

# **DEVELOPING AUTOMATIC SYNTHESIS METHODOLOGIES FOR QUANTUM CIRCUITS USING GENETIC ALGORITHMS**

Teză destinată obținerii  
titlului științific de doctor inginer  
la  
Universitatea "Politehnica" din Timișoara  
în domeniul Știința Calculatoarelor  
de către

**Ing. Cristian Ruican**

Conducător științific:  
Referenți științifici:

Profesor Mircea Vlăduțiu  
Profesor Mircea Petrescu,  
Profesor Dumitru Dumitrescu,  
Profesor Ioan Jurca

Ziua susținerii tezei: 11.06.2009

Seriile Teze de doctorat ale UPT sunt:

- |                        |   |
|------------------------|---|
| 1. Automatică          | 7. Inginerie Electronică și Telecomunicații |
| 2. Chimie              | 8. Inginerie Industrială                    |
| 3. Energetică          | 9. Inginerie Mecanică                       |
| 4. Ingineria Chimică   | 10. Știința Calculatoarelor                 |
| 5. Inginerie Civilă    | 11. Știința și Ingineria Materialelor       |
| 6. Inginerie Electrică |   |

Universitatea „Politehnica” din Timișoara a inițiat seriile de mai sus în scopul diseminării expertizei, cunoștințelor și rezultatelor cercetărilor întreprinse în cadrul școlii doctorale a universității. Seriile conțin, potrivit H.B.Ex.S Nr.14 / 14.07.2006, tezele de doctorat susținute în universitate începând cu 1 octombrie 2006.

Copyright © Editura Politehnica – Timișoara, 2009

Această publicație este supusă prevederilor legii dreptului de autor. Multiplicarea acestei publicații, în mod integral sau în parte, traducerea, tipărirea, reutilizarea ilustrațiilor, expunerea, radiodifuzarea, reproducerea pe microfilme sau în orice altă formă este permisă numai cu respectarea prevederilor Legii române a dreptului de autor în vigoare și permisiunea pentru utilizare obținută în scris din partea Universității „Politehnica” din Timișoara. Toate încălcările acestor drepturi vor fi penalizate potrivit Legii române a drepturilor de autor.

România, 300159 Timișoara, Bd. Republicii 9,  
tel. 0256 403823, fax. 0256 403221  
e-mail: editura@edipol.upt.ro

## **Cuvânt înainte**

Teza de doctorat a fost elaborată pe parcursul activității mele în cadrul Departamentului de Calculatoare al Universității „Politehnica” din Timișoara. Cercetarea a fost efectuată având sprijinul și suportul necondiționat al colegilor din grupul ACSA (Advanced Computing Systems & Architectures).

Mulțumiri deosebite, recunoștință, stimă și respect se cuvin conducătorului de doctorat Prof.Univ.Dr.Ing. Mircea Vlăduțiu care de-a lungul anilor de studiu mi-a dedicat timp prețios și mi-a deschis calea spre cercetare. Discuțiile de înaltă competență despre arhitectura calculatoarelor și despre sinteza circuitelor au constituit un real sprijin în conceperea acestei teze.

Timișoara, Noiembrie 2008

Cristian Ruican

To my daughter, Alexandra Cristiana – for the joy you are.

Ruican, Cristian

**Developing Automatic Synthesis Methodologies for Quantum Circuits using Genetic Algorithms**

Teze de doctorat ale UPT, Seria 10, Nr. 14, Editura Politehnica, 2009,  
198 pagini, 64 figuri, 14 tabele.

ISSN:1842-7707

ISBN:978-973-625-819-0

Cuvinte cheie: sinteză, algoritm genetic, circuit cuantic

Rezumat: titlul tezei descrie trei domenii care vor fi conectate într-o manieră ingenioasă și coerentă în același timp. Scopul este identificarea metodologiei adecvate care să realizeze sinteza circuitelor cuantice logice reversibile.

## Published papers and Impact

This thesis is supported by the following published papers:

- **C. Ruican**, M. Udrescu, L. Prodan, M. Vladutiu, "Automatic Synthesis for Quantum Circuits using Genetic Algorithms", Proceedings ICANNGA'07 "International Conference on Adaptive and Natural Computing Algorithms", LNCS 4431 (Springer-Verlag Berlin Heidelberg), Warsaw, Poland, April 2007, pp. 174–183, ISBN 978-3-540-71589-4 (ISI rank).
- **C. Ruican**, M. Udrescu, L. Prodan, M. Vladutiu, "A Genetic Algorithm Framework Applied to Quantum Circuit Synthesis", Proceedings NISCO'07 "Nature Inspired Cooperative Strategies for Optimization", Series: Studies in Computational Intelligence, Vol. 129, (Springer-Verlag Berlin Heidelberg), Acireale, Italy, November 2007, pp. 419-429, ISBN 978-3-540-78986-4 (ISI rank).
- **C. Ruican**, M. Udrescu, L. Prodan, M. Vladutiu, "Software Architecture for Quantum Circuit Synthesis", Proceedings ICAISC'08 "International Conference on Artificial Intelligence and Soft Computing", Computational Intelligence: Methods and Applications, (IEEE Computational Intelligence Society-Poland Chapter), Zakopane, Poland, June 2008, pp. 562-573, ISBN 978-83-60434-50-5.
- **C. Ruican**, M. Udrescu, L. Prodan, M. Vladutiu, "Quantum Circuit Synthesis with Adaptive Parameters Control", Proceedings EuroGP'09 "European Conference on Genetic Programming", LNCS 5481 (Springer-Verlag Berlin Heidelberg), Tübingen, Germany, April 2009, pp. 339-350, ISBN 978-3-642-01180-1 (ISI rank).
- **C. Ruican**, M. Udrescu, L. Prodan, M. Vladutiu, "Genetic Algorithm Based Quantum Circuit Synthesis with Adaptive Parameters", CEC'09 "IEEE Congress on Evolutionary Computation", Trondheim, Norway, May 2009, pp. 896-903, ISBN 978-1-4244-2959-2 (IEEE rank).
  
- **C. Ruican**, "Prerequisites of Synthesis Methodology for Quantum Circuits", Ph.D. Report 1, Politehnica University of Timisoara, July 2007, pp. 1-87.
- **C. Ruican**, "Genetic Algorithm Framework for Application Tuning", Ph.D. Report 1, Politehnica University of Timisoara, March 2008, pp. 1-66.

# Contents

1.	Introduction .....	15
1.1	Motivation .....	15
1.2	Aims of the Thesis .....	17
1.3	Thesis Outline .....	19
2.	Background.....	20
2.1	Quantum Computation .....	20
2.2	Genetic Algorithms .....	26
2.3	Related Work .....	31
3.	Analysis of the QCS.....	37
3.1	QCS Problem Statement .....	38
3.2	Characteristics of the Tool Support.....	39
3.3	Levels of Design .....	40
3.4	QCS Architecture Overview .....	41
3.4.1	Rationale .....	44
3.4.2	Constraints.....	44
3.4.3	Logical View .....	45
3.4.4	Process View .....	46
3.4.5	Development View .....	47
3.5	Architecture Properties .....	48
3.6	Classical vs. Quantum Digital Circuit Synthesis .....	50
4.	Genetic Algorithm Framework.....	53
4.1	Framework Preliminaries .....	54
4.2	Framework Packages Architecture.....	54
4.2.1	Genetic Algorithm Types .....	55
4.2.2	Genome Implementation .....	57
4.2.3	Population Implementation .....	58
4.2.4	GA Operators.....	59
4.2.1	Framework Utilities.....	61
4.2.2	Framework Statistics .....	63
4.3	Framework Validation.....	65

5.	Genetic Quantum Circuits Synthesis .....	69
5.1	Parser Module .....	70
5.2	Quantum Circuits Database.....	82
5.3	Preparation Steps for Genetic Algorithm.....	86
5.4	Integration within ProGA Framework .....	90
5.4.1	Initial Circuit Configuration .....	91
5.4.2	Synthesis Genetic Algorithm .....	93
5.4.3	Circuit Genome .....	95
5.4.4	Circuit Output Function .....	96
5.4.5	Genome Initialization .....	97
5.4.6	Detecting of the Next Gate .....	98
5.4.7	Performing Mutation .....	99
5.4.8	Performing Crossover .....	102
5.4.9	Fitness Formula Computation .....	105
5.5	Metaheuristic Algorithm.....	106
5.5.1	Parameter Control .....	106
5.5.2	Integration within ProGA Framework.....	107
5.5.3	Adaptive Behavior using Operator Performance .....	108
5.5.4	How the Change is Made .....	112
5.5.5	Performance Meaning .....	112
6.	Experiments Result Evaluation .....	114
6.1	The Experiment Setup.....	114
6.2	Evaluation Approach .....	116
6.3	QCS Tool Verification .....	117
6.3.1	Complete Set of Gates .....	117
6.3.2	Reduced Set of Gates .....	117
6.3.3	Minimal Set of Gates.....	118
6.4	Case Studies.....	119
6.4.1	Three-Qubit Circuit.....	120
6.4.2	Four-Qubit Circuit.....	125
6.4.1	Five-Qubit Circuit .....	130
6.4.3	Six-Qubit Circuit .....	135

6.4.5	Seven-Qubit Circuit .....	141
6.5	Additional Experiments .....	148
6.6	Result Evaluation .....	166
6.7	Going Beyond 7-qubit Circuits .....	166
7.	Conclusions and Perspectives .....	170
7.1	Thesis Impact and Contributions .....	170
7.2	Future Directions .....	172
7.2.1	Refinement and Future Work .....	172
7.2.2	QCS Integration .....	173
8.	Appendix .....	174
8.1	Object Oriented Metrics .....	174
8.1.1	Framework Program Unit Complexity .....	174
8.1.2	Framework Class OO Metrics .....	175
8.2	QCS Initial Genome Solution .....	176
8.3	Statistic Details .....	177
8.4	QCS Genome Implementation Details .....	179
8.5	Quantum Gates Cost .....	183
8.5.1	Cost Details using General Approach .....	183
8.5.2	Cost Details using Function Output .....	183
8.6	GNUplot Script .....	184
9.	Bibliography .....	186



## List of Figures

Figure 1.1: Moore’s Law Representation .....	16
Figure 1.2: Taxonomy for Emerging Research Information [16] .....	18
Figure 2.1: Quantum Computation Details[18] .....	21
Figure 2.2: Quantum Logic Circuit Adder .....	25
Figure 2.3: Abstract Quantum Logic Circuit.....	26
Figure 2.4: Genetic Programming Paradigm [19] .....	30
Figure 2.5: Cosine-Sine Decompositions [47].....	34
Figure 2.6: The Quantum Shannon Decomposition [47].....	34
Figure 2.7: Circuit Equivalence as Proposed by Shende et al. [47].....	35
Figure 3.1: The Design Levels in a Program [62] .....	41
Figure 3.2: Software Behavior Deployment .....	43
Figure 3.3: QSynTool Architecture Layout.....	44
Figure 3.4: System Provided Services .....	45
Figure 3.5: QSynTool Process View .....	46
Figure 3.6: QSynTool Development View .....	47
Figure 4.1: ProGA Framework Architecture .....	54
Figure 4.2: Genetic Algorithm Class Diagram .....	56
Figure 4.3: Genome Class Diagram.....	57
Figure 4.4: Population Class Diagram.....	58
Figure 4.5: Selector Class Diagram.....	60
Figure 4.6: Random Number Generator Architecture .....	62
Figure 4.7: Architecture for Execution Time Measurement.....	63
Figure 4.8: Statistic Class Diagram .....	64
Figure 4.9: Knapsack Class Diagram .....	66
Figure 4.10: Methinks Class Diagram .....	68
Figure 5.1: QHDL Parser Details.....	72
Figure 5.2: Description of the QCS Parser.....	72
Figure 5.3: QHDL Parser Object Model Overview.....	74
Figure 5.4: Generic Parser Sequence Diagram.....	75
Figure 5.5: Quantum Circuit View as List-in-a-List Structure.....	76
Figure 5.6: Internal Data Structure Object Model Overview .....	77
Figure 5.7: QHDL Parser Object Model Overview.....	78
Figure 5.8: Nod Object Model Overview.....	79
Figure 5.9: Topological Optimization .....	81
Figure 5.10: Iterator Pattern Object Model Overview .....	82
Figure 5.11: Database for Quantum Circuits .....	83
Figure 5.12: Database Component Diagram.....	83
Figure 5.13: Quantum Gate Class Diagram .....	84
Figure 5.14: Collection Class for Quantum Gates .....	85
Figure 5.15: Creating Collection Sequence Diagram .....	86
Figure 5.16: Chromosome Encoding .....	89
Figure 5.17: Integration within ProGA Framework.....	91

Figure 5.18: Genome Configuration .....	92
Figure 5.19: Synthesis Genetic Algorithm .....	94
Figure 5.20: Synthesis Genome .....	96
Figure 5.21: Genome Initialization.....	97
Figure 5.22: Mutation Representation .....	100
Figure 5.23: Crossover on Complete Gene(s) .....	102
Figure 5.24: Crossover on Incomplete Gene(s).....	103
Figure 5.25: History List for Solutions .....	106
Figure 5.26: Adaptive Control Integration .....	107
Figure 5.27: Statistic Data .....	109
Figure 5.28: Operator Performance .....	109
Figure 5.29: Adaptive Design.....	110
Figure 5.30: PerformanceData Class Overview .....	111
Figure 6.1 Synthesis of EPR Circuits [46] .....	167
Figure 6.2 Synthesis of Composite Circuits [11].....	168
Figure 8.1: Creation of the Initial Genome .....	177
Figure 8.2: Statistic Methods .....	178
Figure 8.3: Output Function Computation .....	179
Figure 8.4: TypeA/TypeB Mutation .....	180
Figure 8.5: Locus Initialization .....	181
Figure 8.6: Type A Crossover.....	182
Figure 8.7: GNUplot Script .....	185

## List of Tables

Table 1: Single Qubit Gates .....	24
Table 2: Multiple Qubit Gates.....	24
Table 3: Available C/C++/Java Frameworks.....	31
Table 4: Analysis of GA Frameworks .....	32
Table 5. Classic vs. Quantum Circuit Design.....	51
Table 6. Variable Type Classification .....	114
Table 7: Initial and Evolved Circuit – Complete Gates Set .....	117
Table 8: Initial and Evolved Circuits – Reduced Gates Set .....	118
Table 9: Initial and Evolved Circuit – Minimal Gates Set.....	118
Table 10. Test the Convergence .....	168
Table 11. Program Unit Complexity Metric .....	174
Table 12. Class OO Metrics .....	175
Table 13. Gate Costs and Feasibility .....	183
Table 14. Toffoli Gate Costs [51] .....	184

## Abstract

This dissertation addresses an important and hard computational problem, the efficient synthesis of reversible quantum circuits from high-level description language. The novelty of dissertation is the Genetic Algorithm [GA] being proposed for the quantum logic circuit synthesis problem, together with the meta-heuristic algorithm used for the parameters control. A new methodology, with layered synthesis architecture, is proposed by starting with a description of quantum circuits, and then progressing through optimization and synthesis phases.

The goal is to understand the application of evolutionary computing approach to Quantum Circuit Synthesis [QCS] and to provide a software tool that will allow automatic synthesis. The tool allows for defined configuration and helps in deciding about the methods and components that are better suited for the proposed synthesis aim. The problem is motivated by the following observations:

- The necessity to provide more computational power for the actual computers will move the digital circuit research work at the atomic scale, where the quantum laws are governing. The demand to shrinkage is also known as Moore's Law.
- Synthesis tools are necessary, because the technological development is extremely fast and the researcher's needs to develop new circuit functions are high.
- Quantum Computation, as well as its implementation (namely, the quantum circuits) has received a boost in its importance, due to the new discovered algorithms that prove the superiority of this field in solving some specific classes of computational problems.
- There are only few physical quantum circuits available that can manipulate small quantum particles, but in the near future it is possible to have a stunning variety of available circuits. Thus, simulation plays an important role now, and synthesis will eventually play the same role in the near future.

This dissertation introduces a computer aided software tool for automatic quantum circuit synthesis that is designed to be configurable, flexible, fast and easy to use by a wide range of researchers. The design is written using Unified Modeling Language and the software implementation is made in C++ language. The open source code allows for further development and proves our availability for discussions concerning quantum circuit synthesis by evolutionary computing methods. The aim was to develop an open platform that creates all the prerequisites for the quantum synthesis task.

Experiments are performed on several benchmark circuits, thus proving the synthesis algorithm efficiency. The evolutionary approach allows for testing different configurations and, together with the meta-heuristic algorithm, provides a powerful tool even for novice users. The experiments are repeated several times to avoid lucky guesses, the graphics and figures are explained and – at the same time – relevant conclusions are extracted.

## Acknowledgements

The work presented in this dissertation would not have been possible without guidance and support offered by many people. I would like to express my gratitude to Professor Mircea Vlăduțiu for supervising me through both MSc and PhD theses. His insight into circuit design and ability to keep me focused on the big picture, together with his steadiness and patient support were essential for me.

I owe thanks to Mihai Udrescu-Milosav who introduced me into the quantum domain when I started the MSc and up to the present moment has shared with me the ups and downs. His extensive knowledge, essential guidance and permanent encouragement shared during our meetings allowed to follow my vision. His genuine friendship is important for me.

I appreciate Lucian Prodan's helping effort for encouraging me every time I was struggling, and his involvement in formatting our articles and presentations.

I would like to thank my colleagues for making this dissertation a reality. During many days, I discussed with them my software abstract ideas, and in the end, I was able to implement new visions. My colleagues Florin Popa, Florin Ilioiu, Dan Onica, Mihai Schitcu and Victor Tomescu have provided essential pieces of advice during refreshing discussions (especially during breaks). I would like to thank you all for taking the time to read this dissertation, and I hope that you have enjoyed some of your selected topics.

Finally, I am most thankful for the support of my family, who encouraged me to continue the academic research over the years. I would like to thank my wife Violeta-Genoveva and my daughter Alexandra-Cristiana for their patience during the writing of this dissertation.

Many thanks to all,

# Chapter 1

## Introduction

"...it seems that the laws of physics present no barrier to reducing the size of computers until bits are the size of atoms, and quantum behavior holds sway"  
Richard P. Feynman [1]

In the 1970's and 1980's several physicists and computer scientists such as Charles H. Bennett<sup>1</sup>, Paul A. Benioff<sup>2</sup>, David Deutsch<sup>3</sup> and Richard P. Feynman<sup>4</sup> started to analyze the possibility of using quantum mechanics proprieties in computational devices. Considering Moore's Law [2], they understood that in a limited time slot the shrinking technology would reach the atom scale, where the behavior and properties of the circuit are governed by the quantum mechanics laws.

### 1.1 Motivation

In 1982 Feynman proposed a universal quantum simulator, by presenting how a quantum computer may be used to perform computation, and then pointed out, at the same time, the difficulties in simulating quantum mechanical systems on conventional computation devices (exponential memory and time overheads) [3]. He considered that the proposed simulator would allow a physicist to make quantum experiments within of a quantum computer. Few years later, in 1985, Deutsch derived a new version of Church-Turing thesis by considering that any physical process may be perfectly modeled by a quantum computer [4]. After Deutsch's first step, many researchers started to search applications for the quantum computer (i.e. Peter Shor's demonstration for finding the prime factors of a large integer, Lov Grover's database search algorithm, etc) [5] [6].

According to the Moore Law, the quantum level would be reached in 2010-2020 (see Figure 1.1). This is because our technology will continue to shrink up to the atom level in order to advance the computing technology (we discuss about the integration scale and the number of doping impurities within the bases of bipolar transistors that are required for logic). In our days, small integration scale is hard to be obtained, due to the amount of the heat that has to be dissipated. The quantum devices allow reversibility on computation, and therefore it is not dissipating heat. This is possible because the input and the output, for any quantum device, can be obtained by starting from the opposite side (it is called "logical reversibility"). If the circuit may also be run backwardly, then it is called "physically reversible", hence

---

<sup>1</sup> IBM Research Center, <http://www.research.ibm.com/people/b/bennetc/>

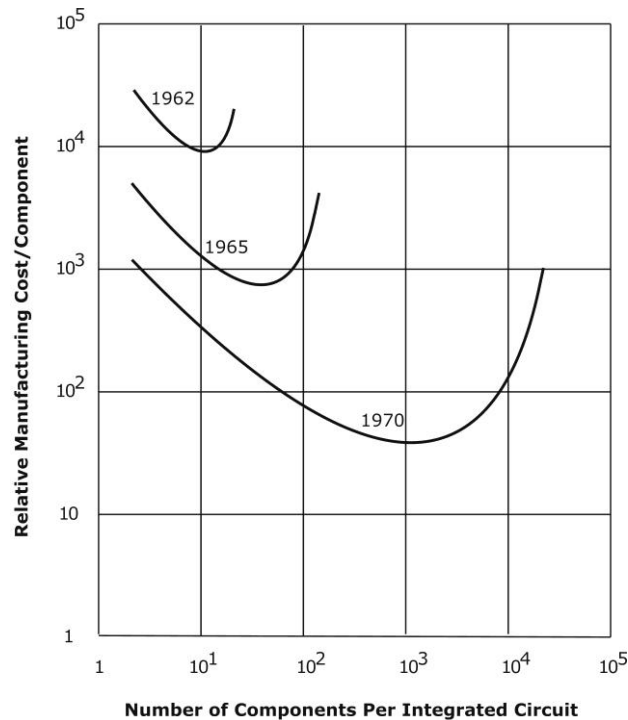
<sup>2</sup> Argonne National Laboratory, <http://www.phy.anl.gov/theory/staff/pab.html>

<sup>3</sup> University of Oxford, <http://www.qubit.org/people/david/>

<sup>4</sup> California Institute of Technology, <http://www.zyvex.com/nanotech/feynmanWeb.html>

## 16 | 1-Introduction

the second law of thermodynamics assures that it dissipates no heat. In a quantum computer, we always may reverse the computation for the final state, because computation is the unitary evolution of an input, over the given system state.



**Figure 1.1: Moore's Law Representation**

Another advantage of quantum computers is its increased computation power. The published quantum algorithms can solve exponential problems from classical computation more efficiently, by reducing their complexity and having a better response during the time spent.

For our research, as specified above, some domains we believe are strongly connected. The merger between quantum computing and genetic algorithms was already made; the field of Evolvable Quantum Information (EQI) has significantly grown over the last years [7]. At first glance, the merging of quantum computation and evolvable computation seems natural and benefic. Indeed, relevant progress has been signaled in the EQI subfield of Quantum-Inspired Genetic Algorithms (QIGA) including the so-called evolvable quantum hardware or the automatic synthesis of quantum circuits by evolvable means. Ongoing developments concerning the other EQI subfield of Quantum Genetic Algorithms (QGA) have been presented previously (as published [7] [8] [9] [10] [11] [12]).

Another motivation is to continue the previous research work from the ACSA [13] group, and to create a solid base for quantum technologies within our



university. Comparing with quantum simulation<sup>5</sup>, where there are many software tools available, the quantum synthesis is still in an incipient phase. One of the reasons for this low development is the unavailability of quantum physical devices and missing information about their characteristics. By taking into consideration the available literature, we consider that quantum synthesis will play an important role in the near future, as well as the fact that the development of quantum applications (with links in other domains) will be of great relevance for the success of quantum computation domain. In this newly created context, the existing software architecture will be adjusted to the quantum technology or created from scratch, because the classical solutions cannot be successfully applied.

### 1.2 Aims of the Thesis

The title for our research "Developing Automatic Synthesis Methodologies for Quantum Circuits using Genetic Algorithms" describes all three domains that we are trying to connect in an apprehensive and coherent manner. Our target is to find an adequate methodology that can perform quantum logic circuit synthesis.

There are many quantum simulators available at this moment, but in the quantum logic circuit synthesis field, there are only few theoretical papers available. That is why we are trying to bring our contribution in this very specific aspect of quantum circuit design. The synthesis relevance has two views. According to the first one, if the progress in technology is extremely fast, and it is outstripping the designer's abilities to make use of the created opportunities; the second view is generated by the situation where the technology is not available on a large scale, and the designers can use only a small set of gates for the design. These are the reasons why the development and the application of new and more suitable design methodologies are of the highest importance for the modern computer system industry.

Quantum synthesis has a bigger relevance when it is related to the simulation results. Any of the result simulation may have a physical implementation with the help of the synthesis algorithm (we can use a quantum circuit database in order to support the available circuit types). Therefore, starting from a program written in a high description language we obtain the physical device with the help of the automated synthesis process.

The main objective is to create a tool chain for the quantum circuit synthesis. We are motivated by the desire of bringing together genetic algorithms and quantum computing. This association is created with the aim of performing closer-to-optimum synthesis. On the other hand, the goal of this dissertation is also related to the Advanced Computing Systems and Architectures (ACSA) Laboratory [13], which aims at fostering the new computing technologies.

Another objective is to continue the work of my colleague Mihai Udrescu, who was also an advisor for this particular project within ACSA, and to create a solid base for quantum technologies in our university. We intend to push forward the

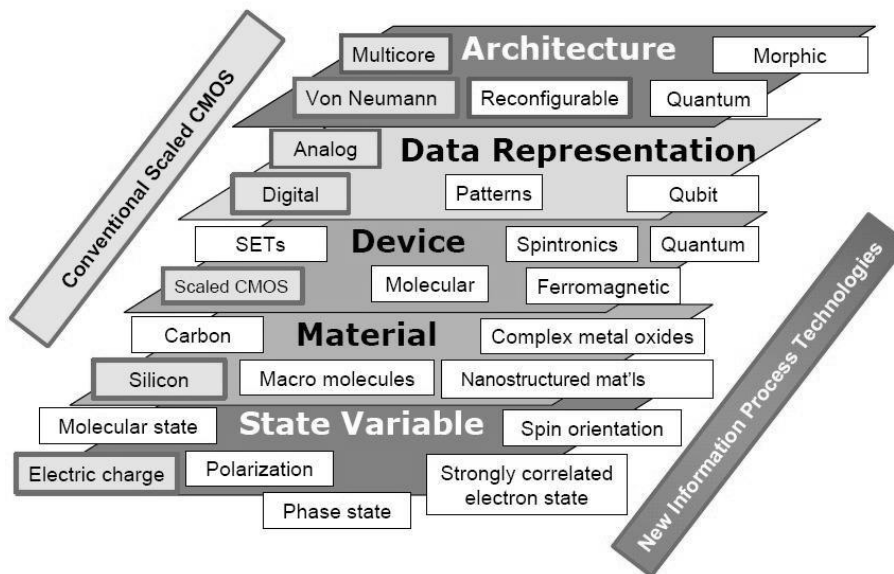
---

<sup>5</sup> [http://www.quantiki.org/wiki/index.php/List\\_of\\_QC\\_simulators](http://www.quantiki.org/wiki/index.php/List_of_QC_simulators)

## 18 | 1-Introduction

research concerning the quantum circuit synthesis, by also involving scientists from the genetic algorithms field, approach that could be of great importance for computer science in general.

The motivation presented in this section may seem theoretical and detached from the actual industry problems. However, the fact is that the industry is seriously taking into consideration the aspects related to the emerging technologies and quantum circuits in particular. The industry representatives have quickly reacted to these emerging problems, and founded a global organization called ITRS (International Technology Roadmap for Semiconductors), which is jointly sponsored by European Semiconductor Industry Association, Japan Electronics and Informational Technology Industries Association, Korea Semiconductor Industry Association, Taiwan Semiconductor Industry Association, and Semiconductor Industry Association from USA. As this organization defines its documents, they are about a continuous evaluation of the semiconductor technology requirements, aimed at increasing the performance of the integrated circuits. This effort is supported by industry, suppliers, academia, research groups, and governments [14][15][16]. The results of the ITRS assessments are published as ITRS reports, which are annually updated [17][16] (see Figure 1.2).



**Figure 1.2: Taxonomy for Emerging Research Information [16]**

Computer Aided Design (CAD) has become over the years an important domain for the development of conventional circuits. To a large scale of developers, the new synthesis techniques have allowed to create new high performance circuits, for a wide range of applications. Two topics are of paramount in the CAD field:

synthesis and optimization. The synthesis is started from logical or architectural models, and the optimization is performed on the intermediate results, that are obtained after applying CAD techniques on the input model. Thus, a designer should elaborate only the circuit specification, and then the CAD application will create the possible design, suitable for implementation on the actual hardware technologies.

### **1.3 Thesis Outline**

In Chapter 2, the background is split into three parts. The first one, that intends to describe the background of quantum computation, the second one in which the genetic algorithm background is introduced, and the third one section, where actual research results are presented. Chapter 3 presents the software analysis issues like requirements and architecture views (use case diagrams and activity diagrams are employed as a common language).

Chapter 4 defines a new genetic algorithm framework that is used to implement different genetic algorithms. Its UML architecture is explained and, at the same time, the source code made available. The framework provides the statistical information that will later be used to compare the algorithm results, and to adapt the implementation in order to obtain a better-evolved solution. The framework utilities, as well as the random number generator and time measurement are useful for the algorithm assessment. Chapter 5 is dedicated to quantum logic circuit synthesis. A new genetic algorithm is presented, the genetic operators are detailed and a new methodology is used for circuit synthesis. The synthesis algorithm was built by using the support provided by our framework. We identify the principles necessary for quantum circuit synthesis and optimization.

Chapter 6 elaborates the evaluation methods used for the quantum logic circuit synthesis assessment and our most general experimental results. The test methods that are used together with the test setup are focused on several test cases, proving that the software requirements are fulfilled by the software implementation. Chapter 7 is dedicated to the dissertation conclusions. The dissertation contributions and some directions for future work are then presented in this chapter.

## Chapter 2

### Background

"...many interesting problems are impossible to solve on a classical computer, not because they are in principal insoluble, but because of the astronomical resources required to solve realistic cases of the problem. The spectacular promise of quantum computers is to enable new algorithms which render feasible problems requiring exorbitant resources..." Nielsen, Michael A. and Chuang, Isaac L. [18]

"...how can computers learn to solve problems without being explicitly programmed? In other words, how can computers be made to do what is needed to be done, without being told exactly how to do it?" Koza, John R. [19]

Without necessarily assuming prior knowledge about quantum computing and genetic algorithms, this section will mainly outline the fundamental concepts. Two relevant references can be pointed out as major milestones in presenting this background: the first one is about quantum computation [18] and the second one is presents the genetic algorithms field [20]. These two textbooks correspond to the two major domains that are under our attention herein. In the last subchapter, the related work is presented, by pointing out the research advances made in the quantum logic circuit synthesis field.

#### 2.1 Quantum Computation

In a broad sense, computation is the processing of the mathematically represented information. In our case, and from a physical standpoint, quantum computation is the computation made with coherent atomic scale dynamics [21]. A quantum computer is a physical device able to perform computation using quantum mechanical phenomenon, like entanglement and data superposition. For a quantum computer, only the quantum mechanics laws are relevant [22][23][24][25][26]. Four quantum mathematical fundamental postulate, which lay the ground rules were verified through different experiments and described in several quantum computation textbooks [18][27][28].

##### **Postulate 1 – state vectors and state space**

A closed quantum system is described by a unit vector in a complex inner product space known as a state space [18].

The quantum states, encoded by qubits (i.e. quantum bits that can physically be implemented by employing photons, electron spin, nuclear spin, etc),

## 2.1 -Quantum Computation|21

are represented as vectors. A simple quantum system may be represented by using the spin 1/2 particle. For example, a spin-down  $|\downarrow\rangle$  and a spin-up  $|\uparrow\rangle$  may be used to represent the binary information  $|0\rangle$  and  $|1\rangle$  (see Figure 2.1 a). In Bra-Ket notation, a qubit is a normalized vector in a two dimensional Hilbert space  $|\psi\rangle = \alpha |0\rangle + \beta |1\rangle \equiv \begin{bmatrix} \alpha \\ \beta \end{bmatrix}$ ,  $|\alpha|^2 + |\beta|^2 = 1$  ( $\alpha, \beta \in C$ ), where  $|0\rangle$  and  $|1\rangle$  are the basis states. The quantum system is represented by a superposition of basis states, while a classical binary system can only settle in one of the basis states "0" or "1". Another advantage of quantum computers is the computational power (see Figure 2.1 b).

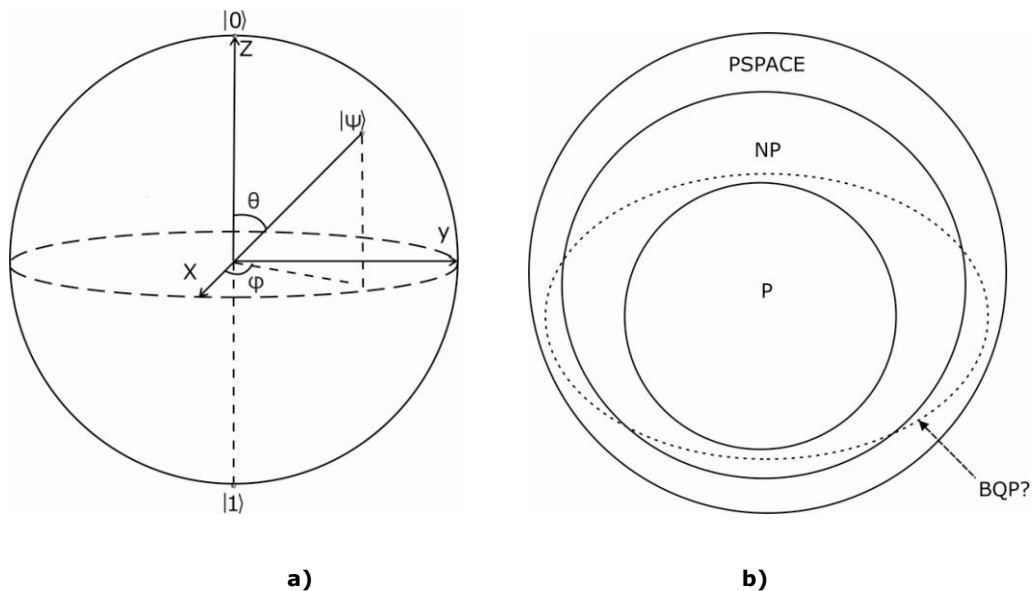


Figure 2.1: Quantum Computation Details[18]

### Inner product<sup>6</sup>

The inner product of two complex vectors  $x = [x_1, x_2, \dots, x_n]$  and  $Y = [y_1, y_2, \dots, y_n]$  is defined as:

$$X \cdot Y = \sum_{i=0}^n x_i^* y_i = \begin{bmatrix} x_1^* & x_2^* & \dots & x_n^* \end{bmatrix} \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} \quad (2.1)$$

<sup>6</sup> known as dot product or scalar product

## 22 | 2-Background

---

### Postulate 2 – unitary evolution

The evolution of a closed quantum system is described by a unitary transformation  $|\psi'\rangle = U|\psi\rangle$  [18].

### Transpose

Given a matrix  $A$ , the corresponding transpose matrix written  $A^T$  is the matrix obtained by exchanging  $A$ 's rows and columns, thus satisfying the identity  $(A^T)^{-1} = (A^{-1})^T$ .

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}^T = \begin{bmatrix} a & d & g \\ b & e & h \\ c & f & i \end{bmatrix} \quad (2.2)$$

### Unitary

A square matrix  $U$  is a unitary matrix if  $U^H = U^{-1}$ , where  $U^H$  denotes the conjugate transpose and  $U^{-1}$  is the matrix inverse.

### Postulate 3 - measurement

The measurement of a quantum state  $|\psi\rangle$  involves a special set of operators. The result will be one of the eigenvalue  $|\omega\rangle$  of the operator  $\Omega$ , with a certain probability when the operator is applied to  $|\psi\rangle$ . Measurement is destructive and will change the measurement state  $|\psi\rangle$  to  $|\omega\rangle$  [18].

Measurement will always find the system in one discrete basis state determined by its outcome; at the same time this is the only way to extract the information from a quantum system. Unfortunately, the measurement destroys the quantum superposition state. The measurement result is one of the operator's eigenvalue (one of the supposed basis states), with a probability given by the squared amplitude of the measured basis state. For example, for the quantum state given as  $0.1|00\rangle + 0.2|01\rangle + 0.3|10\rangle + 0.4|11\rangle$ , the probability to read the second qubit as  $|0\rangle$  is  $0.1^2 + 0.3^2$ . After measurement, the quantum computation is not reversible any longer.

Having the state  $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ , the orthonormal basis  $|+\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}}$

and  $|-\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}}$  the  $\text{Pr}(+) = \left| \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} \alpha \\ \beta \end{bmatrix} \right|^2 = \left| \frac{\alpha + \beta}{\sqrt{2}} \right|^2 = \frac{|\alpha + \beta|^2}{2}$  and the

$\text{Pr}(-) = \frac{|\alpha - \beta|^2}{2}$  when  $\text{Pr}(\pm)$  is the so-called "projector operator".

For example, if we are to measure the state  $|\psi\rangle = \begin{bmatrix} a \\ \beta \end{bmatrix}$  in the basis  $|1\rangle$ , the projective operator is  $|1\rangle\langle 1| = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ , therefore, it can be written that the  $Pr(1) = \langle \psi | 1 \rangle \langle 1 | \psi \rangle = \begin{bmatrix} a^* & \beta^* \end{bmatrix} \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ \beta \end{bmatrix} = \begin{bmatrix} 0 & \beta^* \end{bmatrix} \begin{bmatrix} a \\ \beta \end{bmatrix} = |\beta|^2$ . This example shows that measuring in the computational basis, the probability for getting a 0 or a 1 is the squared magnitude of the associated amplitude.

**Postulate 4 – tensor product**

The state space of a composite physical system is the tensor product of the state spaces of the component systems [18].

For example, two-qubit state space is  $C^2 \otimes C^2 = C^4$  and the computational basis states are  $|0\rangle \otimes |0\rangle = |00\rangle, |0\rangle \otimes |1\rangle = |01\rangle, |1\rangle \otimes |0\rangle = |10\rangle, |1\rangle \otimes |1\rangle = |11\rangle$ .

**Tensor product<sup>7</sup>**

Given a  $[m \times n]$ -size matrix  $A$  and a  $[p \times q]$ -size matrix  $B$ , their direct product  $C = A \otimes B$  is a  $[(mp) \times (nq)]$  matrix with elements defined by  $c_{\alpha\beta} = a_{ij}b_{kl}$ , where  $\alpha \equiv p(i-1) + k$  and  $\beta \equiv q(j-1) + l$ . In order to illustrate this, we compute the tensor product of the following complex-valued matrices,

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}; \quad B = \begin{bmatrix} e & f \\ g & h \end{bmatrix} \tag{2.3}$$

the tensor product operation  $\otimes$  will give

$$A \otimes B = \begin{bmatrix} ae & af & be & bf \\ ag & ah & bg & bh \\ ce & cf & de & df \\ cg & ch & dg & dh \end{bmatrix} \tag{2.4}$$

**Reversibility**

We say that a logic gate is reversible if the number of input wires is equal to the number of output wires and the gate simply permutes the set of input values. We say that a logic circuit is reversible if it contains only reversible gates, and it has no fan-out.

---








<sup>7</sup> known also as Kronecker product

## 24 | 2-Background

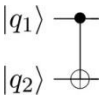
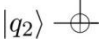
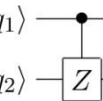

### Quantum Gates

In Table 1 and Table 2, several quantum gates are presented together with the associated symbol and the matrix representation (more details in references such as [29] [30] [31] [23]).

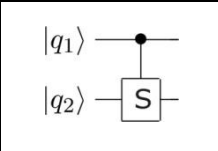
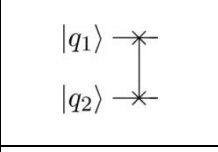
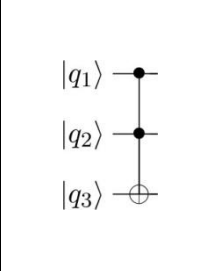
**Table 1: Single Qubit Gates**

Symbol	Name	SW-Notation	Matrix
$ q_1\rangle$ —  —	Hadamard	hadamard	$\frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$
$ q_1\rangle$ —  —	Pauli X	opX	$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$
$ q_1\rangle$ —  —	Pauli Y	opY	$\begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$
$ q_1\rangle$ —  —	Pauli Z	opZ	$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$
$ q_1\rangle$ —  —	Phase	opS	$\begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}$
$ q_1\rangle$ —  —	$\pi / 8$	opT	$\begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix}$
$ q_1\rangle$ —  —	Identity	i	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$

**Table 2: Multiple Qubit Gates**

Symbol	Name	SW-Notation	Matrix
$ q_1\rangle$ —  — $ q_2\rangle$ — 	Controlled-Not	cnot	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$
$ q_1\rangle$ —  — $ q_2\rangle$ — 	Controlled-Z	cZ	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$



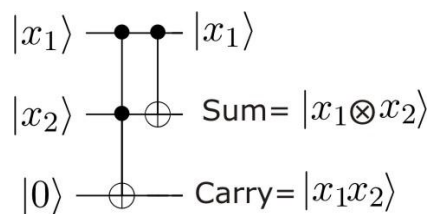
	Controlled-phase	cS	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & i \end{bmatrix}$
	Swap	swap	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
	Toffoli	toffoli	$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$

**Quantum Circuits**

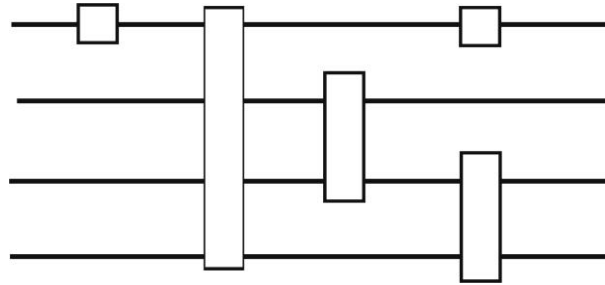
Any quantum phenomenon having a finite number of states can be modeled as a quantum circuit. The quantum gate is a physical device implementing a unitary operator that represents the quantum state transformation. Due to the unitary property, all quantum circuits are reversible; furthermore, these circuits are considered the most feasible implementation solution for quantum algorithms. A good description is also provided within reference [32].

A quantum circuit is composed of one or more quantum gates placed in a defined order. The order is important for quantum computation, which is made by using two quantum registers, an important output for the quantum logic circuit representation. The horizontal lines define the number of qubits, while the vertical lines denote a coupling between various qubits. In other words, a quantum circuit is defined as one or more quantum operations that are applied on a quantum register.

As quantum circuit example, the quantum two-qubits adder is presented in the Figure 2.2. Going to a more abstract level, the quantum circuit may be seen as a sequence of quantum gates placed on the qubit lines as shown in Figure 2.3.



**Figure 2.2: Quantum Logic Circuit Adder**



**Figure 2.3: Abstract Quantum Logic Circuit**

The quantum logic circuit exhibits many similarities with their classical counterparts. The quantum gates are connected through quantum wires that carry qubits encoded quantum information. The most difficult obstacles in the quantum circuit implementation are related to the decoherence of quantum superposition due to the quantum state interaction with the environment, and with the quantum state transformation precision due to subsequent computation steps.

## 2.2 Genetic Algorithms

The Genetic Algorithms are adaptive heuristic search algorithms based on evolutionary ideas of natural selection used to find solutions for optimization and search problems. Genetic Algorithms have been subject to intensive research in the last decade, as many reported results are providing successful solutions to their respective problems. The GA development covers a wide variety of designs and engineering applications. The solutions are, on several cases, more efficient, more elegant, and more complex than the solutions discovered by the human mind [19] [33] [34] [35] [36] [37].

### Biological Terminology

In Nature, the living organisms are composed of cells, whereas a cell contains one or more chromosomes (the organisms with paired chromosomes are called diploid and the organisms with unpaired chromosomes are called haploid) [20]. A collection of chromosomes is called genome. Further, the chromosome is divided into genes (i.e. a gene may encode the eyes color and another gene may encode the hair color). The different values for a gene are called alleles (i.e. blue eyes, green eyes, etc). The position of a gene within the chromosome is called locus.

Another used term is the genotype that refers to a particular set of genes from a set of chromosomes. The phenotype - defined contrary to the genotype - defines characteristics such as morphology, development and behavior. If the genotype is inherited, the phenotype is developed during the life of the biological organism. In Nature, during sexual reproduction, the crossover exchange genes between every chromosome involved; the mutation is the result from copying errors

occurring during the reproduction process. The fitness is associated with the probability of that organism living for reproduction (viability).

In genetic algorithms, the chromosome is a possible candidate solution to a specific problem. Many times, the chromosome is replaced by the term genome, mainly because the candidate solution is composed only of a single chromosome (remember that a collection of chromosomes creates a genome). The genes are considered as being subparts of a chromosome and are used to encode a specific feature or a specific parameter. Different values for the genes define the alleles, which may be for example, in bit string chromosome values of 0 or 1. The crossover operator, in most of the cases, exchange genetic information between two chromosomes, and the mutation operator flips a bit from a random locus.

### Mathematical Foundations

The "Holland's Schema", defined by equation (2.5), is considered to be the foundation basis for explanations regarding the genetic algorithms power [38] [39]. A schema is a template that identifies a subset of strings with similarities at certain string positions. For example, the considered schema 1\*\*0\*1 describes all the set of strings of length 6 where at position 1 and 6 there is a 1, and where at position 4 there is a 0. The symbol "\*" represents a "don't care", thus we may have at positions 2, 3 and 5 either a 0 or 1. The Holland's Schema Theorem or Fundamental Theorem of Genetic Algorithms considers that "short, low-order, schemata with above-average fitness increase exponentially in successive generations" [38] :

$$m(H, t + 1) \geq \frac{m(H, t) * f(H)}{a_t} * [1 - p] \quad (2.5)$$

where:

- $m(H,t)$  is the number of strings belonging to schema H at generation t
- $f(H)$  is the observed fitness of schema H
- $a_t$  is the observed average fitness at generation t
- $p$  is the probability of disruption (because crossover and mutation may destroy the schema) as presented in equation (2.6) [38]

$$p = \frac{o(H)}{l - 1} * p_c * p_{diff}(H,t) + o(H) * p_m \quad (2.6)$$

where:

- $o(H)$ , called "schema order", is the number of fixed positions from the string (for example, in the schema 1\*\*0\*1,  $o(H) = 3$ )
- $l$  is the string length
- $p_m$  is the mutation probability
- $p_c$  is the crossover probability

## 28 | 2-Background

---

- $\delta(H)$ , called "length of schema" is the length between first and the last specific positions  
(for example, in the schema  $1^*0^*1$ ,  $\delta(H) = (6 - 1) = 5$ )
- $p_{diff}(H, t)$  is the probability that the second parent does not match the schema H

Using other terms, Goldberg considers in reference [39], that a schema with "fitness values above the population average will receive an increasing number of samples in the next generation, while schemata with fitness values below the population average will receive a decreasing number of samples".

The following example aims at revealing the differences between different schemas. Thus, we consider the following string S and the two schemata  $H_1$  and  $H_2$ :

S = 0 1 0 1 0 0 0 0  
 $H_1 = * * 1 * * * 0 *$   
 $H_2 = * * * 0 * 1 * *$

If a point crossover is applied between positions 3 and 4, it is easy to observe that schemata  $H_1$  will be destroyed because the "1" from position 3 and the "0" from position 7 will be placed in different offsprings. The schemata  $H_2$  will survive because the "0" from position 4 and the "1" from position 6 will be placed into the same offspring, after the crossover operation. The  $\delta(H_1) = 7 - 3 = 4$  and  $\delta(H_2) = 6 - 4 = 2$ , thus  $H_1$  will be destroyed with  $p_d(H_1) = \delta(H_1) / (l - 1) = 4 / 7$  and  $H_2$  will be destroyed with  $p_d(H_2) = \delta(H_2) / (l - 1) = 2 / 7$ .

Applying mutation for a particular schema, the schema survives when all the  $o(H)$  positions survive. Thus, computing the  $(1 - p_m)^{o(H)}$  we obtain the probability of surviving mutation. Considering that  $p_m \ll 1$  the result for the survival probability is [38]:

$$p_s = 1 - \frac{\delta(H)}{l - 1} * p_c - o(H) * p_m \quad (2.7)$$

The underlying explanation behind this theorem is that instead of trying to construct a complex string in order to represent the solution, better and better strings are constructed using valuable information from the previous samples.

### Design

In his first book [19], Koza creates associations between nature and genetic algorithms: "in nature, the evolutionary process occurs when the following four conditions are satisfied:

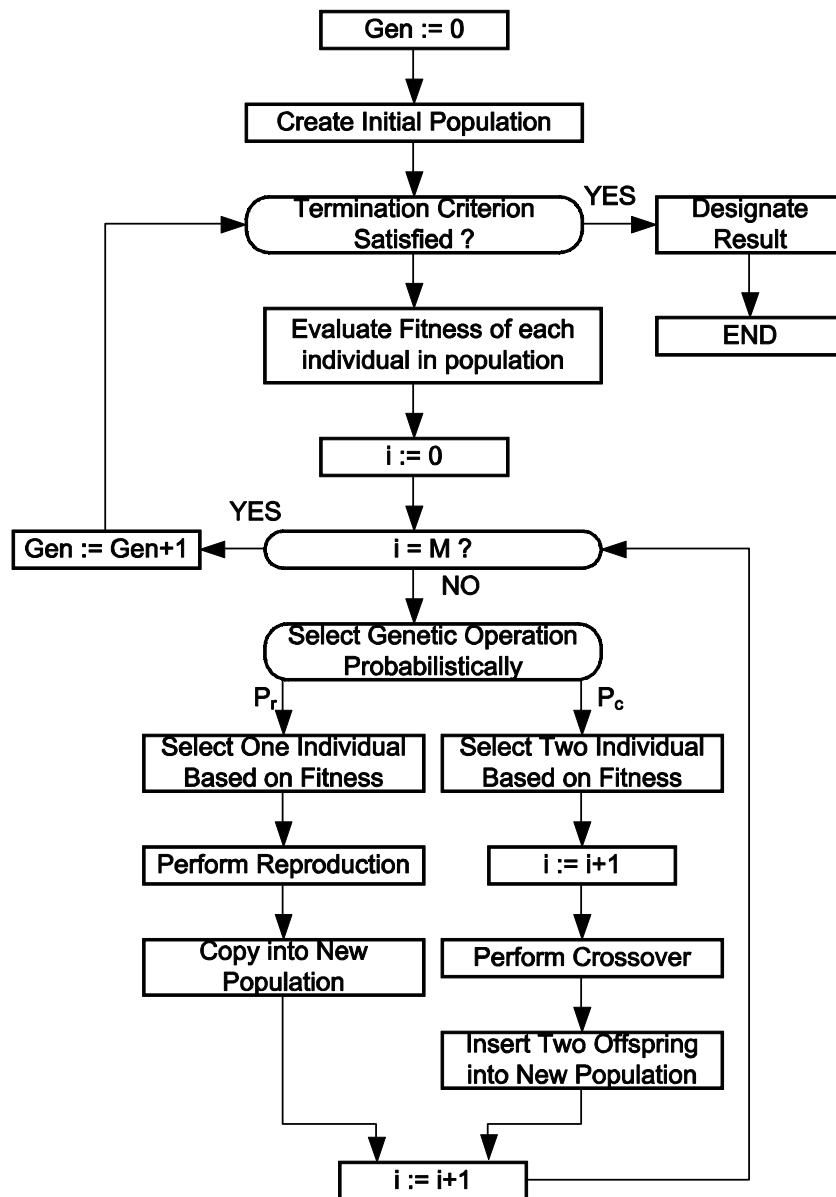
- An entity has the ability to reproduce itself.
- There is a population of such self-reproducing entities.

- There is some variety among the self-reproducing entities.
- Some differences in the ability to survive in the environment is associated with the variety”.

The corresponding steps to be fulfilled by a genetic algorithm (see Figure 2.4), according to Koza [19], will have to map the previous four conditions. Thus, a simple genetic algorithm works as follows:

1. Start the algorithm with a randomly generated population (only the chromosome length is fixed).
2. Repeat the following steps until the termination criterion is satisfied:
  - a. Calculate and evaluate the fitness for each individual from the population.
  - b. Create a new population using the evolution operators with a certain probability. The used operators are:
    - i. Just copy the individuals into the new population.
    - ii. Create two or one offsprings from recombining two individuals (crossover from substrings).
    - iii. Create a new individual by randomly mutate a position from its genome.

The best individual from each step is considered as solution for that iteration.



