In *Proc. of the 7th European Conference on Genetic Programming, EuroGP 2004, on Evolvable Systems: From Biology to Hardware (ICES 2003), Lecture Notes in Computer Science, vol. 3003*, pages 57–66, Berlin, 2004. Springer-Verlag.

[Hil90]   W. D. Hillis. Co-evolving parasites improve simulated evolution as an optimization procedure. *Physica D*, 42(1–3):228–234, June 1990.

[Hol75]   J. H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, 1975.

[Hor03]   G. S. Hornby. Generative representations for evolutionary design automation, PhD thesis. Brandeis University, Department of Computer Science, 2003.

[HP01]   G. S. Hornby and J. B. Pollack. The advantages of generative grammatical encodings for physical design. In *Proc. of the 2001 Congress on Evolutionary Computation*, pages 600–607. IEEE Press, 2001.

[HTvR01]   P. C. 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.

[Hue98]   L. Huelsbergen. Finding general solutions to the parity problem by evolving machine-language representations. In *Proc. of the Third Annual Conference on Genetic Programming*, pages 158–166. Morgan Kaufmann Publishers, 1998.

[IFK00]   K. Imamura, J. A. Foster, and A. W. Krings. The test vector problem and limitations to evolving digital circuits. In *Proc. of The Second NASA/DoD Workshop on Evolvable Hardware*, pages 81–87, Washington, DC, USA, 2000. IEEE Computer Society.

[J. 92]   J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[J. 94]   J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, 1994.

[J. 99]   J. R. Koza and F.H. Bennett and D. Andre and M. A. Keane. *Genetic Programming III: Darwinian Invention and Problem Solving*. Morgan Kaufmann, 1999.

[J. 03]   J. R. Koza and M. A. Keane and M. J. Streeter and W. Mydlowec and J. Yu and G. Lanza. *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Kluwer Academic Publishers, 2003.

[Jui95]   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.

[Kau69]   S. A. Kauffman. Metabolic stability and epigenesis in randomly constructed genetic nets. *Journal of Theoretical Biology*, 22:437–467, 1969.

[Kau93]   S. A. Kauffman. *The Origins of Order*. Oxford University Press, 1993.

[Kit90]   H. Kitano. Designing neural networks using genetic algorithms with graph generation system. *Complex Systems*, 4(4):461–475, 1990.

[Kit96]   H. Kitano. Morphogenesis for evolvable systems. In *E. Sanchez, editor, Towards Evolvable Hardware: The Evolutionary Engineering Approach, Lecture Notes in Computer Science, volume 1062*, pages 99–117, Berlin Heidelberg New York, 1996. Springer-Verlag.

134   References

[Knu98]   D. E. Knuth. *The Art of Computer Programming: Sorting and Searching (2nd ed.)*. Addison Wesley, 1998.

[Koz90]   J. R. Koza. Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems. Technical report, Stanford University Computer Science Department, 1990.

[Kum04]   S. Kumar. Investigating computational models of development for the construction of shape and form, PhD thesis. Department of Computer Science, University College London, 2004.

[Lan84]   C. G. Langton. Self-reproduction in cellular automata. *Physica D: Nonlinear Phenomena*, 10(1–2):135–144, 1984.

[Lan06]   H. W. Lang. Algorithmen in Java, 2. auflage. Oldenbourg Wissenschaftsverlag, 2006.

[Lew99]   B. Lewin. *Genes IV*. Oxford University Press, 1999.

[Lin68]   A. Lindenmayer. Mathematical models for cellular interaction in development, parts I and II. *Journal of Theoretical Biology*, 18:280–315, 1968.

[MCFF00] J. Masner, J. Cavalieri, J. F. Frenzel, and J. Foster. Size versus robustness in evolved sorting networks: Is bigger better? In *Proc. of The Second NASA/DoD Workshop on Evolvable Hardware*, pages 81–87, Washington, DC, USA, 2000. IEEE Computer Society.