**Figure 2.4: Genetic Programming Paradigm [19]**

The flowchart proposed by Koza is the simplest method to evolve a genetic algorithm, and it is the basis for a wide number of applications. The flowchart does not provide detailed information; many details have to be filled in by the developer, at the implementation moment (i.e. population size, encoding, probabilities, etc). In the available literature there are also other versions for this flowchart, but the

structure is mainly the same (depending on the algorithm complexity, other processes are inserted in the flowchart).

## 2.3 Related Work

### Frameworks

There are several published genetic algorithm frameworks (see Table 3), each of them allowing ways of implementing new genetic schemes, with the proposed purpose of finding solutions for a given problem. The programming languages diversity is rediscovered in the available implementation for these frameworks: there are frameworks developed in C++, Java, Lisp, etc.

**Table 3: Available C/C++/Java Frameworks**

Name	Details	Language
Galib [40]	Galib contains a set of C++ genetic algorithm objects. The library includes tools for using genetic algorithms in order to perform optimization in any C++ program by employing any representation and genetic operators.	C++
Open Beagle [41]	A C++ Evolutionary Computation (EC) framework; it provides a high-level software environment in order to implement any kind of EC, with support for tree-based genetic programming.	C++
GPC [42]	The GP kernel is a C++ class library that can be used to apply genetic programming techniques to all kinds of problems.	C++
PGAPack [43]	It is a general-purpose, data-structure-neutral, parallel genetic algorithm library.	C
Splicer [44]	A genetic algorithm tool used to solve searches and optimization problems.	C
GAGS [45]	A C++ class library, which contains classes used to program all the Genetic Algorithm elements.	C++, Java

Comparing our framework with other approaches, several details are to be noticed:

- In Galib, Matthew Bartschi Wall has defined a C++ library of genetic algorithm components. The library implementation evolved over three years since 1993, and even had an update in 2007. We consider that due to its complexity, the library is not so easy to use, even if the author proposes a solution where the

## 32 | 2-Background

---

user has to start a new implementation following an existing example. The work is documented, the source code is available, but the architecture views are missing.

- The OpenBeagle was developed starting from 2002 and provides a high-level software environment for evolutionary computation algorithms, while providing support for different data structures (i.e. tree-based, bit string, integer or real vector, etc). Authors consider that the framework follows strong principles of object oriented programming, by using different levels of abstractions and allowing easy code reuse. The software architecture is explained partially in the provided manual, mainly with focus on the mentioned abstract layers; however, important details about the relations between the (abstract) objects seem to be flawed (more elaborated terms, relations within the data structure are not enough detailed).
- The GPC is one of the first known C++ frameworks for tree-based genetic programming. It was developed mainly between 1993 and 1997, even before a C++ standard was available. Inside the framework there are parts that do not promote a clear object oriented programming, there are no design patterns implemented, and there are many relations and usages between the classes. The library provides support for automated defined functions, tournament and proportional selection, steady state genetic programming, multiple populations, improved random generators, etc.

Another view concerning the mentioned frameworks - this time considering only the C++ approaches - is presented in Table 4 .

**Table 4: Analysis of GA Frameworks**

<b>Name</b>	<b>Com-plexity</b>	<b>Data structure</b>	<b>Architecture views</b>	<b>Abstract layers</b>	<b>Comments</b>
Galib	high	complex	-	-	The library is not so easy to use, even the author proposes a start-up solution
Open Beagle	medium	complex	-	+	The software architecture is partially explained in the provided manual, with focus mainly on the mentioned abstract layers



GPC	low	low	-	-	Within the framework there are parts that do not promote a clear object oriented programming
GAGS	Medium	low	+	-	Only binary haploid representation is possible

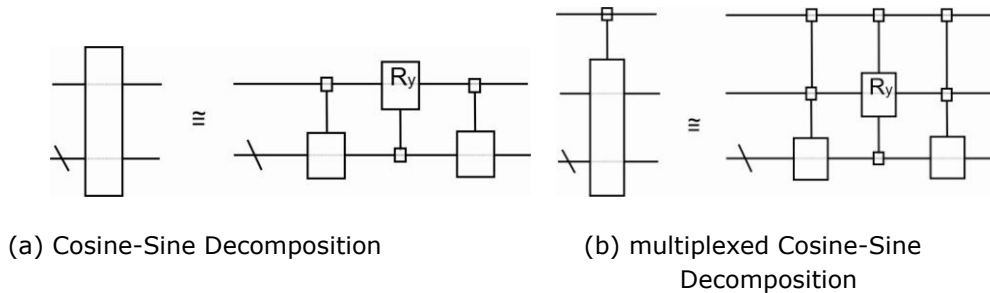
All the frameworks shall allow extensibility (possibility of adding a new genetic algorithm type), the user having to add only small parts of code that are closely connected with the problem definition that will be implemented. In this way, the developer will not lose time with the base part of the genetic algorithm because it is already implemented in the framework, and will focus his attention only on the problem particularities.

### Quantum Circuit Synthesis

In reference [11], Lukac and Perkowski have identified the following question: how to encode the number of wires and the gate position within the circuit, by employing the least complex data structure? They have proposed a transformation of the quantum circuit in an encoded chromosome, in order to be used in a standard genetic algorithm. In the encoded chromosome, the following rules are imposed: equal probability of presence of each gate type, fast individual encoding and decoding, and no other parameters beside basic definitions (no control bits). The potential weak point is that, beside the gate order, there is no information indicating what gate is connected to what wires. Thus, in order to obtain the chromosome, it is required that the quantum circuit be altered by employing swap gates.

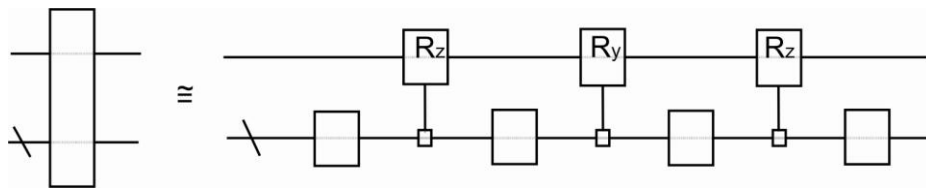
Rubinstein, in reference [46], considers-for the genetic algorithm-a scheme in which a gate has a type, a number of sets for the qubit operands, and some sets of parameters for different categories (the generalized 2-qubit gate takes four real parameters for different types of rotations; the CNOT gate takes a number of control qubits, etc). The quantum circuit is considered as a list of gate structures, where the size of the circuit (number of gates) is variable.

Shende et al. [47] proposed a top-down structure and effective computation, by employing the Cosine-Sine Decomposition (see Figure 2.5 (a)). Decomposition for quantum logic, that is analogous to the well-known Shannon decomposition (see Figure 2.6) of Boolean function  $f = x_i f_{x_i=1} + \bar{x}_i f_{x_i=0}$ , is introduced. It expresses an arbitrary  $n$ -qubit quantum operator in terms of a  $(n-1)$  qubit operator (cofactors), by means of quantum multiplexors (see Figure 2.5 (b)).



**Figure 2.5: Cosine-Sine Decompositions [47]**

With the help of an optimized quantum multiplexor, and applying this decomposition recursively to quantum operators, a synthesis algorithm is obtained in terms of quantum multiplexors.



**Figure 2.6: The Quantum Shannon Decomposition [47]**

In reference [48], it is considered that the heuristic methods employed for synthesis will have non-optimal circuits as a result. The usage of a local optimization technique is proposed under the form of the so-called templates, in order to simplify the circuit output and to compact the circuit levels. Implementation is provided in terms of NOT, CNOT and controlled-sqrt-of-NOT gates. The method output is represented by reductions of the number of quantum and levels. Templates are the generalization of the idea that two circuits, implementing the same function, may be replaced by another circuit (having a small cost in terms of gate number). This is called linear cost metric. Sometimes, the local optimization for a sub-circuit will lead to a global cost optimization of the circuit, called non-linear cost metric. For the level compaction, the usage of a greedy algorithm is proposed in order to apply the reduction rules. It is assumed that every two circuits have the same execution delay, and that the non-intersection gates from the neighborhood can be executed in parallel, with the delay being equal to a single gate delay.

In references [47], several quantum logic circuit pattern optimizations are proposed (see Figure 2.7). The task of circuit synthesis for reversible circuits is approached by employing a minimal number of gates [49]. Sub-circuits suitable for reduction are identified and transformed in simple circuits using the pattern

optimization rules. The resulting circuit is considered optimal if no other equivalent circuit, having a smaller cost in terms of gates number, exists.

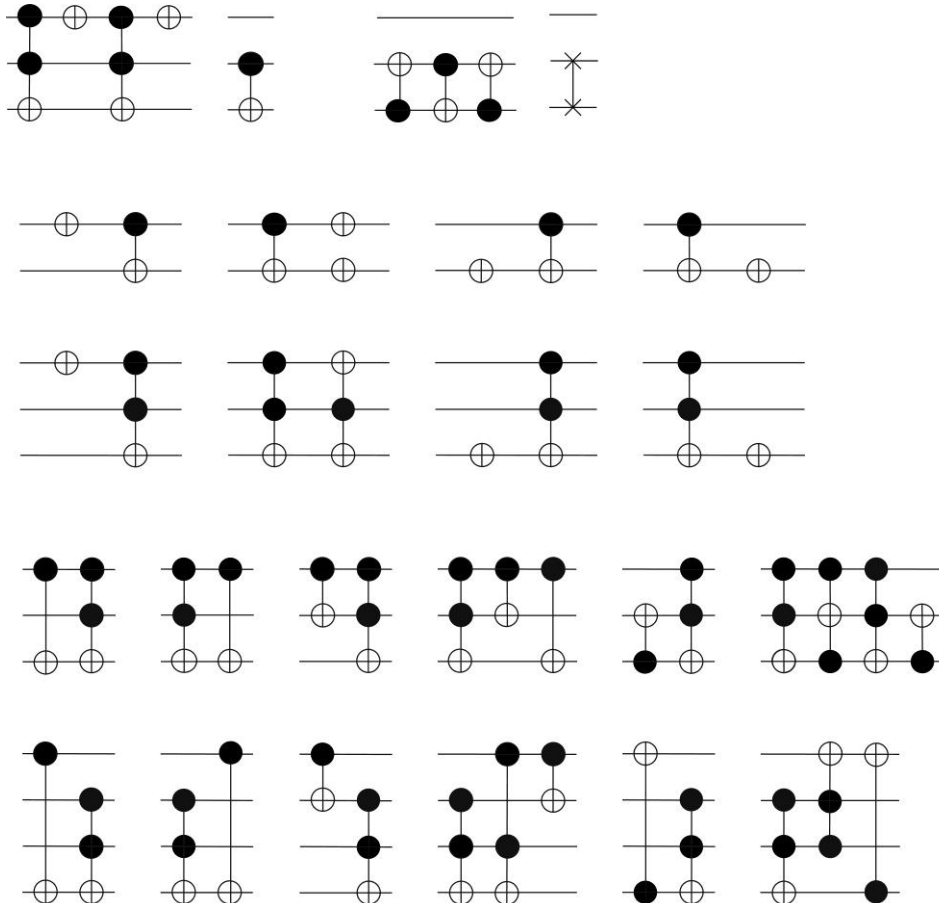


Figure 2.7: Circuit Equivalence as Proposed by Shende et al. [47]

A report from June 2006 [10] considers the logic synthesis for quantum and classical circuits as equally important. It is considered that - in the current implementation technologies - gates that act on three or more qubits are prohibitively difficult to implement in a direct manner. Also, it is considered that a sequence of two-qubit gates is crucial for quantum computation. Quantum circuit blocks are identified and quantum conditions are analyzed, before defining quantum multiplexors that generalize CNOT, Toffoli and Fredkin gates. The quantum multiplexor implements the if-then-else conditionals, when the controlling predicates are evaluated to a non-classical state. In the end of the analysis, an analog

## 36 | 2-Background

---

quantum Shannon decomposition is applied. The most representative published results show the following figures:

- An arbitrary  $n$ -qubit quantum state can be prepared by a circuit containing no more than  $2^{n+1} - 2^n$  CNOT gates
- An arbitrary  $n$ -qubit operator can be implemented with a circuit containing more than  $(23/48)4^n - (3/2)2^n + 4/3$  CNOT gates.

### Quantum Circuit Synthesis

Quantum circuit synthesis is considered as the automatic combination and optimization of quantum circuits in order to implement a specific function. The quantum circuit synthesis is an extensively investigated topic. Several research groups have published results, and positive reductions have been obtained in the gate reduction number, qubit reduction, or even faster runtime.

The team from University of Michigan where - Quantum Circuit Group<sup>8</sup> - that has proposed and resolved several synthesis problems. In one of their latest papers [10], they have “developed efficient quantum logic circuits which performed two tasks: (i) implementing generic quantum computations and (ii) initializing quantum registers”. The constructed generic circuit is considered the most efficient known at that time, in terms of CNOT gates.

In the quantum synthesis field, we have to notice also the work of Dmitri Maslov<sup>9</sup> at the Institute for Quantum Computing from University of Waterloo. They have proposed different approaches for quantum logic circuit synthesis and optimization; for example, in reference [50] the template approach is used to optimize a quantum circuit. In addition, Maslov created the “Reversible Logic Synthesis Benchmarks Page”, where important research results concerning the synthesis are available for further benchmarking [51]. Many papers have been published by this group on the quantum synthesis domain [52], [53], [54], [55].

Important research work on quantum circuit synthesis, using genetic algorithms, has been developed at the Portland State University by Marek Perkowski<sup>10</sup> using their experience from the conventional systems design methodologies and tools; they have proposed new methods for quantum circuit synthesis (i.e. as presented in reference [11]).

One of the first books about the genetic approaches to the quantum-computing domain was published by Lee Spector [7]. A software implementation written in Common Lisp was used to solve several quantum computation problems. The author considers that many quantum problems may be considered as search problems and consequently may be solved with the help of genetic programming.

---

<sup>8</sup> <http://vlsicad.eecs.umich.edu/Quantum>

<sup>9</sup> <http://www.iqc.ca/~dmaslov/>

<sup>10</sup> <http://web.cecs.pdx.edu/~mperkows/>

## Chapter 3

### Analysis of the QCS

This chapter is dedicated to the analysis of the quantum circuit synthesis problem. The requirements are presented in a structured way, starting with a higher granularity level and then going into the smallest details; the architecture is detailed using UML (Unified Modeling Language) terms [56] [57] - which is the modeling language of the object-oriented approach, - thus allowing understanding and a global view on the most important packages involved in the problem definition.

One of the goals of this dissertation is to provide design guidance for the quantum logic circuit synthesis task. The design evaluation involves the examination of design rationale behind the constraints and the comparison of the system proprieties derived from those constraints, with relevance for the system implementation.

Quantum circuit synthesis has an important role in the development of quantum computing technology. This new technology brings advantages in comparison with the classical one (analog or digital). In the last decades, the automatic classical circuit synthesis has improved the use of new circuits (in terms of development time, delay time, integration scale, cost and time to market, etc), allowing developers to be more creative than ever. New complex applications are possible and the classic physical technology limit is pushed to the edges. Physical implementation of quantum circuits is in its incipient phase, and there are only few quantum gates available (this by using an expensive and complicated technology: photons, ion traps, etc) [58]. It is considered that the future of quantum computing technology depends on the existence of reliable physical quantum devices [59], therefore, software synthesis tools are desirable.

Software architecture is an important sub-discipline of software engineering, within the realm of large system development. The architecture gives the possibility to focus on the essential components and their interactions rather than presenting details. A layered architecture allows to individually building each part and, in the end, these parts will work together towards problem implementation. The architecture will present the important proprieties of the system as performance, reliability and extensibility. Software architecture is a complex entity and, in order to describe a specific propriety, a view must be used. When using a specific view, only the interested aspects of the system are presented, while other aspects are intentionally suppressed. This is the reason why within the architecture, different views are necessary to highlight system elements and their relationships. In the following sections, we introduce our system as an abstract object, several views being presented to help in obtaining a better documentation [60].

### 3.1 QCS Problem Statement

Automated quantum circuit synthesis is still only at the beginning of its exploration, at the same time our dissertation tries to create the premises for developing software tools dedicated to this task.

#### QCS: Description

When trying shortly to present the problem definition, we may consider the following phrase as requirement: perform circuit synthesis having a quantum function as target and a set of operators as arguments. After decades of study, solving the problem becomes somehow less demanding when dealing with the digital or analog circuit synthesis. When the problem is moved into the quantum computation context, the situation is different. From the available state-of-the-art, there is no common accepted path to follow for finding a solution; the theoretical basis is not complete yet and the physical devices are not available on a large scale at this moment. There are many questions to be answered, and further research is vital in order to clarify these details.

In an extended manner, the required system must perform quantum circuit synthesis by having a formal description for a circuit and a set of quantum gates, using genetic algorithms:

- A classical computer shall be used for developing a software tool chain used for the quantum circuit synthesis.
- The tool chain shall contain a parser used to interpret the quantum circuit details, as specified in a high-level description language.
- The tool chain shall contain a repository where elementary gates are stored.
  - For each quantum gate, the quantum cost shall be specified.
- The repository shall allow definitions for quantum circuits (a quantum circuit is composed of one or more elementary quantum gates).
  - For each quantum circuit, the feasibility value shall be specified.
- The tool chain shall contain a genetic algorithm that is responsible in evolving an optimal synthesis quantum circuit.
- The genetic algorithm is also responsible with the optimization of the evolved circuit.
  - Minimize the quantum circuit cost.
  - Maximize the quantum logic circuit feasibility value.
- The tool chain shall contain a drawing application allowing quantum circuits visualization.

### **Approaching QCS: Context and Major Challenge**

The relevance synthesis process has two views. According to the first one, if the progress in technology is extremely fast, then it is outstripping the designer's abilities to make use of the created opportunities. The second view is generated by the situation where the technology is not available on large scale and the designers can only use a small set of gates for the design process. The development and the application of new and more suitable design methodologies are important for the modern computer system industry. Quantum circuit synthesis has a bigger relevance when it is related to the simulation results. Any result taken from the simulations may have a physical implementation, with the help of the synthesis algorithm. Therefore, starting from a program written in a high programming language we obtain the physical device by employing the synthesis tools.

The automatic quantum circuit synthesis is a difficult problem, due to the following issues:

- The total number of possible gate permutations is huge and we cannot be sure if the genetic algorithm has evolved the best solution (is considered that the optimum solution can be evolved). In order to overcome this problem, a genetic algorithm will be used due to its capability to evolve solutions into a large search space.
- Optimization of the evolved circuit is hard to tackle. In general, the optimization is an iterative process; hence, an optimized solution is used again as input by the genetic algorithm. The genetic algorithms are suitable for optimization.
- Even a small change in the quantum topology can have a huge impact on the circuit functionality.
- There is no complete quantum circuit benchmark to be used for the quantum logic circuit synthesis solution assessment.
- The quantum gate cost and feasibility are not clearly defined yet (depending on the gate inputs, or depending on the gate implementation complexity, etc).

### **3.2 Characteristics of the Tool Support**

Today's research activity tries to move all the software development into a single application, that can support all the development phases together, while performing all this in a user-friendly manner. Tool support in object-oriented design aims at providing means for capturing requirement, architecture and implementation activities. The tool activities that are necessary for defining and developing a software system are listed below:

- Modeling: well-defined visual notation (i.e. UML) is used by programmers in order to understand the principle upon the system design process.

- Components: the possibility of splitting the development in several components that, eventually, will be integrated together for defining the complete software system is allowed.
- Specification: allow writing the specification within the same tool as for the design, and preserving traceability through them.
- Coding: direct relation between architecture details and code implementation will improve quality and productivity.
- Testing: possibility of defining test cases for each requirement

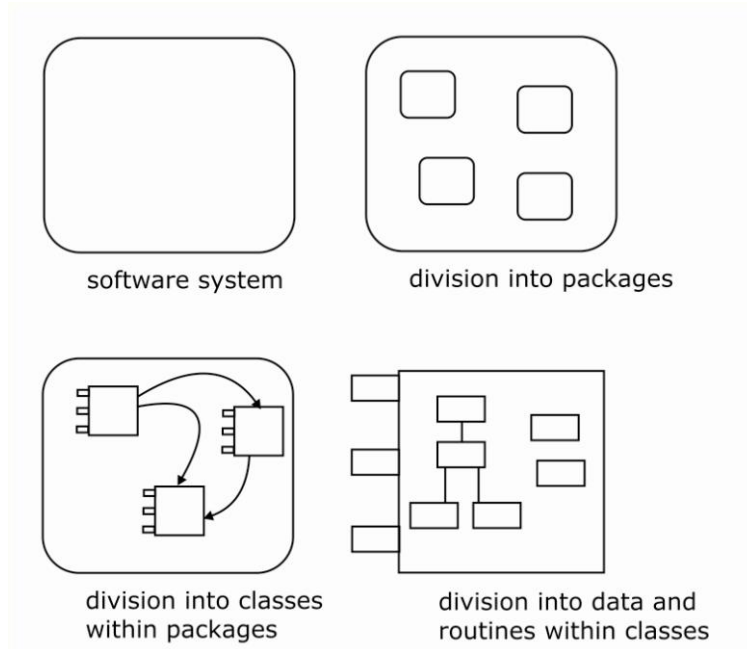
Since Rhapsody is a Model Driven Development (MDD), being widely used by a growing user community, and because it provides support for all the necessary software activities, we decided to use it for our software project. "Telelogic Rhapsody® is an industry-leading UML® 2.1 [...] environment for technical, real-time or embedded systems and software engineering. Rhapsody enables reuse of existing software assets, whether source code or model based, provides a flexible development environment for both function-oriented and object-oriented graphical design techniques to co-exist in one environment and improves productivity and quality through validation of the design early in the development lifecycle when defects are less costly to fix" [61].

The license for Rhapsody Development Edition was offered free of charge by the Telelogic Company through Rhapsody University Program, for a fixed two years period.

### 3.3 Levels of Design

Design is needed at different levels; the first level is the system, and it is a common mistake to jump off the class level when the architecture is started. Then, the next level is the division into packages, where the architect shall identify the major subsystems and their interactions. The intercommunication rules are also defined on this subsystem abstraction level. System dependencies to important packages shall be identified and maybe isolated, thus allowing future extensibility by less couplings.





**Figure 3.1: The Design Levels in a Program [62]**

A package may contain one or more classes, depending on its purpose. Within the package, the class relations shall be well-defined using interfaces, the system being decomposed in fine class functionalities that implements behaviors. We need to make the difference between class and object; an object represents an entity that exists at system run-time while a class is a static code that contains attributes and methods. In the last level, which is responsible with designing classes into data and routines, it is obvious that several behaviors may be implemented with simple routines, while others need data handling and state machines. In this last level, it is important to have only private data and access them via interface methods (“A good class interface is like the tip of an iceberg, leaving most of the class unexposed” [62]).

### 3.4 QCS Architecture Overview

The architecture describes given software systems by using multiple abstracting levels. This dissertation examines all the involved levels, from the system abstractions to the implementation details.

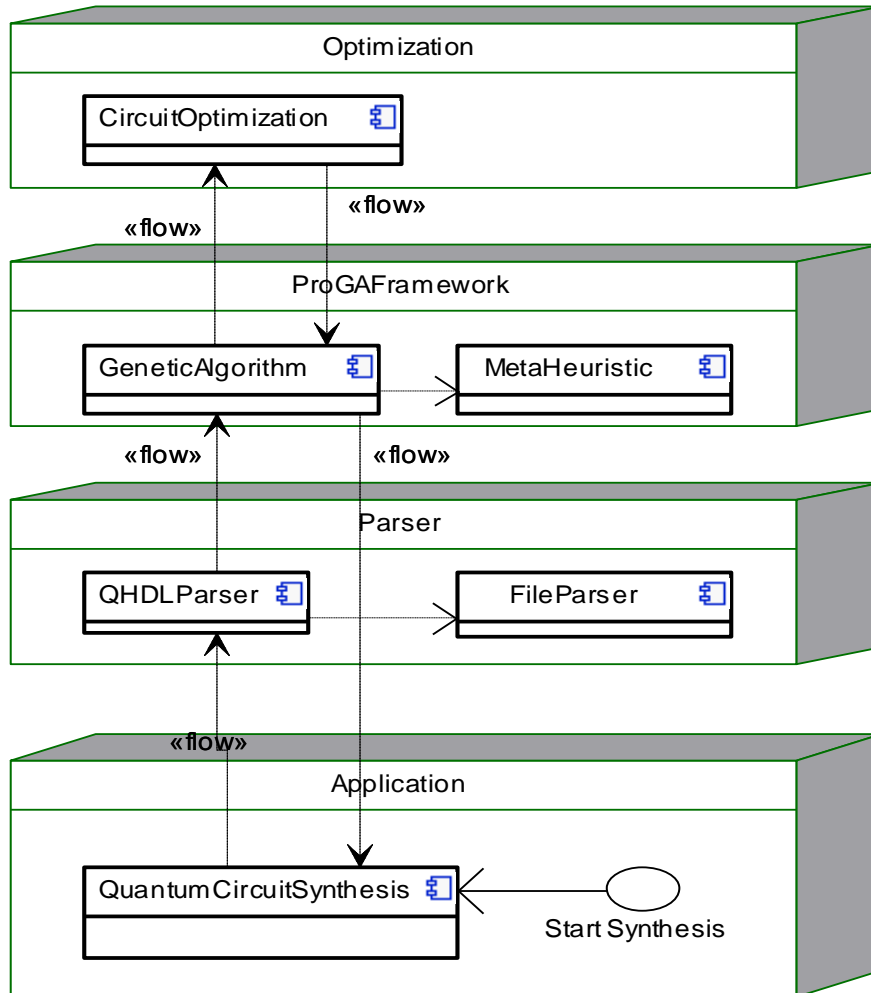
At first glance, different parts may be easily identified: a high-level description language parser used to map the quantum circuit description to a low-level representation, an algorithm responsible for the optimization of the abstract circuit, and a genetic algorithm responsible for the synthesis task. The synthesis

## 42 | 3-Analysis of the QCS

---

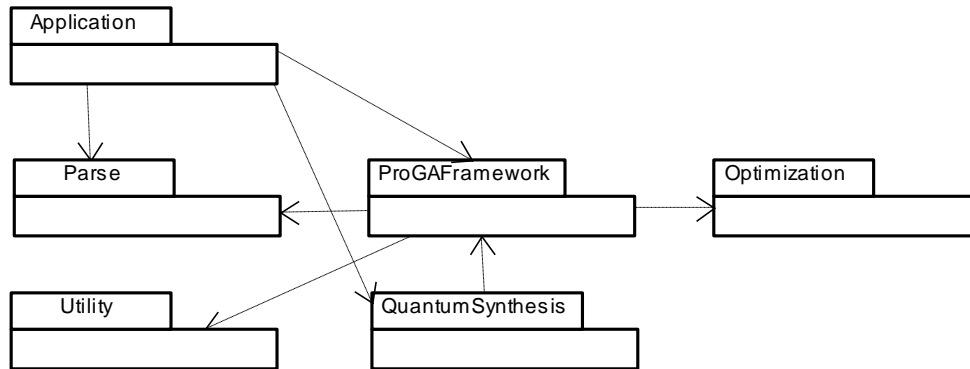
task is performed following the described phases in a cascade manner. Therefore, starting from a program written in a high-level description language we obtain a physical device by employing synthesis tools.

The architecture derives straightforwardly from the described methodology. The proposed breakdown structure indicates a layered software architecture (see Figure 3.2), each layer being responsible for a dedicated task. The ripple computation allows for intermediate results that can be used or maximized at the next layer. In this way, starting from a circuit description in a high-level language after applying all the phases, eventually leads to the corresponding circuit. As intermediate result we have the abstract description of the circuit (which is not hardware dependent), the internal data representation used for the optimization, etc. Explaining in brief terms the deployment diagram, the initialization for QHDL parser will start the synthesis process. The QHDL parser uses a generic file parser to create the internal data structure that is then used by the genetic algorithm. The adjustment for the genetic algorithm parameters control is made by the meta-heuristic component. In the end, the evolved solution is optimized and maybe a new evolution cycle is triggered. The synthesis solution is provided as result to the user under then form of a circuit layout.



**Figure 3.2: Software Behavior Deployment**

The architecture is important in the realm of system development. The system architecture presents the involved parts, along with the specific relations among them. A first view on architecture shall indicate the divide-and-conquer concept about the main functionality, followed by printing how the parts are working together. This view allows building the parts individually and merges them later to solve the problem. The architecture establishes constraints and sets the downstream activities in order to produce artifacts.



**Figure 3.3: QSynTool Architecture Layout**

The QSynTool<sup>11</sup> [63] architecture (see Figure 3.3) is split into five main parts, each part corresponding to one major operation required by the quantum circuit synthesis. Starting from the problem definition, three main components are presented as relevant: parsing (understanding the high-level language), optimization (for the abstract representation) and quantum circuit synthesis (creating the circuit layout). The Utility contains several additional tools.

### 3.4.1 Rationale

Several rationale decisions were made during system architecture development. For example, the existence of a genetic algorithm framework was necessary to allow a facile implementation for a genetic algorithm, and to have access to the statistic results during evolution. The significance of statistical results is used for the self-adaptation of the genetic algorithm parameters. In addition, statistical values are used to prove the algorithm convergence and for later analysis (i.e. for a better tuning process). Another rationale decision was to use complex numbers for the quantum gates representation. Working with many quantum gates or quantum circuits is not straightforward by any means; keeping the entire gate attributes up-to-date and having control on which object may be used for synthesis as imposed by the existence of a repository.

### 3.4.2 Constraints

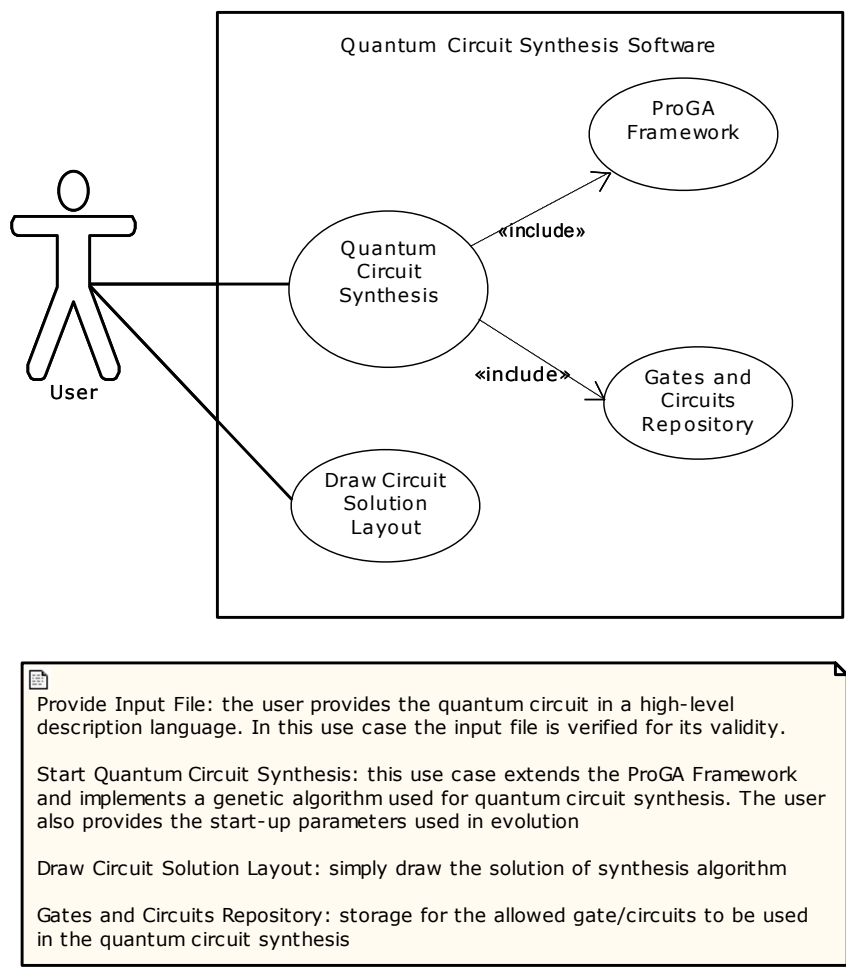
Several constraints are applied to the abovely described system. For example, the usage of the high-level description language was specified to be QHDL<sup>12</sup>. Another constraint was to use a meta-heuristic algorithm for evolving genetic algorithm parameters dynamically during evolution. Working with a repository of gates is our main constraint, only available gates being used by the synthesis process.

<sup>11</sup> QSynTool (Quantum Synthesis Tool) @ Cristian Ruican, 2008

<sup>12</sup> Quantum Hardware Description Language

**3.4.3 Logical View**

This view supports behavioral requirements, thus the services that the system shall provide to the end users, also known as primary system functions or system usages (see Figure 3.4). The problem definition is decomposed into a set of key abstractions, as objects and class objects that incorporates the principles of abstractions, encapsulations and inheritance. At the same time, during system decomposition, we identify the common parts across the system (i.e. utility module).



**Figure 3.4: System Provided Services**

### 3.4.4 Process View

This view is responsible with the system performance, integrity and fault tolerance. There, the focus is on the parts interconnectivity. The sequence diagrams are useful for illustrating the sequential interactions among architectural components, describing scenarios and not complete behavioral specification. Sometimes, during the requirement phase, analysts may use sequence diagrams in order to provide a more formal level of requirements. The sequence diagrams express the requirement transition from use cases into components interaction.

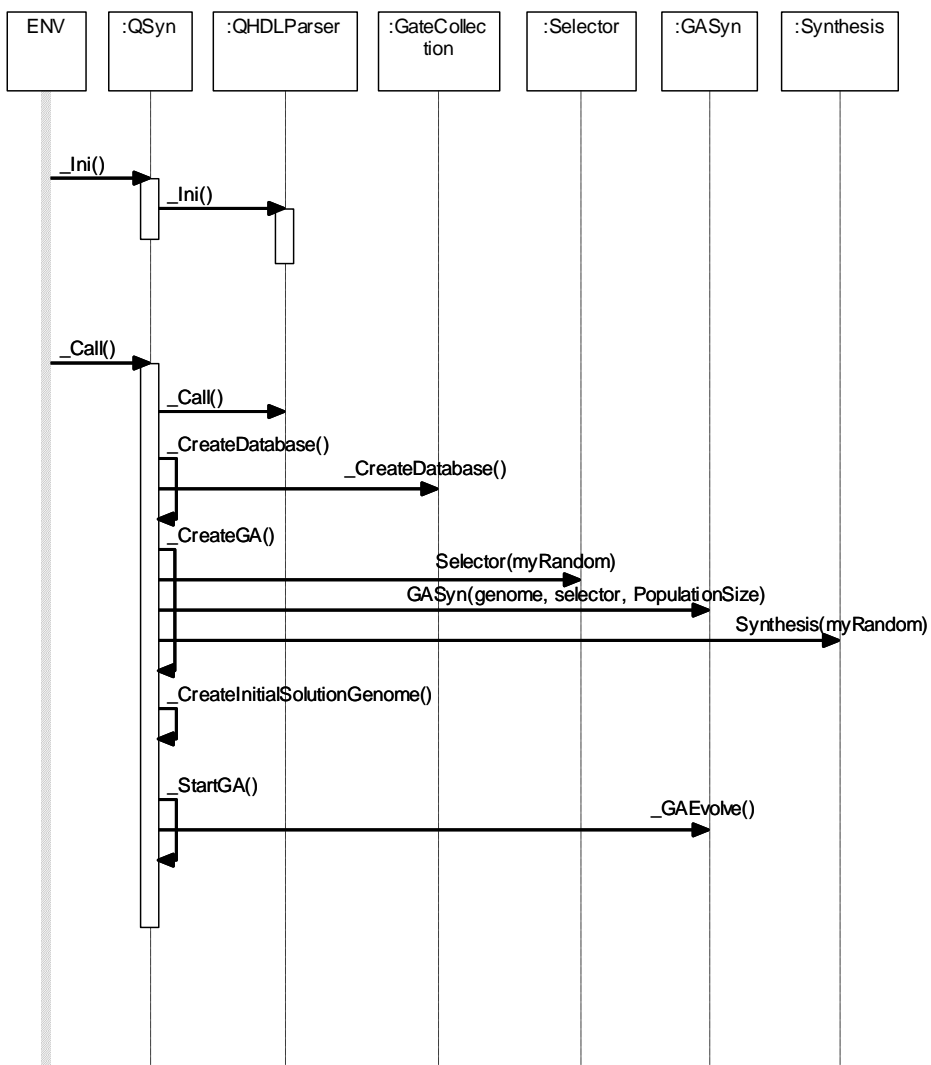


Figure 3.5: QSynTool Process View

### 3.4 -QCS Architecture Overview|47

Two main tasks are described in Figure 3.5. The system initialization is responsible only with the parser initialization, where the input file is provided. The main call triggers the parser operation that creates the internal data structure, triggers the database creation (where the quantum circuits used for synthesis are described), triggers the creation of genetic algorithm and its initial solution and starts the evolution. In the end, the synthesis result is provided to the main task.

#### 3.4.5 Development View

In this view, the focus is on the organization of the software modules in small chunks of software, showing how the system is organized in source code, binaries and libraries. Nevertheless, the class diagrams, as presented in the development view are useful for describing architecture meta-models and are therefore used by developers to design and document system's coded classes.

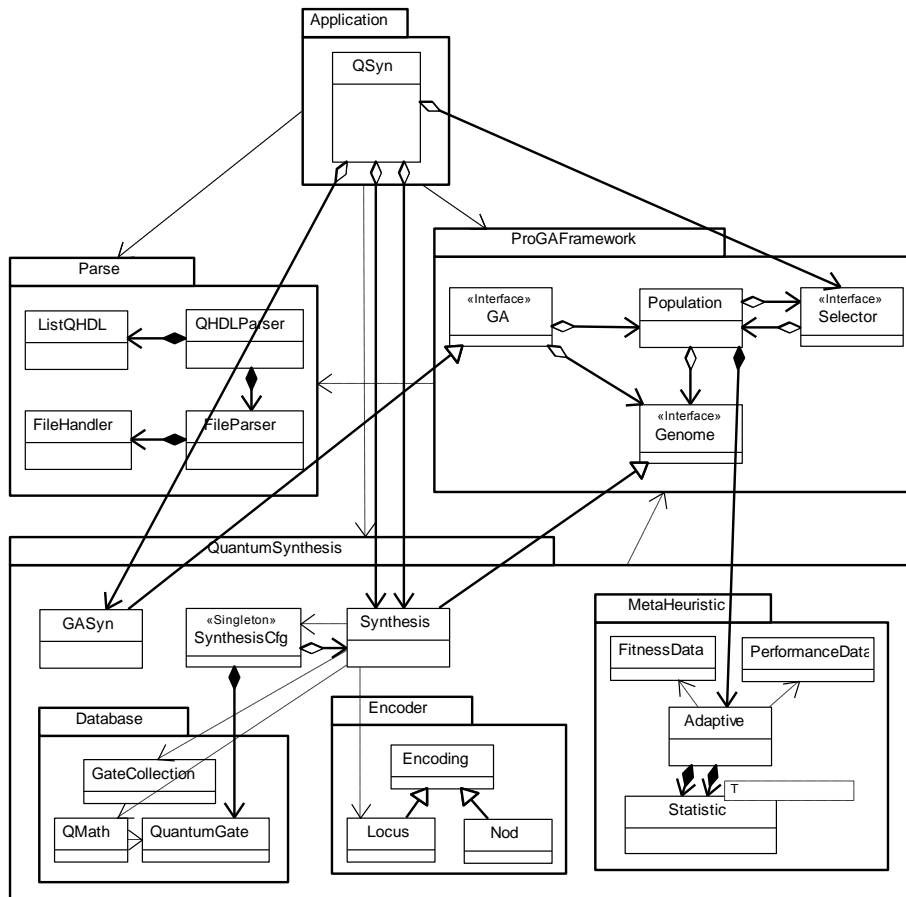


Figure 3.6: QSynTool Development View

The global architecture layout is presented in Figure 3.6. In the Parser package, the QHDLParser has a FileParser object that implements a generic parser and has a ListQHDL object that stores the internal data structure. The ProGA Framework package describes a genetic algorithm framework responsible with the genetic algorithm details (i.e. evolution, population, selector, genome, etc.). The QuantumSynthesis package is considered as the most important package from the architecture, describing the quantum circuit genome, and contains other packages responsible with other tasks. For example, the Database package groups the quantum gates and their mathematical operations in a quantum gate collection that is used for the synthesis task. The Encoding package provides support for common data used in the internal data codification, and is then used in the chromosome genes. The Metaheuristic package creates the possibility of dynamically adjusting the genetic parameter controls using statistical data from the current search state. The creator, Application package, allocates all the necessary objects and creates the links between them. This package triggers the synthesis process and receives the evolved result. More details about each package and about the class diagrams are presented in the corresponding subchapter where the behavior is explained.

### 3.5 Architecture Properties

Designing a system implies modeling high-level structures and presenting them in terms of views, styles and patterns. Several architectural proprieties are used to present different styles in this dissertation. The Rhapsody tool provides supports to define the following properties.

#### Performance

The quantum logic circuit system performance is firstly defined in the system requirements, followed by the implementation. Each phase is important to define the system performance, the decision to change or influence different structures may make the difference between success and failure. The system performance involves additional costs in each component implementation, because performance is the most important feature for our system. It is hard to quantify the performance quality, and in our system, it is mainly based on the capability to obtain a synthesis solution for different types of inputs and by the latency time defined as the period between the system initialization and the system response. The latency time is divided in several particular time intervals: the initialization time (measured between the moment when the initial stimulus is available and the internal data creation event), the time necessary to setup the initial genetic algorithm solution, the time required for saving the statistical data during the genetic evolution and the time required evolving a circuit synthesis solution. Different component interactions may introduce additional time values that may alter the system performance (i.e. the task commutations time in different operating systems).



### **Scalability**

Scalability is a desirable property of system architecture, to support more components within an active configuration. The possibility of having more configurations, allows keeping the components interactions within the tool and controlling them. For example, the measuring time component may be included or not in the active configuration, thus the tool will not generate the corresponding files if necessary. Splitting the components into small parts induce simplicity, because complexity is widely spread within the architecture. The tool support allows to perform fast changes on the generated packages and to modify the interconnectivity between them, everything using configuration files.

### **Modifiability**

It is important to be able to perform a change to the architecture without big impact to other components. Even if the system is fully compliant with the requirements, the requirements will change over the development cycles or time. The synthesis system allows coexistence of different implementations and integration of new ones, being able at the same time to extend its functionality. The existence of different implementations for the same requirements, allows dynamical customization of the active features. Different components also allow configuration of parameter controls, adjusting the available functionality to optimal synthesis purpose. Object-oriented design, together with design patterns, improves the system modifiability value.

### **Portability**

The developed architecture can run in different environments. The object-oriented language used is standard C++, allowing running the software in Windows or UNIX systems. The Windows operating system is used for the development and debugging, while the UNIX system is used for the system assessment.

### **Reliability**

Different parts from development are used, in order to increase the reliability degree to different types of failures. We introduced monitoring for several components, creating incentives for identifying failures from an incipient phase, and avoiding general system failures (i.e. during parsing operation, during quantum gate creation, etc.). Reliability is important in maintaining system functionality when it is operated under stated conditions for, a given amount of time.

### **Minimal Complexity**

It is important to avoid making clever designs, but make simple and easy-to-understand designs instead. This minimal complexity also allows easy maintenance, because somehow the design will be self-explained. Splitting into packages, having deployment diagrams, sequence and class diagrams divide the system complexity into small parts that may be handed individually with less complexity.

### Loose coupling

Keeping the relations between components to a minimum value is important for a good architecture. Using the principles of good abstractions in public interfaces, data encapsulation and inheritance allows for having only few interconnections.

### 3.6 Classical vs. Quantum Digital Circuit Synthesis

In our days, the request for new systems (hardware and software) has dramatically increased. The integrated circuit technology (based on semiconductor materials) has progressed continuously: starting with Intel C4004 (from 1971), the world first single-chip microprocessor, and continuing with our day's microprocessors. The technological development was mainly focused on three major types of circuits:

- Very Large Scale Integration [VLSI], that are used on common applications. VLSI is the process of creating integrated circuits by combining thousands of transistor-based circuits into a single chip.
- Application Specific Integrated Circuits [ASIC] that are used on specific applications, being specialized for a dedicated task (i.e. a chip designed solely to run a cell phone is an ASIC).
- Field-programmable gate array [FPGAS] that are reconfigurable circuits used to implement prototypes. Due to the benefits coming from the theory of reconfigurable computing and evolvable hardware, FPGAs are now used to implement standalone functioning systems, and not just for prototyping.

The term Ultra Large Scale Integration [ULSI] is used to reflect the growth of circuit complexity, and it was proposed for chips of complexity more than 1 million of transistors. From a technical point of view, there is no qualitative leap between VLSI and ULSI, so any ULSI may be considered a VLSI device. Due to this increasing request on the market, the CAD techniques have two major advantages: the reduction of development time and complexity, while reducing the time-to-market, which creates profit for developing companies. CAD techniques allow for an increased quality due to automatic operations (is well known that human complex tasks may introduce errors).

When discussing about the design of the microelectronic circuits, many design styles may be applied, but (usually) just two methodologies are mostly used: custom and semi-custom design styles [64]. The custom style is mainly used for dedicated parts of the circuit (floating point unit or execution unit). The semi-custom style is based on the concept of restricting the circuit primitives and thus reducing the fine-tuning for the circuit parts during the design phase. Semi-custom designs are split in other two subclasses as cell-based and array-based. Cell-based designs are based on libraries of cells that are designed once and stored (called standard cells), or are using the cell-generators for the macro cells (such as memory, gates, etc), where larger cells can be derived by combining the smaller

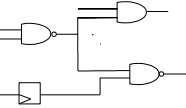
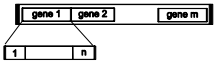
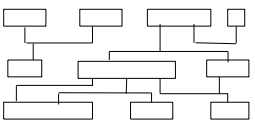
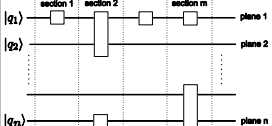
### 3.6 -Classical vs. Quantum Digital Circuit Synthesis|51

ones. Updates to libraries are necessary due to the progress in technology. This task is not an easy one, because each cell needs to be updated, and because many parameters are present (i.e. integration area, delays over ranges, temperature, voltage limits, etc). A major advantage of the cell design is the compatibility between the custom style and the semi-custom style (we can add custom parts to the semi-custom layouts and vice versa).

Array-based design considers the circuit as a matrix of unrelated components. The role of design is to create the connections between these components and to perform their personalization. We may consider a second classification in the prediffused and prewired, also called mask programmable and field programmable gate arrays (MPGAs and FPGAs). A standard cell design approach makes it possible to globally apply advanced optimization algorithms, that reduce the manual effort required, and improve the quality of the synthesized logic during layout. The use of basic standard cell elements reduces complexity to such an extent, that a complete chip design can be handled as flat, by layout and test generation tools, hence removing the need for artificial floor plan boundaries [64].

The model is one of the first artifacts throughout the microelectronics design process. It is used to show the abstract functionality of the circuit, the architecture, without presenting the details (having a great focus on the problem space). Graphical models such as flow diagrams, schematic diagrams, and layout diagrams may describe the functionality from different views, therefore allowing to the designer to focus only on a part of the model at any given time. Any model may introduce different abstractions layers [64]. In Table 1, a comparison between the methods employed by the circuit design in the classical vs. quantum world is presented, highlighting the common parts that are followed by logic circuit design.

**Table 5. Classic vs. Quantum Circuit Design**

Levels	Classic Circuit Design [De Michell]	Quantum Circuit Design
Architectural	<pre> ... PC = PC + 1; FETCH(PC); DECODE(INST); ...                     </pre>	<pre> ... Cnot(d,c); Hadamard(a); OpX(b); OpY(e); ...                     </pre>
Logic		
Geometric		

## 52 | 3-Analysis of the QCS

---

Synthesis tasks are presented on each level (architectural or logic). Synthesis can be seen as a transformation between two axial views and different subsequent tasks are possible:

- Architectural-level synthesis determines the block-level structure of the circuit. It is used to define the circuit function, resources, interconnection and timing. For the quantum circuit synthesis, a specific parser was designed to identify the circuit description blocks and to create the internal data structure that will be later used in the synthesis process.
- Logic-level synthesis creates the logic specification and the interconnections between the architectural blocks. Often is considered to be as a library binding or a technology mapping. Going to the quantum situation, we created a quantum circuit database that stores the quantum gates characteristics. Each gate is randomly selected by a genetic algorithm and used to encode a possible solution for the circuit synthesis problem.
- Geometric-level synthesis is involved in finding the geometrical position of components (the circuit layout). The quantum circuit layout is partially defined by our approach, by splitting the circuit into sections and planes where the gates are placed in a defined order as provided by a topological optimization.

Synthesis without optimization will result in non-optimal circuits. The full circuit optimization is hard to obtain. Optimization may occur on different circuit parts and may have many goals. In classic circuit synthesis, for example, optimizations on the circuit integration area and delay latency are commonly encountered. Sometimes the optimization is based only on approximations. It is possible to optimize the architectural level by improving the dependencies between circuits; it is possible to perform optimizations at the logic level, where function minimization is a goal; and it is possible to have optimization at the physical level (i.e. time reaction, integration space). In the quantum synthesis process, the optimization is seen as improving the circuit quality through the minimization of the number of gates used, as presented in two recent papers [65] [55]. In our approach, this is implemented automatically by the genetic algorithm evolution (the scope is to evolve better-and-better individuals in terms such as number of used gates, circuit cost and circuit feasibility).

## Chapter 4

### Genetic Algorithm Framework

A Genetic Algorithm is a programming technique used as a problem-solving strategy that employs the principles of Darwinian natural selection. The genetic algorithms are using equivalences from the biological evolution in order to solve a specific problem. A genetic algorithm will receive as input a set of potential candidates for the solution (i.e. population), encoded in a useful way (i.e. genome or chromosome), and will then evaluate them by using a fitness function that indicates how good the individual is. Mainly, the set of potential solution candidates is randomly generated, but in some cases, they may be also chosen from any previous partially known solutions. The encoding allows for storing of the candidates and then their manipulation in a facile way. Of course, after the evaluation of the fitness, it is possible, that the candidates (or the individuals) would not fit to a solution. This is the point where the evolution becomes necessary. The process it is made as follows: randomly, with some probabilities, the genetic algorithm selects the best individuals for mate (there are also other selection schemes: roulette wheel, tournament, uniform, etc). The mate is performed as crossover and/or mutation, and the new offspring is introduced inside the new population. As the process from Nature, the copy or the reproduction process introduces errors, thus purely by chance it is possible to obtain several good individuals, which are then copied over the next generation. After the reproduction process, it is important to increase the average fitness value; this fact will guide the algorithm in the following iterations more closely to the solution. In other words, the solution for a given problem is evolved through successive iterations.

A software framework is designed to allow easy development for any type of engineering problem. The framework advantage is that all the low-level details are already implemented in a software library, and the developer can use its time for working only on the specific problem details. A Genetic Algorithm Framework will allow easy integration of different genetic algorithm problems. Software methods and design patterns are applied in order to create the necessary abstract levels for the genetic algorithm [66] [67].

The framework allows for different configurations, and thus the comparison between the characteristics of the emerged solutions becomes straightforward. This design creates incentives for practical solutions, because the inheritance from the defined abstract classes makes possible the creation of new genetic schemes. The patterns allow the framework to extend actual available implementations for their own use. The inheritance also allows - in a small development time - the adding new genomes, new selection schemes and new genetic algorithms dedicated to specific tasks. The applied design patterns aim at describing, in a clear way, both the problem and the core of the solution. The obtained genetic algorithm pattern may be applied many times, running in the same accepted way. The framework

## 54 | 4-Genetic Algorithm Framework

implements the simulation of Nature's genetic processes with the declared purpose of finding a solution for a given problem.

### 4.1 Framework Preliminaries

The framework is described with respect to its architecture, included components, implementation, and application. The framework development was started from the genetic algorithm requirements, as stated in the flowchart proposed by John Koza (see Figure 2.4). The flowchart provides a single, unified approach to the problem of finding a computer program that solves a given problem. The algorithm creates individuals, which are all grouped within a population, that increase their average fitness during genetic evolution by applying the Darwinian principle of natural selection, preservation of the best, and survival over generations. Sexual reproduction (crossover) is used to create offsprings, mutation is used to bring variety into the population, and selection is used to allow stochastic survival of individuals [68].

### 4.2 Framework Packages Architecture

The framework we propose in Figure 4.1, named ProGA [Programming Genetic Algorithms] [69] creates a decoupling of the genetic algorithm from its data structure. It is important to be able to develop new genetic algorithms without being forced to perform modifications on the population or on the genome parts. Details about each class are presented in the following subchapters, with emphasis being put on the abstract classes and on the decoupling objects. Abstract classes are used to allow generic development for new algorithms and data structures.

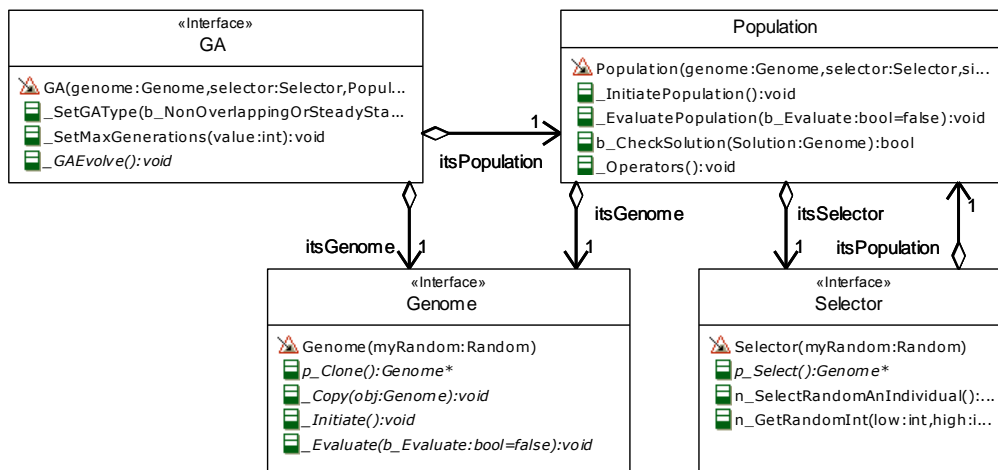


Figure 4.1: ProGA Framework Architecture

The Bridge Pattern [66] is used with the declared purpose of decoupling the genetic algorithm implementation from its several possible implementations (which, in our view, are derived classes from the Genome class). The abstract classes declare the interface to the abstraction, and the concrete subclasses implement it in different ways. Separate class hierarchies are created (one for the genetic algorithms, one for the genomes, and one for the selector), thus allowing object decoupling for the generic implementation. All operations from the derived classes are implemented in terms of operations from the interface. The Bridge Pattern brings several advantages for our framework: both abstractions and implementations are extensible by sub-classing, changes on abstraction have no influence on the implementation and vice versa, permanent binding between abstraction and its implementation is avoided, therefore allowing switching at runtime (in this way, it is even possible to change the algorithm during execution). Moreover, the decoupling encourages the layering of the architecture, having a better-structured system in the end.

Our framework also allows - if necessary - the Decorator Pattern from the same reference [66], for extending the actual architecture. Via decorator, it is possible to add responsibility for genetic algorithm class, or for the genome class, without affecting the other objects. It is important to have an architecture that allows extensibility and - more important - to have an architecture designed for this purpose.

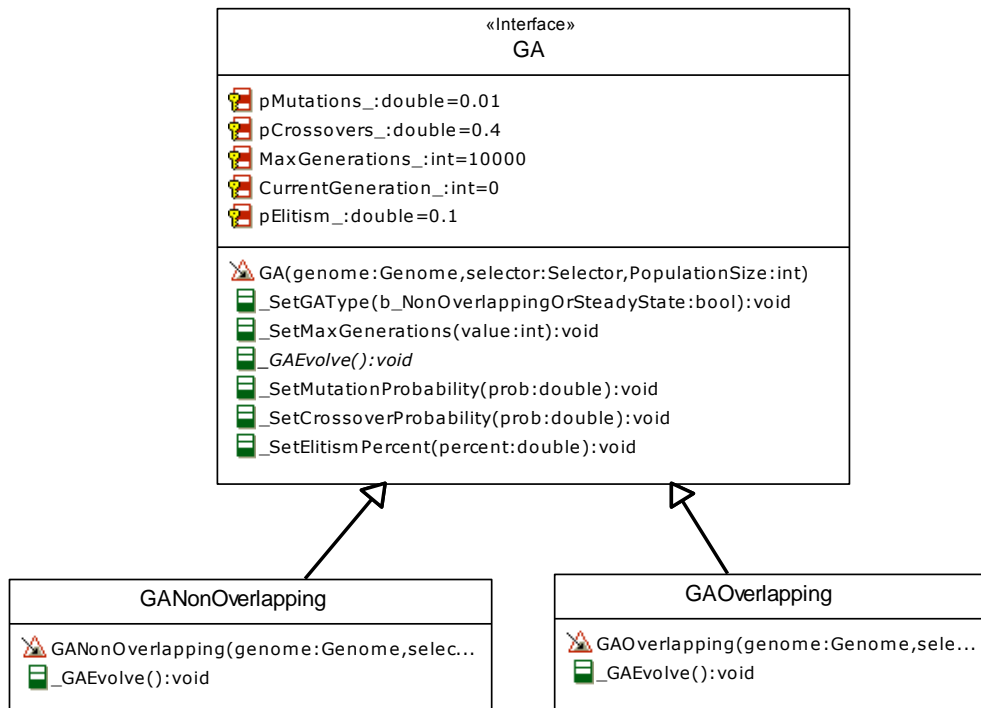
The proposed framework was mainly implemented with the scope of being used for quantum circuit synthesis. It provides support for adding a dedicated genetic algorithm that will be used for circuit synthesis. It is important to have, as a standalone tool, a framework that will provide additional information, which is necessary to adapt the genetic algorithm to the best or optimum performance. The genetic algorithm optimization is necessary, because we do not know how to design a quantum circuit to solve a given problem, and because there is almost no indication about the quantum circuit efficiency. For example, the statistical information will be used for adapting the genetic algorithm, similar to the feedback process, and will allow for finding the most favorable configuration for the synthesis genetic algorithm. We consider the framework as a real help in overcoming many difficulties that appear in the synthesis process.

### 4.2.1 Genetic Algorithm Types

Any kind of genetic algorithm must describe iteration loops that increase the convergence of the individuals towards a solution. Our framework allows the creation of new genetic algorithm as derivations from the GA base class (which is an abstract class), and implementing the only method responsible for the iteration loops of the algorithm (see Figure 4.2). The base class has knowledge about the termination criterion and about the probability used for the natural selection. The GA class is also responsible for creating the population that will be used during the evolution (the client is the one that knows the number of the individuals and the type of the genome used). The GA class was designed as an abstract class, thus the

## 56 | 4-Genetic Algorithm Framework

decoupling from data structure is possible at this level. In addition, the user is forced to implement - in the new derived classes - the abstract method for evolving; framework robustness is achieved in this way.



**Figure 4.2: Genetic Algorithm Class Diagram**

The genetic algorithm class implements the genetic operations necessary for evolution (initialization, mutation, crossover, evaluation, solution verification, etc), by calling the corresponding methods from the aggregated population object. Thus, the decoupling is performed in such a way that the genetic algorithm does not know how to do those operations; it has only knowledge about their existence. The client knows what kind of genetic algorithm intends to run. The client also has the knowledge about the genome used for the evolution, and about the selector type. A random number generator required in order to provide true random numbers, because standard libraries only return pseudo-random numbers. Then, the last operation is to bind the relations between the objects and let the evolution perform the indicated job.



### 4.2.2 Genome Implementation

The Genome class is an abstract class that implements the data structure of the algorithm. Different implementations are possible at this level, the derived genome classes being forced to implement methods as initialization, mutation, crossover and evaluation, for each individual (see Figure 4.3).

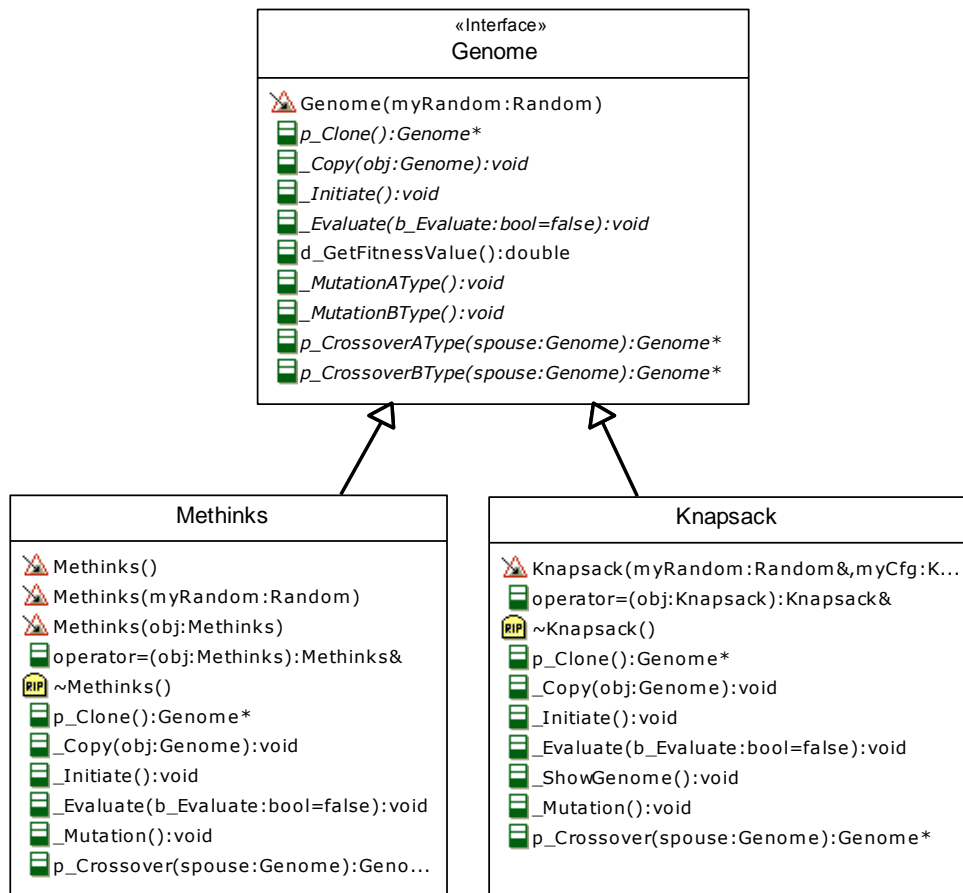


Figure 4.3: Genome Class Diagram

The main attributes are defined in the base class: fitness value, number of mutations, crossovers, evaluations. The *crossover* operator has a specific particularity, because the genome itself does not know about its pair, but knows how to perform the crossover. The knowledge about the crossover is located on the population object, because the individuals involved in the sexual reproduction are

selected there. Due to its design, any derived genome class must implement the genetic operations related to the data structure handling the genome.

### 4.2.3 Population Implementation

The genetic population contains all the individuals that are managed by the genetic algorithm. The population design is a container for the genomes, and its design has generic methods for accessing the genomes. During the algorithm evolution, the population class is responsible for stochastically the individuals that will suffer mutations and/or crossovers and for tracking the evolution for each individual (number of mutations, number of crossovers, etc). The population must have knowledge of the basic operations for its individuals: initialization, mutation, crossover, evaluation, etc). Each generation will store statistic information related to the best/worst/mean fitness, as well as to the total number of mutations, crossovers and evaluations.

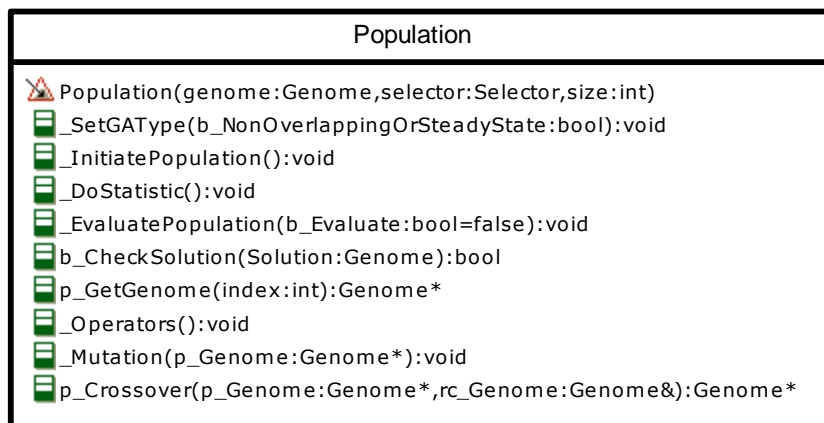


Figure 4.4: Population Class Diagram

Concerning the architecture, the *Population* class (see Figure 4.4) is not abstract, all the necessary methods being implemented at this level. If a user considers that other alternatives are more viable, the *Population* class can be decorated with new functionality, following design patterns methods (i.e. the Decorator Pattern). The population object is created from the genetic algorithm object (it knows how many individuals are within each generation) and has knowledge about the selector and about the type of the genome; a genome that will be used later for population cloning.

### 4.2.4 GA Operators

The role of the genetic operators in the evolution process is to bring diversity into population. The diversity may conduct the algorithm to a solution. A very simple genetic algorithm shall have at least three operators: selection, crossover, and mutation. The genetic operators are used to simulate the Darwinian evolution processes on chromosomes.

#### **Mutation**

The mutation operator introduces an error within the evolution process, and randomly flip-flops some of the locus from the chromosome. The mutation operator also creates variety into population, preventing the population of converging to a local minimum, by introducing new individuals. Different mutation schemes are therefore available: flip-bit, boundary, non-uniform, uniform, Gaussian, etc. The mutation operator is implemented within the derived class according to the problem specification.

#### **Crossover**

The crossover operator mimics biological recombination between two chromosomes. The crossover operator is applied on two individuals (which are randomly selected); a random locus is used to interchange the substrings between the individuals previously selected, and to create one or two offsprings. There are many crossover schemes, the common point being the recombination based on locus interchange (before and after the locus). Different recombinations are possible: one point, two points, uniform, etc. The crossover operator is implemented within the derived class according to the problem specification.

#### **Elitism**

The elitism operator is used to keep the elite of the genomes (composed by genomes that have a higher rank) into the next generation. In this way, from one generation to other, the best-evolved results are reused in order to create new individuals. It is considered, somehow, that good parents may have also good children. The genetic algorithm convergence will be higher when the elitism operator is used.

#### **Selector**

During the evolution, the selection operator selects chromosomes from population for the purpose of reproduction. There are different selection schemes (i.e. rank, tournament, roulette wheel, uniform, etc) which stochastically select single or multiple individuals based on their fitness or other features.

It is impossible to have a genetic algorithm without having a selector method. During evolution, single or multiple individuals are stochastically selected from the current population (based on their fitness or other properties), and modified (recombined and – possibly - randomly mutated) to form a new population. From literature, the most well known selection methods are:

- Rank Selection that will pick the best-ranked individual.
- Roulette Wheel Selection, where each individual has a chance of being picked from a roulette wheel, proportionally with its fitness value.
- Tournament Selection, where two individuals are randomly chosen and the best is selected.
- Uniform Selection, where an individual is randomly picked from the population.

Depending on the selection type, the random number generator is used to ensure the hazard on the selection. In our proposed framework, a simple architecture is implemented, with an abstract class called Selector that knows how to select an individual from the population, while respecting the selecting rules from its derived class (see Figure 4.5). In this way, each derived class is forced to implement the specific select method.

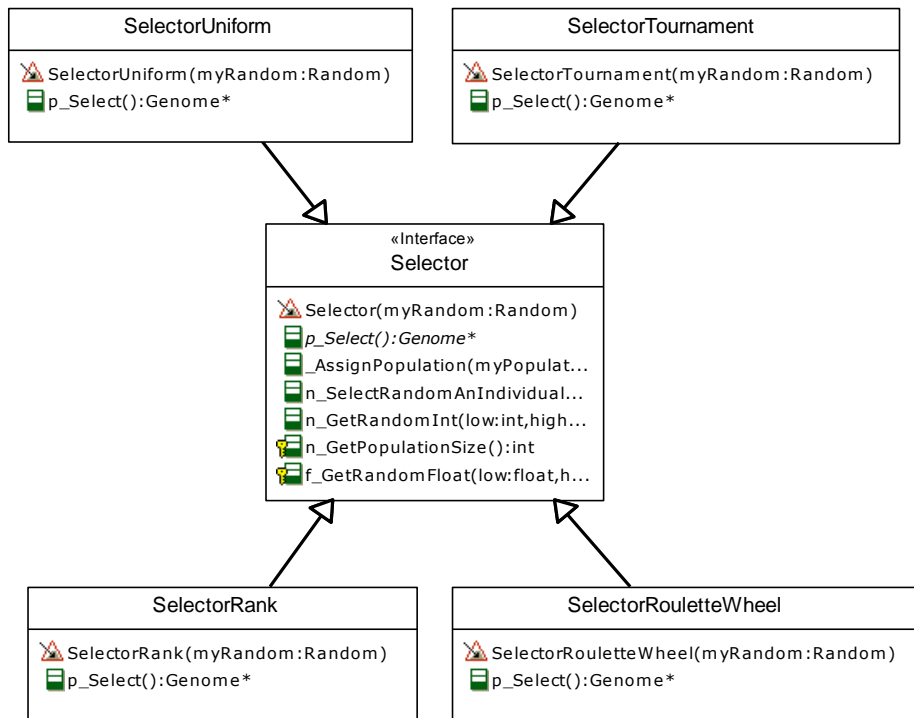


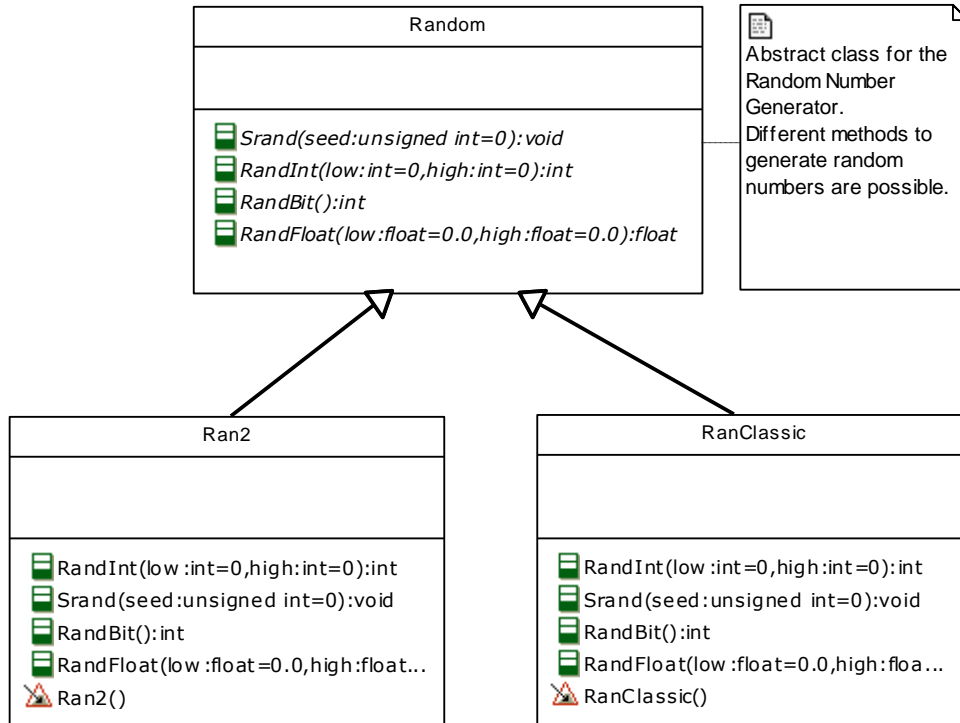
Figure 4.5: Selector Class Diagram

From the architecture, it is obvious that a population knows about the selector method used for the selection, and the selector knows about the genetic population. This population forms the pool of the individuals that are chosen for selection. The selector object is created by the user, because only he/she knows what specific algorithm is to be used for the selection, during the genetic evolution. The selector object is passed to the genetic algorithm in order to create the aggregation between the population and the selector (because the population is allocated dynamically by the genetic algorithm object).

### 4.2.1 Framework Utilities

When discussing about genetic algorithms, the hazard is essential. This is the main reason for which the standard libraries implementations for the random number generator should be avoided (the numbers are generated in a pseudo-random manner, thus a mathematical formula is used for prediction). The pseudo-random is acceptable for many applications, but for the hazard implementations-as roulette wheel or other specific probabilistic operations-, using dedicated algorithms for random numbers yields better results.

In our framework, an abstract class for a random number generator is used, allowing users to have their own random number generator. When defining the derived class, they are forced to implement a method for initialization of the generator (it is possible to specify a given seed, or as default, a variation of the system time is used), and to implement methods for returning random numbers as integers, floats or bits. We provided two implementations (see Figure 4.6) for the random number generator, the first one is inspired from [70], and the second one is an implementation based on the C++ library `stdlib`.



**Figure 4.6: Random Number Generator Architecture**

Measuring execution time plays an important role when different implementations - for the same problem - have to be benchmarked. The execution time is one of the genetic algorithm outcomes that needs to be compared in order to assure a global view about the genetic algorithm characteristics (execution time is important for the algorithm convergence towards a solution). We found out that measuring execution time is not a straightforward job, each implementation having different resolutions, and therefore different values for the same measurement. Moreover, the operating system introduces delays due to task priorities and the switching times between them.

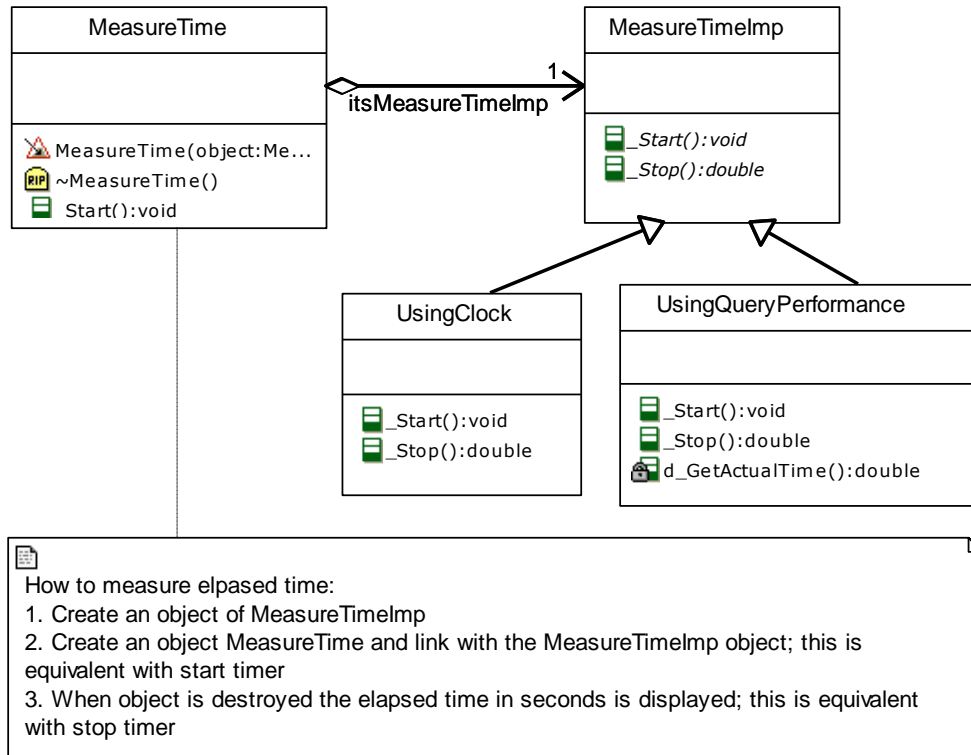


Figure 4.7: Architecture for Execution Time Measurement

We propose a framework that allows time measurement (see Figure 4.7) on different implementations (i.e., using QueryPerformanceTimer and ThreadPriority). The user may choose which implementation is more feasible for his/hers needs, even during the runtime of the algorithm.

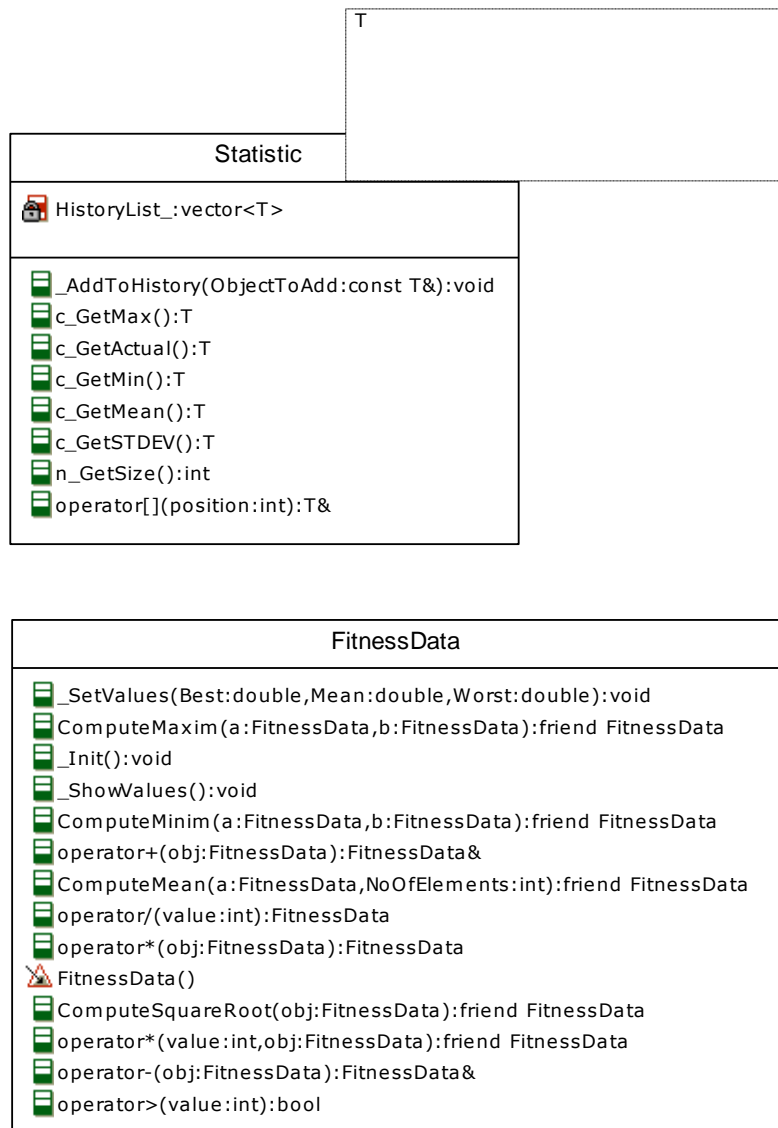
### 4.2.2 Framework Statistics

It is important to be able to compare the genetic algorithm results between different executions (repeating the algorithm) or with different starting parameters (different number of individuals, different probabilities, etc). In our framework, a statistic class was implemented, for providing the user with different statistic information about a specific generation or about the entire population. Statistical information about the maximum, minimum, average, and standard deviation is also provided.

The statistic class is implemented as a template list (using a STL vector), to store all the necessary information related to each generation. We choose to use the template because the statistic methods must be available on any kind of data type (i.e. operation as best, mean, minimum, standard deviation). The user will create the data type for which a statistical analysis is necessary and will instantiate the

## 64 | 4-Genetic Algorithm Framework

template. In our architecture, the population class will have as associated a statistic object (composition) of type FitnessData (storing values related with the best, mean and minimum fitness value). After the evaluation of the genetic algorithm, the statistic information is added into the history list (see Figure 4.8). When the statistic information is requested by the user (it shall be based on request in order not to overload the system), the framework will compute and will provide the results.



**Figure 4.8: Statistic Class Diagram**



For example, the standard deviation is used to measure the spread of the values through generations. The standard deviation is denoted by  $\sigma$ , and it is defined as the square root of the variance.

$$\sigma = \sqrt{\frac{1}{N} \sum_{i=1}^N (x_i - \bar{x})^2} \quad (4.1)$$

where:

- N is the number of samples
- $\bar{x}$  denotes the samples average

For speeding up the software algorithm, a formula equivalent is used, formula that may be obtained after applying standard algebra:

$$\begin{aligned} \sum_{i=1}^N (x_i - \bar{x})^2 &= \sum_{i=1}^N (x_i^2 - 2x_i\bar{x} + \bar{x}^2) = \left(\sum_{i=1}^N x_i^2\right) - (2\bar{x} \sum_{i=1}^N x_i) + N\bar{x}^2 = \\ &= \left(\sum_{i=1}^N x_i^2\right) - 2\bar{x}(N\bar{x}) + N\bar{x}^2 = \sum_{i=1}^N x_i^2 - N\bar{x}^2 \end{aligned} \quad (4.2)$$

thus,

$$\sigma = \sqrt{\frac{1}{N} \left(\sum_{i=1}^N x_i^2 - N\bar{x}^2\right)} \quad (4.3)$$

### 4.3 Framework Validation

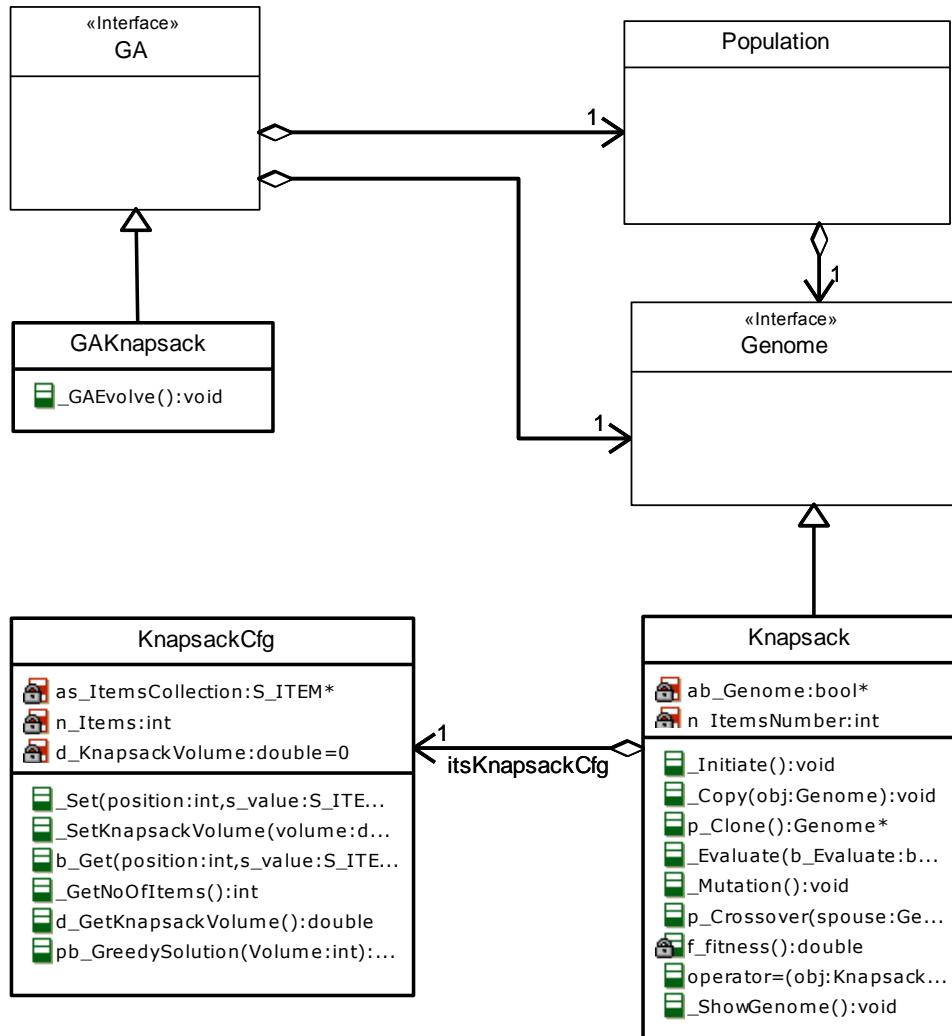
Two well-known genetic problems have been implemented, for proving the framework validation: the knapsack and methinks (more details in second PhD. Report [71]).

The knapsack problem [72] is known in the literature as a combinational optimization problem, and (at least throughout the last two decades) many studies have been proposed [73] [74]. It is considered one of the easy-stated NP-hard problems, and may be solved using different methods (i.e. by using dynamic programming the problem is solved in pseudo-polynomial time, using greedy programming a solution is reached in  $N/2$ , where N is the number of objects, etc). The potential of the genetic algorithms to yield good solutions was proven, and many papers have been published [75] [76].

The knapsack problem was implemented using the framework [77], by adding two new classes. We derived from the GA class in order to define our own genetic algorithm. All these were necessary because there are several particularities that may be configurable via the interface methods provided by the base GA class (i.e. genetic algorithm type, elitism, crossover, mutation, etc). After that, we

## 66 | 4-Genetic Algorithm Framework

derived from the Genome class for creating the chromosome necessary for the encoding of the candidate problem solutions (see Figure 4.9).



**Figure 4.9: Knapsack Class Diagram**

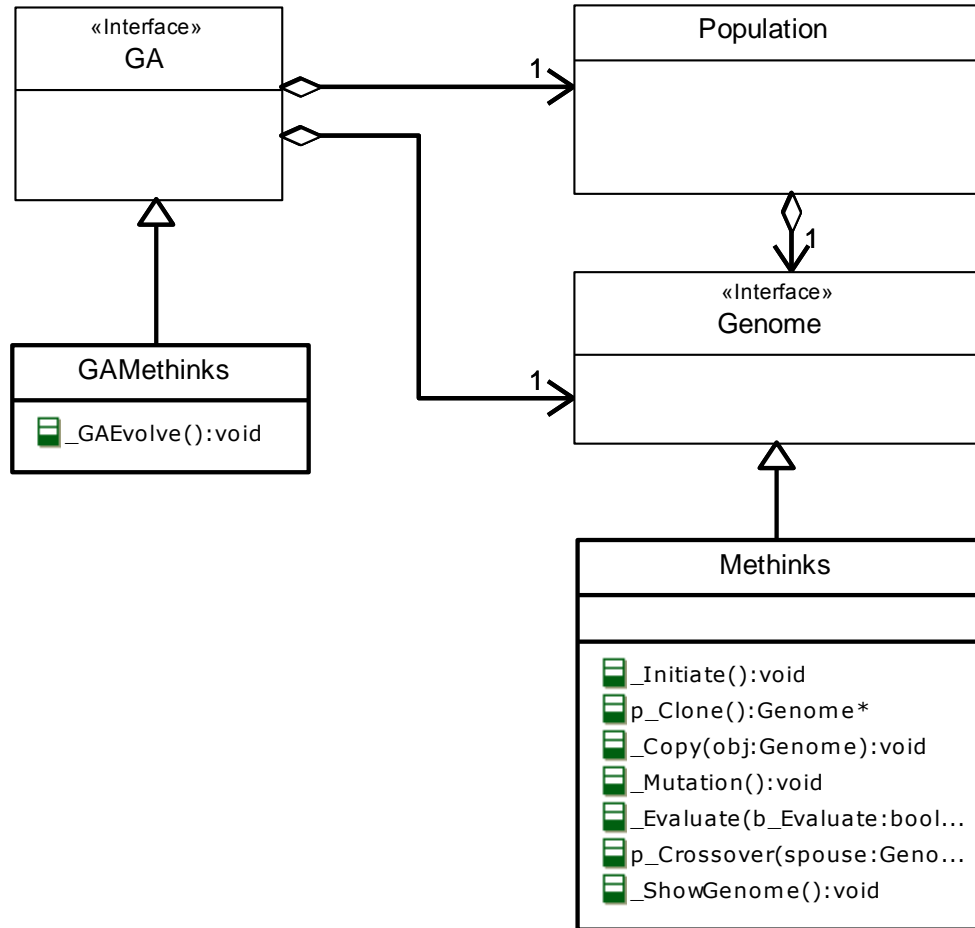
The chromosome was encoded as a bit string with the length equal to the number of items from the collection. A gene equal to 1 indicates to the algorithm the fact that the corresponding item is added into the knapsack and, in the same way, a gene equal to 0 indicates that the items are not used for the candidate solution. An item will store information about the object benefit and volume. The fitness function computes the benefit and the volume for the involved genome, and

then it returns the division from the genome benefit and the benefit of all the items. If the genome volume is higher than the knapsack volume, the fitness operator will add a penalty value and the genome is disqualified. We implemented the penalty fitness, because the fitness shall represent the proportionality of the profit in comparison with an infeasible genome:

$$fitness = \sum_{i=1}^N B_i X_i - \max(B_i / V_i) * (\sum_{i=1}^N V_i X_i - C) \quad (4.4)$$

The mutation operator will randomly flip-flop a bit from the bit string, including or excluding an item from the knapsack. The crossover operator is implemented as a two-point crossover; two chromosomes are stochastically selected and then interchanged from the cut points. The genetic algorithm will start with an initial population where the chromosomes are randomly generated. This means that each chromosome may be considered as a potential solution, even from the beginning. After initialization, the population is evaluated using the fitness function, each chromosome receiving a value according to its efficiency. The next step is to analyze if the best chromosome represent the solution (this is made by a comparison with the solution generated by another approach). When the equality results, the algorithm is considered finished and the evolution is stopped. Otherwise, the genetic operators are applied, in order to perform the algorithm evolution, and a new iteration is started. We used a non-overlapping algorithm for our solution, at the same time allowing the elitism.

When dealing with the evolutionary biology and with the infinite monkey theorem, Richard Dawkins [78] has pointed out that the task of the monkey writing the complete work of Shakespeare may be reduced to a phrase "Methinks it is like a weasel". Moreover, the typewriter shall have a reduced keyboard only with 26 capital letters and a space. He was wondering how much time would be necessary for the monkey to typewrite the phrase. The biological implication after the weasel program was important because Dawkins proved that if a small change in the genome is preserved over generations, and if there is a mechanism for selection, in the end meaningful offsprings will be obtained in short time. We start with the high-level implementation, the genetic class where it is specified what kind of genetic algorithm type is to be used (non-overlapping or steady state). In addition, in the same class, other specific proprieties shall be set (elitism percent, mutation probability, crossover probability, maximum number of generations, etc). This class is derived from an abstract GA class, and it implements all the necessary abstract methods used for the genetic evolution (see Figure 4.10).



**Figure 4.10: Methinks Class Diagram**

The second step is to encode a candidate solution, and for this task a string chromosome equal to the length of solution "METHINKS IT IS LIKE A WEASEL!" is used (thus, 28). The set of allowed characters is restricted to 26 characters and a space: "ABCDEFGHIJKLMNOPQRSTUVWXYZ" like in the problem specification. There are no special implementations for the mutation or crossover operators, the framework allowing the use of existing implementations with a small change (downcast to our new class). In the end, we need to define our fitness method, in order to decide how far from the solution a given individual is. The fitness formula is defined as the sum of the equal corresponding characters from the target and from the genome.

## Chapter 5

### Genetic Quantum Circuits Synthesis

“Current techniques for specifying quantum computations are an *ad hoc* mix of mathematical notation and prose narrative. These methods, however adequate for research reports, will not support the industrial-sized quantum algorithms hoped for in the near future. They are often imprecise; they cannot be directly simulated or compiled or submitted to automated analysis tools; and they will not scale up.” [79]

John Hayes and Igor Markov, as pointed out in reference [10], consider that Quantum Design Automation “is a necessary enabling factor in achieving scalable classical and quantum circuits”. The results from [10] represent the progress, at the middle of year 2006, in the quantum circuit synthesis, testing and simulation field, as the authors attempt to advocate the CAD (Computer Aided Design) techniques for quantum research. At the same time, reference [80] introduces the Genetic Programming as “a set of evolutionary computation techniques for getting computers to solve automatically problems without having to tell them explicitly how to do it”. This reference represents a tutorial for the genetic algorithms, presenting different approach styles and many applications. Taking into consideration the previous affirmations, we consider that the quantum logic circuit synthesis problem may be efficiently solved with genetic programming methods.

The quantum circuit synthesis has been intensely investigated in the last years, and researchers like Dmitri Maslov, Gerhard Dueck and Michael Miller have allocated years of effort for finding efficient synthesis methods [81][82][83][84][85][86][87][88][89][90][91]. In the quantum logic circuit simulation domain, it is considered that the research has already reached an unanimous level where the quantum circuits may be more efficiently simulated on classical computers [92][65][93][94][95][96][97]. Other appropriate references about quantum synthesis and/or simulation are available: [98][99][100][101][102][103][104][105][106][107][108].

A new methodology is presented in our paper [109], together with the corresponding software analysis that creates incentives for reversible quantum circuit synthesis. The focus in the presented methodology is on the layered phases, necessary for synthesis and optimization. We envision the described synthesis task as: perform the synthesis having a circuit description in a high-level description language, a set of unitary operators applied over the qubits and the collection hardware circuits (contained within a database) that will be used for synthesis.

A quantum circuit is composed of one or more quantum gates on which a set of unitary transformations are applied, according to a quantum algorithm. Designing a quantum circuit to implement a given function is not a straightforward task, because even if we know the target unitary transformation we do not know

## 70 | 5-Genetic Quantum Circuits Synthesis

---

how to compose it from primary transformations. Even if the circuit is somehow composed, we do not have information about its efficiency. This is the main reason why we are proposing a genetic algorithm approach for the synthesis task. The genetic algorithms will evolve a possible solution that will be evaluated against other previous solutions obtained, and in the end, the close-to-the-optimum solution will be indicated. In the circuit synthesis process, optimization plays an important role. We are proposing different methods for circuit optimization in reference [110], like topological, behavioral and pattern optimization.

Before applying a Genetic Algorithm for the Quantum Logic Circuit Synthesis problem, several decisions are to be made, in order to define all the necessary steps for quantum circuit synthesis. For example, is necessary to define the high-level description language that will be used to represent the quantum circuits in our system. The quantum circuit database is also necessary to define all the hardware circuits available for the synthesis. The quantum circuits have to be described together with their physical and logical characteristics. Finally yet importantly, the genetic algorithm parameters (number of generations, different probabilities, exit conditions, encoding, fitness evaluation, etc) and the performed self-adaption method are important for a successful quantum circuit synthesis.

In this chapter, the main software modules are presented. The Parser module presents the internal data structure used to translate the high description language into a more convenient representation. The Quantum Circuits Database facilitates quantum circuit description and adding of new quantum circuits by inheritance from a base class that abstracts a quantum circuit. Details about how the quantum synthesis algorithm is integrated into the ProGA Framework are exposed and explained in this chapter. Finally, yet importantly, details about the meta-heuristic algorithm used for parameter control adaption are also presented in the conclusion of this chapter.

### 5.1 Parser Module

The parser module implements all the necessary methods in order to read stream characters from an input file, and then creates the internal data structure that later will be used by other algorithm modules.

#### QHDL Presentation

Quantum Hardware Description Language [QHDL] was designed in order to extend the actual conventional VHDL (Very High Speed Integrated Circuit Hardware Description Language), thus allowing quantum design and classical design on different abstraction layers. The ATC-NY company is actually developing QHDL and its tool support, like parsing and type checking. "QHDL will provide both a shared language of interchange for the research community in quantum computing and a language for programming realistic applications" [79] [111]. The QHDL is intended as a tool that is able to specify an elegant and concise notation for the quantum algorithm. The project aim was to eliminate the imprecise and particular descriptions used by the many paper authors. The description language allows for

writing specification that is machine-readable, compiled and analyzed by using computer computation rules. QHDL also allows for describing complex quantum algorithms and, what is more important, provides a common language for all the involved researchers, by avoiding all the communication problems that were previously encountered (i.e. the notation mixing between narrative description and mathematical equation). It is considered that "a standard machine-readable notation for the full range of quantum algorithms is an essential element for building a wide range of automated tools and for allowing them to interoperate smoothly" [79]. Thus, the usage of a quantum language imposes correct syntax and type check when quantum algorithms are described. This language is also quoted in the QuIDDPro user guide [95], which is considered as actual state-of-the-art in quantum circuit simulators.

### **Generic Parser and QHDL Parser**

Taking under consideration all the abovely presented reasons, we decided to use the QHDL as our high-level quantum description language, mainly because the QHDL creates the common substrate for future quantum computing tools. The only available implementation related to QHDL was proposed by the National Institute of Standards and Technology [NIST] when QCSim [112] (Quantum Computation Simulator) was developed. The QCSim parser implements only a subset of QHDL, the one that is necessary to specify the quantum gates and the initial system state. We considered that this subset is relevant and it suffices for the quantum synthesis task that falls under the scope of our thesis. The proposed parser implementation was partially reused (thanks to the related source code and to the benefic email correspondence with Paul E. Black) for reading the qubit variables and the quantum gates used to describe the quantum algorithm from a text input file. We extended the number of recognized gates, but also removed some of them, in order to allow the description of quantum algorithms using the elementary gates presented in Table 1 and Table 2.

When discussing about the generic parser, we should think about a software tool that takes a large amount of data, mainly text data, and breaks it up into small pieces of data. In our case, the parser will read characters from the input file following several imposed grammar language rules. The lexical analyzer reads by file scanning characters and passes them to the syntax analyzer that recomposes the known language symbols. Last but not least, the semantic analyzer performs the type checking, thus trying to identify the possible errors (see Figure 5.1).

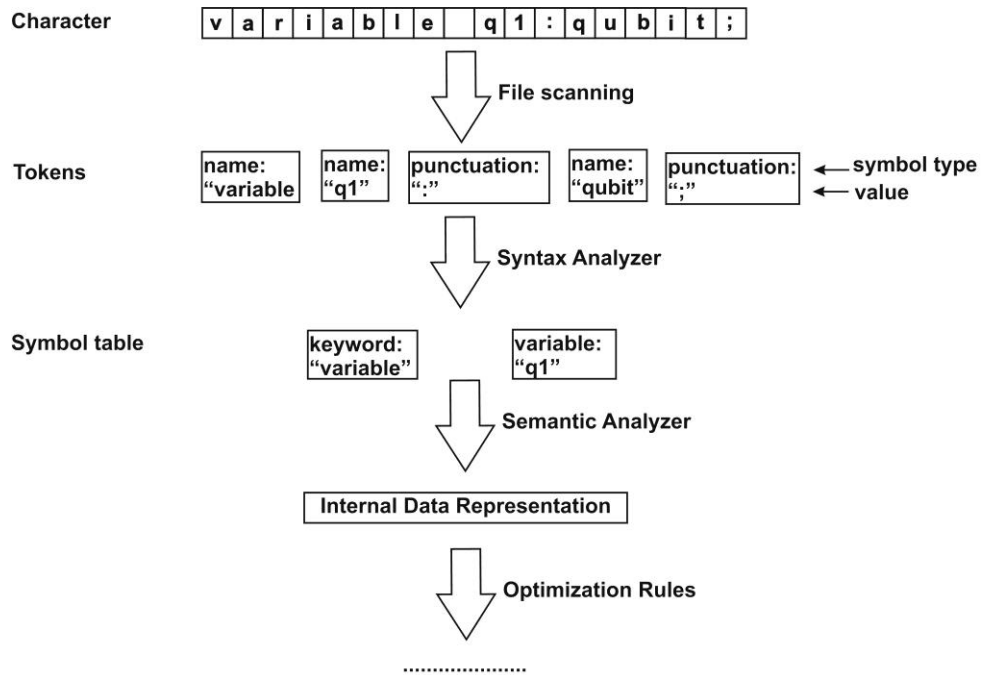


Figure 5.1: QHDL Parser Details

The generic parser is responsible with symbol detection, that may be a name (string of characters), a number (real or complex) or punctuation (i.e. "|", "#", ";", ","). The symbols are passed to the QHDL parser that will create the internal data structure, like in a two-pass compiler (see Figure 5.2). The drawback is represented by the memory consumption and by the time processor necessary for creating the internal data structure. On the other hand, the advantages are more important and dominant for our proposed synthesis task: better portability by cutting the relation between front end and back end, easy optimization to be performed on internal data, and if later will be necessary easy adaption of a new abstract layer between existing layers.

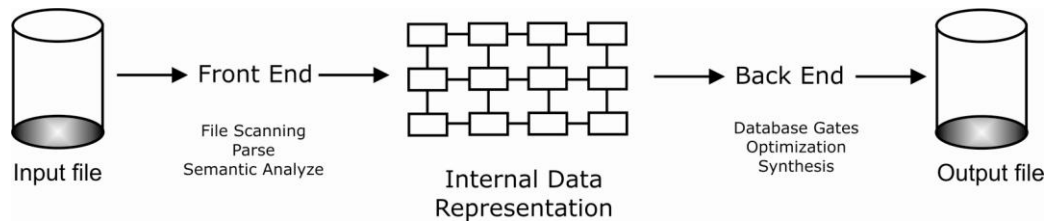


Figure 5.2: Description of the QCS Parser



The circuit is described in a text file, with the QHDL formalism. The QHDL provides a common understanding for all engineers because it is acting as a shared language for exchanging the research results. The semantic is easy, and - with its help - any kind of specific quantum algorithm can be specified.

The blocks of the input file, defining the file format, are:

- Variable block - specifies the variables (qubits) of the quantum algorithm. We expect to read, from the input file, the keyword "variable" followed by a space, then the identifier for the variable name followed by ":", and then the "qubit" keyword plus ";". Some validity checks are also performed in this phase (i.e. variable duplicate).
- Gate(s) block - is the place where the quantum algorithm is written, using the language semantic. We expect to read the identifier for the gate name, followed by the operator, which in our case is "(" and then the identifiers for the affected qubits, followed by the operators ")" and ";". For a quantum gate, it is important to know its name and the corresponding parameters. A validity gate check is performed (the known name and the expected number of parameters are equal with the read parameters). A second check for the parameters is performed in order to be sure that the parameters belong to the variable block part.
- Comment block(s) can appear anywhere in the file. Any character following the "#" character is considered a comment, and it is not important from the parse point of view. A comment line is ended with the line end.

When discussing architecture (see Figure 5.3), the QHDLParser is the main object from the parser module, its relation with the FileParser (our generic parser implementation) and with the ListQHDL being a strong one due to composition. Thus, the generic parser provides for the QHDLParser (via `s_GetSymbolRead` method) the symbols necessary to construct the internal data structure thru `_AppendQubit` and `_AppendFunction` methods. This class diagram map the system functional requirements to the objects involved in different processing steps (i.e. file opening by FileHandler).

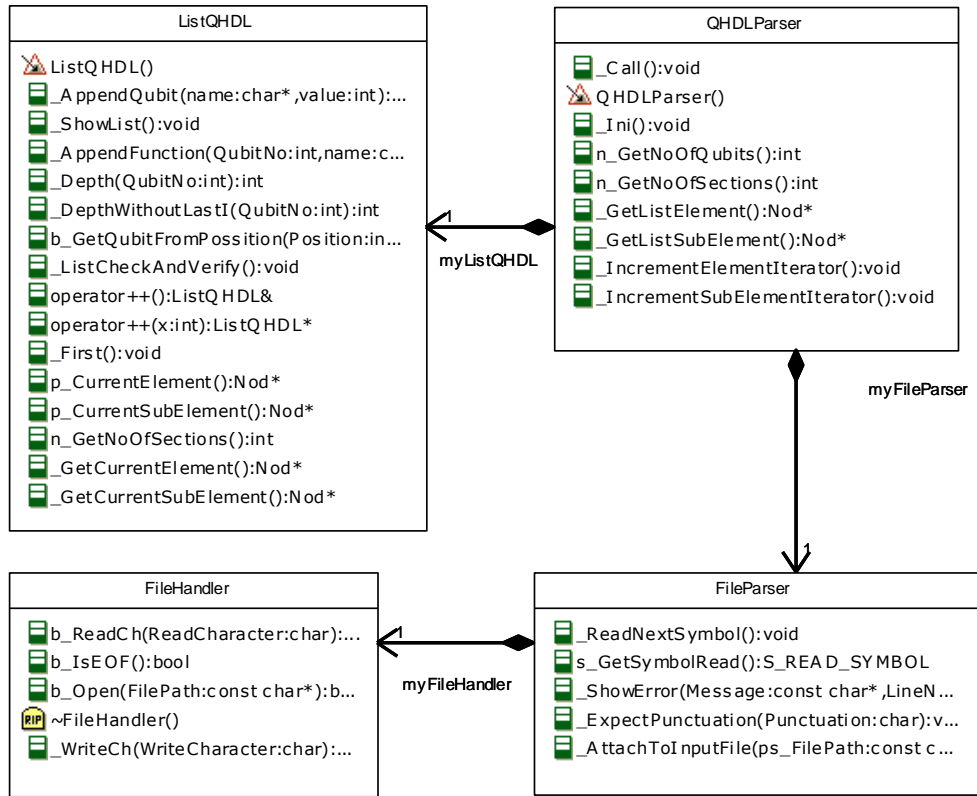


Figure 5.3: QHDL Parser Object Model Overview

The sequence diagram from Figure 5.4 illustrates the sequence of events in the parsing use case, by showing the objects interaction. The QHDL parser client is attached to an input file executing the `_AttachToInputFile` method and will scan the stream characters in order to find new symbols by calling the `_ReadNextSymbol` method. The `_ExpectPunctuation` and `_ShowError` are used by the client from its state machine to perform type checking. The main output from the FileParser is the detected symbol that will be provided via `s_GetSymbolRead`.