[Mil03]   J. F. Miller. Evolving developmental programs for adaptation, morphogenesis and self-repair. In *Advances in Artificial Life. 7th European Conference on Artificial Life, Lecture Notes in Artificial Intelligence, volume 2801*, pages 256–265, Dortmund DE, 2003. Springer.

[MJ00a]   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.

[MJ00b]   J. F. Miller and D. Job. Principles in the evolutionary design of digital circuits – part II. *Genetic Programming and Evolvable Machines*, 3(2):259–288, July 2000.

[MT00]   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, 2000. Springer.

[MYK+96] M. Murakawa, S. Yoshizawa, I. Kajitani, T. Furuya, M. Iwata, and T. Higuchi. Hardware evolution at function level. In *Proc. of the 4th International Conference on Parallel Problem Solving from Nature, PPSN 1996, Lecture Notes in Computer Science, volume 1141*, pages 206–217, London, UK, 1996. Springer-Verlag.

[Nor94]   P. Nordin. *A Compiling Genetic Programming System that Directly Manipulates the Machine-Code*. MIT Press, Cambridge, MA, 1994.

[Nor97]   P. Nordin. Evolutionary program induction of binary machine code and its applications, PhD thesis. University of Dortmund, Department of Computer Science, 1997.

[P. 99]   P. J. Bentley (ed.). *Evolutionary Design by Computers*. Morgan Kaufmann, San Francisco CA, 1999.

[PL90]   P. Prusinkiewicz and A. Lindenmeyer. *The algorithmic beauty of plants*. Springer-Verlag, 1990.

[RA97]   G. Rozenberg and A. Salomaa (eds.). *Handbook of Formal Languages, volume 3*. Springer-Verlag, 1997.

[Rec73]   I. Rechenberg. *Evolutionstrategie: Optimierung Technischer Systeme nach Prinzipien der Biologischen Evolution.* Frommann-Holzboog, Stuttgart, 1973.

[RFF07]   D. Roggen, D. Federici, and D. Floreano. Evolutionary morphogenesis for multi-cellular systems. *Genetic Programming and Evolvable Machines*, 8(1):61–96, 2007.

[RSP08]   R. Růžička, L. Sekanina, and R. Prokop. Physical demonstration of polymorphic self-checking circuits. In *Proc. of the 14th IEEE Int. On-Line Testing Symposium*, pages 31–36. IEEE Computer Society, 2008.

[S. 03]   S. Kumar and P. J. Bentley (eds.). *On Growth, Form and Computers.* Elsevier Academic Press, 2003.

[SB05]   L. Sekanina and M. Bidlo. Evolutionary design of arbitrarily large sorting networks using development. *Genetic Programming and Evolvable Machines*, 6(3):319–347, 2005.

[Sch65]   H.-P. Schwefel. Kybernetische evolution als strategie der experimentellen forschung in der stroemungstechnik, Master thesis. Technische Universitaet, Berlin, 1965.

[Sek04]   L. Sekanina. Evolving constructors for infinitely growing sorting networks and medians. *Lecture Notes in Computer Science*, 2004(2932):314–323, 2004.

[Sek05]   L. Sekanina. Evolutionary design of gate-level polymorphic digital circuits. *Lecture Notes in Computer Science*, 2005(3449):185–194, 2005.

[Sip97]   M. Sipper. *Evolution of Parallel Cellular Machines – The Cellular Programming Approach, Lecture Notes in Computer Science, volume 1194.* Springer-Verlag, Berlin, 1997.

[SKL06]   E. Stomeo, T. Kalganova, and C. Lambert. Generalized disjunction decomposition for evolvable hardware. *IEEE Transactions on Systems, Man and Cybernetics – Part B*, 36:1024–1043, 2006.

[SM03]   K. O. Stanley and R. Miikkulainen. A taxonomy for artificial embryogeny. *Artificial Life*, 9:93–130, 2003.