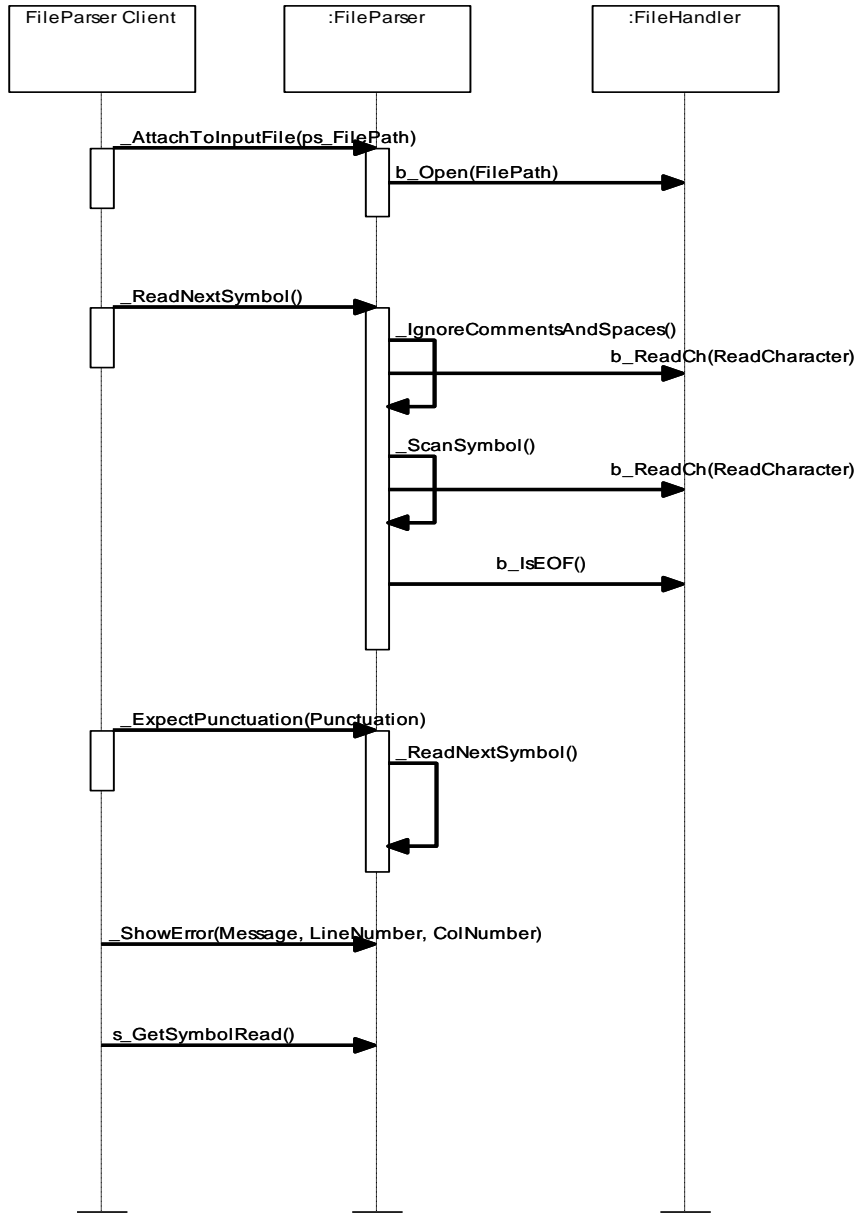
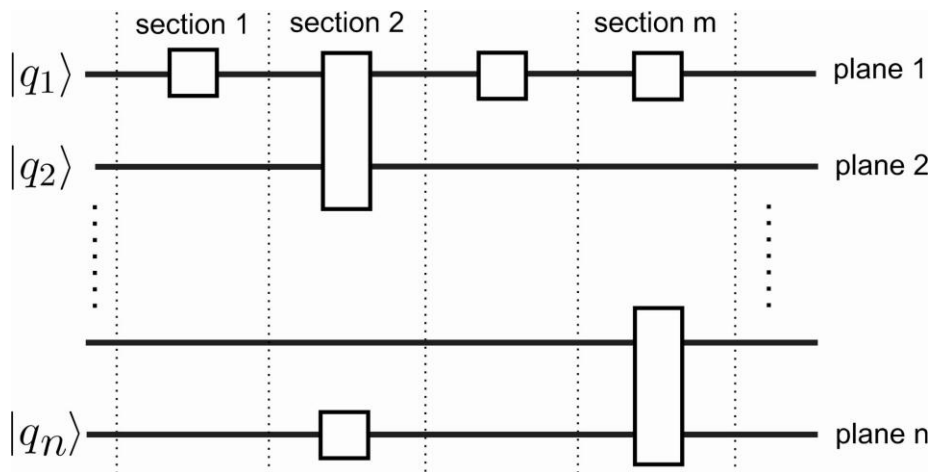


Figure 5.4: Generic Parser Sequence Diagram

The adopted granularity level (as high-level) presents only the public operations that are carried out, and which are the messages sent during the time required to implement a generic parser module.

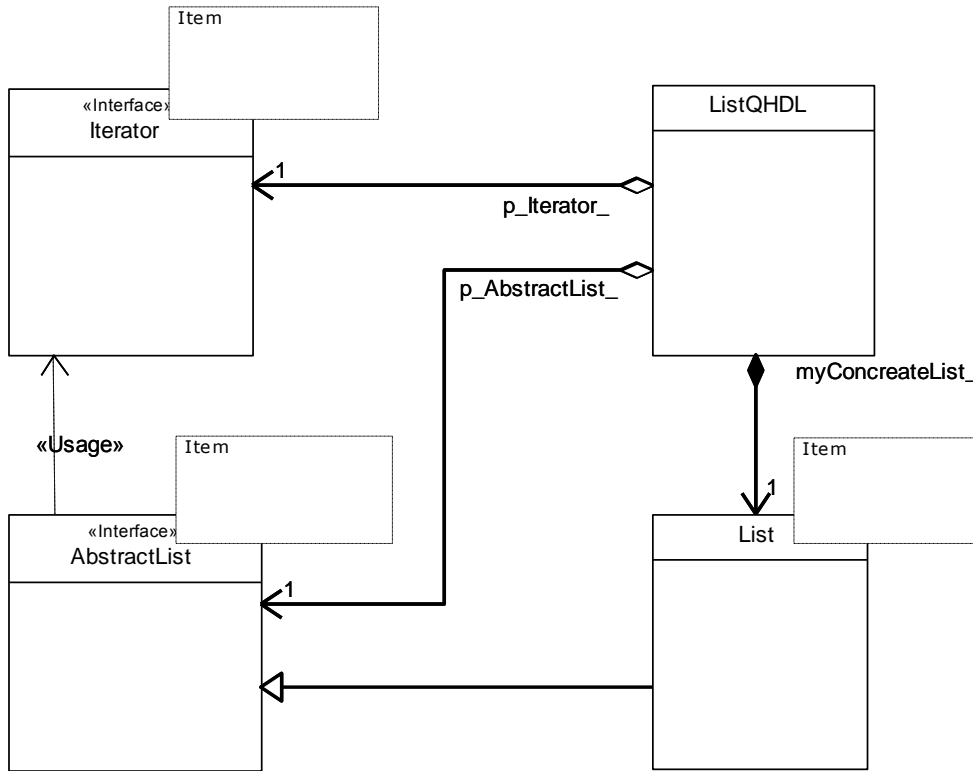
### Internal Data Structure

The representation used for keeping the internal data structure was chosen to be a list-in-a-list. The reason behind this choice is our new view on the quantum circuit: the splitting in sections and plains [113]. This splitting allows access to sections elements, which is later used for optimization, and creates the prerequisites for the circuit encoding, which is then used by the genetic algorithm.



**Figure 5.5: Quantum Circuit View as List-in-a-List Structure**

The used high-level granularity from Figure 5.6, tries to emphasize only the object relations, our focus being here on how the Iterator pattern is used to implement this particular type of list. The List class is derived from the AbstractList interface, thus will implement the methods for working with list elements, while `p_Iterator_` pointer is used to iterate the list. Additional data members are used to memorize the iterator position and to allow vertical and horizontal iterations.



**Figure 5.6: Internal Data Structure Object Model Overview**

The sequence diagram from Figure 5.7 presents the method calls, in a time sequence, which is necessary for parsing a QHDL input file. First, during the system initialization, the QHDLParser is attached to the specified input file. The next main call will read the declaration part from the file, where the qubits variables are defined, and will create the horizontal lines from the internal data structure by the `_AppendQubit` method. The rest of the calls, until the end of the input file, are used to detect the involved gates and to fill the internal data structure with the corresponding gate nodes, by using the `_AppendFunction` method.

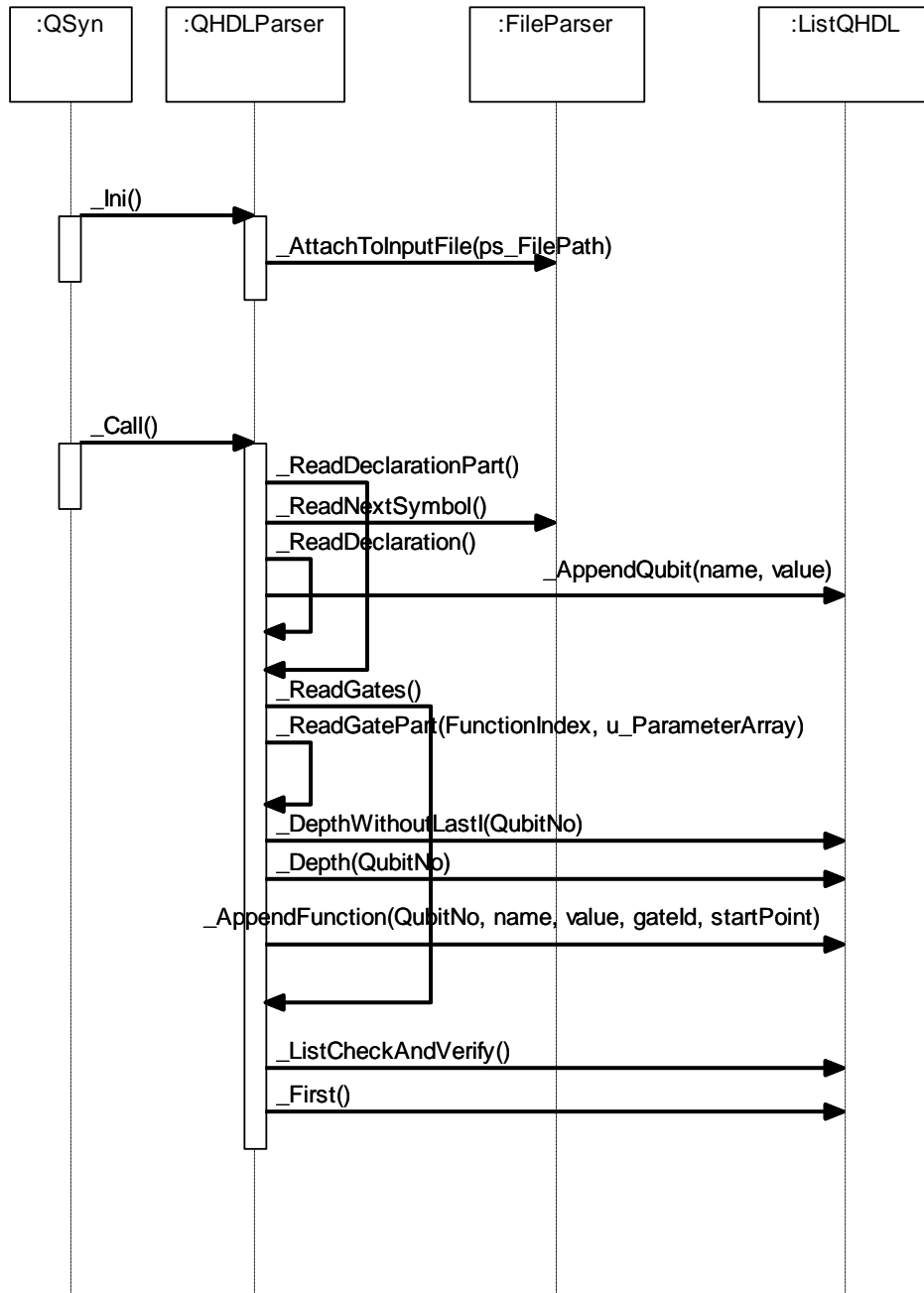


Figure 5.7: QHDL Parser Object Model Overview

The created ListQHDL object is used as a comparison object, with the evolved output emerged by the genetic algorithm. On one side, we have the described quantum circuit represented by the internal structure and, on the other side, there is the best evolved chromosome. In case of equality, the evolved solution may be considered as a solution (not necessarily the optimum one).

Each object from the ListQHDL has the type of Nod, wherein Nod objects declare the pointers necessary to create the horizontal and vertical relations. Others data are used from the composite object Locus, such as gate name, gate identifier, number of gates, etc (see Figure 5.8). Public methods are also present in the class diagram in order to provide get access on members.

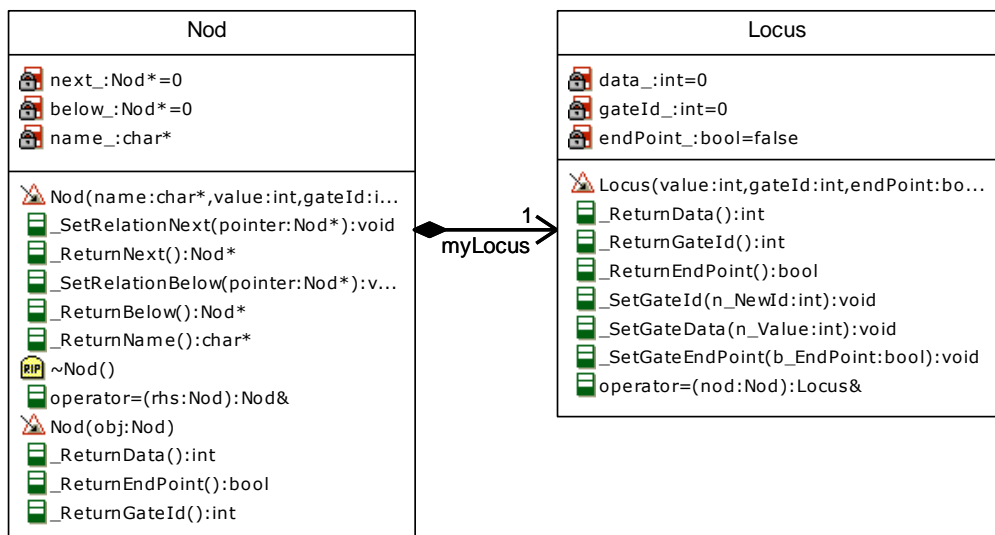


Figure 5.8: Nod Object Model Overview

We needed to distinguish between Nod and Locus because not all the class members from Nod are necessary in order to define the chromosome positions (also known as locus). The reason behind this decision is simple; we need to have a small object, in terms of memory consumption, in order to decrease the computation time of the genetic algorithm evolution. At the chromosome level, it is not important to store the gate name, because it suffices to know its identifier. The replacing of id's with a name, for showing a readable solution, is still possible since its name may be obtained from the QHDLParser class (it is defined static there).

**Optimization**

It is important to optimize the internal data, therefore enhancing the circuit quality before proceeding with the synthesis task. It is considered that synthesis without optimization will generate uncompetitive circuits. It is also important to maximize

## 80 | 5-Genetic Quantum Circuits Synthesis

---

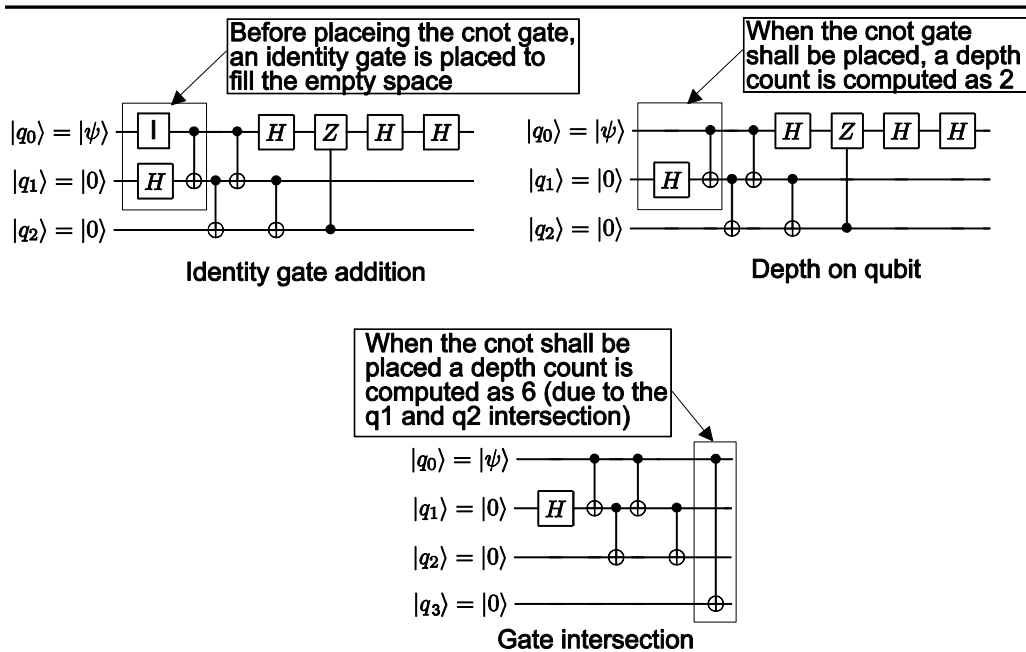
the circuit quality, a situation that will generate superior performance for the synthesized circuit. Optimization implies analyzing the circuit from a structural view and from a behavioral view. Another factor in assuring the quality of a circuit is the computation time (each gate introduces a specified delay). Our analysis will consider all the factors that may influence the quality of a quantum circuit design. Optimization will preserve the circuit output function, while decreasing the total number of used gates. Optimization is an iterative process that shall be applied several times; one optimization being able to create incentives for a second one.

The full circuit optimization is hard to obtain. Optimization may occur on different circuit parts and may have many distinct goals. In classic synthesis, optimizations on the circuit integration area and delay latency are common, and sometimes the optimization is based only on approximations. The structural view allows for interpreting the circuit as a sequence of gates, which are placed on quantum wires (see Figure 5.5). At this stage, it is important to have a clear structure that will be used in computing the quantum state on each section.

An enumeration of the rules that need to be followed for implementing the topological optimization [110] is presented below:

- Rule 1, order of placement: the gates are placed in the order described within the input file.
- Rule 2, identity gate(s) addition: if there are empty sections in front of the placement section, they will be filled with identity gates. In this way, we assure a complete internal data representation, which is later necessary for computing the output function.
- Rule 3, qubit depth: if the gate affects more than one qubit, it shall be placed on the section defined by the highest depth on the composed qubits.
- Rule 4, gate intersection: if there are other gates intersecting with the gate that needs to be placed in the section defined by the previous rule, then a new depth (which takes into consideration also the intersected qubits) is calculated. The gate will be placed on the section defined by this new depth.





**Figure 5.9: Topological Optimization**

The structural view is created when the circuit description is read from the input file, each gate being placed on the corresponding qubits when the file is parsed. The placement algorithm is responsible with the topological optimization, by defining the gate placement methodology (see Figure 5.9).

### Iterator Pattern

An abstract list class is defined, in order to provide a common interface to access and manipulate a particular list structure. In the same way, the abstract iterator provides a common iteration interface. The pattern was necessary because we need to have more list structures in our architecture, and for each of them different iterations is necessary. The list object is responsible for creating the corresponding iterator; the client will request an iterator object via the CreateIterator method (almost an example from the Factory Pattern [66]).

Using this pattern, we defined several concrete lists, which can memorize the intermediate results of the synthesis algorithm. For example, a first list will keep information about the circuit as it is described in the input file, and a next list will follow the optimized circuit pursuant to the optimization phase.

The abstract class Iterator provides an interface for any kind of list iteration. The methods are abstract and need to be redefined in the concrete ListIterator. The abstract class allows for an easy implementation for any kind of iterations that have to be performed on the concrete list. Template usage allows different object types to be iterated. Without an iteration class, all the details related to list traverse need to

be stored in the implemented class list (a negative effect: the class will become bushy).

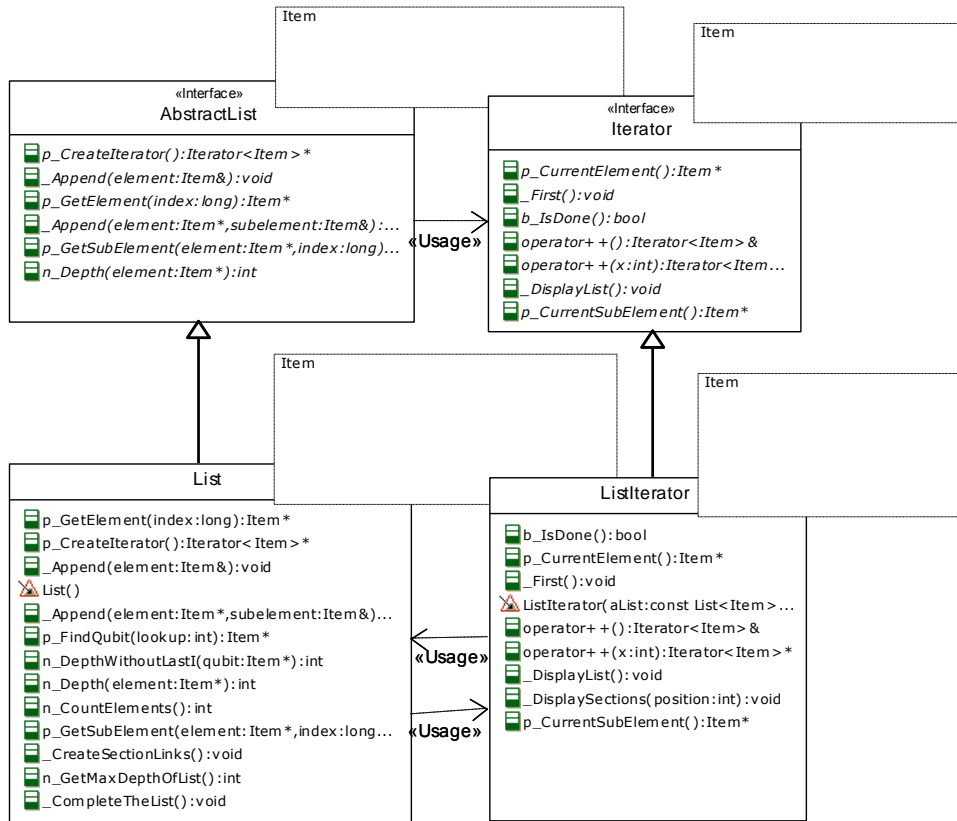


Figure 5.10: Iterator Pattern Object Model Overview

Figure 5.10 shows how the iterator pattern is implemented into the architecture. The interface allows for insertion and removal of elements for any kind of lists. Additional methods are presented for finding or for checking the depth of an added element.

## 5.2 Quantum Circuits Database

For the purpose of synthesis, it is important to create and update a database with quantum circuit specifications. The synthesis will be performed only by using circuits from that database (see Figure 5.11). The quantum circuits are described from behavioral and physical points of view. The behavior is important in order to verify if the circuit satisfies the logic-nature expectations, while it implements the required function; the physical characteristics are also important for defining the circuit

layout (i.e. creating a circuit layout was not under our focus throughout this thesis, but the database may be easily extended to allow this propriety).

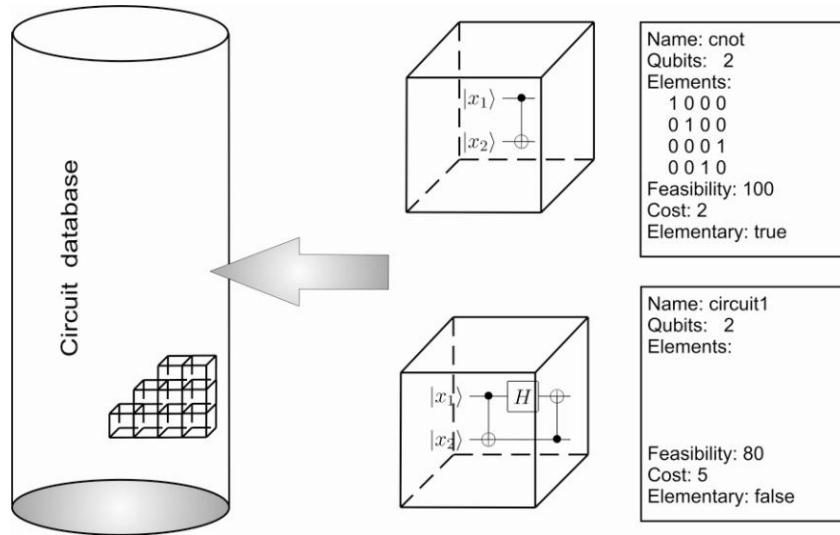


Figure 5.11: Database for Quantum Circuits

The Database was designed to allow extensibility. Thus, the module database contains three essential classes (see Figure 5.12). The QuantumGate class is used to describe a quantum circuit and its proprieties, while the QMath class implements the necessary corresponding operations. The GateCollection class is just a container for these gates, which are used for the purpose of synthesis.

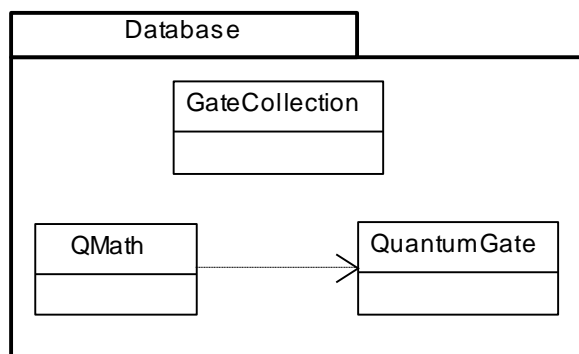


Figure 5.12: Database Component Diagram

## 84 | 5-Genetic Quantum Circuits Synthesis

The database contains elementary gate circuits and arrays of circuits (composed by elementary gate circuits). The elementary gates are used to synthesize the circuit function, in order to find an optimal circuit composed only of elementary gates. The complex circuits are important in the optimization phase, the previously obtained circuit being changed so that it will also contain complex circuits; this optimization is required to reduce the circuit cost, by using a small number of capsules.

### Quantum Gates

A quantum gate is described by using the matrix representation. In order to do that, we used the STL<sup>13</sup> vector template. We defined the quantum gate as a double dimension vector of complex elements. This kind of structure allows facile access to the gate elements and a low consumption of execution time. The class members define the quantum gate proprieties such as number of input qubits, the gate cost and feasibility (see appendix 8.5 Quantum Gates Cost), the gate name, and if the gate is elementary or complex. The constructor allows the creation and, at the same time, the initialization of gate elements with the corresponding values. In fact, the constructor will create the object and the private methods, which are called within the constructor (i.e. `_MakeHadamard` will fill the matrix values according to the Hadamard logic function) will initialize its elements.

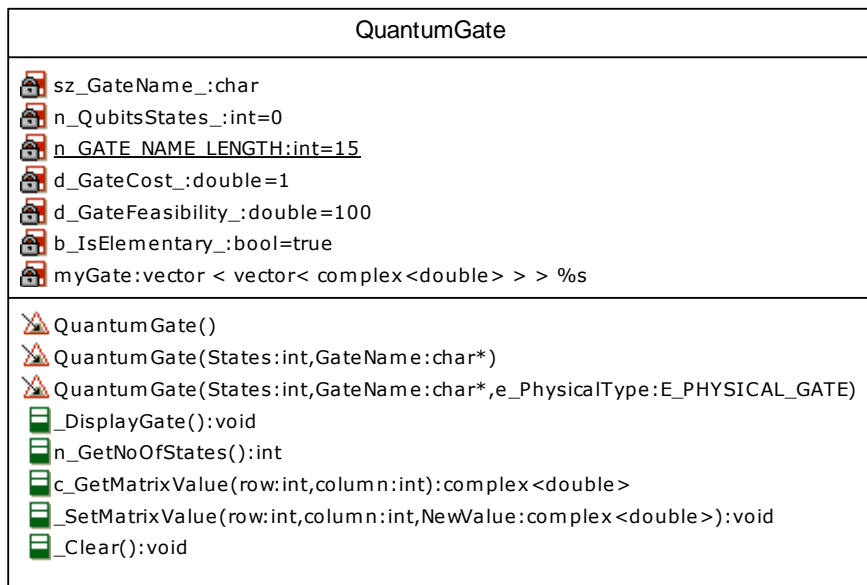


Figure 5.13: Quantum Gate Class Diagram

<sup>13</sup> Standard Template Library

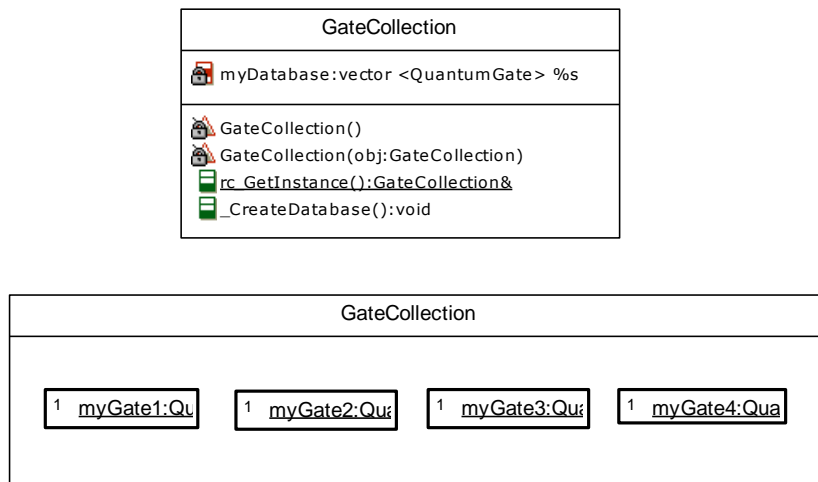
Discussing extensibility, just by adding a derived class from the QuantumGate and, inside it, a new private method it is enough to define a new quantum gate into the system.

**Gate’s Math**

This class is responsible with quantum gate math’s implementation. Methods necessary for multiplying, compute the tensor product or just for the equality degree of two quantum gates are implemented in this class.

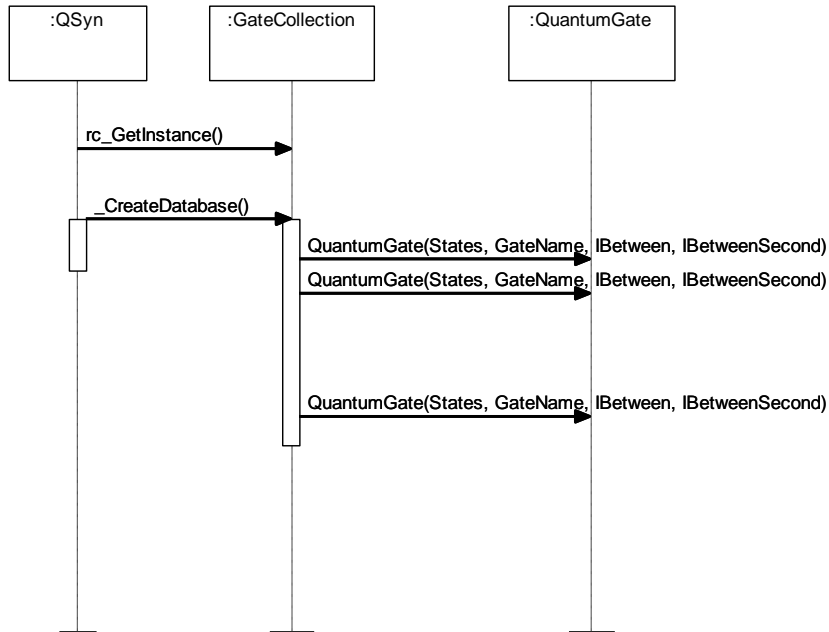
**Gate Collection**

The gate collection class was designed as a container of quantum gates. The gates are declared as derived QuantumGate objects and defined within the collection, as composite objects into a STL vector object. The class was implemented by using the Meyers Singleton pattern, thus only one of its instances is allowed in the system. The access to the gates is allowed through get methods. The main characteristic of this class is that the gate particularities are not important at this level, as opposed to their presence. In Figure 5.14 the specification and structured views presents the GateCollection class.



**Figure 5.14: Collection Class for Quantum Gates**

It is easy to extend the collection with a new gate object, by creating a new gate in the GateCollection class as a composite object (i.e. myGate5) and adding the object in the GateCollection constructor into the vector.



**Figure 5.15: Creating Collection Sequence Diagram**

The abovely presented Figure 5.15 presents the necessary operations during the time line for the gate collection creation. For the first time, the static GateCollection object is created and then - via corresponding creators calls (constructors) - the quantum gates are created and added into the STL vector object.

### 5.3 Preparation Steps for Genetic Algorithm

A dedicated genetic algorithm is used to emerge a circuit synthesis solution. The obtained solution is not guaranteed as being the best, thus giving reasons for tuning of the algorithm. The most important parameters required by the genetic algorithm shall be interactively set by the user when the algorithm is initialized (for example: the number of individuals, the number of generations, the mutation/crossover percentage, the accepted number of circuits in the evolved result, etc), and dynamically adjusted by the meta-heuristic algorithm.

Within our proposal, it is required that several decisions be taken before going into genetic algorithm details (also called preparation steps):

- “What is the *terminal set*?”

### 5.3 -Preparation Steps for Genetic Algorithm|87

---

- What is the *function set*?
- What is the *fitness measure*?
- What *parameters* will be used for controlling the run?
- What will be the *termination criterion*, and what will be designated the *result of the run*?" [114]

#### **Objective**

The objective here is to find the efficient reversible quantum circuit synthesis from a high-level description. Considering the discrete space search  $X$  and the objective function  $f : X \rightarrow \mathfrak{R}$ , our scope is to find the  $\max_{x \in X}(f)$ , where  $x$  is a vector of decision variables  $f(x) = f(x_1, x_2, \dots, x_n)$ . It is a maximization problem, because we try to find the optimum quantum circuit that implements a given input function.

#### **Terminal Set**

The terminal set used in the quantum logic circuit synthesis process is composed from the following components:

- Quantum gates (any gate from the database may be randomly used for the chromosome encoding).
- Implemented methods to generate random numbers (used in the selector probabilities and in the gate selector when genetic operators are applied)
- Constant gate characteristic values (i.e. quantum circuit cost and its efficiency).

#### **Function Set**

The function set for a genetic algorithm is derived from the nature of problem. For the quantum synthesis problem, the function set is composed of the mathematical functions necessary to evaluate the circuit output function (tensor product, multiplication, equality, etc.). A function set needs to fulfill two proprieties:

- Closure: it is assured because any function from the function set can accept as arguments any value returned by any other function from the function set. In our case, the return result from any function is a circuit gate, which is a terminal in the terminal set. In other words, any function from the function set is well defined and closed for any argument combination that it may receive.
- Sufficiency: it is assured by theory, because it is possible to express a solution by hand computation, combining the elements of the primitive set.

### **Fitness Measure**

The fitness measure specifies what the user expects from the synthesis algorithm. Therefore, the fitness assignments to a chromosome indicate how close the individual output is to the algorithm target. A good method not to repeat the fitness calculation for a chromosome is to maintain a flag showing if any change has been performed since the last evaluation. Any genetic operator applied over an individual will affect the evaluation flag, hence allowing a new fitness re-evaluation in that case.

### **Selection**

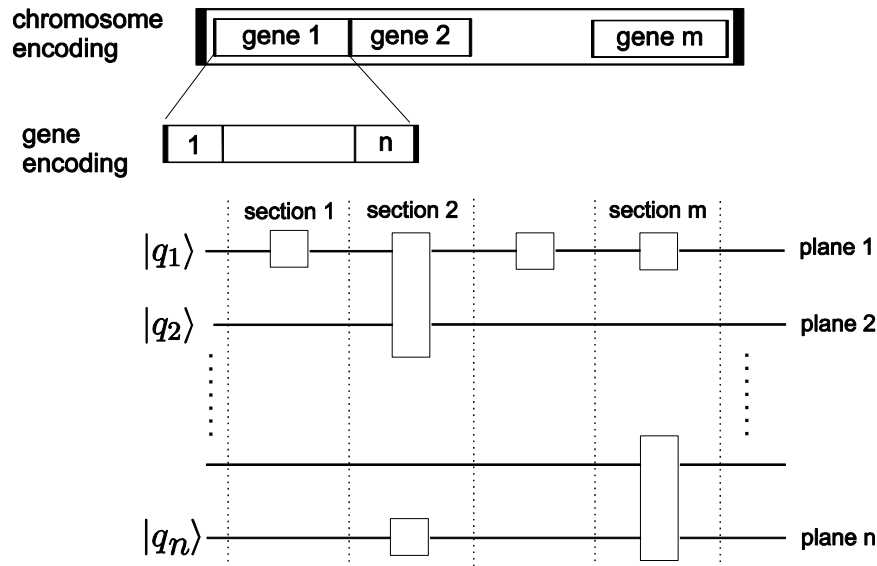
Different types of selector methods are used for quantum circuit synthesis (i.e. rank, roulette wheel, tournament and uniform). The possibility to change the selection is important in order to compare different evolved solutions. The ProGA framework provides support for selection methods, the user having the possibility of choosing one of them, or even of dynamically changing the selector during run-time.

### **Encoding**

The circuit representation is important for the chromosome encoding. A chromosome shall represent a possible candidate solution for the given problem. Our approach was to split the circuit representation in sections and plains [113] [109], a representation that will be used in the chromosome definition.

Following Nature, where a chromosome is composed of genes, in our chromosome the genes are the circuit sections. In this way, we are able to encode the circuit within the chromosome, and to represent a possible candidate solution (as presented in Figure 5.16). A gene will store the specific characteristic of a particular section and the genetic operators will be applied at the gene level or inside the gene.





**Figure 5.16: Chromosome Encoding**

The possibility of applying the genetic operators at the gene level is promoted here for the pattern optimization. For example, replacing a gene with another one has its correspondent in the circuit section replacement (a circuit section may be replaced by another, having the same functionality but with a better efficiency/cost). The patterns will be externally stored and used in the pattern replacement. Moreover, it is possible to exchange more related sections with an equivalent section (which is, for example, more attractive from the efficiency point of view).

The parser will provide information for the genetic algorithm, about the number of qubits ( $n$ ) and about the number of sections ( $m$ ). These values are taken from the internal data representation, after applying the topological optimization. At the beginning, the chromosome will be composed of  $m$  genes and each gene by  $n$  gates, because our scope is to optimize the evolved circuit and not to have an increase number of gates in comparison with the given (HDL described) circuit. Later, due to the evolution process, and due to pattern optimization, it may be possible to reduce the number of used genes by compaction of the adjoining genes, or by removing a gene if the added characteristic does not produce a transformation for the previous one. This is made possible by using genes composed only of identity gates, the removed gene being placed at the end of the chromosome (it is important not to change the chromosome length).

Each chromosome represents a possible solution, which is computed by applying the tensor product for all gate genes and then multiplying all the results.

### **Initial Population**

The initial population is created using random quantum gates inside the chromosome genes. The number of individuals may be set by the user, when the genetic algorithm is started.

### **Controlling Parameters**

It is impossible to make general recommendations for specifying optimal run-time parameter values. For this reason, a meta-heuristic algorithm is used in order to determine the controlling parameters as operator probabilities. Other parameter controls, as population size and fitness function, have static values, and the dynamical adjustability was not necessary during the algorithm evolution.

### **Evaluation**

The genetic algorithm will start with random candidates, having the chromosome length and the number of genes defined by the parser and by the topological optimization. The chromosome values are randomly generated at the beginning, having random gates inside. The created population is evaluated in order to check if there is an acceptable solution. This check is made by comparing the chromosome output function with the given circuit function and by applying the penalty value, if such it is the case. The genetic operators are applied afterwards, allowing population evolution by generating a new population, (the type of genetic algorithm is non-overlapping). We have also proposed to use elitism for our algorithm (a specified number from the best chromosomes are kept during the genetic evolution, allowing an increased convergence for the algorithm). The scope is to maximize the fitness values, thus any value higher than 1 is considered to be an acceptable solution; this acceptable solution will have the same output function as the given circuit and, in addition to this, the number of composing gates will be lower).

### **Termination Criteria**

The genetic algorithm is finished when the number of allowed generations is reached. Other methods for stopping the algorithm are not used, because our scope is to maximize the fitness function using all the possible number of generations (i.e. the possibility of stopping the algorithm when there is no improved solution evolved during several generations, or when the fitness is sufficient for deciding that the algorithm has already evolved a better solution, etc.). The maximum number of allowed generations is a parameter control for the synthesis algorithm and may be set by the users when the genetic algorithm is started.

## **5.4 Integration within ProGA Framework**

Four main blocks are identified within Figure 5.17. The ProGA framework is responsible with genetic algorithm implementation details, and the Adaptive component is responsible with the dynamically adjustment of the algorithm parameter controls. The Synthesis component is responsible with the quantum

## 5.4 -Integration within ProGA Framework|91

synthesis process, implementing the particularities necessary for the genetic evolution. The fourth module, the Database, represents a collection of gates that are used in the synthesis process, random gates being introduced into genetic population as locus into chromosomes.

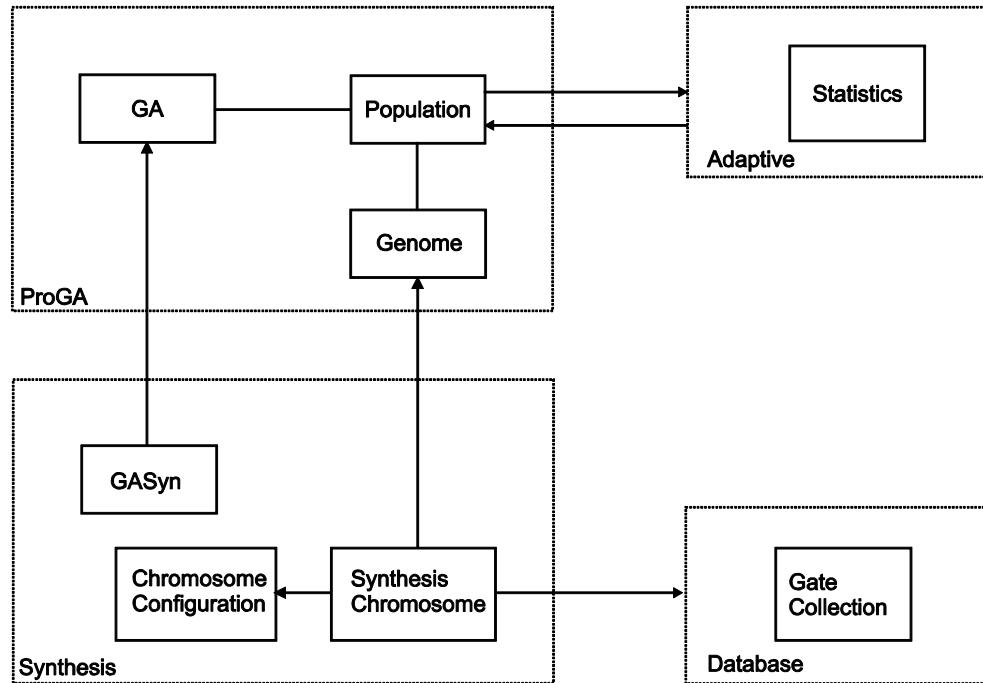


Figure 5.17: Integration within ProGA Framework

The ProGA framework allows for testing different configurations and comparing their results. It is possible to have different values for the parameters control, different type of selectors, different random number generators, different time measures implementations, etc.

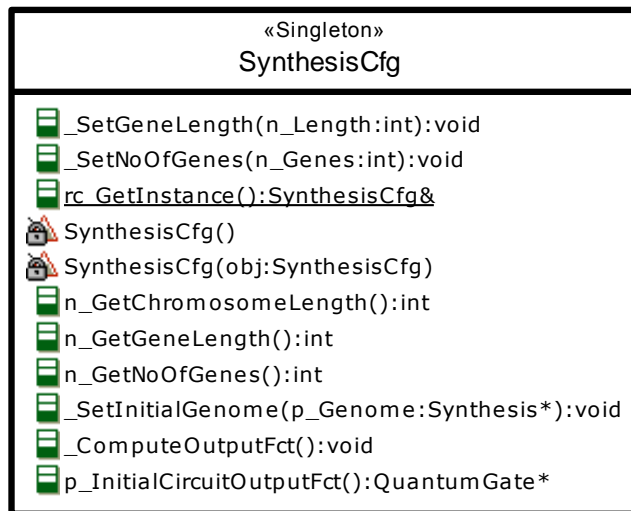
### 5.4.1 Initial Circuit Configuration

The first information that is necessary in the synthesis configuration is related to the *number of genes* from a chromosome, and the *length for a gene*. These values have their correspondent in the number of sections and planes (like in Figure 5.5) that are computed during the parsing operation. Then, in a second step, the initial chromosome is constructed from the internal data structure, using the Locus information instead of Nod (see Figure 5.8). Thus, we have encoded the input high-level circuit description into a synthesis chromosome, that will be used as a comparison base for all the evolved results (remember, that our scope is to evolve a

## 92 | 5-Genetic Quantum Circuits Synthesis

---

better chromosome). In order to allow a facile comparison, the output function result is saved into a quantum circuit object (called `myInitialFctOutput`, as it is described in Figure 5.19). This quantum circuit will not be included into the Quantum Gate Collection, because it will not be used as input for the synthesis algorithm.



**Figure 5.18: Genome Configuration**

The algorithm implemented in the Application class, responsible with the creation of the initial solution is:

---

Algorithm used for transformation of the internal data structure (created by parser) into the initial chromosome

---

1. Create an initial chromosome.
2. Get the number of qubits (known by parser module).
3. While element exists into Internal Data Structure
  - a. Copy all sub-elements into the current gene.
  - b. Go to next element from the Internal Data Structure.
4. Compute the output function and memorize the result into a quantum gate circuit.

---

The configuration class is required in order to transform the parsing information into chromosome and to give facile access to the initial solution. Details about code implementation are given in 8.2 (QCS Initial Genome Solution) section.

### 5.4.2 Synthesis Genetic Algorithm

The Synthesis Genetic Algorithm is considered as an extension for the ProGA framework, the classes belonging to this component being derived from the interface declared in ProGA. Thus, as presented in Figure 5.19, GASyn is derived from the interface class GA and Synthesis is derived from the interface class Genome. By this inheritance, the ProGA framework functionality is extended with the synthesis behavior.

---

#### Algorithm used for the genetic synthesis

---

1. Set the genetic algorithm type: overlapping or non-overlapping.
  2. Set the mutation and the crossover probabilities.
  3. Set the elitism percentage.
  4. Set the maximum number of allowed generations and individuals.
  5. Get the number of qubits (known by the parser module).
  6. While generations are still allowed or a solution has not been evolved:
    - a. Initiate the population.
    - b. Evaluate the individuals.
    - c. Store the statistic information.
    - d. Check for a solution.
    - e. Apply the genetic operators (mutation and crossover).
  7. Display the best chromosome.
-

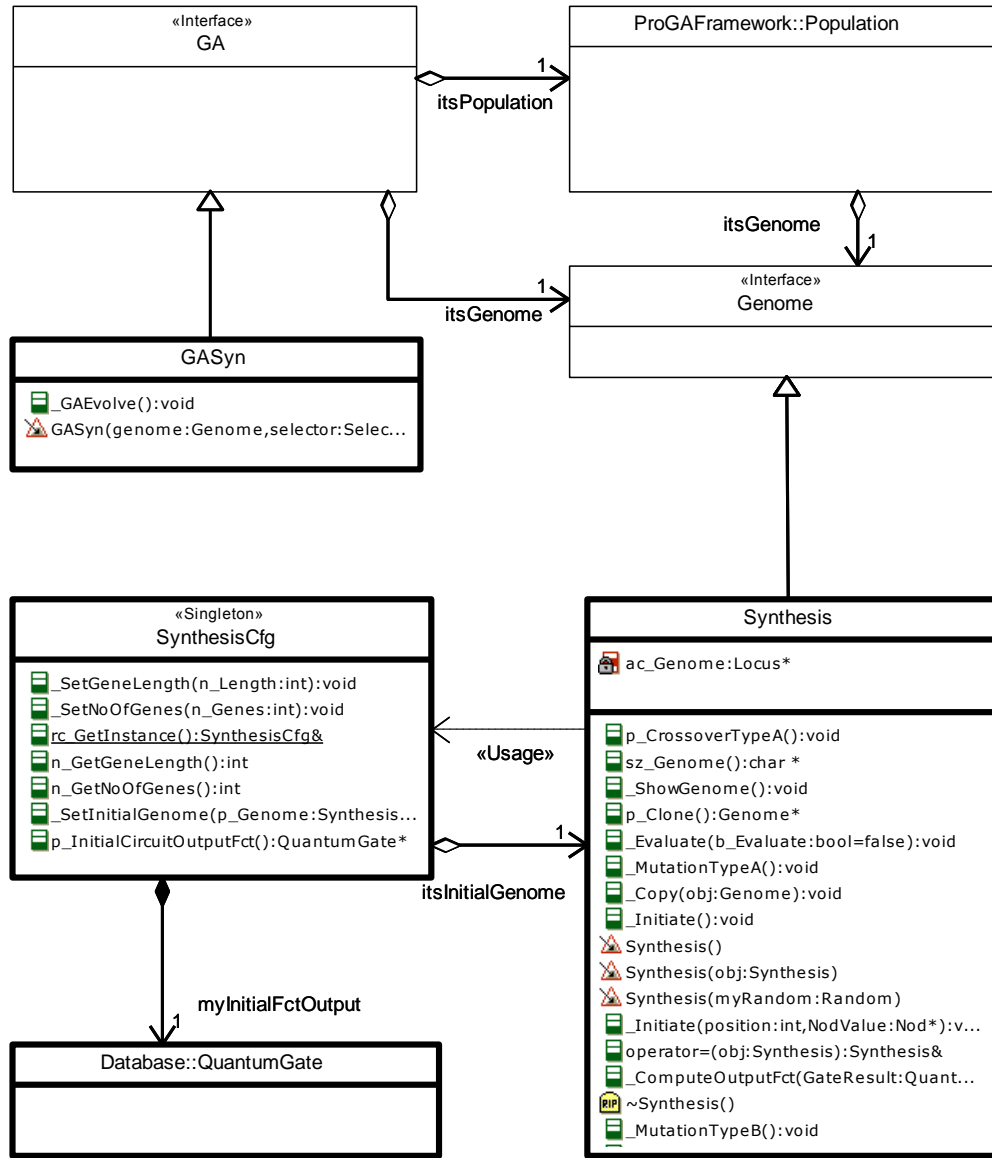


Figure 5.19: Synthesis Genetic Algorithm

For the user, the inheritance from the ProGA framework gives access to all the genetic algorithm parameters control. For example, the genetic algorithm type, the operator probabilities, the number of individuals can be adjusted within this GASyn class. More than this, in this class, the order of genetic operators or the termination criteria may also be changed. This class defines all the algorithm parameters control, all the operation executed, and – at the same time - their order. This class

was design to have only base information about the genome, selector and population type, which are used for the genetic algorithm declaration.

### 5.4.3 Circuit Genome

The Synthesis class implements the genome, which encodes the circuit representation. As already stated, a genome could represent a possible solution for the synthesis task. The genome representation is an array of random quantum gates that are chosen randomly from the database, with the only constraint that a quantum gate cannot be split on two genes. Thus, the genome (or the chromosome) is composed of one or more genes, their number being known by the parser module and provided to the Synthesis class via SynthesisCfg class.

This class implements the main functionality of the synthesis task, the operators and their behavior being defined at this level. Another important method is responsible with the random gate initialization for the chromosomes. We defined two types of operators, called A and B. The A type is responsible with changes inside of the gene, while the B type is responsible with changes at the chromosome level (thus, affecting one or more genes). The initial probability, defined by user, is split in two parts and assigned to the A and B operators. Later, during the run-time, their values will be automatically adjusted by the meta-heuristic algorithm (if it is configured to perform in this manner). The object model overview for the Synthesis class is presented in Figure 5.20. The genetic operators are applied at the gene level and inside the gene. When applied at the gene level, an entire gene is replaced with the content derived from the genetic operator. When applied inside of the gene, the gene content is only modified.


























Synthesis	
	ac_Genome:Locus*
	n_GeneIndex:int=0
	b_NewSection:bool=false
	sz_Genome():char *
	_ShowGenome():void
	p_Clone():Genome*
	_Evaluate(b_Evaluate:bool=false):void
	_Copy(obj:Genome):void
	_Initiate():void
	Synthesis()
	Synthesis(obj:Synthesis)
	Synthesis(myRandom:Random)
	_Initiate(position:int,NodValue:Nod*):void
	operator=(obj:Synthesis):Synthesis&
	d_fitness():double
	rc_DetectNextGate():QuantumGate&
	b_CheckNewSection():bool
	b_CheckValidIndex():bool
	_ComputeOutputFct(GateResult:QuantumGate):void
	~Synthesis()
	_InitiateBetween(n_StartPosition:int,n_EndPosition:i...
	_MutationAType():void
	_MutationBType():void
	p_CrossoverAType(spouse:Genome):Genome*
	p_CrossoverBType(spouse:Genome):Genome*

Figure 5.20: Synthesis Genome

#### 5.4.4 Circuit Output Function

The genome output function is computed by the `_ComputeOutputFct` method that will return the result as a quantum gate result, which is later used for comparison with the initial genome function. Information is exchanged from left to right, with the upper wires representing the most significant qubits (as presented in Figure 5.16). Each individual represents a possible solution, which is computed by applying the tensor product individually over all horizontal rows, and then multiplying all the row results.



---

Algorithm used for the genome output function

---

1. While gene index is less than chromosome length
    - a. Read the current quantum gate.
    - b. If the read operation has changed the section and the first gene was read
      - i. Only the first section was read.
      - ii. Reinitialize the Locus position.
    - c. If the read operation has changed the section
      - i. Reinitialize the locus position.
      - ii. Perform multiplication.
      - iii. Save the multiplication result.
    - d. If it is the first locus
      - i. Save gate value under the form of tensor result.
    - e. If it is not the first locus
      - i. Perform the tensor multiplication.
  2. Perform the last multiplication.
  3. Return the output function result.
- 

Details about algorithm code implementation are presented in the 8.4 (QCS Genome Implementation Details) section.

### 5.4.5 Genome Initialization

On the initialization phase, the genome will receive random gates, as values from the quantum circuit database. The genome attributes are reset to the initial values and the evaluation flag is cleared in order to force the evaluation on the next execution cycle. The initialization is performed only once during the algorithm lifetime, and is responsible with the genome creation (as presented in Figure 5.21).

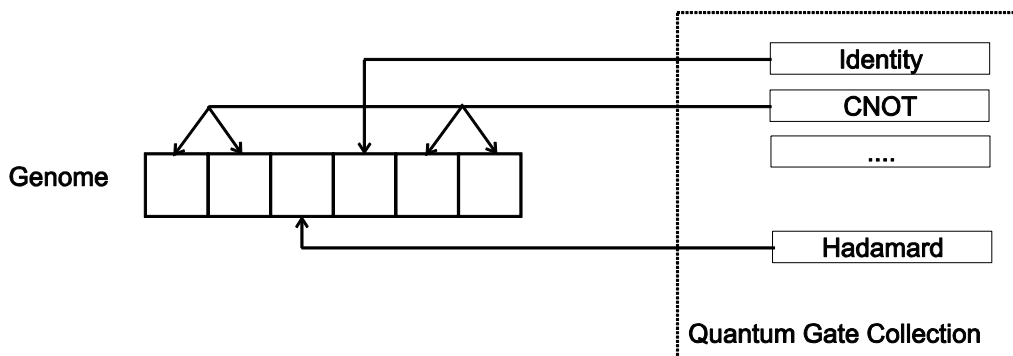


Figure 5.21: Genome Initialization

One of the design constraints needs to be respected, as being related to the impossibility of splitting one quantum gate between two genes (or sections). For example, if the gene has a length equal to 3, is possible - at the initialization phase - to receive a gate with 3 inputs, or a gate with 2 inputs plus one gate with one input, or a gate with one input plus the following with two inputs, or just 3 gates with 1 input. The initialization algorithm is repeated until the genome receives the initial quantum gates with the constraint being that the maximum allowed number of inputs is given by the gene length.

### 5.4.6 Detecting of the Next Gate

The most sophisticated part of synthesis class is the gate detection from the genome. It is important to be able to read the quantum gates one by one from the encoded information, and to provide the quantum gate type read. This method is essential when the output function is computed, and it is essential in the genetic operators because the quantum gates are affected by changes there. The complexity is given by the locus-encoding mode, being possible to split one physical gate on several locus positions. One encoding improvement is coming from the parsing design, because it is possible to have a quantum gate, located only within one gene without being spread over several genes. If the next read gate is a singular qubit gate, then its reference is returned in a fast manner by the `rc_DetectNextGate` method, and the index from the chromosome is incremented with one (allowing reading the next possible gate from that new position). When the next gate read affects more qubits, it is necessary to check all the other possible types of gates that will have as first input the gate described by the current locus (i.e. considering the first locus a CNOT gate, then it is known, for sure, that the gate will affect at least two qubits). Another constraint is due to the topological optimization; it is not allowed to intersect different types of gates within the same circuit section. The method returns a reference to one known quantum gate from the quantum gate collection and the gate values will be used by the output function methods, in order to compute the circuit function.

---

Algorithm used for detecting the next quantum gate

---

1. Read a quantum gate id and increment the position into the chromosome.
  2. Translate the id into a gate type.
  3. If the gate type was identified, return the reference to it.
  4. Repeat
    - a. Read the next quantum gate id and increment the position.
    - b. Check the gate termination point.
    - c. Translate the id into a gate type.
    - d. If the gate type was identified, return the reference to it.
  5. Meanwhile, a new section is not started.
-

### 5.4.7 Performing Mutation

The mutation operator has the role of producing a change within the chromosome, hence allowing the search algorithm to explore new spaces; it is important not to converge to a local optimum (see Figure 5.22). In this way, when the mutation operator is applied inside the gene, the principle to be followed is:

- Randomly select a chromosome using a mutation probability.
- Randomly select a gene.
- Randomly select a locus from the gene.
- Replace the entire gate from the selected locus (more positions may be affected, if it is a gate on more qubits) with one or more random quantum gates; in the end the same number of inputs need to be replaced.

When the mutation operator is applied at the gene level, then the necessary steps are to:

- Randomly select a chromosome using a mutation probability.
- Randomly select a gene for mutation operation.
- Replace the complete gene contents with new random gene content.

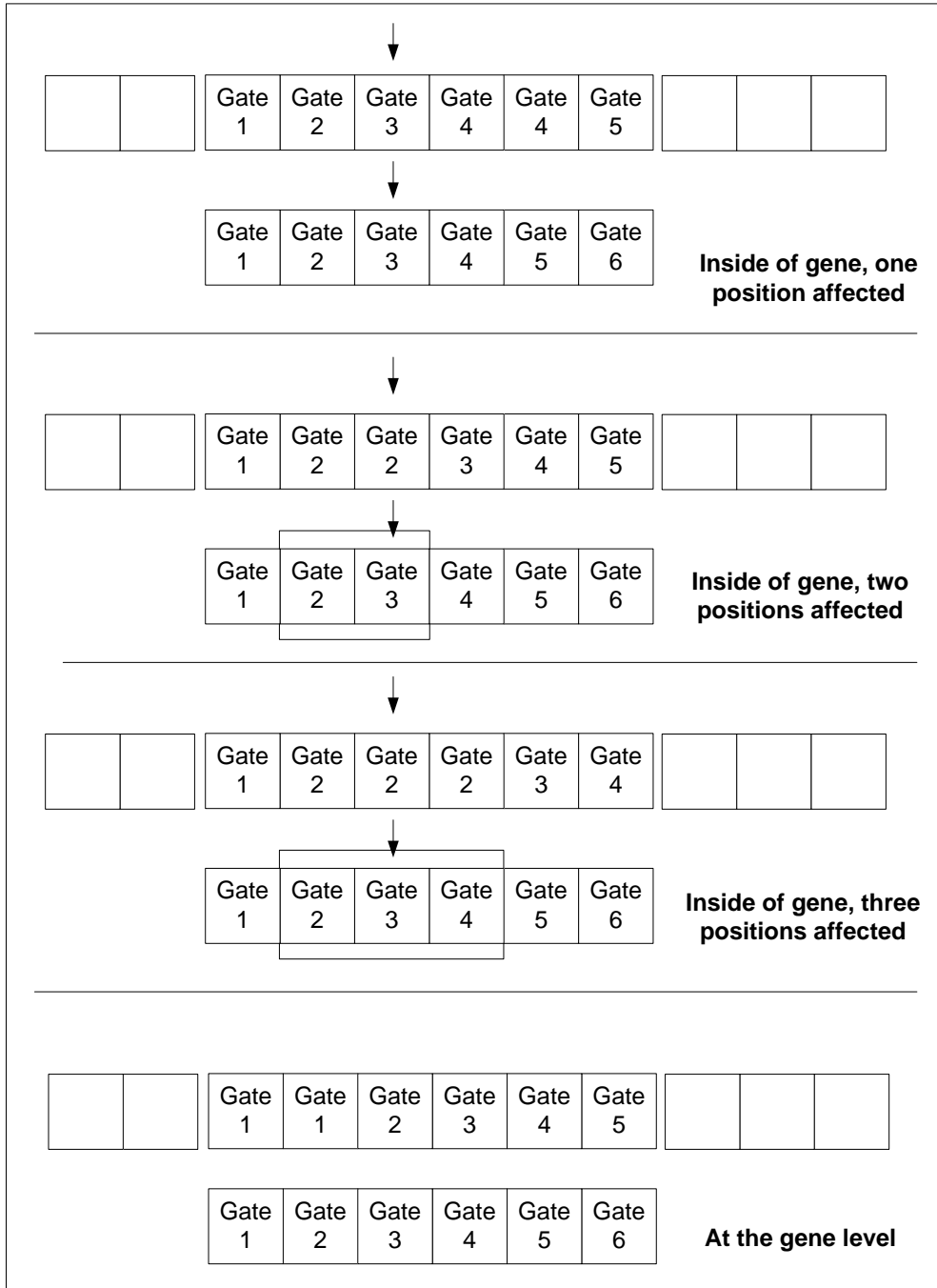


Figure 5.22: Mutation Representation

## 5.4 -Integration within ProGA Framework|101

---

In the synthesis algorithm, it is possible to configure singular or multiple mutation operators (user configurable). Using a locally generated probability, it is possible to affect the genome only once through mutation, or to allow multiple mutations over the previous values.

From an encoding point of view, each gate from the genome has an id number. When the mutation is applied, there are high chances of having a different number of gates into the chromosome, thus it is mandatory to compute again each gate id, starting from the point where the mutation was finished. A part from the mutation algorithm is presented below, while the entire source code is available in section 8.4 (QCS Genome Implementation Details).

---

### Algorithm used for mutation at the gene level

---

1. Get the number of present genes.
  2. Get the gene length.
  3. Randomly select a gene.
  4. Store the last gate id before the mutation point.
  5. Perform the gene mutation by replacing the complete contents with new randomly generated gates having the id number starting with the last memorized gate id increased by one.
  6. Affect the rest of the gates' id in order to be consecutive numbers
    - a. It is possible to have more gates, thus the gates id need to be increased with a shift value.
    - b. Alternatively, it is possible to have fewer gates, thus the gates id need to be decreased with a shift value.
- 

### Algorithm used for mutation inside of gene level

---

1. Get the number of present genes.
2. Get the gene length.
3. Randomly select a gene.
4. Randomly select a locus.
5. Detect the gate corresponding to the selected locus
  - a. If the gate is an identity used between the qubits of a quantum gate, then move the selected locus to the right until a different gate id is detected (used to detect the complete quantum gate).
6. Search to left for the neighboring gate and memorize its locus.
7. Search to right for the neighboring gate and memorize its locus.
8. Generate a new random quantum gate or more quantum gates between the left and right locus positions.
9. Affect the rest of the gates id to be consecutive numbers
  - a. It is possible to have more gates, thus the gates id need to be increased with a shift value.

- b. Alternatively, it is possible to have fewer gates, thus the gates id need to be decreased with a shift value.

### 5.4.8 Performing Crossover

The crossover operator is much complex than mutation. In this case, the gates selected from parents are used to create offsprings, by copying their contents and proprieties. Thus, by using a crossover probability, two genes are selected for reproduction and by applying, one or two, cut points (user configurable) the content between these points is exchanged (see Figure 5.23 and Figure 5.24).

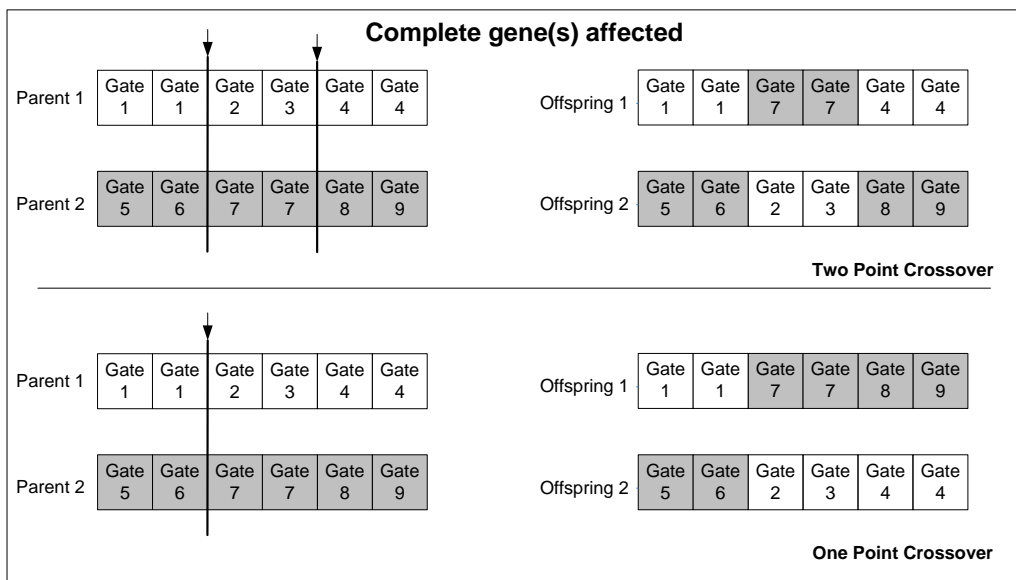


Figure 5.23: Crossover on Complete Gene(s)

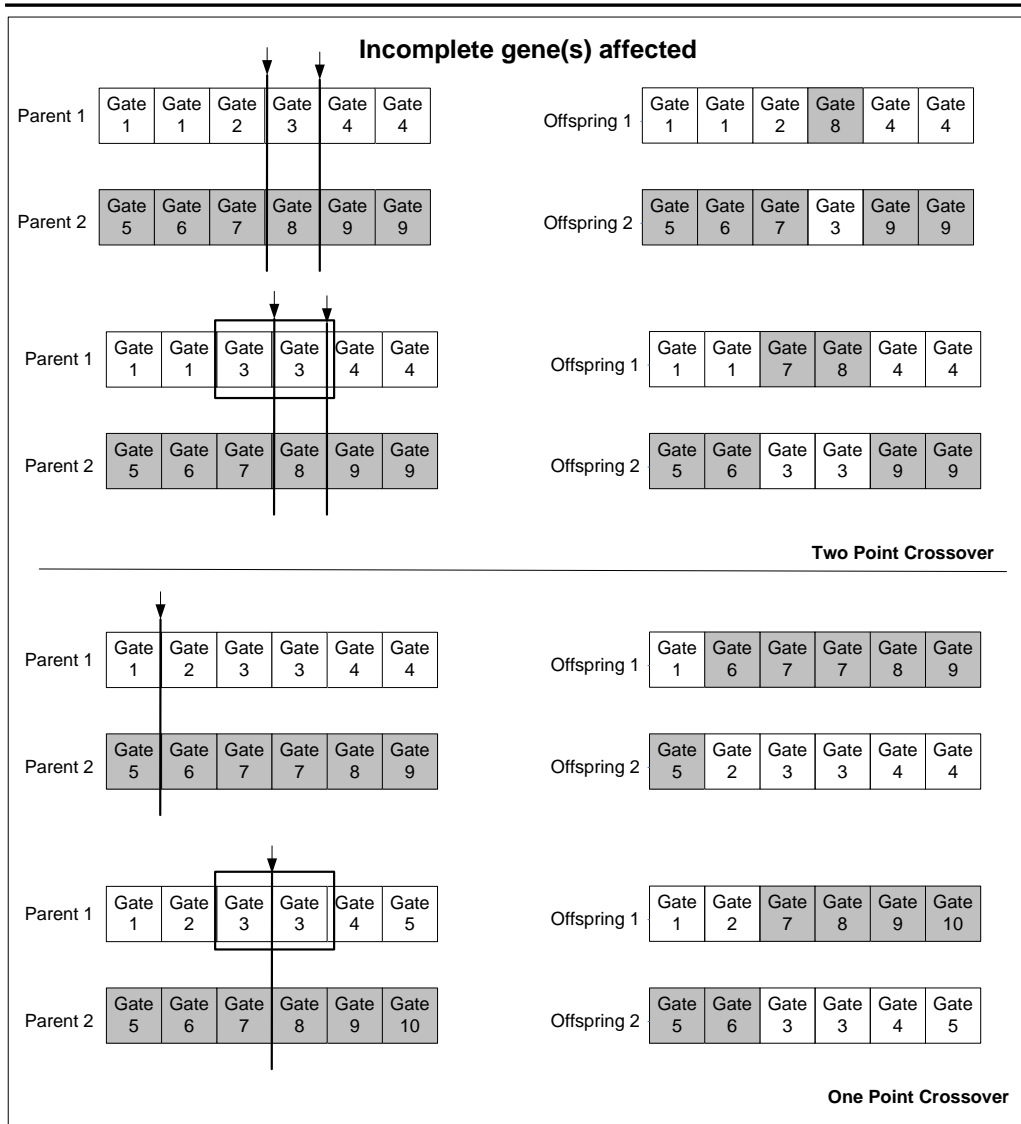


Figure 5.24: Crossover on Incomplete Gene(s)

There are two different situations: the first one where the cut points will affect a complete gene within the crossover operation and, the second situation when the cut points affect only partially the locus from a selected gene. In that last case, the algorithm becomes complicated needed to detect if the cut points affect a single input quantum gate (situation that is easy to copy into the new offspring), or a multiple input quantum gate (situation where the cut point is shifted to the left or

## 104 | 5-Genetic Quantum Circuits Synthesis

---

to the right in order to copy the entire quantum gate). Both algorithms are presented below:

---

### Algorithm used for crossover at the gene level

---

1. Create a new offspring.
  2. Copy parent1 details into the offspring.
  3. Randomly select a cut1 point between 0 and number of genes minus one.
  4. If the crossover operator is with one cut point
    - a. Calculate Start index as  $Cut1 * GeneLength$ .
    - b. Calculate Stop index as the chromosome length.
  5. Else if crossover operator is with two cut points
    - a. Randomly select a cut2 point between 1 and number of genes.
    - b. Repeat selection for cut2 until it is different from cut1.
    - c. Order the cut1 and cut2 points.
    - d. Calculate Start index as  $Cut1 * GeneLength$ .
    - e. Calculate Stop index as  $Cut2 * GeneLength$  minus one.
  6. Exchange the elements of offspring and parent2 between the Start and Stop index.
  7. Shift gate id's for the gates between Start index and Stop index.
  8. Shift gate id's for the gates after the Stop index.
- 

---

### Algorithm used for crossover inside of the gene level

---

1. Create a new offspring.
2. Copy parent1 details into the offspring.
3. Randomly select a cut1 point between 0 and chromosome length minus one.
4. Detect the gate corresponding to the selected locus
  - a. If the gate is an identity used between the qubits of a quantum gate, then move the selected locus to the right until a different gate id is detected (used to detect the complete quantum gate).
  - b. Search to the left the neighboring gate and memorize its index.
  - c. Search to the right the neighboring gate and memorize its index.
5. If the crossover operator is with one point
  - a. Calculate Start index as Cut1 point.
  - b. Calculate Stop index as the chromosome length.
6. Else if the crossover operator is with two cut points
  - a. Randomly select a cut2 point between right index and chromosome length minus one.
  - b. Calculate Start index as left index.
  - c. Calculate Stop index as Cut2 point.
7. Exchange the elements of offspring and parent2 between the Start and Stop index.
8. Shift gate id's for the gates between Start index and Stop index.



9. Shift gate id's for the gates after the Stop index.

---

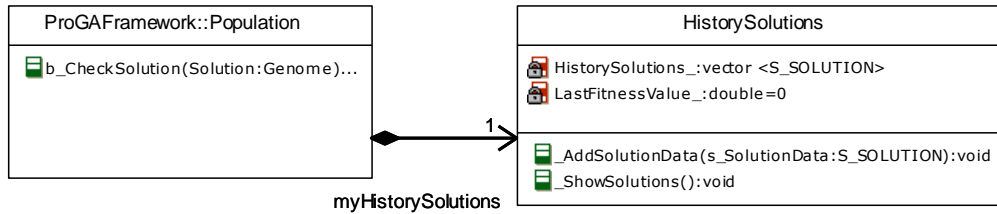
### 5.4.9 Fitness Formula Computation

The most important genetic operator involved in the algorithm is the fitness function. The fitness operator is implemented as a comparison between the output function of the chromosome and the output function of the given circuit. The comparison shall reveal the approximation between them. A penalty function is used in order to indicate a more efficient chromosome than the given circuit (penalty has a good meaning in this case). The penalty function is implemented as the difference between the number of gates from the evolved circuit, and from the given circuit, divided by the number of given gates; it is applied only when the evolved circuit has the same functionality as the given circuit. The penalty is considered as a constraint for the algorithm, and it is used to assure (as result) a better circuit than the given, starting one.

$$f = \frac{\text{function}(\text{evolved})}{\text{function}(\text{given})} + \left(1 - \frac{\text{gates}(\text{evolved}) - \text{gates}(\text{given})}{\text{gates}(\text{given})}\right) \quad (5.1)$$

It is important to discuss about other quantum gate proprieties that may be introduced within the fitness value computation, as a composite formula. For example, the quantum gate cost for each solution is computed and may be used to differentiate them. In the same way, the quantum circuit feasibility is computed and may be introduced into the fitness formula. In order to not introduce weak or partially unknown proprieties into the fitness computation, for the moment the quantum synthesis algorithm only provides those values for manual analysis and will not use them into the genetic evolution. Details about the quantum gate costs are presented in the Quantum Gates Cost subchapter. Concerning the quantum gate feasibility, the research is still ongoing and more results are expected in the near future.

Any fitness higher or equal to one is considered as solution for the synthesis problem. Because the genetic algorithm may generate solutions that are more likely and because each of them is important when the algorithm assessment is performed, a history list for all solutions was integrated within the population architecture (see Figure 5.25). In this way, when the genetic algorithm is finished (according to the termination criterion) all the solutions may be investigated in order to decide which one is optimal or close-to-optimal. Due to genetic evolution, the order in the history is given from non-optimal solutions to the optimal solutions (the backing reason is simple: an evolved solution is maintained into the population while a better one is considered as evolved only if it has an increased fitness value).



**Figure 5.25: History List for Solutions**

For each solution, statistical data is saved for later analysis: the generation number when the solution is evolved, the resulted fitness value, the chromosome values that generated the solution, and the time required by evolution until that moment for the current generation. Identical solutions are not saved into the history list because is not important, from an algorithmic point of view, to analyze identical data values. In this way, the history list will contain better and better solutions for the studied synthesis problem.

## 5.5 Metaheuristic Algorithm

The behavior of the genetic algorithm is determined by the balance between exploitation and exploration. The meta-heuristic algorithm is applied for quantum circuit synthesis in order to adjust dynamically the control parameters. The adaptive parameter control is based on statistical data analysis for each genetic operator type. The objective is to offer the appropriate exploration and exploitation during the algorithm run, without user intervention. The performance measurement is intended to highlight the “good” parameters and to introduce an intuitive meaning for the statistical results.

The problem of setting values for different control parameters is crucial in the context of the algorithm performance [115]. We introduce an adaptive genetic algorithm, in order to evolve quantum circuits. Our ProGA framework is used for the genetic algorithm implementation, its architecture being decorated with related statistical information. The statistical data are analyzed on the fly by the adaptive algorithm, and the results are used for adjusting of the genetic parameters control during the runtime processes.

Meta-heuristic approaches are already used to solve different problems, this domain being of particular interest for the researchers in the last decade [116] [117] [118] [119] [120] [121] [122] [123] [124].

### 5.5.1 Parameter Control

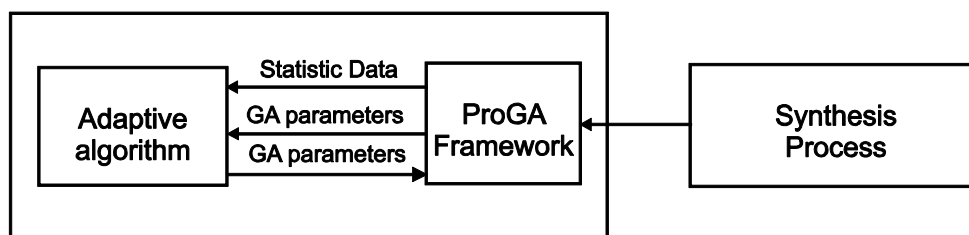
The parameter controls involved in a genetic algorithm are population size, mutation probability, crossover probability, selection type, etc. Each of them is responsible

with controlling the evolution path towards the optimal solution. There are two major forms of setting the parameter values for a genetic algorithm [125]:

- Parameter tuning: the parameter values are fixed before the algorithm run (i.e. using information from experiments) and remains fixed during the algorithm run. There are several disadvantages for the tuning: finding good parameters for the parameters before the run may be time consuming for a human expert, the evolution being a dynamic process, while the tuning is static, and it is possible not to have optimal parameters for all the phases.
- Parameter control: the initial parameter values are changed during the algorithm run, keeping the dynamic spirit of evolution. The adaption algorithm uses the feedback values from the process and adjusts the parameters for better performance (i.e. initially, mutation probability may be higher in order to allow large exploration space and, later, it may be decreased to permit the solution fine-search).

### 5.5.2 Integration within ProGA Framework

The ProGA framework is responsible with the genetic algorithm implementation. The choice concerning the algorithm type (steady state or non-overlapping), the population structure, the encoding of the genome and the initial settings for the parameter controls is made within the framework. An important framework characteristic is the possibility of extending its functionality. Thus, as it is presented in Figure 5.26, the Adaption Control can use the framework interface, therefore allowing its integration into the system.



**Figure 5.26: Adaptive Control Integration**

The framework provides all the data necessary for statistic analysis and the actual values of the parameters control while the Adaptive Control component will return the new adjusted values for the parameters control back to the framework. The Adaptive Control is considered as an external tool for the genetic algorithm implementation, and it is responsible only with the cleverly update of the parameters control.

### **5.5.3 Adaptive Behavior using Operator Performance**

Almost every practical search algorithm is controlled by several parameters. In fact, the genetic algorithms are controlled by more parameters than other algorithms: population size, selector type, mutation probability, crossover probability, etc. From meta-heuristic point of view, it is considered that genetic algorithms contain all necessary information for adaptive behavior. Nevertheless, in the following subchapters, we present how the adaptive behavior optimizes the circuit synthesis algorithm (from the user's point of view the setting of parameters is far from being a trivial task). Two types of statistical data are used as input for the adaptive algorithm (see Figure 5.27). The first type is represented by the fitness results for each population corresponding to the best, mean and worst chromosomes. The second type is represented by the operator performance. Following an idea proposed in reference [126], the performance records are essential in order to decide on operators reward. The statistical data indicate different levels of operator performance as described in Figure 5.28):

- Absolute - when the resulted offspring has a higher fitness than the best fitness from the previous generation.
- Relative - when the resulted offspring has a better fitness than its parents, but it is not absolute.
- In Range - when the resulted offspring has a fitness situated between the fitness values of its parents.
- Worse - when the resulted offspring has a fitness value that is lower than that of its parents.

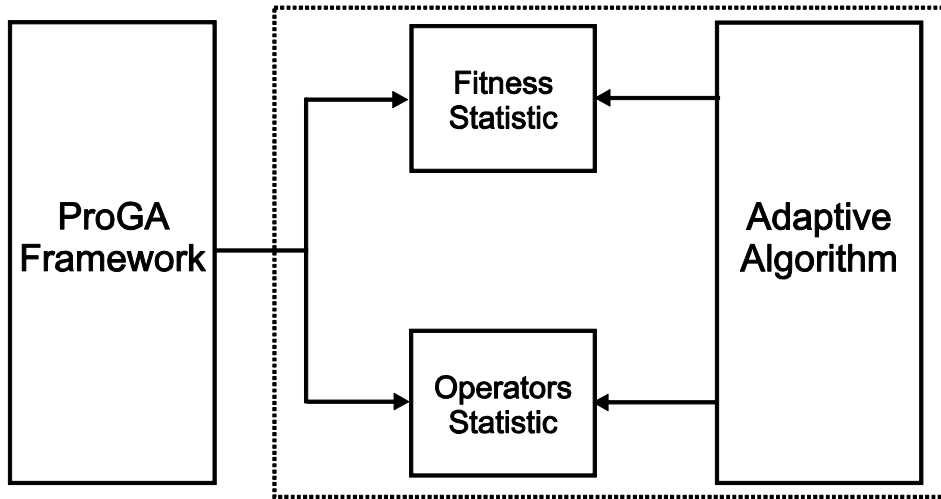


Figure 5.27: Statistic Data

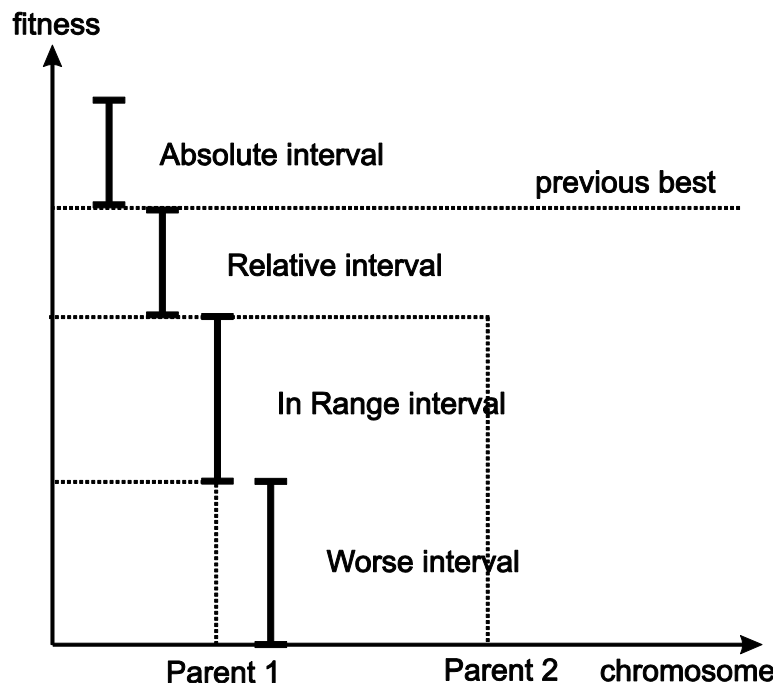
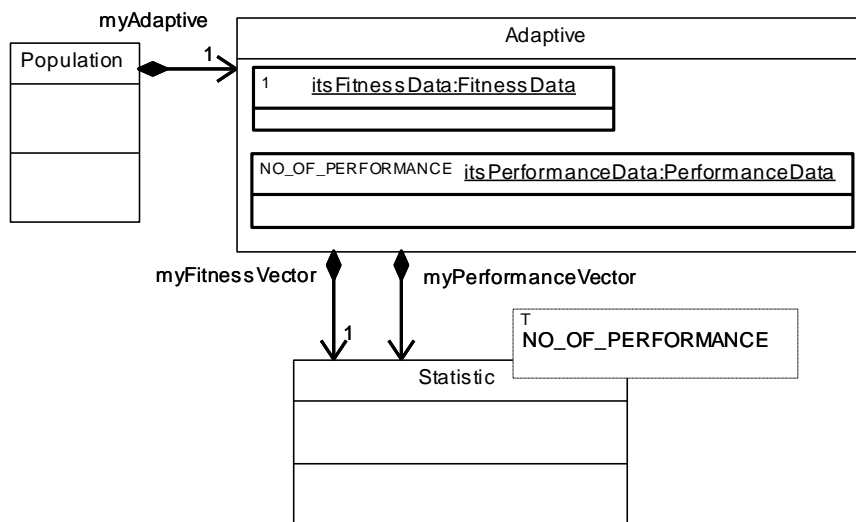


Figure 5.28: Operator Performance






















## 110 | 5-Genetic Quantum Circuits Synthesis

For the circuit synthesis algorithm, the mutation and crossover probabilities are two major parameters that need to be dynamically controlled. Because we defined two mutation and two crossover operators, four related statistical data have to be memorized and later analyzed in order to perform parameter control adjustment. Each operator offspring result is important and needs to be recorded (see Figure 5.29).



**Figure 5.29: Adaptive Design**

Details about Statistic class implementation are available in 8.3 Statistic Details) section. As described in 4.2.2 (Framework Statistics) section, basic statistic functions such as maximum, minimum, average and standard deviation need to be available for the adaptive controlled types. Thus, the design overview allows interface declaration for the statistical functions in the Statistic class, and –also– their definition in each object class instance (i.e. the FitnessData class will define its own statistic function on its data attributes, while the PerformanceData will do the same but on different attributes). In this way, as assured by design, any class instance from the statistic has its own implementations for the corresponding functions, and at the same time keeps a common interface for accessing statistic results (see Figure 5.30).

PerformanceData	
	Absolute_:double=0
	InRange_:double=0
	Relative_:double=0
	Worse_:double=0
	ComputeMaxim(a:PerformanceData,b:PerformanceData):friend PerformanceData
	_IncrementAbsolute():void
	_IncrementRelative():void
	_IncrementInRange():void
	_IncrementWorse():void
	_Init():void
	ComputeMinim(a:PerformanceData,b:PerformanceData):friend PerformanceData
	_ShowValues():void
	operator+(obj:PerformanceData):PerformanceData&
	operator/(value:int):PerformanceData
	ComputeMean(a:PerformanceData,NoOfElements:int):friend PerformanceData
	operator*(obj:PerformanceData):PerformanceData
	operator-(obj:PerformanceData):PerformanceData&
	operator*(value:int,obj:PerformanceData):friend PerformanceData
	operator>(value:int):bool
	ComputeSquareRoot(obj:PerformanceData):friend PerformanceData
	n_ComputeReward():double

**Figure 5.30: PerformanceData Class Overview**

The first type of mutation, called mutation A, is responsible with the gate mutations inside of genes, while the second type of mutation, called mutation B, is applied at the chromosome level. The same rules are defined for the crossover operator (applied at the gene level - called crossover A, and at the chromosome level - called crossover B). The operators are implemented within the ProGA framework, and only adding an extra-layer for the adaptive algorithm is sufficient for the meta-heuristic implementation. From the meta-heuristic point of view, it is not essential to know the operator implementation details; instead, one has to be informed about the number of operators because, for each of them, a separate statistical structure will be reserved. The adaptive algorithm will receive breeding feedback from each operator and will analyze the data, in order to compute the operator performance and decide on its adjustment rate.

### **5.5.4 How the Change is Made**

The change of parameter controls is made by using the feedback data from the search (stored as statistical data). The adaptive algorithm distinguishes between the qualities of solutions evolved by different operators and adjusts the rates based on merits. As described, the adaptive algorithm is external to the genetic algorithm framework, the only interaction being on the transfer of parameter rates and feedback data.

The following algorithm presents the important steps that are necessary for the operator's rate adjustment:

---

Algorithm used for dynamically adjusting the operators' rate

---

1. Initialize operator rates.
  2. While the algorithm does not reach the stop condition
    - a. Initiate the operator rates using the Adaptive Algorithm.
    - b. Evaluate the individuals.
    - c. Save the statistical data (fitness and operators performance).
    - d. Compute the new rates using the Adaptive Algorithm.
    - e. Check for a solution.
    - f. Apply the operators.
- 

### **5.5.5 Performance Meaning**

When discussing about statistical data, several statistic functions may provide valuable information about data distribution. Functions as Maximum, Minimum, Average and Standard Deviation may be applied on any kind of statistical data. For each generation the maximum, average and minimum fitness values are provided by the genetic algorithm framework and stored in the statistical data. When the genetic evolution is finished (i.e. when a solution has been evolved), more important statistical functions are computed: maximum for all the generation maximum fitness, average on all the maximum values, etc. Thus, we defined statistical functions on each generation and statistical functions over all generations.

The second type of statistical data for the operator performance is computed when the operator is applied, following the rules presented in Figure 5.28). For example, when the Crossover B result is available, the resulted offspring fitness value is compared against the previous best fitness, and if it is higher, the Absolute value is increased with one-step. If it is lower, the comparison continues, and if the offspring fitness is higher than its parents' fitnesses, then the Relative value is increased, etc. The algorithm identifies the possible solutions obtained during the genetic algorithm evolution and their distribution over the involved generations. After each generation, the operator performance is updated with statistical data. Following the 1/5 Rechenberg rule, after 5 generations it is time to make the



analysis of the acquired data. The operator reward is updated according to the following formula:

$$\sigma(\text{operator}) = a * \text{Absolute} + \beta * \text{Relative} - \gamma * \text{InRange} - \delta * \text{Worse} \quad (5.2)$$

Parameters  $a$ ,  $\beta$ ,  $\gamma$  and  $\delta$  are introduced to rank the operator performance. Thus, an absolute improvement will have a higher importance as compared with the relative improvement; a worse result will drastically decrease the operator rank.

## Chapter 6

### Experiments Result Evaluation

During the execution of the performed experiments, several variables were used to measure, control and manipulate the application results. The proposed synthesis tool allows two different types of statistical data; the correlation research helps measuring the statistical data and – at the same time - looking for possible relationships between some given sets of variables, while in the experimental research some variables are being influenced in order to observe their effect on other variable sets. The data analysis of the experimental results also creates correlations between the manipulated variables and those affected by the manipulation. The correlation results were interpreted in causal terms by using different theories, and the experimental results conclusively prove causality.

#### 6.1 The Experiment Setup

In our experiments, configurable variables were used to define the genetic algorithm controlled parameters. A further classification is performed by splitting variables into independent and dependent types (as in Table 6). Only the dependent variables are measured (i.e. the execution clock cycles until an optimal solution is evolved), while the independent variables are manipulated in order to allow a faster convergence for the genetic algorithm. The goal of our scientific analysis was to find the best relation between variables that produced the best execution runtime for the optimal evolved solution, and to provide a meaning for the obtained results. The experiments will also prove, by validation testing, the correctness of the algorithm implementation for quantum logic circuit synthesis.

Globally speaking, in any experiment where variables are measured there is a measurement error present. It is considered that variables differ in how well they can be measured [127]. In our experiments, we are using interval variables (i.e. number of generation may be 50, 100, 150 or 200), that allow data values rank order and comparison of the data differences over experiment runs.

**Table 6. Variable Type Classification**

Variable name		Independent	Dependent
Genetic algorithm type	Overlapping	√	
	Non-Overlapping		
Number of generations	(50, 100, 150 or 200)	√	
Population size	(50, 100, 250 or 300)	√	

### 6.1 -The Experiment Setup|115

Elitism percent	0% means "No Elitism"		√	
Crossover type	One point	√	√	
		√		
	Two points	√		
		√		
Mutation type	Singular	Chromosome level	√	
		Inside gene		
	Multiple	Chromosome level		
		Inside gene		
Selection type	Uniform		√	
	Rank			
	Roulette wheel			
	Tournament			
Mutation probability			√	√ (only with adaptive behavior)
Crossover probability			√	√ (only with adaptive behavior)
Statistical Data	Available		√	
	Not Available			
Meta-heuristic Algorithm	Used (only with Statistic Data Available)		√	
	Not Used			
Meta-heuristic Increase			√	
Meta-heuristic Decrease			√	
Random generator type	Classic generator		√	
	Ran2 generator			
Statistical information for fitness data and for each genetic operator (crossover and mutation types)	Maximum			√
	Minimum			
	Average			
	Standard deviation			

## 116 | 6-Experiments Result Evaluation

Algorithm runs	10	√	
Gate set type used for synthesis	Complete	√	
	Reduced		
	Minimum		
History solution time (in processor clocks)			√
Number of quantum circuit gates			√
Quantum circuit cost			√
Quantum circuit feasibility			√
Number of generations until optimal solution			√

The experiments were conducted on a computer with the following configuration: Intel Pentium M processor at 1.862GHz, 1GB RAM memory and Open SuSe 10.3 as operating system. In order to avoid lucky guesses the experiments have been repeated for 10 times, the average result being used for comparison in the provided graphics.

### 6.2 Evaluation Approach

The C++ source code<sup>14</sup> was compiled using the "gcc" compiler under the Eclipse v.15 software on Open SuSe 10.3 operating system. The executable released binary was fed with input files (the configuration parameters and the circuit description) and then used to execute the quantum logic circuit synthesis algorithm. The graphs are created automatically by using the GNUplot<sup>16</sup> software for which we have written a script file in order to multiplot the graphs, by using the quantum logic circuit synthesis algorithm output results (see 8.6 GNUplot Script).

To measure the performance of an application, it is common to measure the time spent until a solution is evolved. Because the results may appear within a small period, a fine granularity for time measurement was necessary. We used the RDTSC (Read Data Time Stamp Counter) to measure the processor ticks in order to provide excellent, high-resolution information. A stopwatch was used in order to measure the processor ticks between the moment when the genetic algorithm evolution was started and the event generated by finding a solution. For evaluation of other solutions that may succeed, the stopwatch is re-started to time the next solution event. The number of ticks is independent from the processor platform and it accurately measure events of short duration (with laptops or systems supporting Intel@Speed Technology the processor frequency will change as a result of CPU utilization when running on batteries). If the user wants to derive the time duration, the number of ticks should be divided by the processor frequency.

<sup>14</sup> [http://www.cs.utt.ro/~crys/index\\_files/public/qsyn.tar.gz](http://www.cs.utt.ro/~crys/index_files/public/qsyn.tar.gz)

<sup>15</sup> Eclipse, <http://www.eclipse.org/>

<sup>16</sup> GNUplot, <http://www.gnuplot.info/>

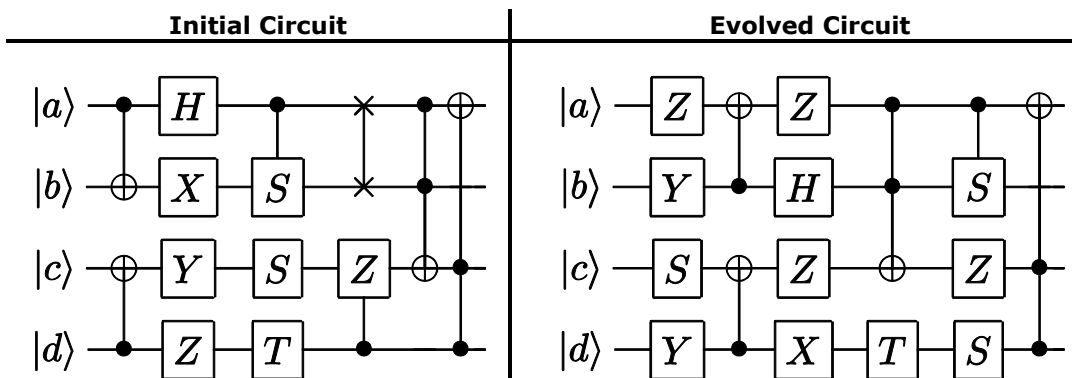
### 6.3 QCS Tool Verification

The test is intended to prove the genetic algorithm convergence towards a circuit solution. All the recognized gates are used to define a test circuit and, after algorithm execution, other equivalent circuits are expected to emerge by using quantum circuits from the specified gate set. In this test, we do not evaluate configurations or parameters because the target is only to evolve an equivalent solution for the given circuit description.

#### 6.3.1 Complete Set of Gates

The entire set of recognized quantum gates will be used in order to evolve a circuit solution. This test will prove the complete functionality of quantum logic circuit synthesis.

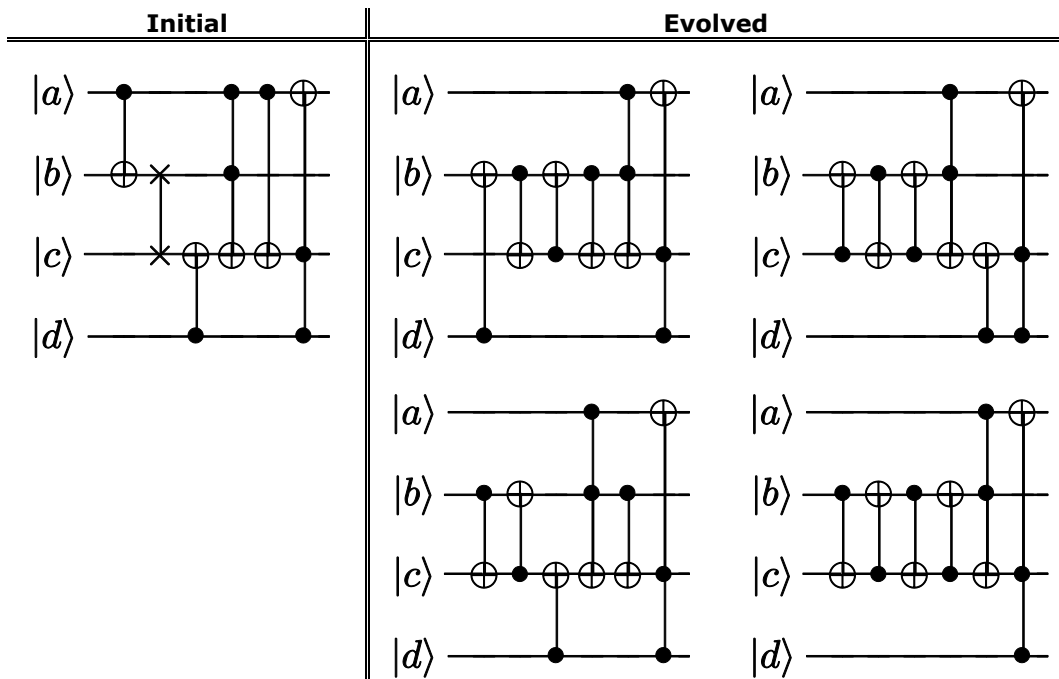
Table 7: Initial and Evolved Circuit – Complete Gates Set



#### 6.3.2 Reduced Set of Gates

In this test only Controlled-NOT, Identity and Toffoli gates were used to evolve a circuit solution. This reduced set of gates was introduced to allow compatibility with the available benchmark circuits [51][128][11][46][47] where a limited number of circuit types are used.

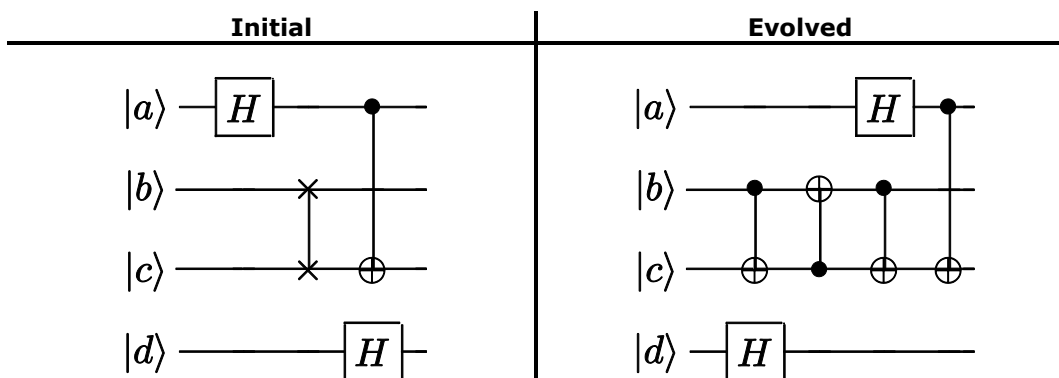
Table 8: Initial and Evolved Circuits – Reduced Gates Set



### 6.3.3 Minimal Set of Gates

In this test, only Hadamard, Identity and Controlled-NOT gates were used to evolve a circuit solution. This set of gates is recommended for simple circuits where a fast synthesis solution may be emerged.

Table 9: Initial and Evolved Circuit – Minimal Gates Set



## **6.4 Case Studies**

Each case study is started with a benchmark quantum circuit that is used for the synthesis algorithm evaluation. On the benchmark, the name of the circuit is presented along with its number of qubits (including garbage qubits, if present) and the circuit cost. Three synthesis configurations are used to evolve a synthesis solution, different parameters being manipulated during the test evolution (i.e. the adaptive behavior will dynamically adjust the mutation and the crossover probabilities), while the results are presented as graphs. The result analysis section is used to explain the algorithm behavior and to extract a conclusion concerning the optimal parameter control values. All the benchmark circuits are kept up-to-date by Dmitri Maslov [51] on his web page<sup>17</sup> dedicated to quantum circuit synthesis.