[Smi80]   S. F. Smith. A learning system based on genetic adaptive algorithms, PhD thesis. University of Pittsburgh, 1980.

[SZG$^+$04]   A. Stoica, R. S. Zebulum, X. Guo, D. Keymeulen, M. I. Ferguson, and V. Duong. Taking evolutionary circuit design from experimentation to implementation: some useful techniques and a silicon demonstration. In *Computers and Digital Techniques, IEE Proceedings - Volume 151, Issue 4*, pages 295–300, 2004.

[SZK01]   A. Stoica, R. S. Zebulum, and D. Keymeulen. Polymorphic electronics. In *Proc. of International Conference on Evolvable Systems: From Biology to Hardware, Lecture Notes in Computer Science, volume 2210*, pages 291–302. Springer-Verlag, 2001.

[SZKL02]   A. Stoica, R. S. Zebulum, D. Keymeulen, and J. Lohn. On polymorphic circuits and their design using evolutionary algorithms. In *Proc. of IASTED International Conference on Applied Informatics, AI2002*, Innsbruck AU, 2002.

[TH05]   G. Tufte and P. C. Haddow. Towards development on a silicon-based cellular computing machine. *Natural Computing*, 4(4):387–416, 2005.

[Tor03]   J. Torresen. Evolving multiplier crcuits by training set and training vector partitioning. In *Proc. of the 5th International Conference on Evolv-*

able Systems: From Biology to Hardware, ICES 2003, Lecture Notes in Computer Science, volume 2606, pages 228–237. Springer-Verlag, 2003.

[Tuf04]   G. Tufte. Development of digital circuits on a virtual sblock FPGA, PhD thesis. Department of Computer and Information Science, Norwegian University of Science and Technology, 2004.

[Tur52]   A. M. Turing. The chemical basis of morphogenesis. *Philosophical Transactions of the Royal Society of London. Series B: Biological Sciences*, 237(641):37–72, 1952.

[VJM00]   V. Vassilev, D. Job, and J. Miller. Towards the automatic design of more efficient digital circuits. In *Proc of the Second NASA/DoD Workshop on Evolvable Hardware*, pages 151–160, Palo Alto, CA, 2000. IEEE Computer Society.

[vK93]   H. von Koch. On a continuous curve without tangents constructible from elementary geometry. In *Classics on fractals (G. Edgar, ed.)*, pages 25–45. Addison-Wesley, Reading, 1993.

[VM00]   V. Vassilev and J. F. Miller. Scalability problems of digital circuit evolution. In *Proc. of the 2nd NASA/DoD Workshop of Evolvable Hardware*, pages 55–64, Los Alamitos, CA, US, 2000. IEEE Computer Society.

[vN66]   J. von Neumann. *The Theory of Self-Reproducing Automata.* A. W. Burks (ed.), University of Illinois Press, 1966.

[Wak01]   J. F. Wakerly. *Digital Design: Principles and Practice.* Prentice Hall, New Jersey, US, 2001.

[WK88]   L. D. Whitley and J. Kauth. Genitor: A different genetic algorithm. pages 118–130, 1988.

[WL96]   M. L. Wong and K. S. Leung. Learning recursive functions from noisy examples using generic genetic programming. In *Proc. of the First Annual Conference on Genetic Programming*, pages 238–246. The MIT Press, 1996.

[Wol07]   L. Wolpert. *The Principles of Development.* Oxford University Press, Oxford, UK, 2007.

[WS90]   L. D. Whitley and T. Starkweather. Genitor II: A distributed genetic algorithm. *Journal of Experimental ans Theoretic Artificial Intelligence*, 2(3):189–214, 1990.

[ZB08]   M. J. Zvelebil and J. O. Baum. *Understanding Bioinformatics.* Garland Science, Taylor & Francis Group, 2008.