---

<sup>17</sup> <http://webhome.cs.uvic.ca/~dmaslov/>

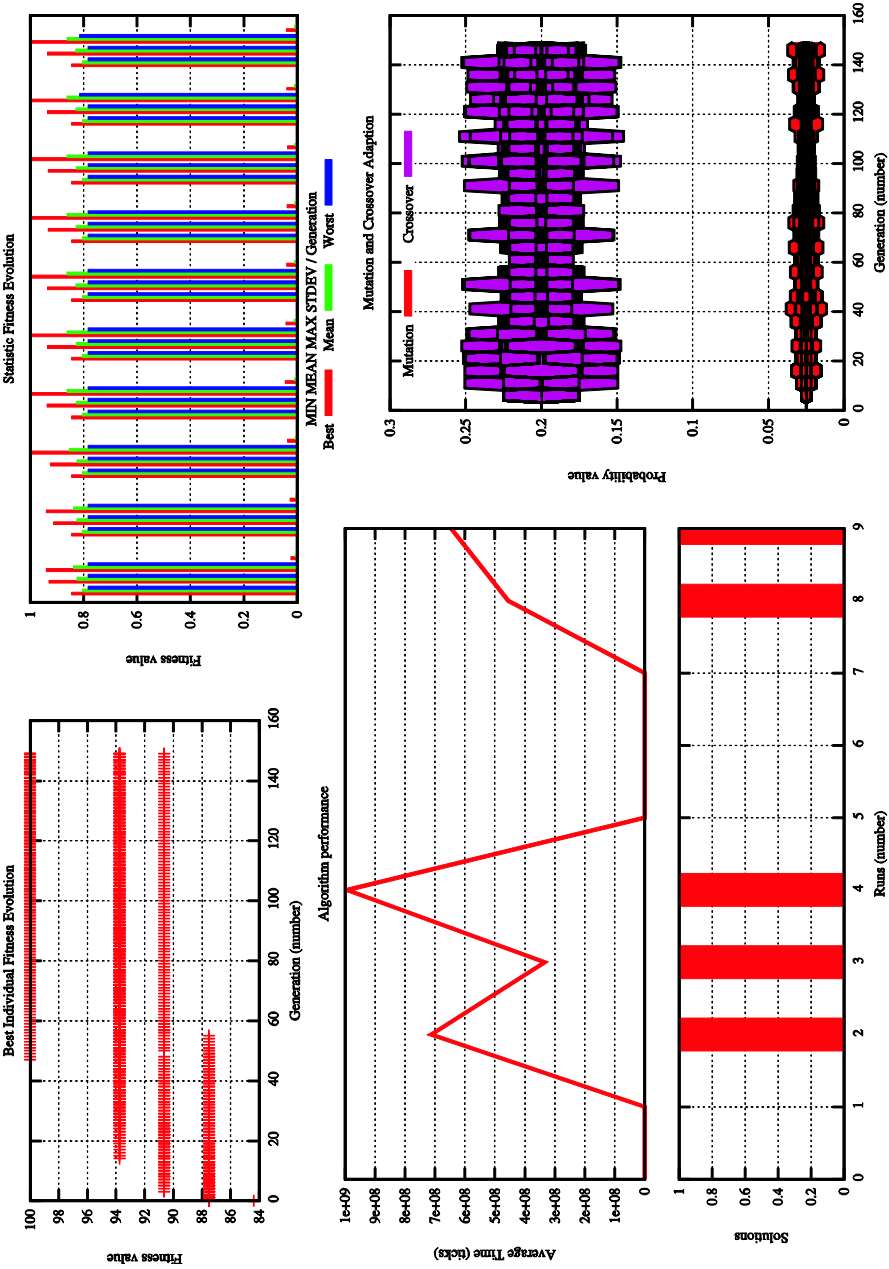
### 6.4.1 Three-Qubit Circuit

<b>Benchmark:</b> ham3 [51]					
<b>Gate count</b>	5	<b>Cost</b>	7	<b>Garbage</b>	0
Function <i>ham3</i> is the size 3 Hamming optimal coding function.					
<b>Configuration</b>	<b>1</b>	<b>2</b>	<b>3</b>		
Number of Generations	150				
Population Size	100				
Elitism percent	0.05				
Crossover Type	Two Points	One Point	Two Points		
Mutation Type	Multiple	Singular	Multiple		
Crossover Probability	0.4				
Mutation Probability	0.05				
Selection Type	RouletteWheel	Uniform	RouletteWheel		
Performance Statistic	Available				
Meta Heuristic	Available		NA		
Adaptive Increase	0.15		NA		
Adaptive Decrease	0.1		NA		
Gate Set	Reduced				
Genetic Algorithm Type	Non Overlapping				
Random Generator	Ran2				
Algorithm Runs	10				
<b>Solution Runtime (average) Clocks</b>	6.29e+08	3.69e+08	8.98e+08		

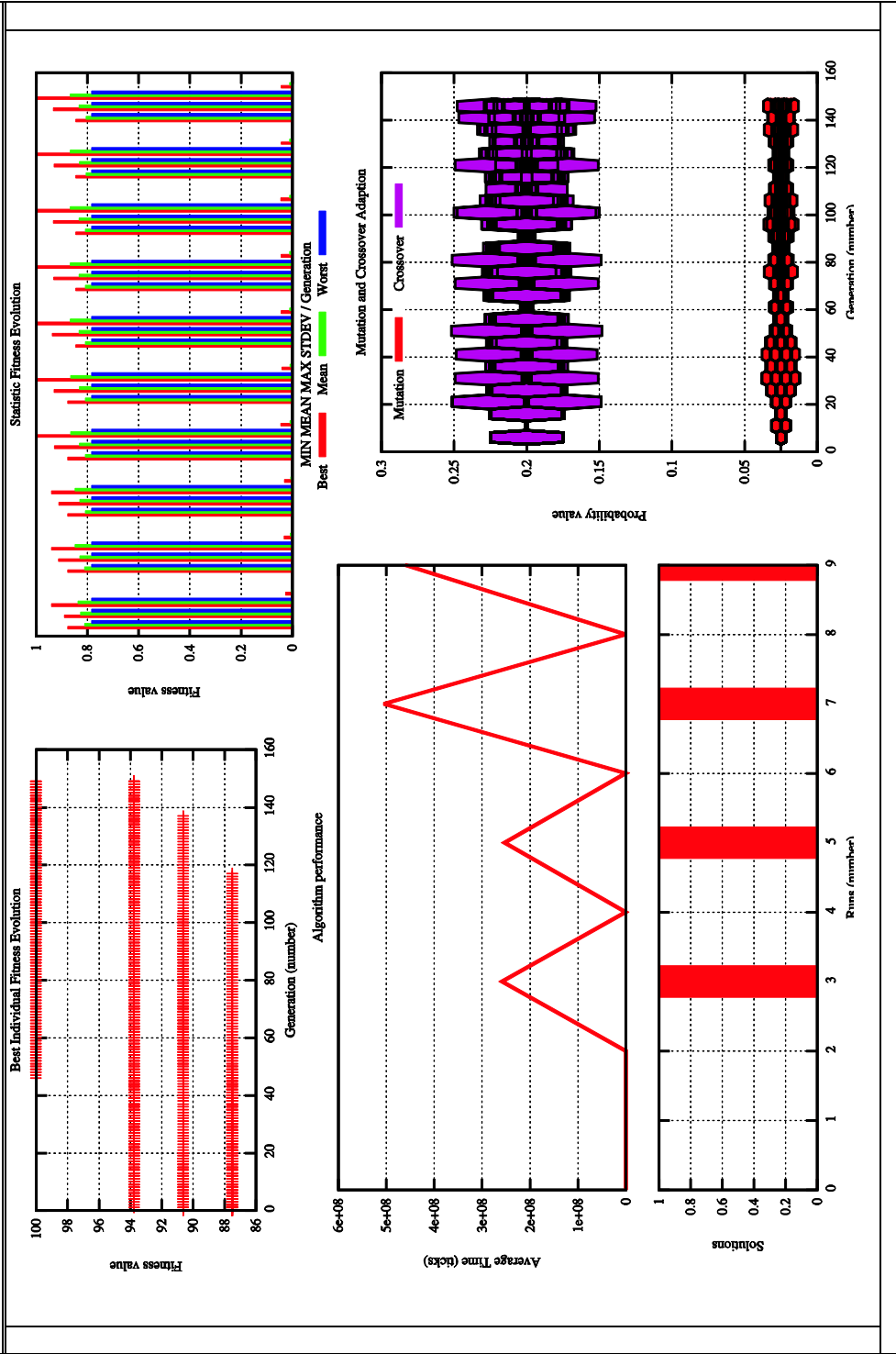


Statistic Results for the Evolved Solutions

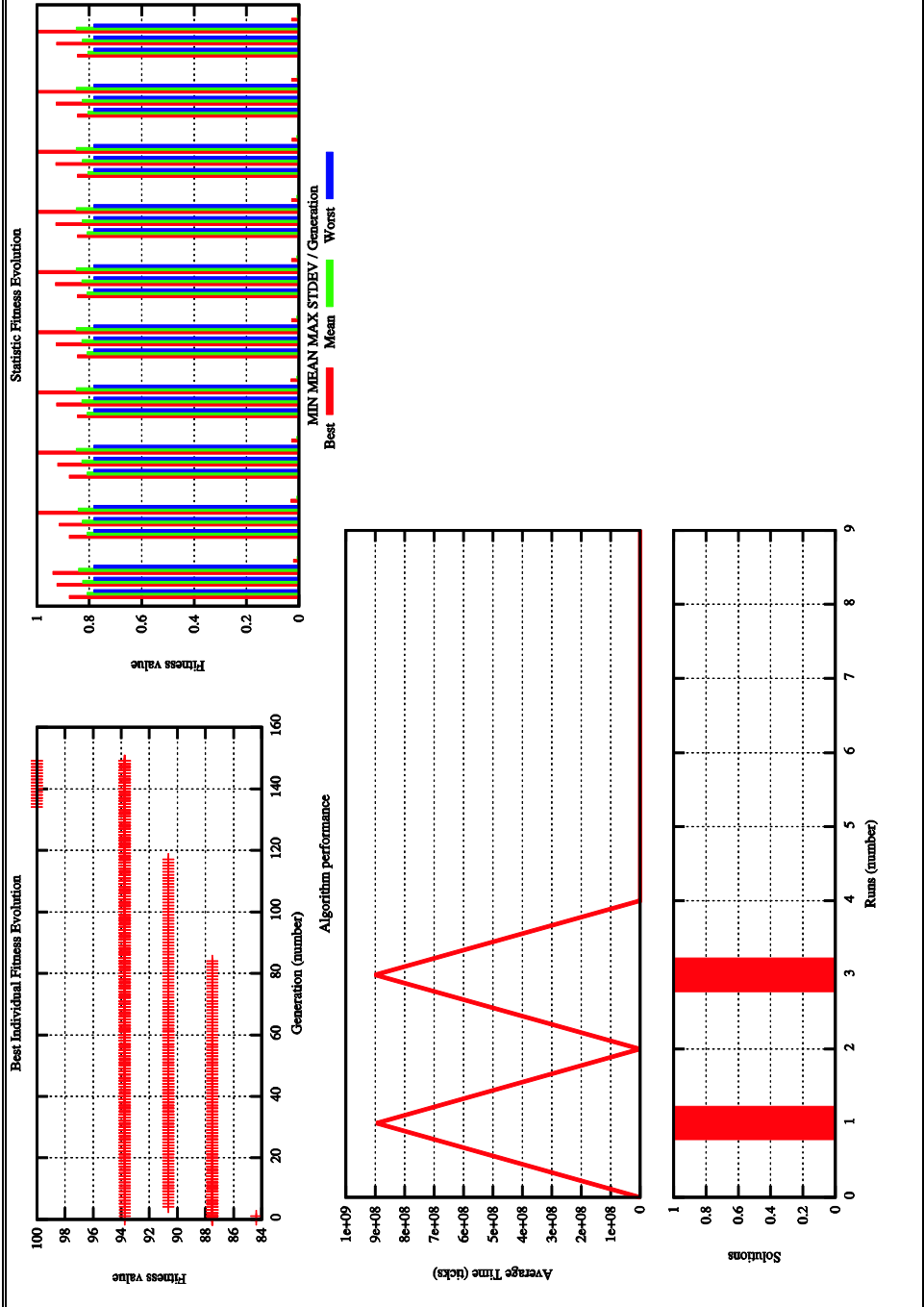
Configuration 1



Configuration 2



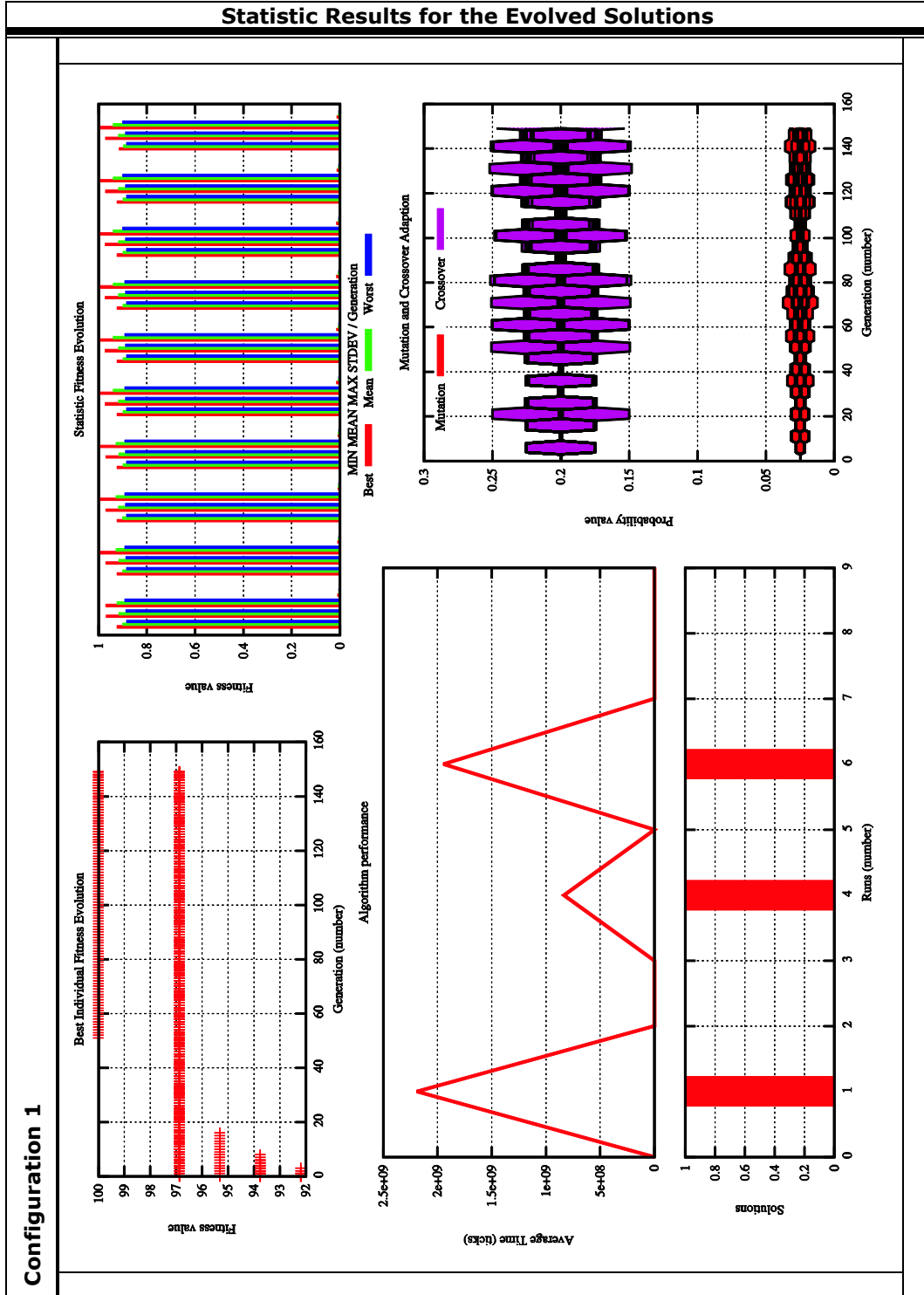
Configuration 3



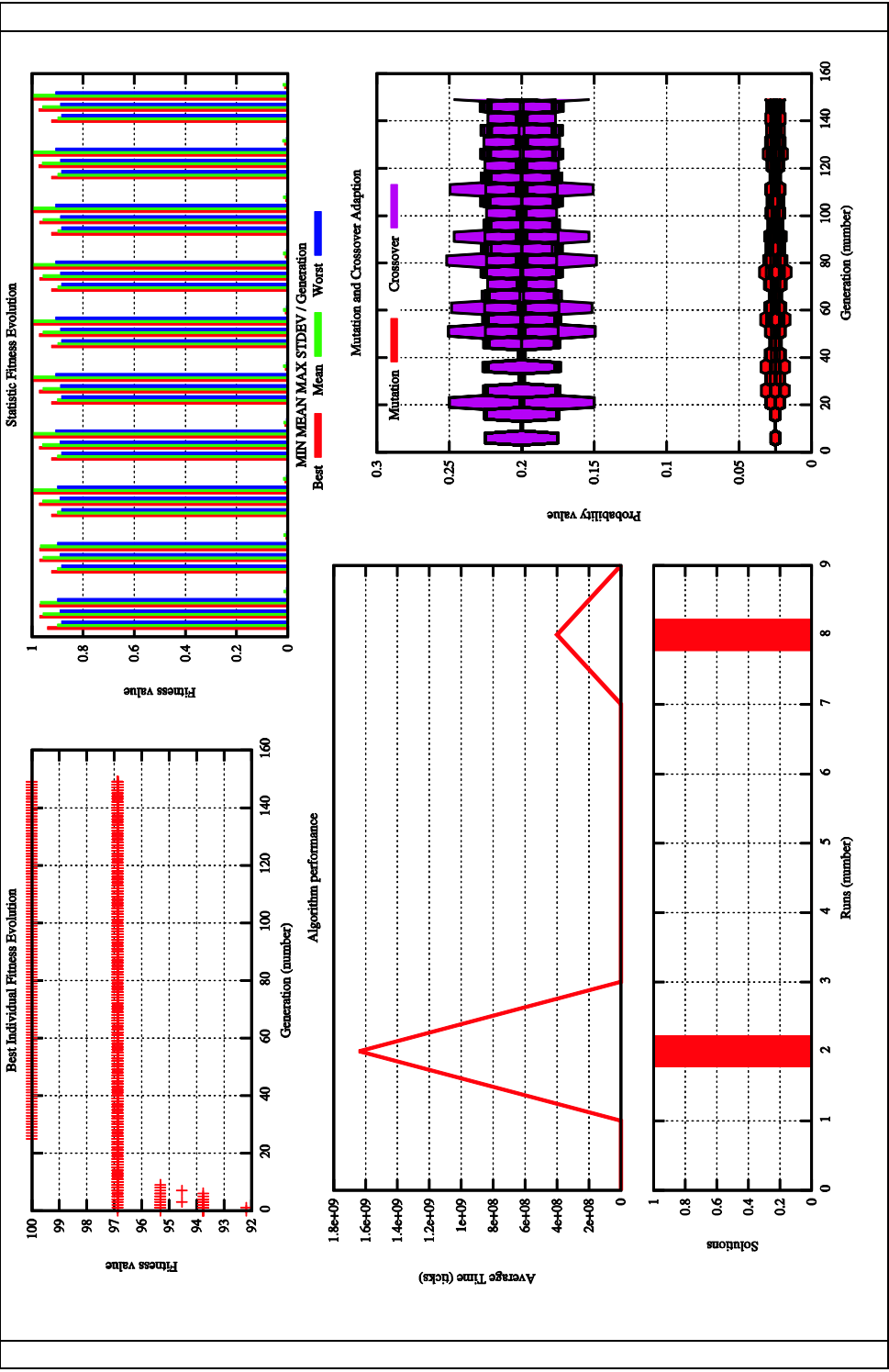
Circuits Evolved Solutions									
Configuration 1/2/3									
	<b>1</b>	<b>2</b>	<b>3</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>1</b>	<b>2</b>	<b>3</b>
	5	11	100	5	11	100			
Note: 1 = Gates count, 2 = Circuit Cost*, 3 = Circuit Feasibility* * see their definition in 8.5.1 (Cost Details using General Approach) section									

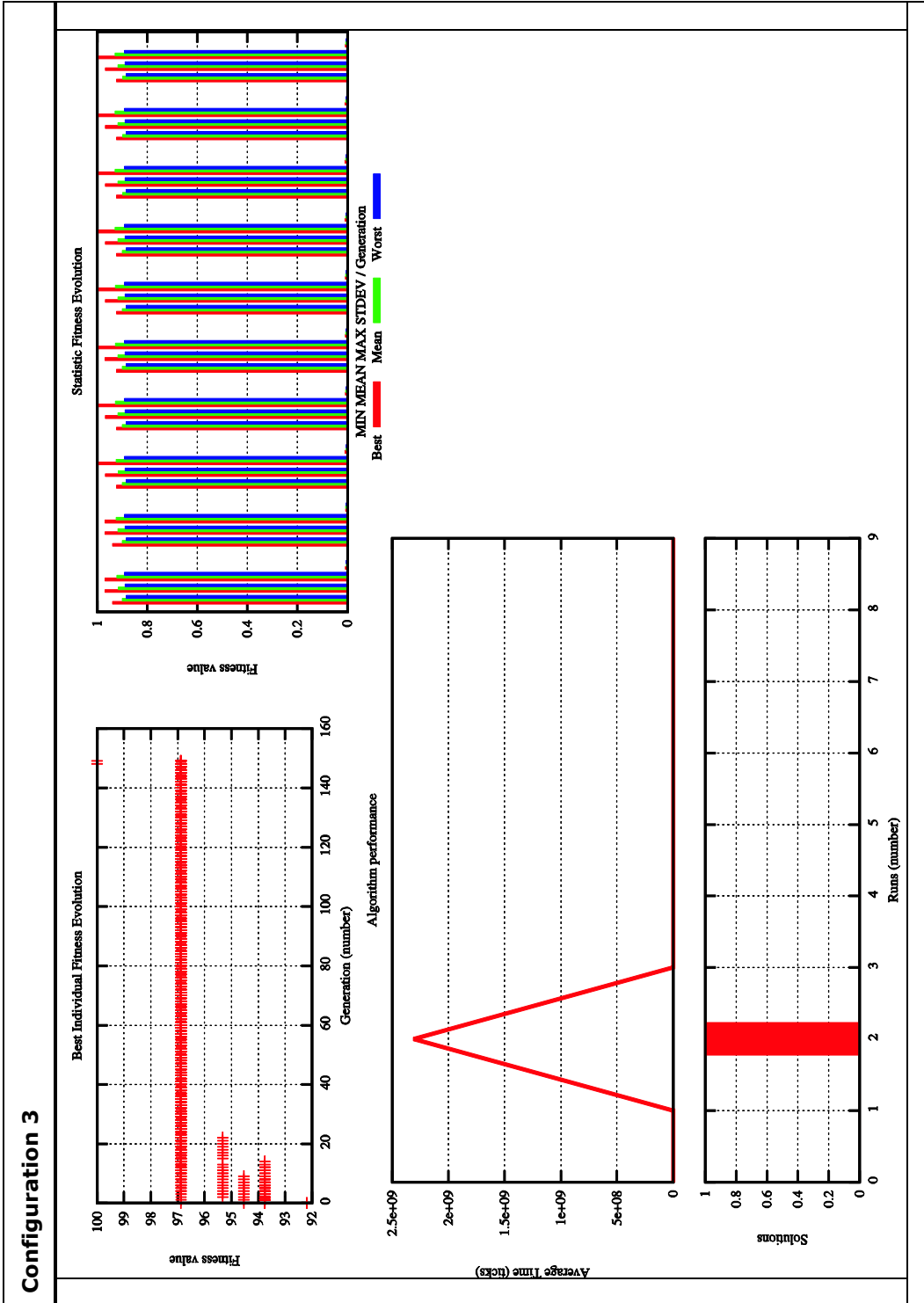
6.4.2 Four-Qubit Circuit

<b>Benchmark:</b> rd32 [51]			
<b>Gate count</b>	4	<b>Cost</b>	8
		<b>Garbage</b>	2
Its output is binary representation of the number of 1s in its input.			
<b>Configuration</b>	<b>1</b>	<b>2</b>	<b>3</b>
Number of Generations	150		
Population Size	100		
Elitism percent	0.05		
Crossover Type	Two Points	One Point	Two Points
Mutation Type	Multiple	Singular	Multiple
Crossover Probability	0.4		
Mutation Probability	0.15		
Selection Type	RouletteWheel	Tournament	RouletteWheel
Performance Statistic	Available		
Meta Heuristic	Available		NA
Adaptive Increase	0.15		NA
Adaptive Decrease	0.1		NA
Gate Set	Reduced		
Genetic Algorithm Type	Non Overlapping		
Random Generator	Ran2		
Algorithm Runs	10		
<b>Solution (average) Runtime (Clocks)</b>	1.66e+09	1.02e+09	2.31e+09



Configuration 2







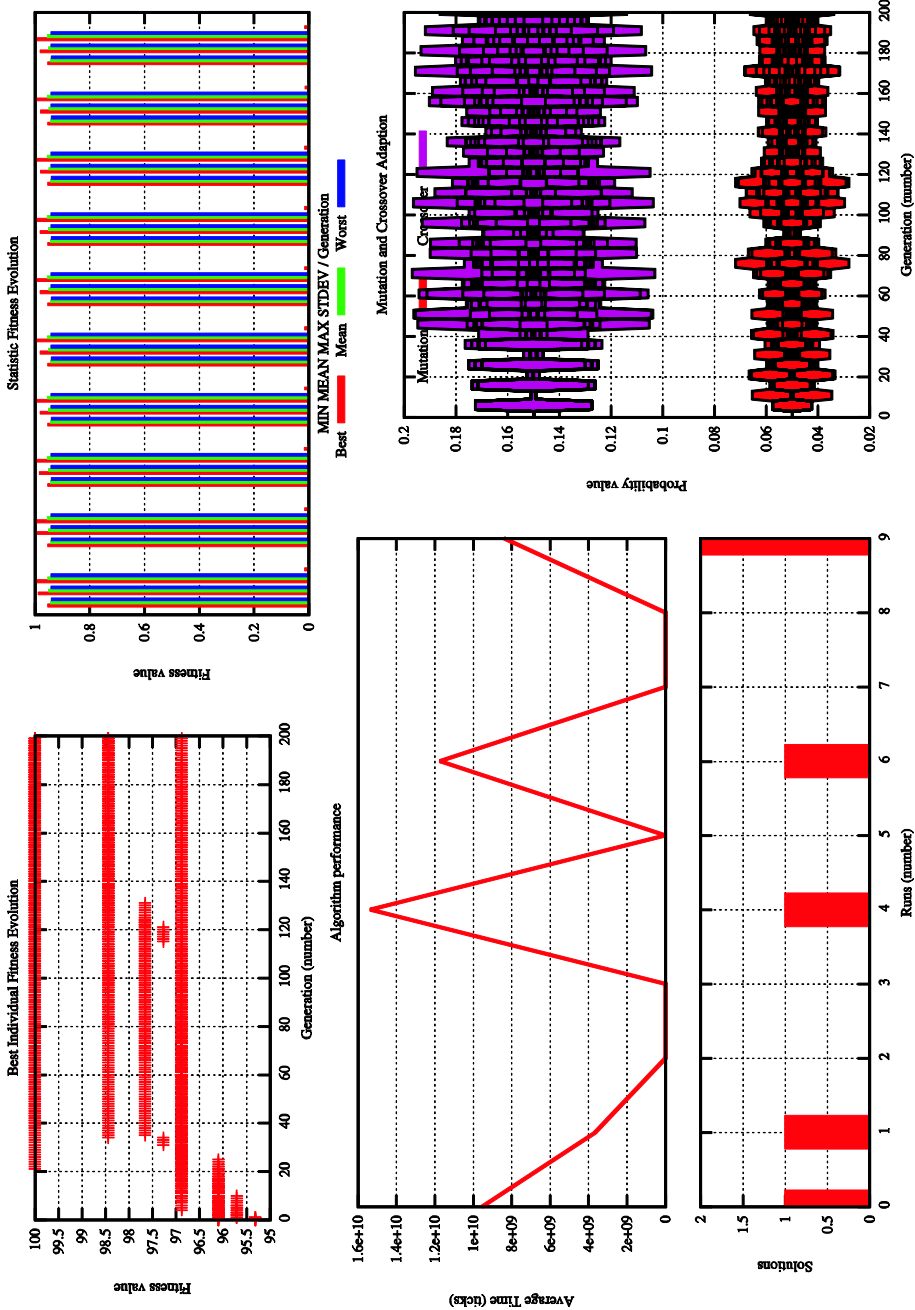
Circuits Evolved Solutions										
Configuration 1/2/3										
	<b>1</b>	<b>2</b>	<b>3</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>1</b>	<b>2</b>	<b>3</b>	
	4	10	93.75							
	Note: 1 = Gates count, 2 = Circuit Cost*, 3 = Circuit Feasibility* * see their definition in 8.5.1 (Cost Details using General Approach) section									

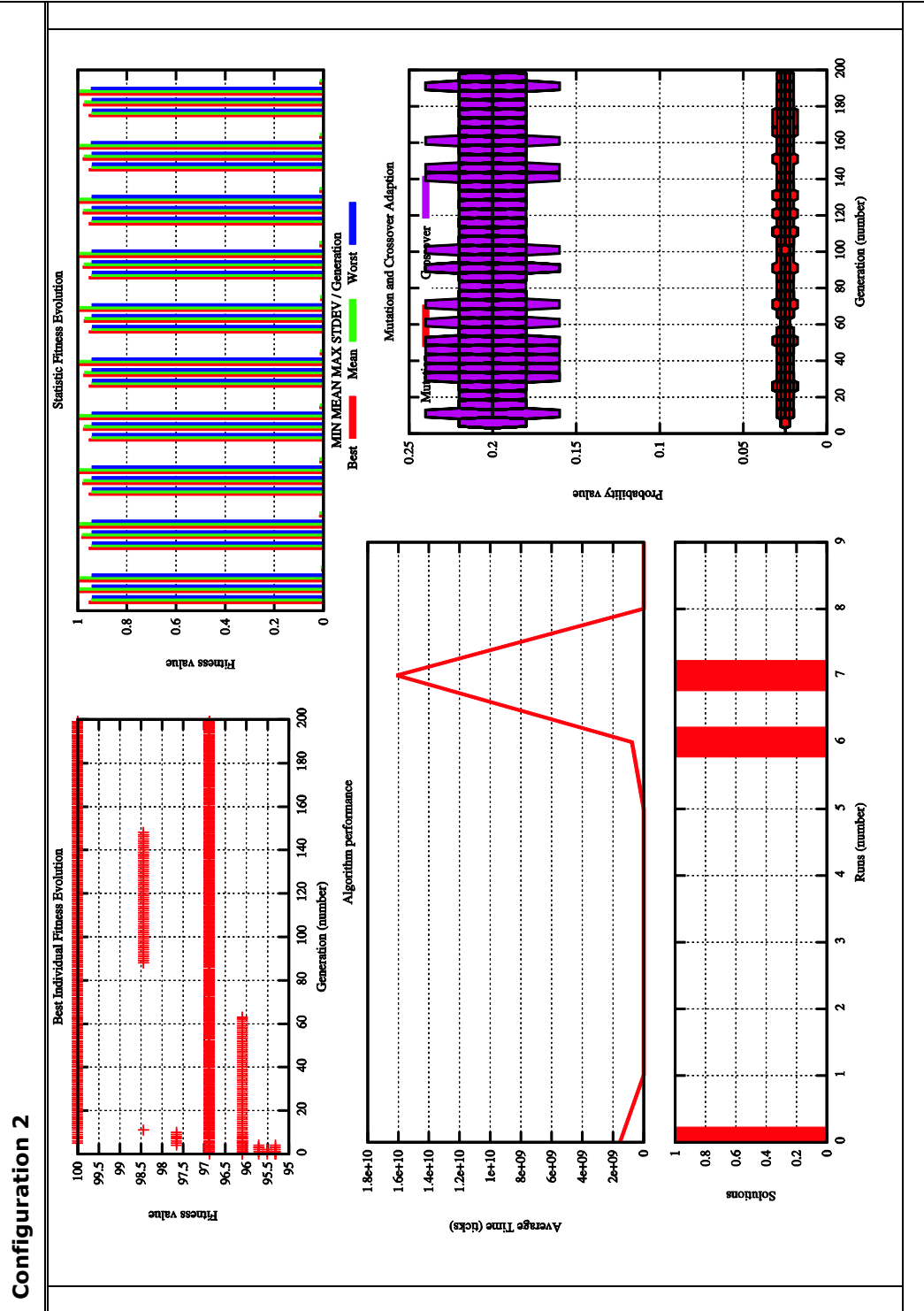
## 6.4.1 Five-Qubit Circuit

<b>Benchmark:</b> xor5 [51]			
<b>Gate count</b>	4	<b>Cost</b>	4
		<b>Garbage</b>	4
Its output is EXOR of all its variables.			
<b>Configuration</b>	<b>1</b>	<b>2</b>	<b>3</b>
Number of Generations	200		
Population Size	150		
Elitism percent	0.05	0.1	0.05
Crossover Type	Two Points		
Mutation Type	Multiple		
Crossover Probability	0.3	0.4	0.25
Mutation Probability	0.1	0.05	0.05
Selection Type	RouletteWheel	Rank	RouletteWheel
Performance Statistic	Available		
Meta Heuristic	Available		NA
Adaptive Increase	0.2	0.1	NA
Adaptive Decrease	0.1		NA
Gate Set	Reduced		
Genetic Algorithm Type	Non Overlapping		
Random Generator	Ran2		
Algorithm Runs	10		
<b>Solution (average) Clocks</b>	9.74e+09	6.12e+09	7.55e+09

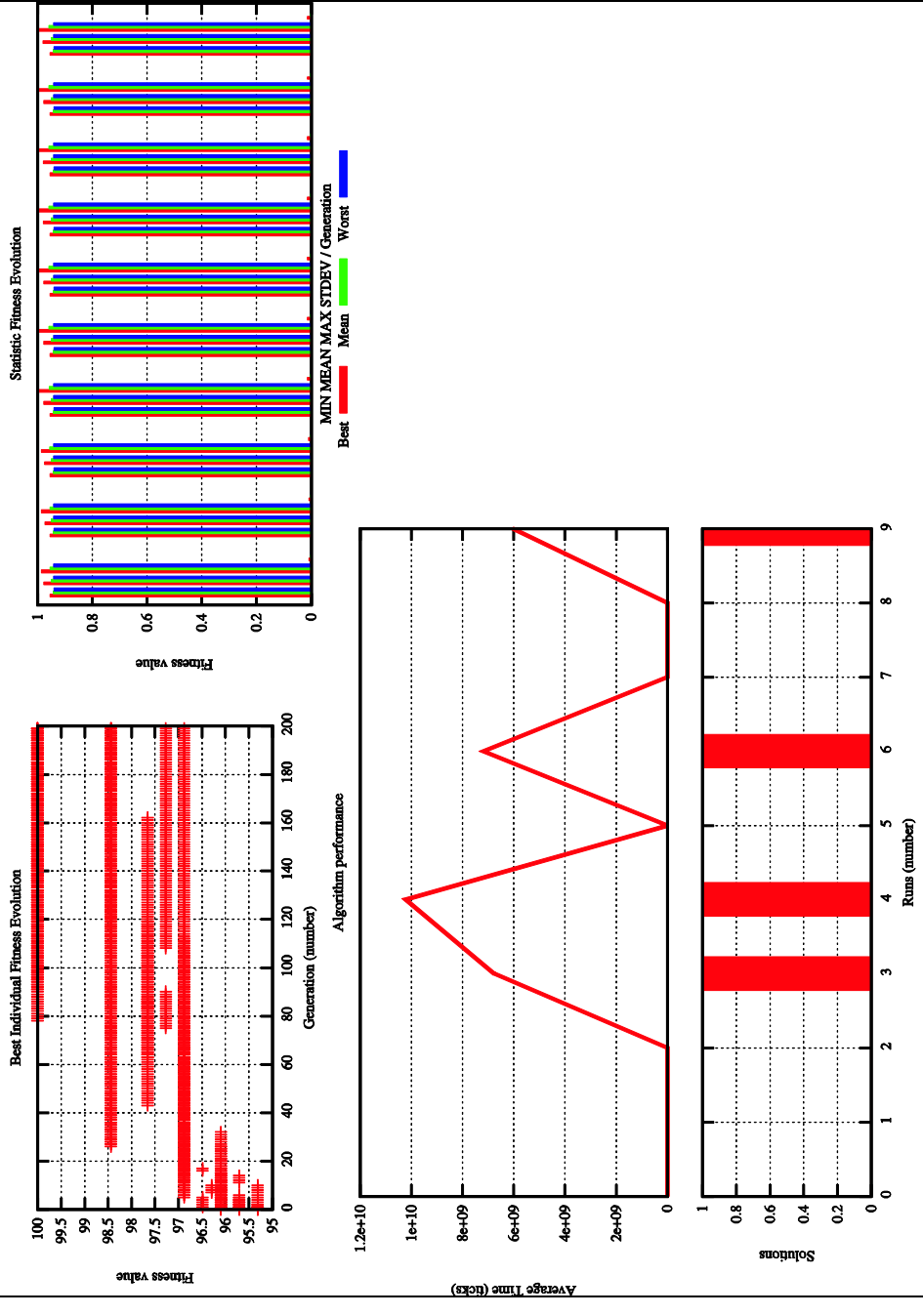
Statistic Results for the Evolved Solutions

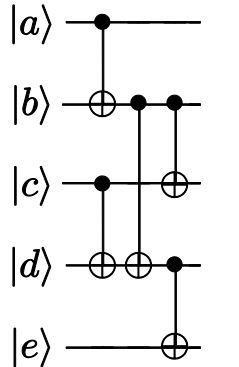
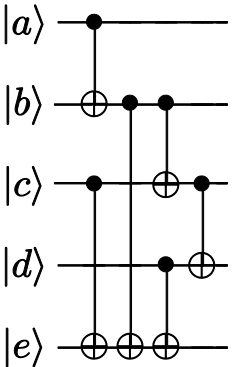
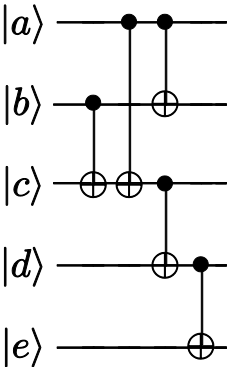
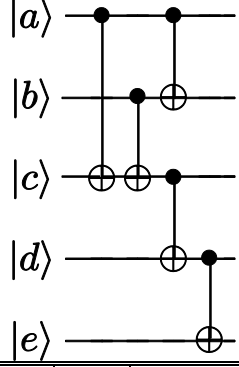
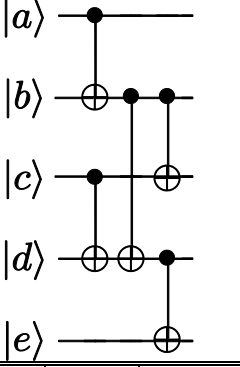
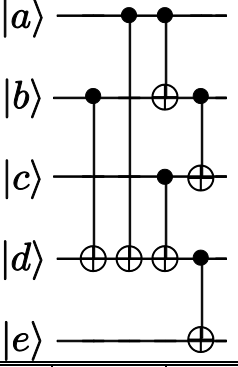
Configuration 1





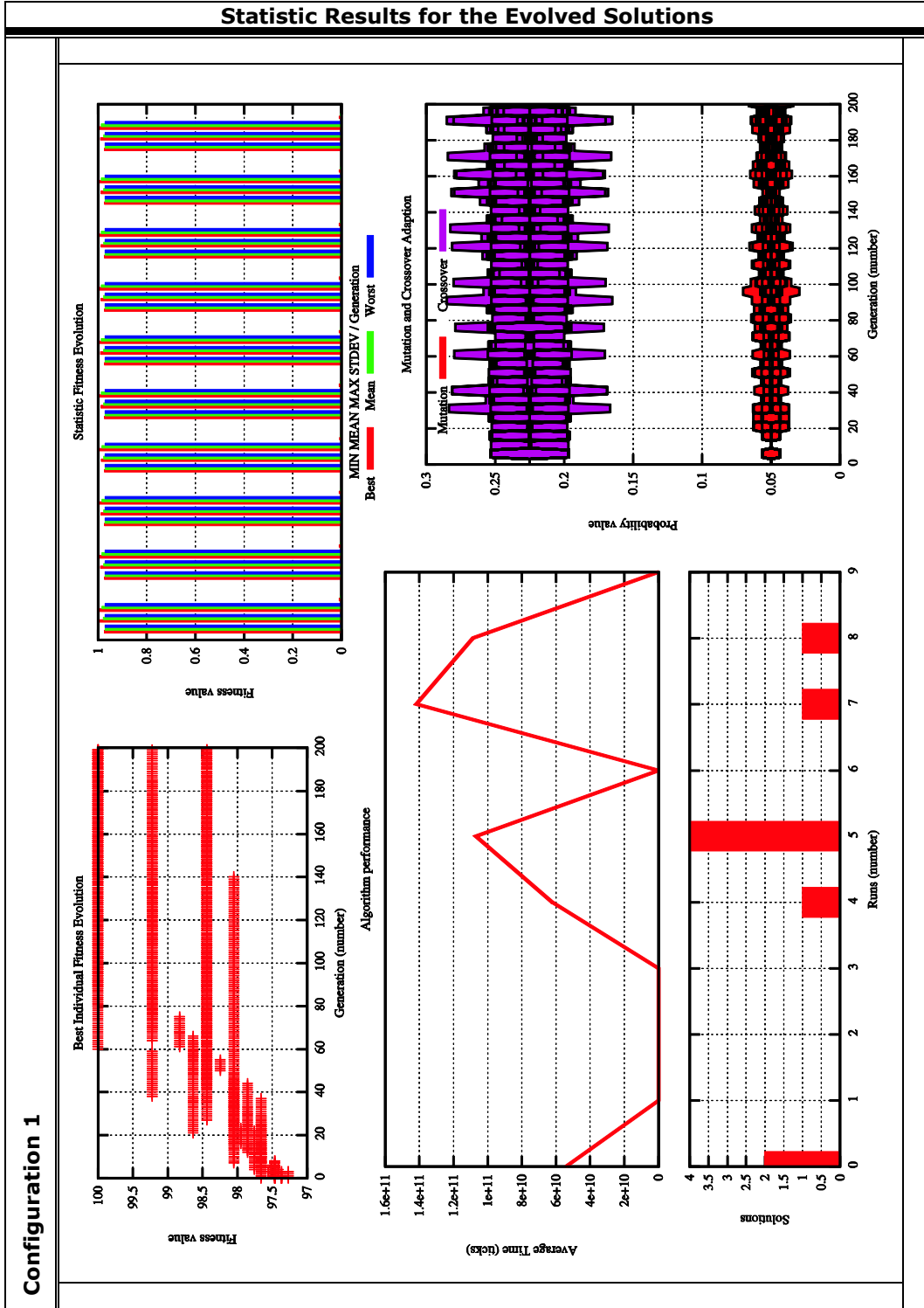
**Configuration 3**



Circuits Evolved Solutions										
Configuration 1/2/3										
	<b>1</b>	<b>2</b>	<b>3</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>1</b>	<b>2</b>	<b>3</b>	
	5	11	93.33	6	15	86.11	5	11	9.33	
										
	<b>1</b>	<b>2</b>	<b>3</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>1</b>	<b>2</b>	<b>3</b>	
	5	11	93.33	5	11	93.33	6	15	86.11	
	Note: 1 = Gates count, 2 = Circuit Cost*, 3 = Circuit Feasibility* * see their definition in 8.5.1 (Cost Details using General Approach) section									

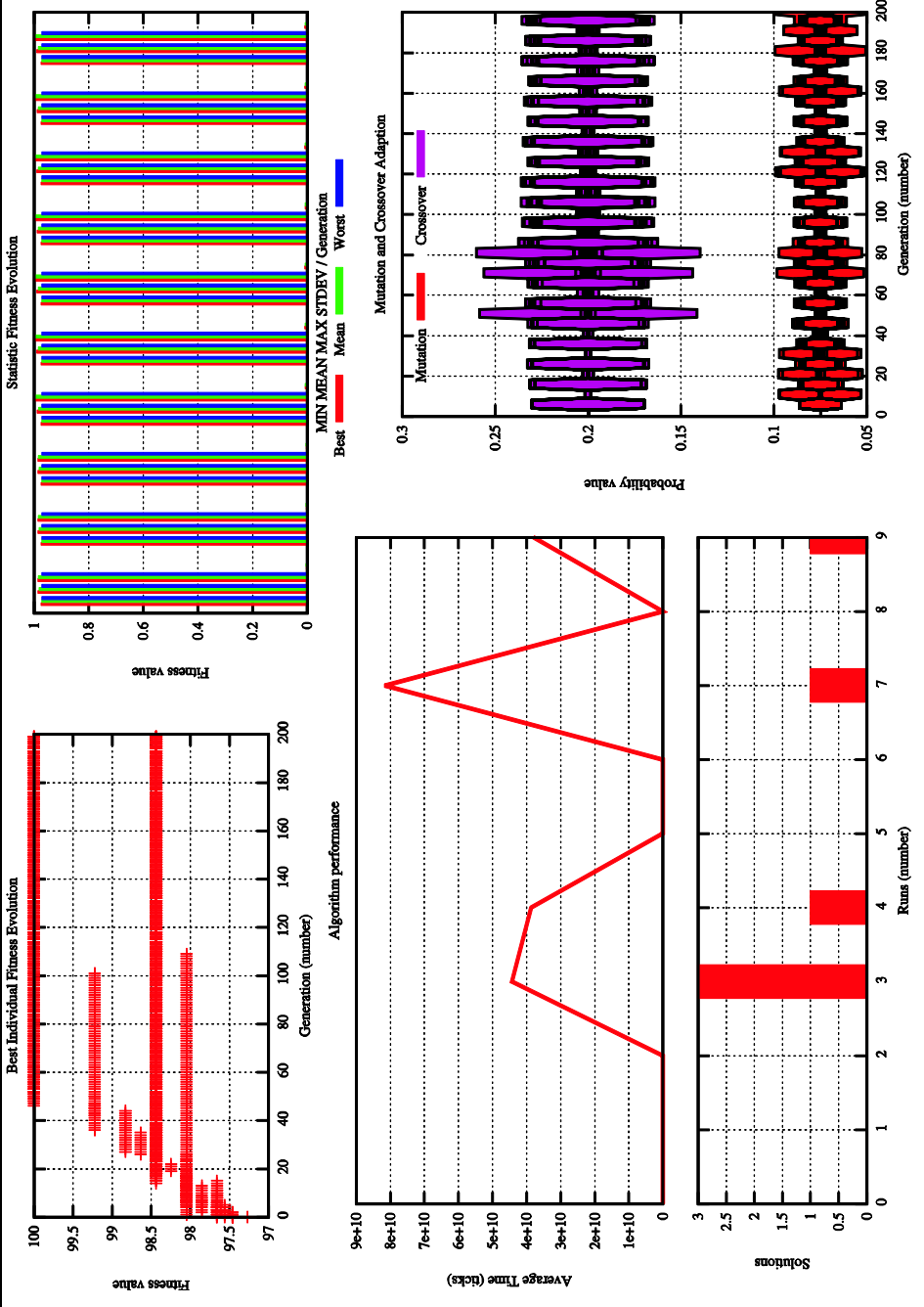
6.4.3 Six-Qubit Circuit

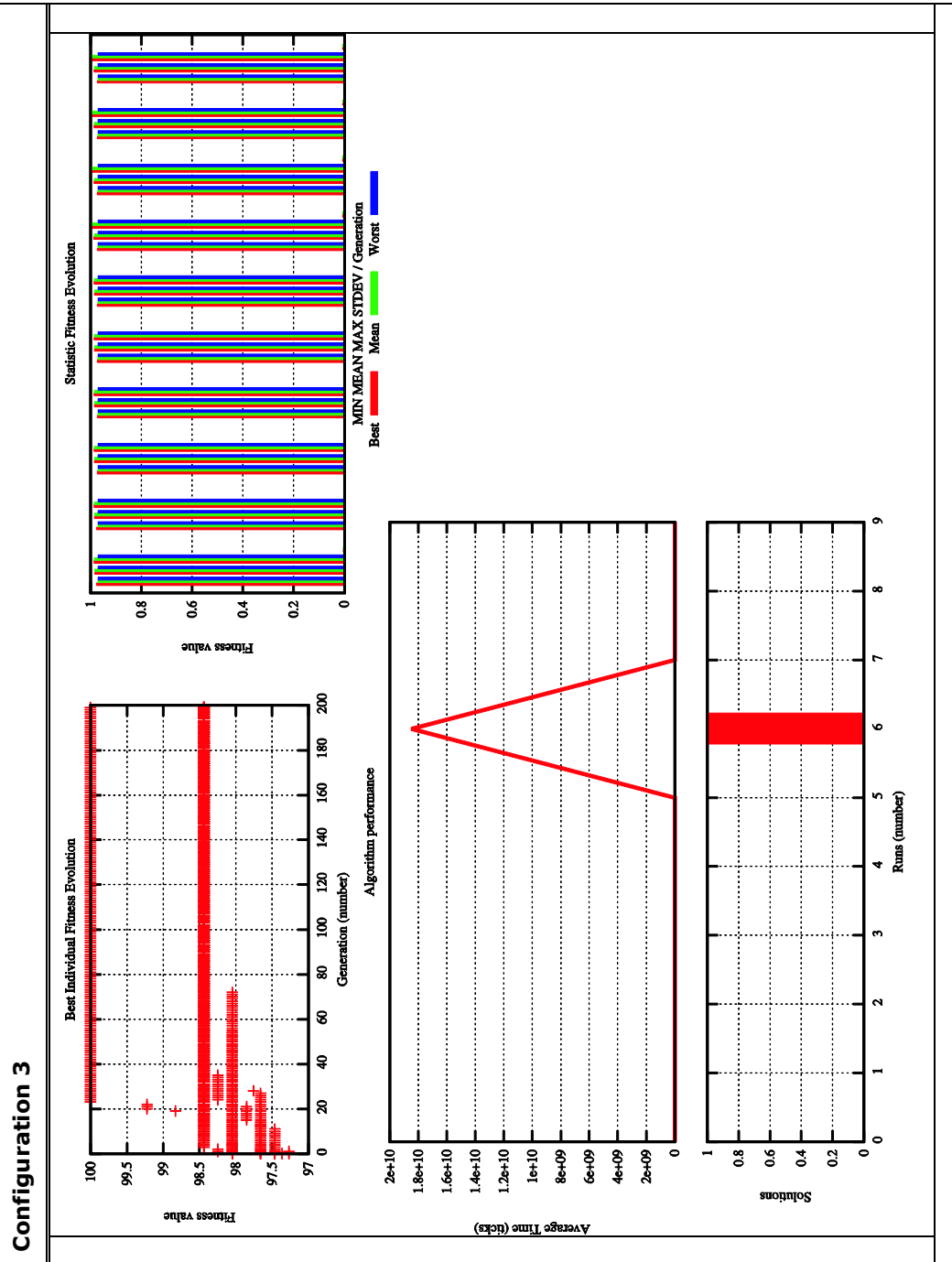
<b>Benchmark:</b> gray code [51]					
<b>Gate count</b>	5	<b>Cost</b>	5	<b>Garbage</b>	0
If the circuit for such function is run in reverse, the output is the ordinal number of the corresponding Gray code pattern.					
<b>Configuration</b>	<b>1</b>	<b>2</b>	<b>3</b>		
Number of Generations	200				
Population Size	150				
Elitism percent	0.1	0.15	0.1		
Crossover Type	One Point	Two Points	One Point		
Mutation Type	Singular			Multiple	
Crossover Probability	0.45	0.4			
Mutation Probability	0.1	0.15	0.1		
Selection Type	Uniform	Tournament	Rank		
Performance Statistic	Available				
Meta Heuristic	Available			NA	
Adaptive Increase	0.15	0.1	NA		
Adaptive Decrease	0.1	0.2	NA		
Gate Set	Reduced				
Genetic Algorithm Type	Non Overlapping				
Random Generator	Ran2				
Algorithm Runs	10				
<b>Solution Runtime (average) Clocks</b>	9.74e+10	5.06e+10	1.85e+10		





Configuration 2





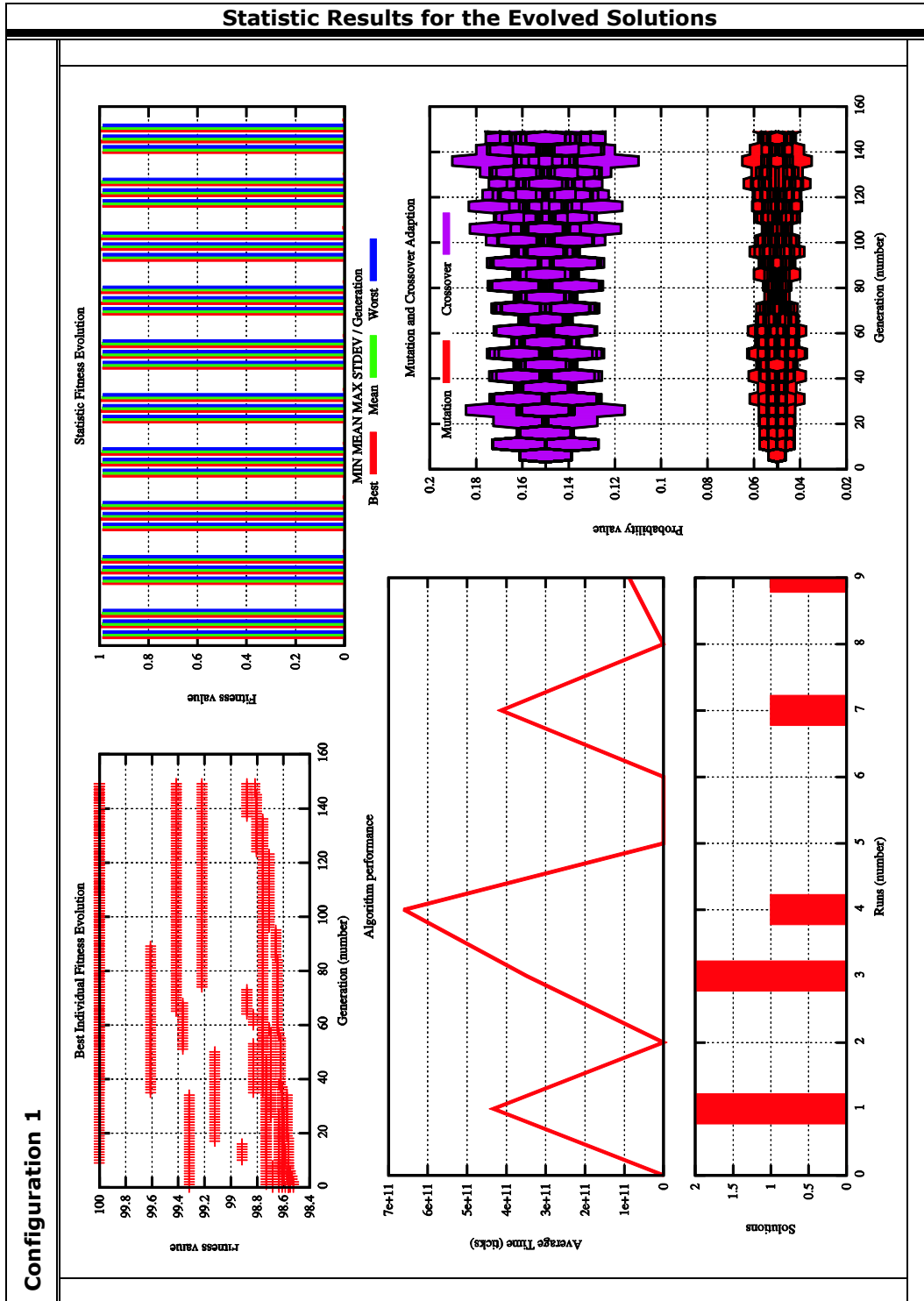
Circuits Evolved Solutions									
Configuration 1/2/3									
	<b>1</b>	<b>2</b>	<b>3</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>1</b>	<b>2</b>	<b>3</b>
	7	16	90.47	6	13	94.44	10	23	96.66
	<b>1</b>	<b>2</b>	<b>3</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>1</b>	<b>2</b>	<b>3</b>
	9	20	100	8	19	95.83	8	17	95.83

140 | 6-Experiments Result Evaluation

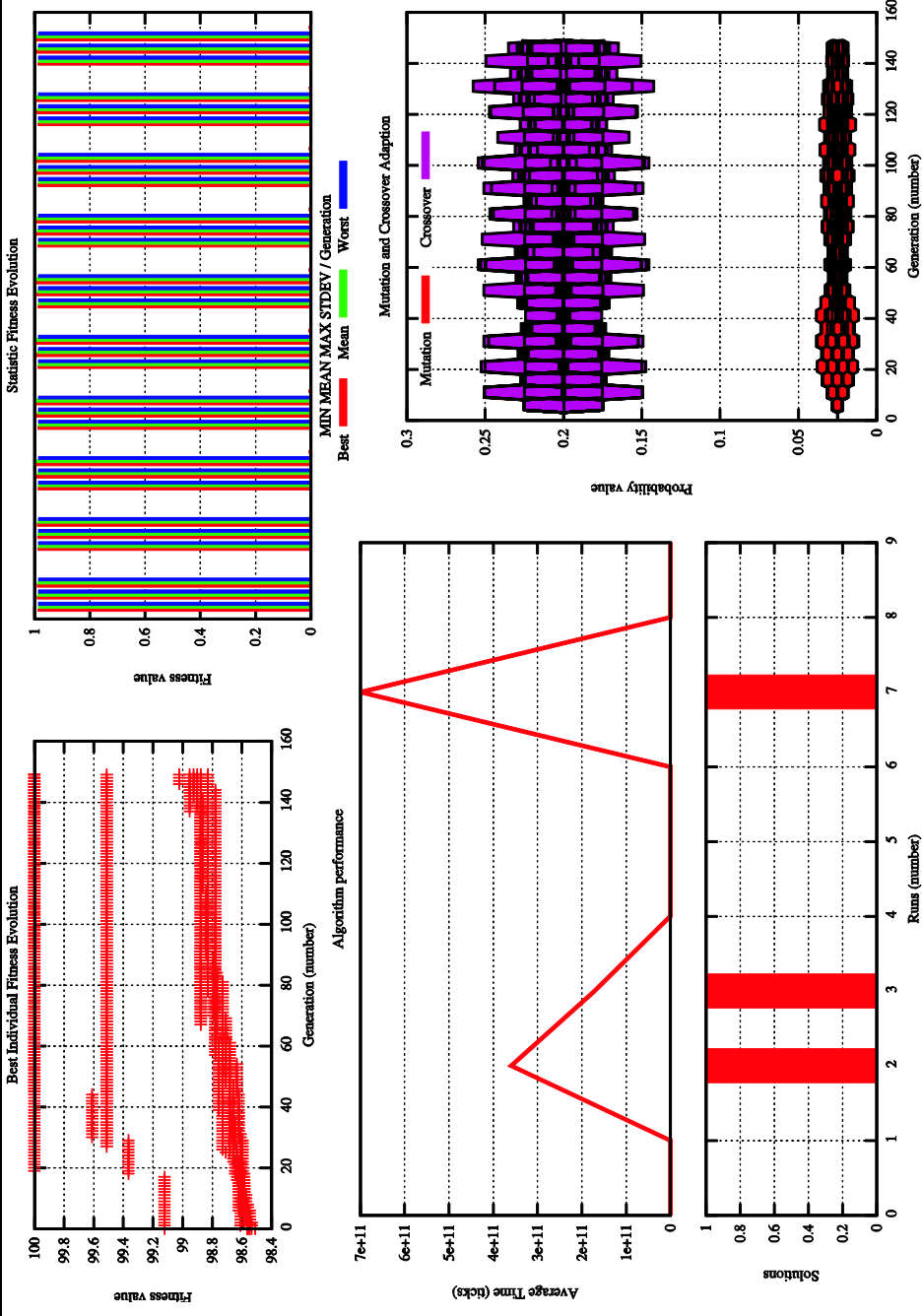
<b>1</b>	<b>2</b>	<b>3</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>1</b>	<b>2</b>	<b>3</b>
6	13	94.44	8	17	95.83	7	17	88.09
<b>1</b>	<b>2</b>	<b>3</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>1</b>	<b>2</b>	<b>3</b>
9	20	92.59	8	20	85.41			
<p>Note: 1 = Gates count, 2 = Circuit Cost*, 3 = Circuit Feasibility*            * see their definition in 8.5.1 (Cost Details using General Approach) section</p>								

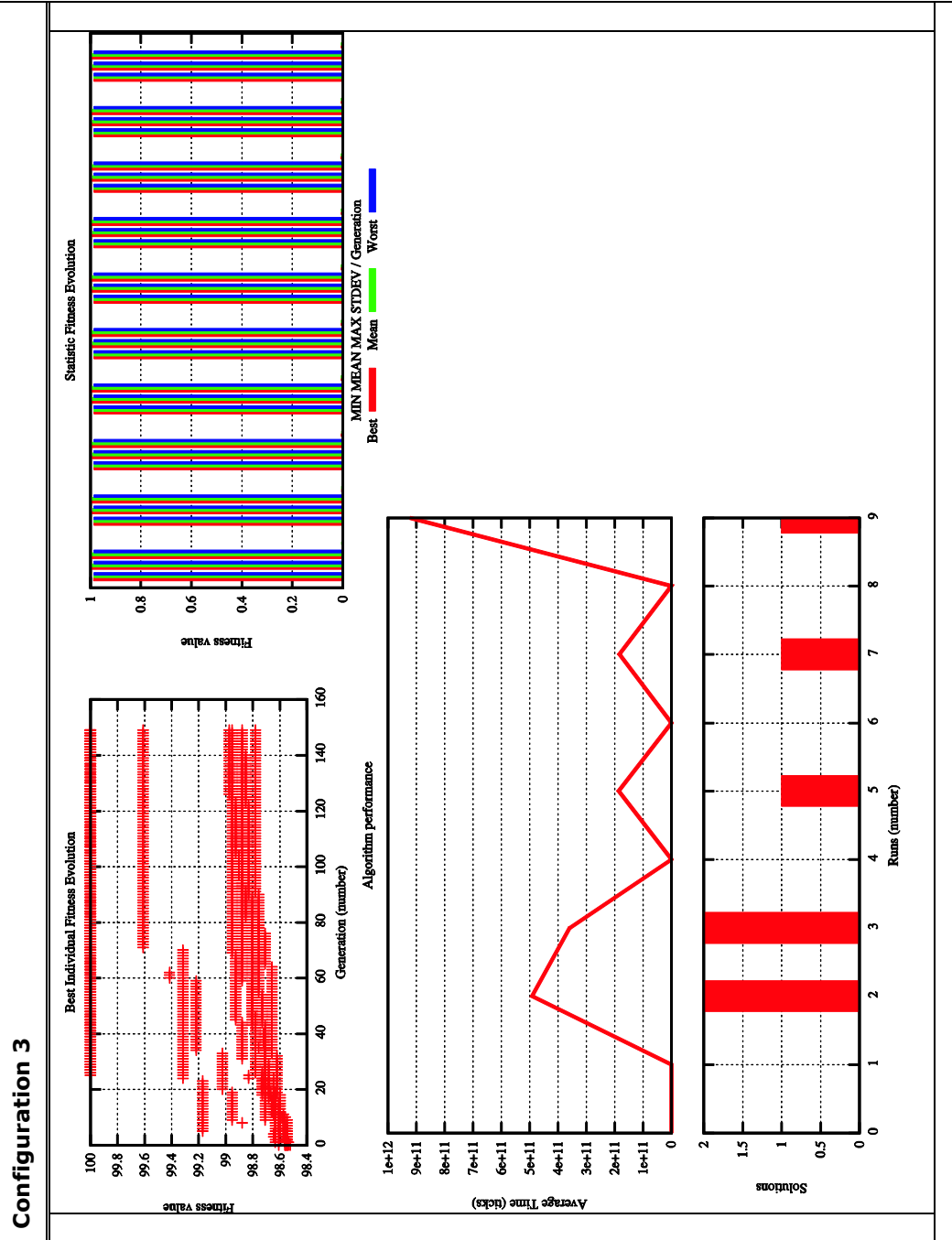
6.4.5 Seven-Qubit Circuit

<b>Benchmark:</b> 2of5 [51]					
<b>Gate count</b>	12	<b>Cost</b>	32	<b>Garbage</b>	6
Its output is 1 if number of 1s in its input equals two.					
<b>Configuration</b>	<b>1</b>	<b>2</b>	<b>3</b>		
Number of Generations	150				
Population Size	100				
Elitism percent	0.1	0.15	0.05		
Crossover Type	One Point	Two Points	One Point		
Mutation Type	Multiple	Singular	Multiple		
Crossover Probability	0.3	0.4	0.3		
Mutation Probability	0.1	0.05	0.04		
Selection Type	RouletteWheel	Uniform	Tournament		
Performance Statistic	Available				
Meta Heuristic	Available			NA	
Adaptive Increase	0.1	0.15	NA		
Adaptive Decrease	0.05	0.1	NA		
Gate Set	Reduced				
Genetic Algorithm Type	Non Overlapping				
Random Generator	Ran2				
Algorithm Runs	10				
<b>Solution Runtime (average) Clocks</b>	3.88e+11	4.11e+11	4.28e+11		



Configuration 2







<b>Circuits Evolved Solutions</b>							
<b>Configuration 1/2/3</b>							
	<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 33%; text-align: center;"><b>1</b></td> <td style="width: 33%; text-align: center;"><b>2</b></td> <td style="width: 33%; text-align: center;"><b>3</b></td> </tr> <tr> <td style="text-align: center;">12</td> <td style="text-align: center;">40</td> <td style="text-align: center;">85</td> </tr> </table>	<b>1</b>	<b>2</b>	<b>3</b>	12	40	85
	<b>1</b>	<b>2</b>	<b>3</b>				
	12	40	85				
<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 33%; text-align: center;"><b>1</b></td> <td style="width: 33%; text-align: center;"><b>2</b></td> <td style="width: 33%; text-align: center;"><b>3</b></td> </tr> <tr> <td style="text-align: center;">14</td> <td style="text-align: center;">44</td> <td style="text-align: center;">87.14</td> </tr> </table>	<b>1</b>	<b>2</b>	<b>3</b>	14	44	87.14	
<b>1</b>	<b>2</b>	<b>3</b>					
14	44	87.14					

Configuration 1/2/3			
	<b>1</b>	<b>2</b>	<b>3</b>
	14	46	87.14
Configuration 1/2/3			
	<b>1</b>	<b>2</b>	<b>3</b>
	13	43	84.23

Configuration 1/2/3			
	<b>1</b>	<b>2</b>	<b>3</b>
	13	43	83.58
<p>Note: 1 = Gates count, 2 = Circuit Cost*, 3 = Circuit Feasibility*          * see their definition in 8.5.1 (Cost Details using General Approach) section</p>			

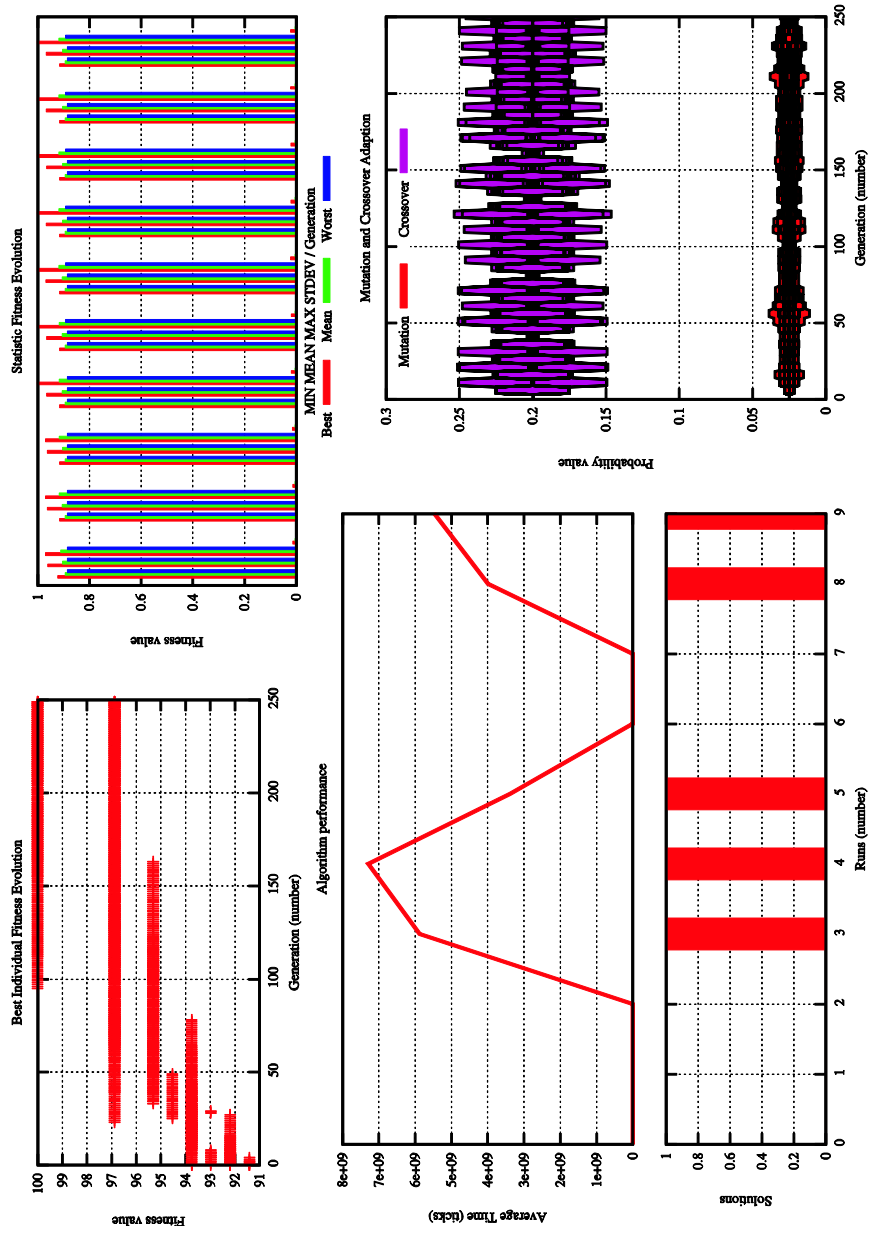
## 6.5 Additional Experiments

Each case study is started with a quantum circuit that is used for the synthesis algorithm evaluation. Only one synthesis configuration is used to evolve a synthesis solution, the adaptive behavior will dynamically adjust the mutation and the crossover probabilities, and the rest of the parameters will remain fixed during the evolution, while the results are presented as graphs.

<b>Circuit from reference [129], proposed by M. Mohammadi in 2008</b>	
<b>Configuration</b>	<b>1</b>
Number of Generations	250
Population Size	150
Elitism percent	0.05
Crossover Type	Two Points
Mutation Type	Multiple
Crossover Probability	0.4
Mutation Probability	0.05
Selection Type	RouletteWheel
Performance Statistic	Available
Meta Heuristic	Available
Adaptive Increase	0.15
Adaptive Decrease	0.1
Gate Set	Reduced
Genetic Algorithm Type	Non Overlapping
Random Generator	Ran2
Algorithm Runs	10
<b>Solution Runtime (average) Clocks</b>	5.20e+09

Statistic Results for the Evolved Solutions

Configuration 1



150 | 6-Experiments Result Evaluation

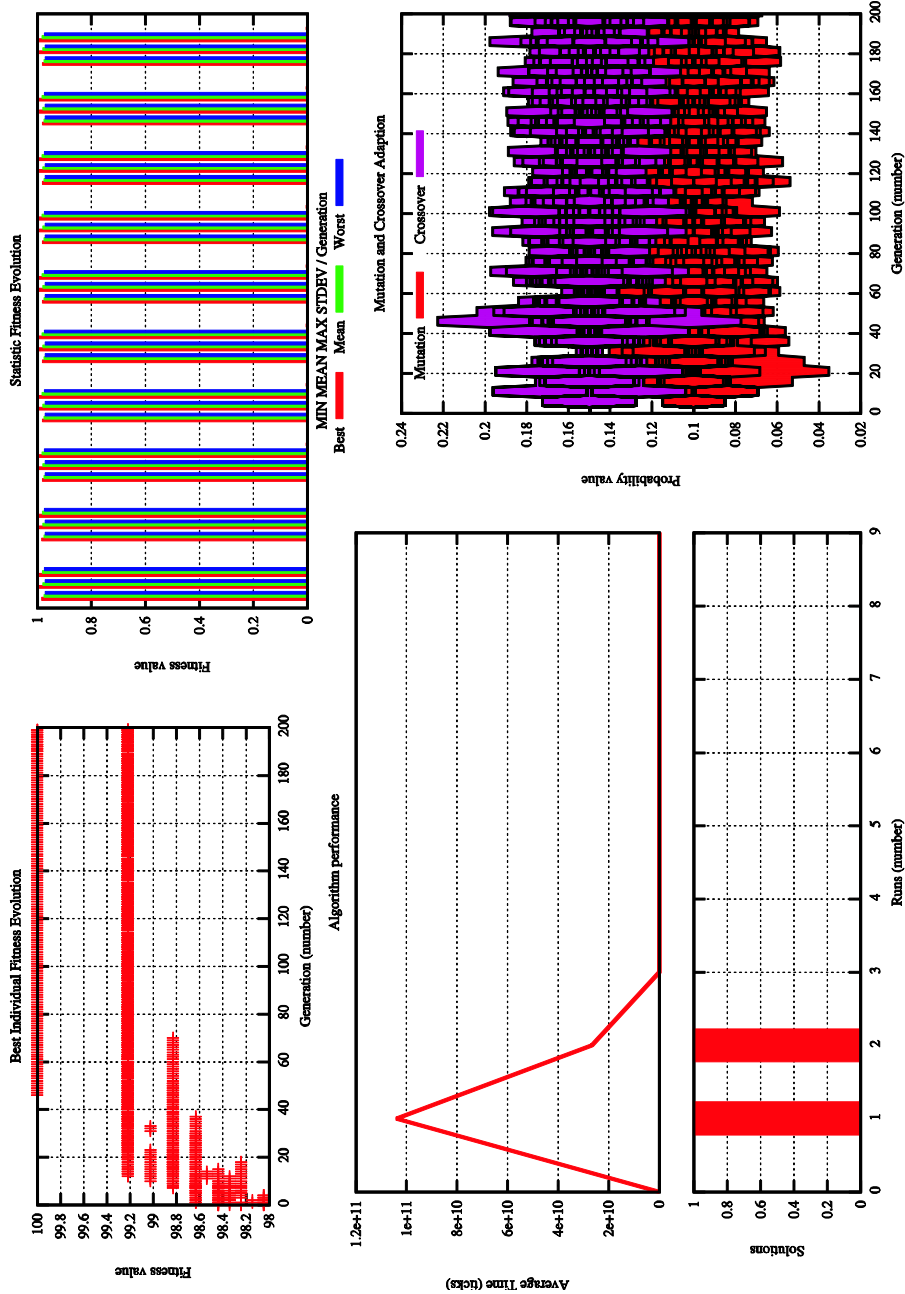
Circuits Evolved Solutions									
Configuration 1									
	<b>1</b>	<b>2</b>	<b>3</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>1</b>	<b>2</b>	<b>3</b>
	7	20	91.66	6	18	90.27			
<p>Note: 1 = Gates count, 2 = Circuit Cost*, 3 = Circuit Feasibility*</p> <p>* see their definition in 8.5.1 (Cost Details using General Approach) section</p>									

6.5 -Additional Experiments|151

Circuit from reference [130] proposed by Van Meter et al. in 2007	
Configuration	1
Number of Generations	200
Population Size	150
Elitism percent	0.1
Crossover Type	Two Points
Mutation Type	Singular
Crossover Probability	0.3
Mutation Probability	0.2
Selection Type	RouletteWheel
Performance Statistic	Available
Meta Heuristic	Available
Adaptive Increase	0.2
Adaptive Decrease	0.1
Gate Set	Reduced
Genetic Algorithm Type	Non Overlapping
Random Generator	Ran2
Algorithm Runs	10
Solution Runtime (average) Clocks	6.54e+10

Statistic Results for the Evolved Solutions

Configuration 1





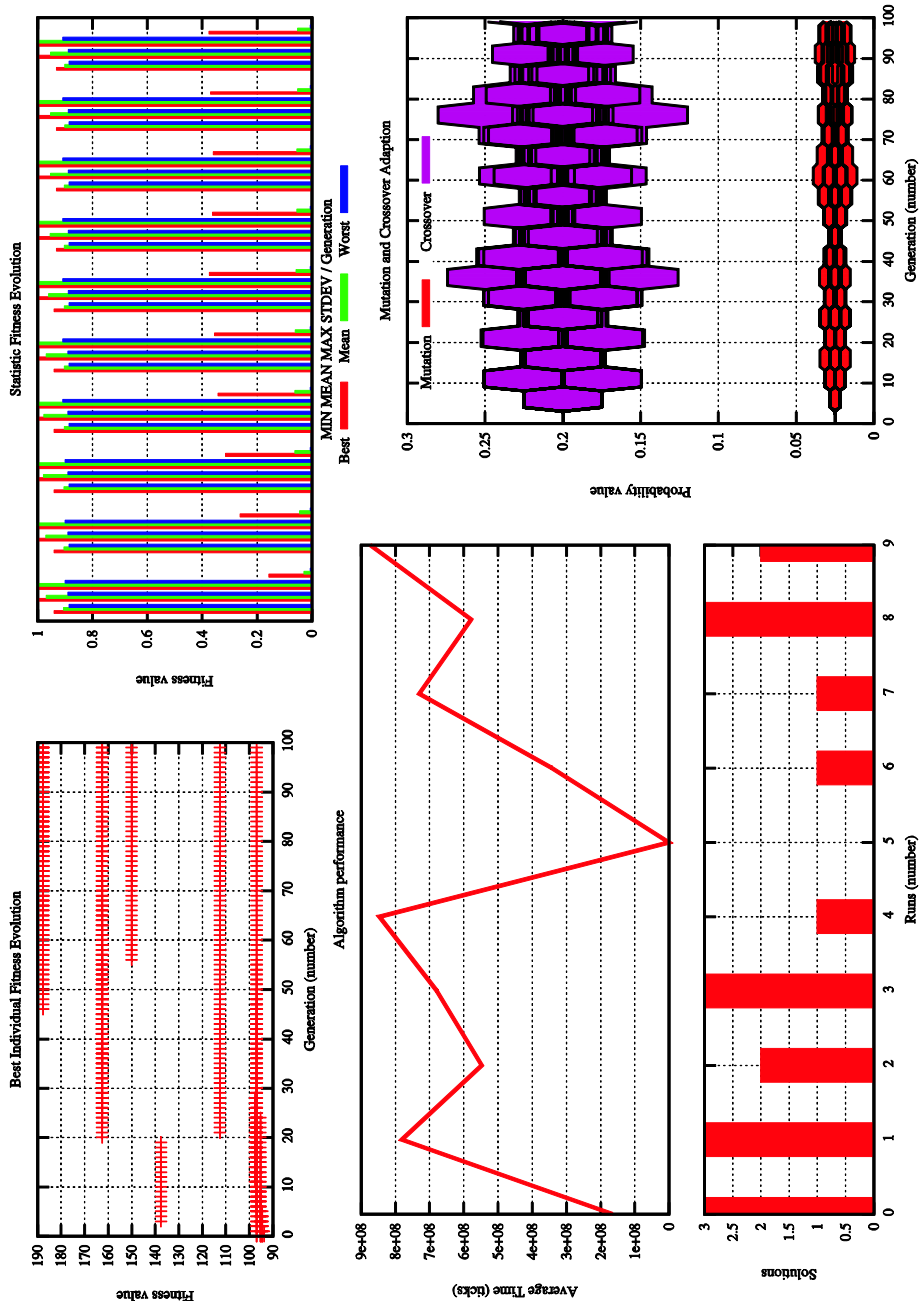
Circuits Evolved Solutions										
<b>Configuration 1</b>										
	<b>1</b>	<b>2</b>	<b>3</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>1</b>	<b>2</b>	<b>3</b>	
	4	14	93.75							
Note: 1 = Gates count, 2 = Circuit Cost*, 3 = Circuit Feasibility* * see their definition in 8.5.1 (Cost Details using General Approach) section										

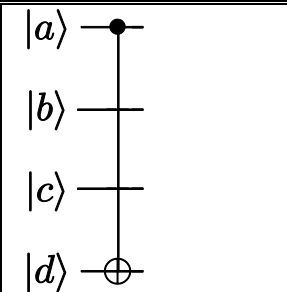
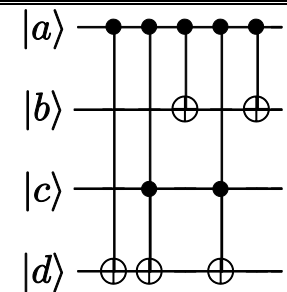
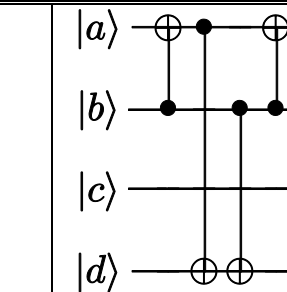
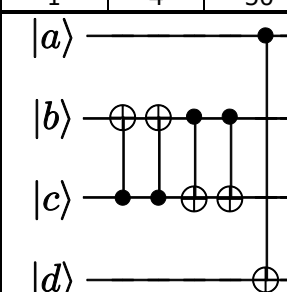
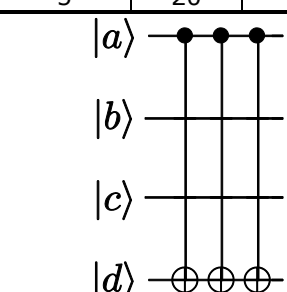
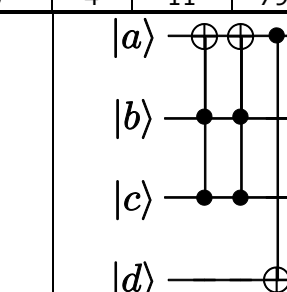
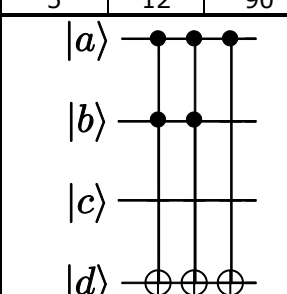
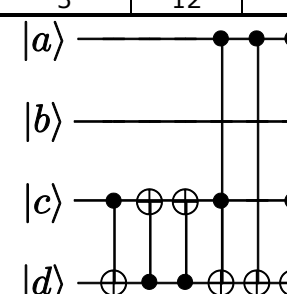
154 | 6-Experiments Result Evaluation

Circuit from reference [131] proposed by Shende et. al in 2006	
Initial	Evolved
Configuration	1
Number of Generations	100
Population Size	50
Elitism percent	0.05
Crossover Type	Two Points
Mutation Type	Multiple
Crossover Probability	0.4
Mutation Probability	0.05
Selection Type	RouletteWheel
Performance Statistic	Available
Meta Heuristic	Available
Adaptive Increase	0.15
Adaptive Decrease	0.1
Gate Set	Reduced
Genetic Algorithm Type	Non Overlapping
Random Generator	Ran2
Algorithm Runs	10
Solution Runtime (average) Clocks	6.16e+08

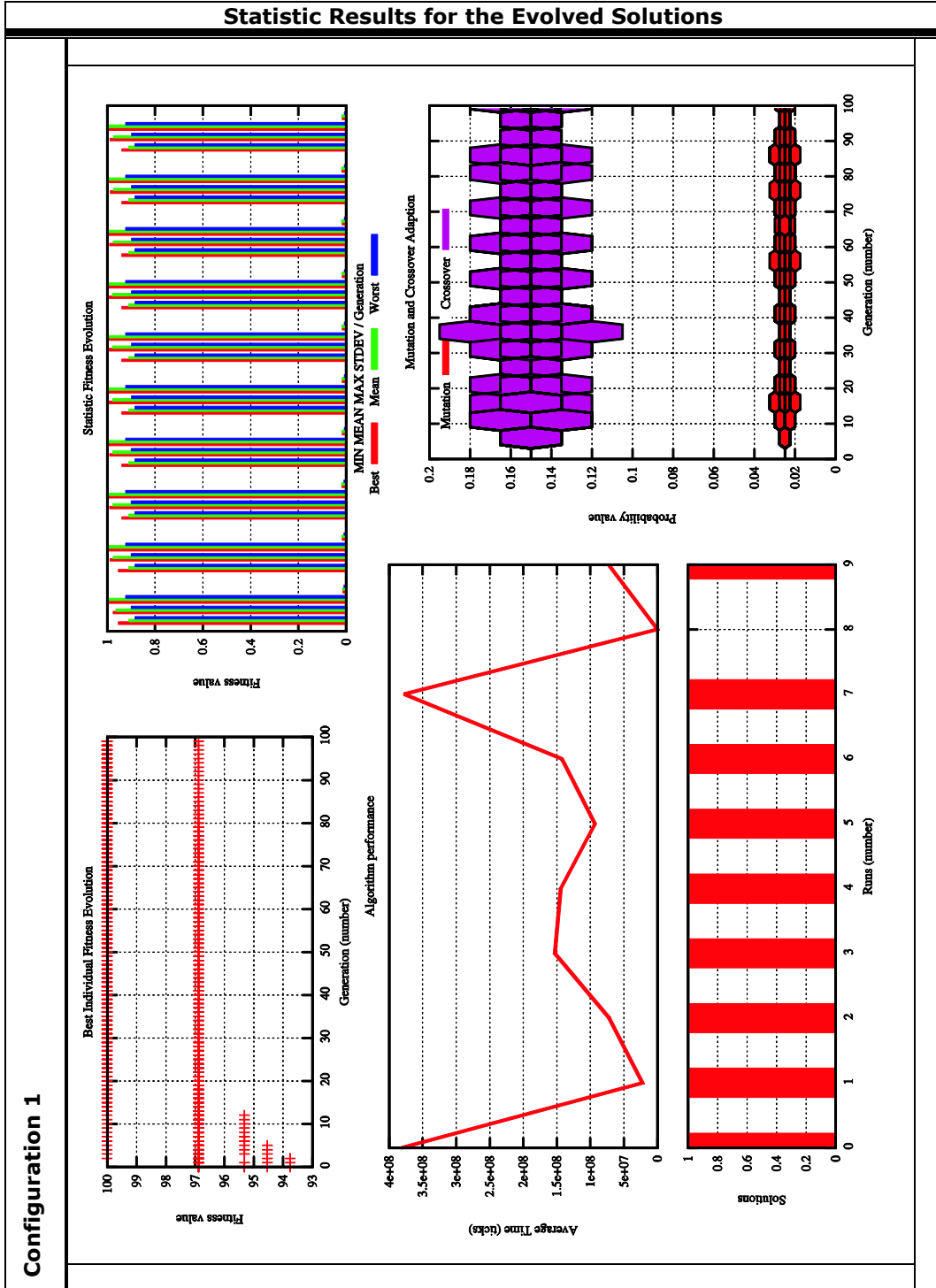
Statistic Results for the Evolved Solutions

Configuration 1



Circuits Evolved Solutions									
Configuration 1									
	<b>1</b>	<b>2</b>	<b>3</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>1</b>	<b>2</b>	<b>3</b>
	1	4	50	5	20	80	4	11	79.16
									
	<b>1</b>	<b>2</b>	<b>3</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>1</b>	<b>2</b>	<b>3</b>
	5	12	90	3	12	50	3	10	83.33
									
	<b>1</b>	<b>2</b>	<b>3</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>1</b>	<b>2</b>	<b>3</b>
	3	10	66.66	7	24	85.71	3	8	83.33
	Note: 1 = Gates count, 2 = Circuit Cost*, 3 = Circuit Feasibility* * see their definition in 8.5.1 (Cost Details using General Approach) section								

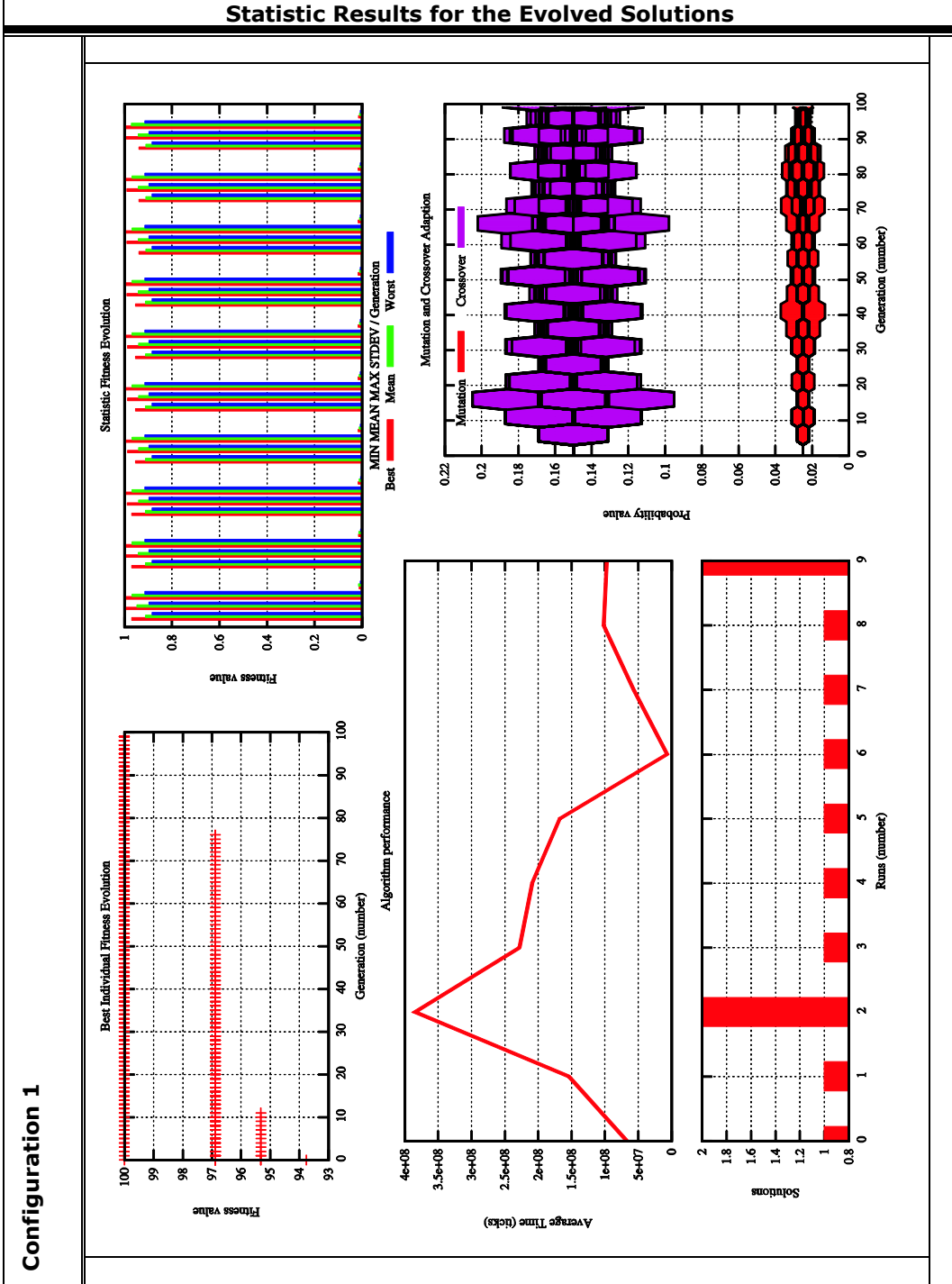
<b>Circuit from reference [47] proposed by V. Shende et al. in 2003</b>	
Initial	Evolved
<b>Configuration</b>	<b>1</b>
Number of Generations	100
Population Size	50
Elitism percent	0.05
Crossover Type	One Point
Mutation Type	Singular
Crossover Probability	0.3
Mutation Probability	0.05
Selection Type	Rank
Performance Statistic	Available
Meta Heuristic	Available
Adaptive Increase	0.1
Adaptive Decrease	0.1
Gate Set	Reduced
Genetic Algorithm Type	Non Overlapping
Random Generator	Ran2
Algorithm Runs	10
<b>Solution (average) Runtime (Clocks)</b>	1.62e+08



Circuits Evolved Solutions									
<b>Configuration 1</b>									
	<b>1</b>	<b>2</b>	<b>3</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>1</b>	<b>2</b>	<b>3</b>
	2	7	75	3	10	83.33			
Note: 1 = Gates count, 2 = Circuit Cost*, 3 = Circuit Feasibility* * see their definition in 8.5.1 (Cost Details using General Approach) section									

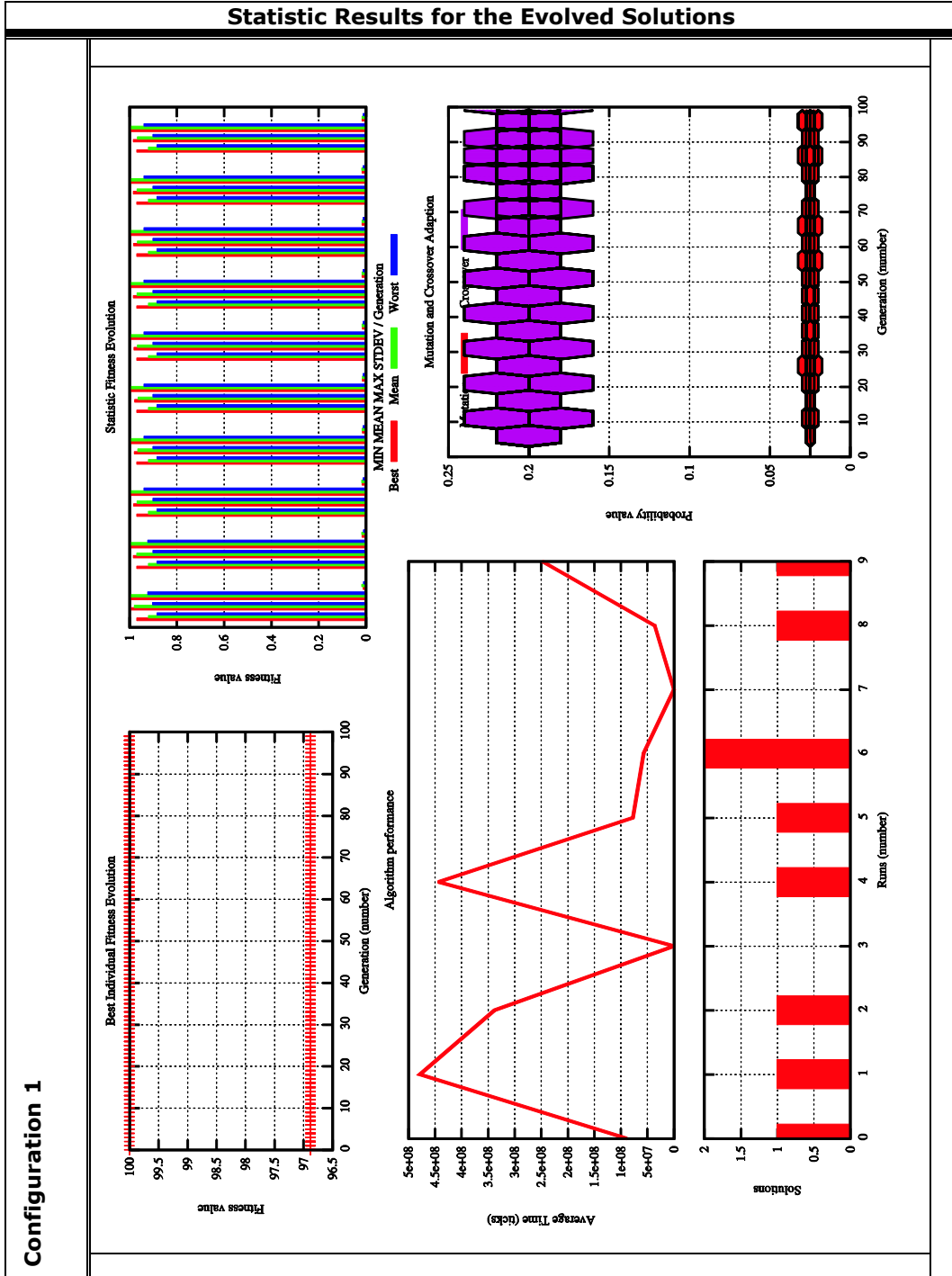
<b>Circuit from reference [47] proposed by V. Shende et al. in 2003</b>	
Initial	Evolved
<b>Configuration</b>	<b>1</b>
Number of Generations	100
Population Size	50
Elitism percent	0.1
Crossover Type	One Point
Mutation Type	Multiple
Crossover Probability	0.3
Mutation Probability	0.05
Selection Type	Uniform
Performance Statistic	Available
Meta Heuristic	Available
Adaptive Increase	0.1
Adaptive Decrease	0.15
Gate Set	Reduced
Genetic Algorithm Type	Non Overlapping
Random Generator	Ran2
Algorithm Runs	10
<b>Solution Runtime (average) Clocks</b>	1.47e+08





Circuits Evolved Solutions									
Configuration 1									
	<b>1</b>	<b>2</b>	<b>3</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>1</b>	<b>2</b>	<b>3</b>
	2	5	100	3	11	91.66	4	9	100
Note: 1 = Gates count, 2 = Circuit Cost*, 3 = Circuit Feasibility* * see their definition in 8.5.1 (Cost Details using General Approach) section									

<b>Circuit from reference [47] proposed by V. Shende et al. in 2003</b>	
Initial	Evolved
<b>Configuration</b>	<b>1</b>
Number of Generations	100
Population Size	50
Elitism percent	0.05
Crossover Type	One Point
Mutation Type	Singular
Crossover Probability	0.4
Mutation Probability	0.05
Selection Type	Tournament
Performance Statistic	Available
Meta Heuristic	Available
Adaptive Increase	0.1
Adaptive Decrease	0.1
Gate Set	Reduced
Genetic Algorithm Type	Non Overlapping
Random Generator	Ran2
Algorithm Runs	10
<b>Solution Runtime (average) Clocks</b>	2.21e+08



Circuits Evolved Solutions									
Configuration 1									
	<b>1</b>	<b>2</b>	<b>3</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>1</b>	<b>2</b>	<b>3</b>
	2	5	100	2	5	100	4	9	100
	Note: 1 = Gates count, 2 = Circuit Cost*, 3 = Circuit Feasibility* * see their definition in 8.5.1 (Cost Details using General Approach) section								

## **6.6 Result Evaluation**

Summarizing our results for these test cases (whose characteristics are considered to inherit the smallest number of gates necessary in order to implement the required function – as being benchmark circuits), we note that our quantum synthesis tool always converge towards a solution. In addition, the meta-heuristic approach, when used for configuration 1 and 2, prove its benefit in evolving a faster convergence by increasing the operators probability, when better offsprings are evolved. Others experimental results prove that the roulette wheel selection has generated an increased number of synthesis results in comparison with other selection methods. Although not obvious at first glance, at the same time with increasing the number of circuit qubits, more generations are necessary in order to evolve synthesis solutions (it is also worth mentioning, that an increased number of individuals is not necessary). The experimental results consider significantly more difficult circuits than other evolutionary programming approaches and provide a stable base for further quantitative comparison (i.e. feasibility, cost assessment and runtime values).

The experimental results were obtained in a relative short time period by using an ordinary computer. Synthesis results for bigger quantum circuits cannot be reported here due to computational resource limitation, mainly runtime (we design our synthesis system to allow a dynamic defined number of qubits and circuits). In order to be relevant, experimental results need to be performed on better computer systems (i.e. John Koza performed genetic algorithms employing 1000 parallel computers in Mountain View, California, and the results were patented as six electronic circuits<sup>18</sup>).

## **6.7 Going Beyond 7-qubit Circuits**

Prashant, in reference [128], has described a genetic algorithm used to evolve quantum circuits (it was proposed in Nov. 2005 and revised in Jan. 2007). The genetic algorithm automatically searches for the appropriate circuit design that yields the desired output state. The fitness function evaluates the current output with the desired output, the search being stopped when a close match is found. The experimental results are presented only for 2-qubit systems and only a few quantum gates are used. The author is suggesting that a further optimization has to be applied in order to run the algorithm for multi-qubit systems (i.e. to revise the fitness function and to include the circuit efficiency as well).

In reference [132], Shengchao et al. (Oct. 2006) has proposed a hybrid quantum evolutionary algorithm for the implementation of quantum oracles. The quantum circuits are encoded using numerical values and the fitness function consider a “cost” value for the evaluation. In the end it is specified that the presented approach is not to be considered for circuit synthesis or optimization, mainly because the experimental part consist only for 2-qubit circuits.

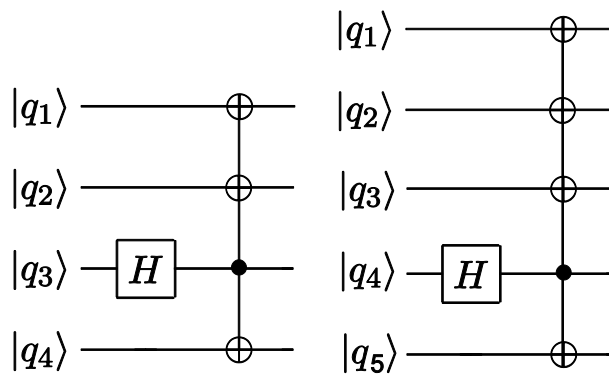
---

<sup>18</sup> <http://www.genetic-programming.com/johnkoza.html>

## 6.7 -Going Beyond 7-qubit Circuits|167

In Tim Reid's master thesis [133], which was defended in 2005, the proposed goal is to understand the application of the evolutionary approach to the quantum circuit design. A toolkit named "wabisabi" has been proposed in order to evolve basic quantum circuits such as CNOT, CPHASE, SWAP, TOFFOLI, etc. The toolkit experiments are limited to 3-qubit circuits and the symbolic circuit representation cannot be extended by using the proposed approach.

Rubinstein, in reference [46], considers for the genetic algorithm a scheme in which a gate has a type, a number of sets for the qubit operands and some sets of parameters for different categories (the generalized 2-qubit gate takes four real parameters for different types of rotations; the CNOT gate takes a number of control qubits, etc). The quantum circuit is considered as a list of gate structures, where the size of the circuit (number of gates) is variable. The proposed approach has been used to evolve the EPR (Einstein, Podolsky and Rosen) circuits with 2, 3, 4 and maximum 5 qubits (see Figure 6.1). The approach was dedicated only to EPR circuit synthesis and does not provide runtime values, and thus cannot be used for further comparisons.



**Figure 6.1 Synthesis of EPR Circuits [46]**

Other genetic algorithm based approaches present effective solution only for three or four-qubit circuits [11][46]. As stated in reference [11], the main difficulties encountered where: complexity of performing Kronecker tensor product for large matrixes, a high number of individuals used for the total population (the evolved result may be found out in less generations, but employing longer runtime of fitness evaluation), and the complexity of encoding a specific quantum gate. Our approach tackles these problems firstly by using an OOP (object-oriented programming) environment backed by a framework architecture that employs optimization techniques; this improves the effectiveness of using quantum operations (including the tensor product). Secondly, our chromosome representation and meta-heuristic approach allow for using small populations (about 300 individuals) within the genetic evolution process. Also, another improvement comes

## 168 | 6-Experiments Result Evaluation

from the fact that our approach uses a more flexible encoding scheme for the quantum gates, allowing runtime definitions. As a result, the experiments can be performed within our synthesis framework for 3 to 7 qubit circuits in a reasonable time (less than 1 minute).

The first two circuits from Figure 6.2 are used to produce the “EPR” experiment and the third is a “send” circuit used for quantum teleportation. The evolved results from reference [11] are presented in Table 10, together with our results that are highlighted in the last table column. A one-point crossover is used, and mutation can change the chromosome length. All results have been averaged over 20 runs on each gate type set used. The used configuration is: Intel Pentium M processor at 1.862GHz, 1GB RAM memory and SuSe 10.3 as operating system.

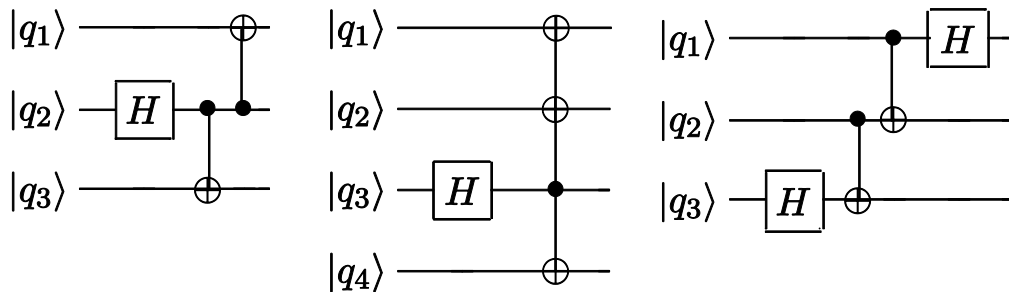


Figure 6.2 Synthesis of Composite Circuits [11]

Table 10. Test the Convergence

Number of inputs per q-gate	Number of generations	Population size	pM	pC	Real time [11] (average 20 runs)	Our real time (average 20 runs)	
						sec	clocks
3-inputs (a)	<150	50	0.4	0.6	<1min	<1	3.10e+8
3-inputs (a)	<300	50	<0.2	0.6	<2min	<1	3.96e+8
3-inputs (a)	100	50	0.1	0.3	Not reported	<1	6.91e+7
3-inputs (c)	<150	50	0.4	0.6	<1min	<1	3.68e+8
3-inputs (c)	<300	50	<0.2	0.6	<2min	<1	4.51e+8
3-inputs (c)	100	50	0.1	0.3	Not reported	<1	8.20e+7
4-inputs	<350	50	0.6	0.4	<2min	<2	2.89e+9
4-inputs	<900	50	<0.2	0.4	<3min	<2	3.41e+9
4-inputs	100	50	0.1	0.3	Not reported	<1	4.36e+8



## 6.7 -Going Beyond 7-qubit Circuits| 169

---

Therefore, our proposed genetic quantum synthesis methodology offers, at least for the considered experiments, better performance in terms of runtime (the values are presented in seconds and in processor clocks). Moreover, if the proposed number of generations is decreased (i.e. 100) we are still able to evolve solutions and the runtime becomes even better. Our motivation was to evolve more complex quantum circuits, and not the benchmark evaluation of the convergence along with the effectiveness of the genetic algorithm, although the presented results seem to outperform those obtained with the previous approaches.

Even so, attempting to perform synthesis over a larger number of qubits will also have to confront the complexity problem of matrix multiplication. However, we intend to further investigate this matter and optimize our framework, in order to extend the effectiveness of our approach for even larger quantum circuits, in order to obtain an evolved solution in a reasonable time, by using an ordinary commercial computer system.

## Chapter 7

### Conclusions and Perspectives

The pursuit for performance in computers is relentless. If in classical computers the acquired experience is vast, developed over almost half a century, in quantum computers the race has started relatively recently, in the 1980's. Even from today's view, it can be foreseen that in the next decade the quantum computer will be available. Thus, any improvement on quantum domain is important to be followed.

This PhD thesis has addressed an important aspect in quantum computing's reversible circuit logic, the automated quantum logic circuit synthesis. This research is relevant for the field, as the industry and academia are preoccupied within the quantum circuit development where the synthesis problem has the potential of playing an important role in the CAD tool evolution [16] [59][10].

In order to deal with the problems of decoherence and gate support and – therefore – bring the entire quantum computing field into a more clear view, the researchers from the physics and materials science fields still have a significant way to go. In addition, the engineers have many problems to solve, in order to bring a real quantum computer, as a super machine based on solid-states qubits with high performance, into our daily life.

#### 7.1 Thesis Impact and Contributions

This dissertation acknowledges the research directions as defined by ITRS, and presents a new quantum logic circuit synthesis methodology, based on genetic algorithms. As it is stated in the title, this thesis brings together, in an apprehensive and already verified manner, three domains: automatic synthesis, genetic algorithms and quantum computing. Following this idea, a software tool chain was developed, creating the prerequisites for automatic quantum circuit synthesis. Thus, starting with a quantum high-level circuit description and using a specific genetic algorithm configuration, the software tool will possibly provide – due to evolutionary nature of genetic algorithms - one or more evolved circuit synthesis solutions.

An improved file parser was developed for the quantum circuit blocks identification. The parser creates, in the second phase, the internal data structure that is used for circuit optimization. The optimization is made using topological placement of the recognized quantum gates. In addition, the validation of the input data is one of the main parser features.

A new genetic algorithm framework was developed, mainly, for the quantum logic circuit synthesis assessment. The framework allows genetic algorithm implementations and provides the software tools (e.g. runtime measurements, statistic data acquisition, random number generators, different selectors, etc) necessary for any algorithm evaluation. It was shown that object oriented

## 7.1 -Thesis Impact and Contributions|171

---

programming and design patterns are able to create an extensible architecture for developing new genetic solutions.

A meta-heuristic approach is presented for the automatic tuning of parameters control, as defined in a genetic algorithm that is used for the purpose of quantum logic circuits synthesis. Statistical data are saved on each generation, and then analyzed by an adaptive algorithm that dynamically adjusts the parameter control values. In addition, the methodology presented here offers a strategy that implements the Rechenberg rule and the operators performance analysis in a circuit synthesis algorithm.

A quantum circuit repository was created for providing the quantum circuit characteristics to the synthesis genetic algorithm. Within the repository, the available quantum gates characteristics are dynamically generated by following the corresponding mathematical formula. In this way, the repository will contain only valid gates for the user-defined number of qubits (which is also computed at runtime).

A UML software model was created for the quantum synthesis problem, allowing a facile presentation for this complex problem. The object-oriented approach has enhanced the software reusability, extensibility and maintenance, hence allowing - at the same time - better runtime execution.

We have evaluated the performance of our quantum synthesis methodology by building a software tool that implements all the proposed concepts. The experiments have been conducted using quantum benchmark circuits [51] as inputs, and the evolved solutions implement the same functionality as their counterpart inputs. The resulting reduction in evolved solution runtime, as compared with other approaches, comes mainly from two sources: object-oriented implementation and meta-heuristic adaption of the application parameter controls.

As a remainder, the main contributions to quantum logic circuit synthesis and optimization methodologies presented in this PhD thesis, and also published in papers, are:

- A genetic algorithm dedicated to quantum logic circuit synthesis. A new view on quantum circuits, the split in sections and planes, is used in order to encode the quantum circuit. New genetic operators are applied at the gene level, as well as inside the gene. A quantum circuit repository stores the quantum circuit characteristics and provides a unique interface point for new circuit addition.
- A new framework architecture, that allows construction of different genetic algorithms; examples that validate the framework effectiveness are included: methinks and knapsack. The framework is used for the quantum logic circuit synthesis process, the statistical information being used to optimize the algorithm.

## 172 | 7-Conclusions and Perspectives

---

- A meta-heuristic method for adaptation of the genetic algorithm parameters, allows finding the optimum genetic algorithm control parameters without user intervention.
- A quantum logic circuit synthesis methodology transformed into a software application, developed for quantum logic circuit synthesis. The experiments and the source code availability prove the efficiency of this approach for task of quantum logic circuit synthesis.

Due to these improvements, our genetic-algorithm-based quantum circuit synthesis methodology is able to obtain significant runtime gains over other GA-based quantum circuits approaches [128][132][133][46][11]. Moreover, synthesis can be performed effectively even for large circuits (6-7-qubit), as it is shown, in the experiments section. It is worth noticing that other mentioned approaches are able to perform synthesis only for 3-4 qubit circuits, due to the high complexity of the matrix multiplication.

### 7.2 Future Directions

Although many of the questions identified during this research process have been dealt with, some of them still worth further investigation in the near future.

#### 7.2.1 Refinement and Future Work

- Quantum circuit representation: even if improved matrix representation and optimized library methods are used now, some improvements may be applied to allow fast operations on many qubits. The actual solution for the large matrix multiplication was presented by Don Coppersmith and Shmuel Winograd in 1990 and has the complexity of  $O(n^{2.376})$ .
- Quantum gate cost and feasibility: new theoretical results might come into the public view in the near future and the solution refinement will be necessary in order to follow the state-of-the-art trend. This adjustment is necessary because only few details are known at this moment.
- Quantum circuit database: the definition of additional, more complex quantum logic circuits will increase the range of synthesis algorithm and will allow for evolving new possible solutions.
- Controlled gate: the representation needs to be improved in order to allow target qubit definition at arbitrary positions within the gate. For this purpose, the QHDL needs a language extension in order to allow this complex representation. This change will complete the proposed basic

quantum gate set and will allow for more quantum gate combinations on the chromosome initialization.

### 7.2.2 QCS Integration

The genetic algorithm framework and the quantum-synthesis-based genetic algorithm may become interesting in the near future for other research applications. The design and source code availability allows for easy adaptation to other CAD tools. In addition, it will be possible to integrate (within a standalone environment) a quantum simulator and a quantum synthesis tool. Several discussions are ongoing, at this moment [134][10][55][99][93], keeping alive the hope that a complete tool chain for quantum logic circuits will be soon available.

# Appendix

## 8.1 Object Oriented Metrics

Having good software also means having software quality assurance. There are several ways to reach software quality: following a defined development process (from requirements through testing), performing reviews in order to find a second opinion and flaws from the very beginning, by using automatic software tools that indicate (via metric-based checks) the software quality reached during ongoing development. In this appendix, two major metrics parts are described: program unit complexity and class object-oriented metrics. The software used to detect flaws and to measure the software indicators is "Understand" [135], that was developed by the SCITools. The metric definitions are copied from the tool help pages. On the long run of the software domain, the software metrics have always been associated with software engineering, and there is no way to avoid using them.

### 8.1.1 Framework Program Unit Complexity

The program unit complexity reports the McCabe (Cyclomatic) complexity value for the methods defined within a module. A higher number for this metric indicates that the program unit is more likely to be tested with difficulty and maintained without error. In the following table (Table 11), only the methods that overreach the average values are described.

Table 11. Program Unit Complexity Metric

Method	Cyclo matic	Modi fied	Str ict	Esse ntial	Nest ing	Path Count
ToffoliTCC::_MakeToffoli_t_c_c	11	11	12	1	4	98
Population::_Operators	12	12	16	1	4	90
QHDLParser::_ReadGate	12	12	16	4	4	42
QSyn::_CreateGA	13	13	13	1	1	405
Synthesis::p_CrossoverBType	13	13	15	1	2	216
Synthesis::_GenerateGate	17	16	21	1	4	27
QHDLParser::_ReadGates	21	21	21	1	5	9858
Adaptive::_ShowStatisticPerformanceValues	22	14	22	1	2	143
QSyn::b_ReadCfgFile	23	23	26	1	1	2359296
Synthesis::p_Translate	31	17	33	15	4	63

The definitions according to the reference [135] are:

## 8.1 -Object Oriented Metrics|175

- Cyclomatic: "The measure of the complexity of a function's decision structure; it is also the number of basis, or independent, paths through a module. Sometimes also called the McCabe Complexity after its originator."
- Modified: "Cyclomatic except each case statement is not counted; the entire switch counts as 1."
- Strict: "Same as Cyclomatic complexity except that && and || also count as 1."
- Essential: "Measures the amount of unstructured code in a function."
- Nesting: "Shows the maximum nesting level of control constructs in program unit."

### 8.1.2 Framework Class OO Metrics

The Class OO Metrics provides the object-oriented code metrics for the analyzed classes. In the following table (Table 12), only the methods that overreach the average values are described.

Table 12. Class OO Metrics

Class	LCOM	DIT	IFANIN	CBO	NOC	RFC	NIM	NIV	WMC
Adaptive	75	0	0	4	0	13	13	12	13
FileParser	65	0	0	1	0	9	9	7	9
GA	75	0	0	4	1	22	22	8	22
GateCollection	68	0	0	19	0	11	10	2	11
Genome	75	0	0	2	1	20	20	5	20
HistorySolutions	57	0	0	2	0	7	7	5	7
Population	79	0	0	9	0	27	27	18	27
QHDLParser	79	0	0	4	0	23	23	8	23
QSyn	51	0	0	16	0	8	8	7	8
QuantumGate	70	0	0	0	16	19	19	8	19
Synthesis	70	1	1	10	0	49	29	4	29
SynthesisCfg	60	0	0	2	0	12	11	4	12

The definitions according to reference [135] are:

- LCOM: "Percent Lack of Cohesion is 100% minus the average cohesion for class data members. A method is cohesive when it performs a single task."
- DIT: "Max Inheritance Tree is the maximum depth of the class in the inheritance tree."
- IFANIN: "Count of Base Classes is the number of immediate base classes."
- CBO: "Count of Coupled Classes is the number of other classes coupled to this class."
- NOC: "Count of Derived Classes is the number of immediate subclasses this class has."
- RFC: "Count of All Methods is the number of methods this class has, including inherited methods."
- NIM: "Count of Instance Methods is the number of instance methods this class has."
- NIV: "Count of Instance Variables is the number of instance variables this class has."
- WMC: "Count of Methods is the number of local methods this class has."

## **8.2 QCS Initial Genome Solution**

The initial genome is created by the synthesis algorithm, using the parsed information from the input file. The initial genome is necessary for further comparison with the evolved solutions (method source code is available in Figure 8.1).



```

void QSyn::_CreateInitialSolutionGenome() {
    // create the initial solution genome
    itsInitialGenome = new Synthesis(*itsRandom, s_Config_.e_MutationType,
        s_Config_.e_CrossoverType) ;

    // create the relation in SynthesisCfg
    // it is important to have at that level the initial genome
    SynthesisCfg::rc_GetInstance()._SetInitialGenome(itsInitialGenome) ;

    // get the number of qubits
    int qubits = myQHDL.n_GetNoOfQubits() ;
    int i = 0 ;      // for subelements
    int j = 0 ;      // for elements
    bool b_valid = true ;

    // get the current element
    boost::shared_ptr<Nod> Element = myQHDL._GetListElement(i,j, b_valid) ;
    while ( (Element!=0) && (b_valid) )
    {
        // get the current subelement
        boost::shared_ptr<Nod> SubElement = myQHDL._GetListElement(i+1,j, b_valid) ;
        while ( (SubElement!=0) && (b_valid) )
        {
            // initiate the genome with the subelement
            itsInitialGenome->_Initiate((j+(i*qubits)), *SubElement) ;

            // goto next subelement
            i++ ;
            SubElement = myQHDL._GetListElement(i+1, j, b_valid) ;
        }

        // reinitialize the subelement and increment the element
        i = 0, j++ ;

        // goto next element
        Element = myQHDL._GetListElement(i, j, b_valid) ;
    }

    // compute the output function and save it into the
    // QuantumGate at the SynthesisCfg level
    SynthesisCfg::rc_GetInstance()._ComputeOutputFct() ;
}

```

Figure 8.1: Creation of the Initial Genome

### 8.3 Statistic Details

The Statistical class is implemented as a container for different object types using a template parameter. The class provides basic methods that are implemented in the class instances, for the statistical functions: maximum, minimum, average and standard deviation (method source code is presented in Figure 8.2).

```

template <class T> void Statistic<T> ::_AddToHistory(const T& ObjectToAdd) {
    HistoryList_.push_back(ObjectToAdd) ;
}

template <class T> T Statistic<T> ::c_GetMean() {
    T result = HistoryList_[0] ;
    // no. of elements in vector
    int lsize = int(HistoryList_.size()) - 1 ;

    for(int i=0; i<lsize ; i++)
    {
        // first calculate the sum
        result = result + HistoryList_[i+1] ;
    }
    // then calculate the mean
    result = ComputeMean(result, (lsize+1)) ;
    return result ;
}

template <class T> T Statistic<T> ::c_GetSTDEV() {
    T result, empty ;
    T mean = c_GetMean() ;      // get the mean value

    int lsize = int(HistoryList_.size()) ;
    // stdev only between two objects
    if (lsize>1)
    {
        // sum Xi*Xi
        for(int i=0; i<lsize ; i++)
        {
            result = result + HistoryList_[i]*HistoryList_[i] ;
        }
        // sum Xi*Xi - N*Xmean^2
        result = result - lsize*(mean*mean) ;

        if ( result>0 ){
            result = ComputeSquareRoot(result / lsize) ;
            return result ;
        }
    }

    // otherwise return an empty object
    return empty ;
}

```

Figure 8.2: Statistic Methods

## 8.4 QCS Genome Implementation Details

In this section, the genome implementation details are described, by highlighting the most important methods from the Synthesis class.

```

void Synthesis::_ComputeOutputFct(QuantumGate& GateResult) {
    // temporary gates
    QuantumGate GateMultiplicationTmp, GateTensorialTmp ;
    QuantumGate GateMultiplication, GateTensorial ;
    // actual read gate
    QuantumGate ReadGate ;
    // index inside of gene
    bool FirstLocus = true , FirstGene = true ;

    // compute first the output function
    while (n_GeneIndex < SynthesisCfg::rc_GetInstance().n_GetChromosomeLength())
    {
        ReadGate = rc_DetectNextGate() ; // read actual gate
        // save the result for doing multiplication
        if (b_NewSection && FirstGene)
        {
            FirstGene = false ;
            // only the first section was read, thus save it into Tmp
            GateMultiplicationTmp = GateTensorialTmp ;
            FirstLocus = true ; // re-initialize the locus
        }
        else if (b_NewSection)
        {
            FirstLocus = true ; // re-initialize the locus
            // do the multiplication
            (void) QMath::b_Multiply(GateMultiplicationTmp, GateTensorialTmp,
                GateMultiplication) ;
            // save it for further multiplication
            GateMultiplicationTmp = GateMultiplication ;
        }
        // save result for product tensor
        if (FirstLocus)
        {
            FirstLocus = false ;
            // only the first gate was read, thus save it into Tmp
            GateTensorialTmp = ReadGate ;
        }
        else
        {
            // do the tensor product
            QMath::_Tensorial(GateTensorialTmp, ReadGate, GateTensorial) ;
            // save it for further tensor product
            GateTensorialTmp = GateTensorial ;
        }
    }
    // perform the last multiplication
    (void) QMath::b_Multiply(GateMultiplicationTmp, GateTensorialTmp, GateMultiplication);
    n_GeneIndex = 0 ; // reset the gene index for a further output computation
    GateResult = GateMultiplication ; // return it via argument
}

```

**Figure 8.3: Output Function Computation**

The TypeA Mutation is responsible with the mutation operator applied at the gene level (the complete gene is replaced by others randomly generated values). The TypeB Mutation is responsible with the mutation operator applied inside the gene level (one quantum gate is replaced by other randomly generated gate). If the selected locus for mutation is used by a quantum gate with more than one input, the locus is moved to the right, and start/stop indexes are computed in order to allow the complete quantum gate detection.

```
void Synthesis::_MutationAType()
{
    b_WasEvaluated_ = false ;

    int NoOfGenes = SynthesisCfg::rc_GetInstance().n_GetNoOfGenes() ;
    int GeneLength = SynthesisCfg::rc_GetInstance().n_GetGeneLength() ;

    // select a gene from the chromosome
    int SelectGene = itsRandom->RandInt(0, NoOfGenes-1) ;

    // perform mutation by replacing the complete gene values (only one gene is affected)
    _InitiateLocus(SelectGene*GeneLength, (SelectGene+1)*GeneLength - 1) ;
}

void Synthesis::_MutationBType()
{
    b_WasEvaluated_ = false ;

    int NoOfGenes = SynthesisCfg::rc_GetInstance().n_GetNoOfGenes() ;
    int GeneLength = SynthesisCfg::rc_GetInstance().n_GetGeneLength() ;

    // select a gene from the chromosome
    int SelectGene = itsRandom->RandInt(0, NoOfGenes-1) ;
    int SelectLocus = itsRandom->RandInt(0, GeneLength-1) ;

    int StartIndex = SelectGene*GeneLength+SelectLocus ;

    // perform mutation
    _InitiateLocus(StartIndex, StartIndex) ;
}
```

**Figure 8.4: TypeA/TypeB Mutation**

#### 8.4 -QCS Genome Implementation Details| 181

```
void Synthesis::_InitiateLocus(int StartIndex, int EndIndex)
{
    int NoOfGenes = SynthesisCfg::rc_GetInstance().n_GetNoOfGenes() ;
    int GeneLength = SynthesisCfg::rc_GetInstance().n_GetGeneLength() ;

    // detect the possible different start and stop index
    _DetectGateStartLeft(StartIndex, StartIndex) ;
    _DetectGateStopRight(EndIndex, EndIndex) ;

    int StartGateId = 1;
    if (StartIndex > 0)
        StartGateId = ac_Genome[StartIndex - 1]._ReturnGateId() + 1 ;

    // generate random values
    for(int i=StartIndex; i<=EndIndex; i++)
    {
        int MaxGateInputs = GeneLength - (i%GeneLength) ;
        // special case when not the complete gene is affected
        if ((EndIndex - StartIndex + 1) < GeneLength )
            MaxGateInputs = EndIndex - StartIndex + 1 ;

        // generate one or more quantum gate, but with Max No. of Inputs defined
        _GenerateGate(MaxGateInputs, i, i, StartGateId) ;
    }

    // affect the rest of the gates id (to be consecutive numbers)
    _ShiftIds((EndIndex+1), NoOfGenes*GeneLength, StartGateId) ;
}
```

**Figure 8.5: Locus Initialization**

```

boost::shared_ptr<Genome> Synthesis::p_CrossoverAType(Genome& spouse) {
    int ChromosomeLength = SynthesisCfg::rc_GetInstance().n_GetChromosomeLength();
    int GeneLength = SynthesisCfg::rc_GetInstance().n_GetGeneLength();
    int NoOfGenes = SynthesisCfg::rc_GetInstance().n_GetNoOfGenes();
    // create a new offspring
    boost::shared_ptr<Synthesis> offspring(new Synthesis(*itsRandom, e_MutationType_,
        e_CrossoverType_));
    // copy the value from its parent
    *offspring = *this;
    offspring->_ClearEvaluatedFlag();
    // select random cut
    int CutPoint1 = itsRandom->RandInt(0, (NoOfGenes-1));
    // detect the locus within chromosome
    int StartIndex = 0, StopIndex = 0;
    int LastIndex = ChromosomeLength - 1;

    if (e_CrossoverType_ == Genome::e_OnePoint) // type of crossover is OnePoint
    {
        StartIndex = CutPoint1*GeneLength; // compute the start index
        StopIndex = LastIndex; // stop index is at the chromosome length
    }
    else if (e_CrossoverType_ == Genome::e_TwoPoints) // type of crossover is TwoPoints
    {
        int CutPoint2;
        do
        {
            CutPoint2 = itsRandom->RandInt(1, NoOfGenes);
        } while( CutPoint2 == CutPoint1);
        if (CutPoint2 < CutPoint1) // order the cut points
        {
            int tmp = CutPoint1;
            CutPoint1 = CutPoint2;
            CutPoint2 = tmp;
        }
        StartIndex = CutPoint1*GeneLength; // compute the start index
        StopIndex = (CutPoint2*GeneLength) - 1; // compute the stop index value
    }
    // perform the crossover, copy the gene or genes between start index and stop index
    for (int i=StartIndex; i<=StopIndex; i++)
        // do the exchange
        offspring->ac_Genome[i] = dynamic_cast<Synthesis&>(spouse).ac_Genome[i];
    int StartGateId = 1;
    if (StartIndex >= 1)
        StartGateId = offspring->ac_Genome[StartIndex-1]._ReturnGateId() + 1;

    // affect the gates id (to be consecutive numbers)
    offspring->_ShiftIds(StartIndex, StopIndex, StartGateId);
    if (StopIndex != LastIndex)
    {
        StartGateId = offspring->ac_Genome[StopIndex]._ReturnGateId() + 1;
        // affect the rest of gates id (to be consecutive numbers)
        offspring->_ShiftIds(StopIndex+1, LastIndex, StartGateId);
    }
    return offspring; // return the child
}

```

Figure 8.6: Type A Crossover

## 8.5 Quantum Gates Cost

In the available articles, the subject of quantum gate cost is not completely covered by information about physical quantum gates. Therefore, from a theoretical point of view, several possibilities are possible in order to define and introduce the quantum gate cost propriety.

### 8.5.1 Cost Details using General Approach

Each quantum gate physical implementation complexity is evaluated by introducing two proprieties: cost and feasibility. Because the genetic algorithm approach does not know any details about the circuit output function, or about the circuit garbage qubits, the number of quantum qubits involved in the quantum gate is part of a mathematical formula with the scope of defining the quantum circuit cost value. For example, having an increased number of qubits into a quantum gate will increase the cost and will decrease its feasibility. In the following table (Table 13), the gates and their associated formula for cost and feasibility are also presented. In this way, it is possible to compare and grade intermediate evolved results.

**Table 13. Gate Costs and Feasibility**

Gate Name	I's between	Quantum Feasibility	Quantum Cost
Hadamard	0	100	1
X	0	100	1
Y	0	100	1
Z	0	100	1
S	0	100	1
T	0	100	1
I	$0 \div n$	100	1
CNOT	$0 \div n$	$100 * 2 / (2 + n)$	$2 + n$
CZ	$0 \div n$	$100 * 2 / (2 + n)$	$2 + n$
CS	$0 \div n$	$100 * 2 / (2 + n)$	$2 + n$
Swap	$0 \div n$	$100 * 2 / (2 + n)$	$2 + n$
Toffoli	$0 \div n$	$100 * 3 / (3 + n)$	$3 * (1 + n)$

### 8.5.2 Cost Details using Function Output

Each quantum gate has (as associated) a cost reported in comparison with Toffoli and generalized Fredkin gates. The following table (Table 14) describes the quantum costs of the generalized Toffoli gates, according to the paper published by Barenco.et.al. [32], by taking into consideration the a priori knowledge about the circuit function (number of garbage qubits). This cost cannot be used in a genetic algorithm approach, because on randomly generated gates the number of garbage gates remains unknown.

Table 14. Toffoli Gate Costs [51]

Size (n)	Garbage	Name	Quantum Cost
1	0	NOT, t1	1
2	0	CNOT, t2	1
3	0	Toffoli, t3	5
4	0	Toffoli, t4	13
5	0	t5	29
5	2	t5	26
6	0	t6	61
6	1	t6	52
6	3	t6	38
7	0	t7	125
7	1	t7	80
7	4	t7	50
8	0	t8	253
8	1	t8	100
8	5	t8	62
9	0	t9	509
9	1	t9	128
9	6	t9	74
10	0	t10	1021
10	1	t10	152
10	7	t10	86
n>10	0	tn	$2^n-3$
n>10	1	tn	$24n-88$
n>10	n-3	tn	$12n-34$

The Fredkin gate may be efficiently simulated by using  $n$  Toffoli gates and two additional CNOT gates. Therefore, the cost for the  $n$  Fredkin gate is computed as the cost of  $n$  Toffoli gates plus two (the CNOT gate cost is equal to one).

## 8.6 GNUplot Script

The following script was used to generate automatically all the statistic graphs that were presented throughout this dissertation. The script receives, as input, the quantum synthesis results coded as text files and, after processing, it plots the graphics. The conversion of the postscript result file into EPS, JPEG, GIF, TIFF or PDF was performed by using the ImageMagick<sup>19</sup> software.

<sup>19</sup> ImageMagick, <http://www.imagemagick.org/script/index.php>



```

set size 1.0, 1.0
set origin 0.0, 0.0
set term postscript enhanced color "Arial" 12
set output 'fig.ps'
set multiplot
set size 0.45,0.35
set origin 0.0,0.65
set grid
unset key

set title "Best Individual Fitness Evolution" # Plot Best Individual
set ylabel "Fitness value"
set xlabel "Generation (number)"
plot 'fitness.txt' using 1:2 every ::0::9

set size 0.6,0.25 # Plot Algorithm Performance
set origin 0.0,0.0
set notitle
set xrange [ 0 : 9 ] noreverse nowriteback
set xlabel "Runs (number)"
set ylabel "Solutions"
set boxwidth 0.9 absolute
set style fill solid 1.00 border -1
set style histogram clustered gap 1
set style data histograms
set lmargin 11
plot 'stat_solution.txt' using 2 ti col

set size 0.6,0.4 # Plot time Performance
set origin 0.0,0.25
set title "Algorithm performance"
set xrange [ 1 : 10 ] noreverse nowriteback
unset xlabel
set noxtics
set ylabel "Average Time (ticks)"
set style data lines
plot 'stat_solution.txt' using 1:3 lw 4

set size 0.6,0.4 # Plot Statistic Fitness evolution
set origin 0.4,0.6
set autoscale xy
set title "Statistic Fitness Evolution"
set key inside
set ylabel "Fitness value"
set xlabel "MIN MEAN MAX STDEV / Generation"
set boxwidth 0.9 absolute
set style fill solid 1.00 border -1
set style histogram clustered gap 1
set style data histograms
set noxtics
set yrange [ 0 : 1 ] noreverse nowriteback
plot 'stat_fitness.txt' using 5 index 0 ti col, " u 6 index 0 ti col, " u 7 index 0 ti col

```

Figure 8.7: GNUplot Script

## Bibliography

- [1] Richard P. Feynman, "Quantum mechanical computers," vol. 11, pp. 11-20, 1985.
- [2] Gordon E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, p. 4, April 1965.
- [3] Richard P. Feynman, "Simulating Physics with Computers," *International Journal of Theoretical Physics*, vol. 21, no. 6/7, pp. 467-488, 1982.
- [4] David Deutsch, "Quantum theory, the Church-Turing principle and the universal quantum computer," in *Proceedings of the Royal Society of London*, 1985, pp. 97-117.
- [5] Peter W. Shor, "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer," in *35th Annual Symposium on Foundations of Computer Science*, Santa Fe, 1994, p. 28, [http://arxiv.org/PS\\_cache/quant-ph/pdf/9508/9508027v2.pdf](http://arxiv.org/PS_cache/quant-ph/pdf/9508/9508027v2.pdf).
- [6] Lov K. Grover, "A fast quantum mechanical algorithm for database search," in *28th Annual ACM Symposium on the Theory of Computing*, Philadelphia, 1996, pp. 212-219, [http://arxiv.org/PS\\_cache/quant-ph/pdf/9605/9605043v3.pdf](http://arxiv.org/PS_cache/quant-ph/pdf/9605/9605043v3.pdf).
- [7] Lee Spector, *Automatic Quantum Computer Programming: A Genetic Programming Approach.*: Springer Science-Business Media, 2004, <http://www.springer.com/computer/programming/book/978-0-387-36496-4>.
- [8] Lynch J. Benjamin. (2006, February) Optimizing with Genetic Algorithms. [Online]. <http://static.msi.umn.edu/tutorial/scientificcomp/OptimizingWithGA.pdf>
- [9] Andrei Bautu and Elena Bautu, "Quantum Circuit Design by Means of Genetic Programming," *Romanian Physics*, vol. 52, no. 5-7, pp. 697-704, 2007, [http://www.nipne.ro/rjp/2007\\_52\\_5-7/0697\\_0705.pdf](http://www.nipne.ro/rjp/2007_52_5-7/0697_0705.pdf).
- [10] John P. Hayes and Igor L. Markov, "Quantum Approaches to Logic Circuit Synthesis and Testing," p. 10, June 2006.
- [11] Martin Lukac and Marek Perkowski, "Evolving quantum circuits using genetic algorithm," in *Proceedings NASA/DoD Conference on Evolvable Hardware*, 2002, pp. 177 - 185, <http://ieeexplore.ieee.org/Xplore/login.jsp?url=/iel5/8000/22122/01029883.pdf?arnumber=1029883>.
- [12] Andrea Malossini, Enrico Blanzieri, and Tommaso Calarco, "QGA: A Quantum Genetic Algorithm," 2004.
- [13] ACSA. [Online]. <http://www.acsa.utt.ro/>

- [14] Kang L. Wang, "Microelectronics roadmap: from ultimate CMOS to quantum information systems," , Hong Kong, 2001, [http://ieeexplore.ieee.org/xpl/freeabs\\_all.jsp?tp=&arnumber=946904&isnumber=20499](http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?tp=&arnumber=946904&isnumber=20499).
- [15] Helve Jaouen, "ITRS/Upper+ suggestions," 2004.
- [16] (2007) International Technology Roadmap for Semiconductors. [Online]. [http://www.itrs.net/Links/2007ITRS/2007\\_Chapters/2007\\_ERD.pdf](http://www.itrs.net/Links/2007ITRS/2007_Chapters/2007_ERD.pdf)
- [17] (2005) International Technology Roadmap for Semiconductors. [Online]. <http://www.itrs.net/Links/2005ITRS/ERD2005.pdf>
- [18] Michael A. Nielsen and Isaac L. Chuang, *Quantum Computation and Quantum Information.*: Cambridge University Press, 2000.
- [19] John R. Koza, *Genetic Programming: On the programming of computers by Means of Natural Selection.* London, England: The MIT Press, Cambridge, Massachusetts, 1998.
- [20] Mitchell Melanie, *An Introduction to Genetic Algorithms.*: The MIT Press, 1999.
- [21] John P. Hayes, "Tutorial: basic concepts in quantum circuits," in *IEEE Proceedings Design Automation Conference*, 2003, <http://ieeexplore.ieee.org/Xplore/login.jsp?url=/iel5/8647/27397/01219146.pdf?arnumber=1219146>.
- [22] The free-content WWW resource in quantum information science. [Online]. <http://www.quantiki.org/>
- [23] Artur Ekert, Patrick Hayden, and Hitoshi Inamori, "Basic concepts in quantum computation," February 2008, <http://arxiv.org/abs/quant-ph/0011013>.
- [24] Emanuel Knill et al., "Introduction to Quantum Information Processing," *LA Science*, p. 48, <http://arxiv.org/abs/quant-ph/0207171>.
- [25] John H. Reif, "Quantum Information Processing: Compression, Coding, and Related Computations," 1999, <http://www.cs.duke.edu/reif/paper/qsurvey.ps>.
- [26] Andrew Steane, "Quantum computing," *Reports on Physics Progress*, pp. 117-173, 1998, <http://www.iop.org/EJ/abstract/0034-4885/61/2/002/>.
- [27] Ramamurti Shankar, *Principles of Quantum Mechanics.*: Plenum Pub Corp, 1994.
- [28] Michael D. Fayer, *Elements of Quantum Mechanics.*: Oxford University Press, 2001.
- [29] David Deutsch, Adriano Barenco, and Artur Ekert, "Universality in Quantum Computation," in *Proceedings of the Royal Society of London Ser. A*, vol. 449, 1995, pp. 669-677, <http://citeseer.ist.psu.edu/deutsch95universality.html>.

- [30] David P. DiVincenzo, "Quantum Gates and Circuits," in *ITP Conference on Quantum Coherence and Decoherence*, 1996, p. 18, <http://arxiv.org/abs/quant-ph/9705009>.
- [31] David P. DiVincenzo, "Two-Bit Gates Are Universal for Quantum Computation," *Phys. Rev. A*, vol. 51, no. 2, pp. 1015-1022, 1995, <http://citeseer.ist.psu.edu/divincenzo95twobit.html>.
- [32] Adriano Barenco et al., "Elementary Gates for Quantum Computation," *Physical Review*, p. 24, March 1995, <http://citeseer.ist.psu.edu/11452.html>.
- [33] John R. Koza, *Genetic Programming II: Automatic Discovery of Reusable Programs.*: MIT Press, 1994.
- [34] John R. Koza, H Forrest Bennett, David Andre, and Martin A. Keane, *Genetic Programming III: Darwinian Invention and Problem Solving.*: Morgan Kaufmann Publishers, 1999.
- [35] John R. Koza et al., *Genetic Programming IV: Routine Human-Competitive Machine Intelligence.*: Kluwer Academic Publishers, 2003.
- [36] John R. Koza, "Introduction to Genetic Programming Tutorial," in *Genetic and Evolutionary Computation Conference*, Seattle, 2004, p. 137, <http://www.genetic-programming.com/gecco2004tutorial.pdf>.
- [37] John R. Koza, "A Genetic Programming Tutorial," in *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*, Department of Computer Science, Ed. California: Stanford University / Kluwer Academic Publishers, p. 40, <http://citeseer.ist.psu.edu/cache/papers/cs/27780/http:zSzzSzwww.genetic-programming.comzSzzSzgptutorialburke.pdf/a-genetic-programming-tutorial.pdf>.
- [38] John H. Holland, *Adaption in Natural and Artificial Systems*, Massachusetts Institute of Technology, Ed.: First MIT Press, edition 1992, first edition 1975, The University of Michigan Press.
- [39] David E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning.*: Addison Wesley Longman Inc., 1989.
- [40] Matthew B. Wall. (1996-2007) Galib, A C++ Library of Genetic Algorithm Components. [Online]. <http://lancet.mit.edu/ga/>
- [41] Christian Gagne and Marc Parizeau. (2002-2007) Open BEAGLE, a versatile EC framework. [Online]. <http://beagle.gel.ulaval.ca/>
- [42] Adam Fraser and Thomas Weinbrenner. (1993-1997) GPC++ - Genetic Programming C++ Class Library. [Online]. <http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/weinbenner/gp.html>
- [43] David Levine. (1996) PGAPack Parallel Genetic Algorithm Library. [Online]. [http://www-fp.mcs.anl.gov/CCST/research/reports\\_pre1998/comp\\_bio/stalk/pgapack.html](http://www-fp.mcs.anl.gov/CCST/research/reports_pre1998/comp_bio/stalk/pgapack.html)

- [44] Open Channel Foundation. [Online].  
<http://www.openchannelfoundation.org/projects/SPLICER/>
- [45] J. J. Merelo. (1997) GAGS genetic algorithm C++ class library. [Online].  
<http://kal-el.ugr.es/GAGS/>
- [46] Benjamin I. P. Rubinstein, "Evolving Quantum Circuits using Genetic Programming," in *Genetic Algorithms and Genetic Programming at Stanford 2000.*: Stanford Bookstore 002-0-00-002365-B, 2000, pp. 325-334, <http://citeseer.ist.psu.edu/543423.html>.
- [47] Vivek Shende, Aditya K. Prasad, Igor L. Markov, and John P. Hayes, "Synthesis of Reversible Logic Circuits," in *IEEE Transaction on CAD* 22, 2003, pp. 710-722, [http://ieeexplore.ieee.org/xpl/freeabs\\_all.jsp?arnumber=1201583](http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1201583).
- [48] Dmitri Maslov and Gerhard W. Dueck, "Level Compaction in Quantum Circuits," in *IEEE Congress on Evolutionary Computation*, 2006, pp. 2405-2409, [http://ieeexplore.ieee.org/xpl/freeabs\\_all.jsp?arnumber=1688606](http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1688606).
- [49] Kazuo Iwama, Yahiko Kambayashi, and Shigeru Yamashita, "Transformation Rules for Designing CNOT-based Quantum Circuits," in *Design Automation Conference*, 2002, p. 6, <http://csdl2.computer.org/persagen/DLAbstoc.jsp?resourcePath=/dl/proceedings/&toc=comp/proceedings/dac/2002/2402/00/2402toc.xml&DOI=10.1109/DAC.2002.1012662>.
- [50] Dmitri Maslov, Gerhard W. Dueck, Michael D. Miller, and Camille Negreva, "Quantum Circuit Simplification and Level Compaction," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 3, pp. 436-444, March 2008, [http://arxiv.org/PS\\_cache/quant-ph/pdf/0604/0604001v2.pdf](http://arxiv.org/PS_cache/quant-ph/pdf/0604/0604001v2.pdf).
- [51] Reversible Logic Synthesis Benchmarks Page. [Online].  
<http://www.cs.uvic.ca/%7Edmaslov/>
- [52] Dmitri Maslov, Gerhard W. Dueck, and Michael D. Miller, "Toffoli Network Synthesis with Templates," in *Computer-Aided Design of Integrated Circuits and Systems*, 2005, pp. 807-817, [http://ieeexplore.ieee.org/xpl/freeabs\\_all.jsp?arnumber=1432873](http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1432873).
- [53] Dmitri Maslov, Gerhard W. Dueck, and Michael D. Miller, "Techniques for the Synthesis of Reversible Toffoli Networks," *ACM Transactions on Design Automation of Electronic Systems*, vol. 12, no. 4, p. 20, 2007, <http://portal.acm.org/citation.cfm?id=1278349.1278355&coll=GUIDE&dl=GUIDE&CFID=64559792&CFTOKEN=69330897>.
- [54] Dmitri Maslov, Gerhard W. Dueck, and Michael D. Miller, "Quantum Circuit Simplification and Level Compaction," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2008, pp. 436-444, <http://ieeexplore.ieee.org/Xplore/login.jsp?url=/iel5/43/4454003/04378213.pdf?temp=x>.

- [55] Dmitri Maslov, Sean M. Falconer, and Michele Mosca, "Quantum Circuit Placement: Optimizing Qubit-to-qubit Interactions through Mapping Quantum Circuits into a Physical," in *Annual ACM IEEE Design Automation Conference*, 2007, pp. 962-965, <http://portal.acm.org/citation.cfm?id=1278717>.
- [56] Unified Modeling Language. [Online]. <http://www.uml.org/>
- [57] Paul Harmon and Mark Watson, *Understanding UML: The Developer's Guide*. San Francisco, USA: Morgan Kaufmann Publishers Inc., 1997, <http://www.amazon.com/Understanding-UML-Developers-Engineering-Programming/dp/1558604650>.
- [58] Goong Chen et al., *Quantum Computing Devices: Principles, Designs, and Analysis*.: Chapman & Hall/CRC, 2007, <http://www.amazon.com/Quantum-Computing-Devices-Principles-Mathematics/dp/1584886811>.
- [59] (2004) ARDA. [Online]. [http://gist.lanl.gov/pdfs/rm\\_intro.pdf](http://gist.lanl.gov/pdfs/rm_intro.pdf)
- [60] Francis D. Desmond and Cameron W. Alan, *Objects, Components, and Frameworks with UML*.: Addison-Wesley, 1999.
- [61] Telelogic. (2005) Rhapsody Development Edition. [Online]. <http://www.telelogic.com/products/rhapsody/index.cfm>
- [62] Steve McConnell, *Code Complete, Second Edition*. Redmond, Wa: Microsoft Press, 2004.
- [63] **Cristian Ruican**. (2008) Personal Web Site. [Online]. [http://www.cs.utt.ro/~crys/index\\_files/public/qsyn.tar.gz](http://www.cs.utt.ro/~crys/index_files/public/qsyn.tar.gz)
- [64] Giovanni De Micheli, *Synthesis and Optimization of Digital Circuits*.: McGraw-Hill, Inc, 1994.
- [65] George F. Viamontes, Igor L. Markov, and John P. Hayes, "Checking equivalence of quantum circuits and states," in *IEEE/ACM International Conference on Computer-Aided Design*, 2007, pp. 69-74, [http://www.ieeeexplore.ieee.org/xpl/freeabs\\_all.jsp?isnumber=4397223&arnumber=4397246&count=153&index=22](http://www.ieeeexplore.ieee.org/xpl/freeabs_all.jsp?isnumber=4397223&arnumber=4397246&count=153&index=22).
- [66] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software*.: Addison-Wesley, 1994.
- [67] Shalloway Alan and Trott R. James, *Design Patterns Explained - A New Perspective on Object-Oriented Design*, Addison-Wesley, Ed., 2002.
- [68] **Cristian Ruican**, Mihai Udrescu, Lucian Prodan, and Mircea Vladutiu, "A Genetic Algorithm Framework Applied to Quantum Circuit Synthesis," in *Nature Inspired Cooperative Strategies for Optimization*, vol. 129/2008, Acireale, 2007, pp. 419-429, <http://www.springerlink.com/content/5871541r3571h608/?p=8c93e746e39744f59c7a1c6f0cf0cab4&pi=0>.
- [69] **Cristian Ruican**. (2007) Personal Web Site. [Online].

[http://www.cs.utt.ro/~crys/index\\_files/public/framework.zip](http://www.cs.utt.ro/~crys/index_files/public/framework.zip)

- [70] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling, *Numerical recipes in C, 2nd.*: Cambridge University Press, 1992.
- [71] **Cristian Ruican**, "Genetic Algorithm Framework for Application Tuning," University Politehnica Timisoara, PhD Report No.2 2008.
- [72] Paul E. Black. (2007) Knapsack problem in Dictionary of Algorithms and Data Structures. [Online].  
<http://www.nist.gov/dads/HTML/knapsackProblem.html>
- [73] Farhad Djannaty and Saber Doostdar, "A Hybrid Genetic Algorithm for the Multidimensional Knapsack Problem," *Math. Sciences*, vol. 3, no. 9, pp. 443-456, 2008, <http://www.m-hikari.com/ijcms-password2008/9-12-2008/djannatyIJCMS9-12-2008.pdf>.
- [74] David Pisinger, "Where are the hard knapsack problems?," *Computers and Operations Research*, vol. 32, no. 9, pp. 2271-2284, 2005, <http://portal.acm.org/citation.cfm?id=1063640>.
- [75] Anabela Simoes and Ernesto Costa, "An Evolutionary Approach to the Zero/One Knapsack Problem: Testing Ideas from Biology," in *Neural Networks and Genetic Algorithms*, 2001, pp. 236-239, <http://www.citeseer.ist.psu.edu/437920.html>.
- [76] Sami Khuri, Thomas Back, and Jorg Heitkotter, "The Zero/One Multiple Knapsack Problem and Genetic Algorithms," in *ACM Symposium on Applied Computing*, 1994, pp. 188-193, <http://portal.acm.org/citation.cfm?id=326694>.
- [77] **Cristian Ruican**. (2008) Personal Web Site. [Online].  
[http://www.cs.utt.ro/~crys/index\\_files/public/QKnapsack.zip](http://www.cs.utt.ro/~crys/index_files/public/QKnapsack.zip)
- [78] Richard Dawkins, *The Blind Watchmaker*. New York: W.W Norton & Company, Inc., 1986, <http://www.amazon.com/Blind-Watchmaker-Evidence-Evolution-Universe/dp/0393315703>.
- [79] Matt Stillerman, David Guaspari, and Wolfgang Polak, "Final Report - A Design Language for Quantum Computing," Odyssey Research Associates, Inc. (ATC-NY), ORA TR-03-0001 2003.
- [80] Riccardo Poli, William B. Langdon, Nicholas F. McPhee, and John R. Koza, "Genetic Programming An Introductory Tutorial and a Survey of Techniques and Applications," in *Computational Intelligence: A Compendium.*: Springer-Verlag, 2008, vol. Studies in Computational Intelligence (SCI), ch. 22, <http://www.springer.com/engineering/book/978-3-540-78292-6>.
- [81] Dmitri Maslov, "Eficient Reversible and Quantum Implementations of Symmetric Boolean Functions," in *Circuits, Devices and Systems*, 2006, pp. 467-472, [http://ieeexplore.ieee.org/xpl/freeabs\\_all.jsp?arnumber=4015856](http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=4015856).
- [82] Dmitri Maslov and Gerhard W. Dueck, "Garbage in Reversible Designs of

- Multiple Output Functions," in *Representations and Methodology of Future Computing Technologies*, 2003, pp. 162-170, <http://www.iqc.ca/~dmaslov/papers/rm03a.pdf>.
- [83] Dmitri Maslov and Gerhard W. Dueck, "Improved Quantum Cost for n-bit Toffoli Gates," *Electronics Letters*, vol. 39, no. 25, pp. 1790-1791, <http://ieeexplore.ieee.org/Xplore/login.jsp?url=/iel5/2220/28093/01255714.pdf?arnumber=1255714>.
- [84] Dmitri Maslov and Gerhard W. Dueck, "Reversible Cascades with Minimal Garbage," in *Computer-Aided Design of Integrated Circuits and Systems*, 2004, pp. 1497-1509, [http://ieeexplore.ieee.org/xpl/freeabs\\_all.jsp?arnumber=1350877](http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1350877).
- [85] Dmitri Maslov and Michael D. Miller, "Comparison of the Cost Metrics for Reversible and Quantum Logic Synthesis," *IET Computers & Digital Techniques*, vol. 1(2), pp. 98-104, 2007, <http://arxiv.org/abs/quant-ph/0511008>.
- [86] Dmitri Maslov, Gerhard W. Dueck, and Michael D. Miller, "Fredkin/Toffoli Templates for Reversible Logic Synthesis," in *Computer Aided Design*, 2003, pp. 256-261, [http://ieeexplore.ieee.org/xpl/freeabs\\_all.jsp?arnumber=1257667](http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1257667).
- [87] Dmitri Maslov, Gerhard W. Dueck, and Michael D. Miller, "Synthesis of Fredkin-Toffoli Reversible Networks," in *Very Large Scale Integration (VLSI) Systems*, 2005, pp. 765-769, [http://ieeexplore.ieee.org/xpl/freeabs\\_all.jsp?arnumber=1458796](http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1458796).
- [88] Dmitri Maslov, Christina Young, Michael D. Miller, and Gerhard W. Dueck, "Quantum Circuit Simplification Using Templates," in *Design, Automation and Test in Europe*, 2005, pp. 1208-1213, [http://ieeexplore.ieee.org/xpl/freeabs\\_all.jsp?arnumber=1395758](http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1395758).
- [89] Michael D. Miller, Dmitri Maslov, and Gerhard W. Dueck, "Synthesis of Quantum Multiple-Valued Circuits," *Journal of Multiple-Valued Logic and Soft Computing*, no. 5-6, pp. 431-450, 2006, <http://webhome.cs.uvic.ca/~mmiller/publications/JMVL0404.pdf>.
- [90] Afshin Abdollahi and Massoud Pedram, "Analysis and Synthesis of Quantum Circuits by Using Quantum Decision Diagrams," in *Proceedings of Design, Automation and Test in Europe*, 2006, <http://ieeexplore.ieee.org/Xplore/login.jsp?url=/iel5/11014/34699/01656898.pdf?temp=x>.
- [91] Gerhard W. Dueck and Dmitri Maslov, "Reversible Function Synthesis with Minimum Garbage Outputs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 11, pp. 1497-1509, 2004, [http://ieeexplore.ieee.org/xpl/freeabs\\_all.jsp?arnumber=1350877](http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1350877).
- [92] Paul E. Black and Andrew W. Lane, "Modeling quantum information systems," in *Proceedings SPIE*, Orlando, Florida, 2004, pp. 340-347,



<http://hissa.nist.gov/~black/Papers/modelQuantSimSPIE04.html>.

- [93] George F. Viamontes, Igor L. Markov, and John P. Hayes, "Graph-Based Simulation of Quantum Computation in the Density Matrix Representation," *Quantum Information & Computation*, vol. 5, no. 2, pp. 113-130, 2005, <http://citeseer.ist.psu.edu/633681.html>.
- [94] George F. Viamontes, Igor L. Markov, and John P. Hayes, "High-Performance QuIDD-based Simulation of Quantum Circuits," in *Conference on Design, automation and test in Europe*, 2004, p. 2, <http://citeseer.ist.psu.edu/669521.html>.
- [95] George F. Viamontes, Igor L. Markov, and John P. Hayes. (2008) QuIDDPro: High-Performance Quantum Circuit Simulation. [Online]. [http://vlsicad.eecs.umich.edu/Quantum/qp/qp\\_manual\\_3.1.pdf](http://vlsicad.eecs.umich.edu/Quantum/qp/qp_manual_3.1.pdf)
- [96] George F. Viamontes, Manoj Rajagopalan, Igor L. Markov, and John P. Hayes, "Gate-Level Simulation of Quantum Circuits," in *Proc. of ACM/IEEE Asia and South-Pacific*, Kitakyushu, Japan, 2003, p. 17, <http://citeseer.ist.psu.edu/viamontes03gatelevel.html>.
- [97] Michael D. Miller, Mitchell A. Thornton, and David Goodman, "A Decision Diagram Package for Reversible and Quantum Circuit Simulation," in *IEEE Congress on Evolutionary Computation*, 2006, pp. 2428-2435, <http://ieeexplore.ieee.org/Xplore/login.jsp?url=/iel5/11108/35623/01688610.pdf?tp=&isnumber=&arnumber=1688610>.
- [98] Vladimir P. Gerdt, Robert Kragler, and Alexander N. Prokopenya, "A Mathematica Program for Constructing Quantum Circuits and Computing Their Unitary Matrices," p. 18, 2007, <http://lit.jinr.ru/QPC2007/pdf/GerdtKraglerProkopenya.pdf>.
- [99] Andre Leier and Wolfgang Banzhaf, "Comparison of Selection Strategies for Evolutionary Quantum Circuit Design," in *Genetic and Evolutionary Computation*, Lecture Notes in Computer Science, Ed.: Springer Berlin / Heidelberg, 2004, pp. 557-568, <http://www.springerlink.com/content/tyfaybp8cp56fkaq/>.
- [100] Ketan N. Patel, Igor L. Markov, and John P. Hayes, "Evaluating Circuit Reliability Under Probabilistic Gate-Level Fault Models," *International Workshop on Logic and Synthesis*, p. 6, 2003, <http://citeseer.ist.psu.edu/644440.html>.
- [101] Marek Perkowski et al., "A Hierarchical Approach to Computer-Aided Design of Quantum Circuits," in *6 th Int'l Symp. on Representations and Methodology of Future Computing Technology*, 2003, <http://citeseerx.ist.psu.edu/viewdoc/summary;jsessionid=C90FA37D8B49FA9A62BBA3CCC88FF25B?cid=4578332>.
- [102] Bart Rylander, Terry Soule, James Foster, and Jim Alves-Foss, "Quantum Evolutionary Programming," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2001, pp. 1005-1011,

<http://www.cs.bham.ac.uk/~wbl/biblio/gecco2001/d11.pdf>.

- [103] Susan Stepney and John A. Clark, "Evolving quantum programs and protocols," in *Handbook of Theoretical and Computational Nanotechnology*: American Scientific Publishers, 2006, p. 60, <http://portal.acm.org/citation.cfm?id=1118010>.
- [104] Mihai Udrescu, "Quantum Circuits Engineering: Efficient Simulation and Reconfigurable Quantum Hardware," University Politehnica, Timisoara, PhD Thesis 2005.
- [105] Mihai Udrescu, Lucian Prodan, and Mircea Vladutiu, "The Bubble Bit Technique as Improvement of HDL-Based Quantum Circuits Simulation," in *38th Annual Simulation Symposium*, 2005, pp. 217-224, <http://ieeexplore.ieee.org/Xplore/login.jsp?url=/iel5/9628/30438/01401968.pdf?temp=x>.
- [106] Colin P. Williams and Alexander G. Gray, "Automated Design of Quantum Circuits," in *Quantum Computing and Quantum Communications*, Lecture Notes in Computer Science, Ed.: Springer Berlin / Heidelberg, 1999, pp. 113-125, <http://www.springerlink.com/content/5lyw885hfne5rln7/>.
- [107] Martin Lukac, Normen Giesecke, Sazzad Hossain, Marek Perkowski, and Dong H. Kim, "Quantum Behaviors: Synthesis and Measurement," in *37th International Symposium on Multiple-Valued Logic*, 2007, p. 8, <http://web.cecs.pdx.edu/~mperkows/transfer/LukacPerkowski-quantum-behaviors-synthesis-measurement.pdf>.
- [108] P. Gupta, A. Agrawal, and N. K. Jha, "An Algorithm for Synthesis of Reversible Logic Circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 11, pp. 2317-2330, Nov 2006.
- [109] **Cristian Ruican**, Mihai Udrescu, Lucian Prodan, and Mircea Vladutiu, "Software Architecture for Quantum Circuit Synthesis," in *International Conference on Artificial Intelligence and Soft Computing*, vol. Computational Intelligence: Methods and Applications, Zakopane, 2008, pp. 562-573, <http://www.exit.pl>.
- [110] **C. Ruican**, "Prerequisites of Synthesis Methodology for Quantum Circuits," Timisoara, 2007.
- [111] Odyssey Research Associates. (2006-2008) QHDL: A Design Language for Quantum Computing. [Online]. <http://www.oracorp.com/>; <http://www.dodsbir.net/selections/abs021/darpaabs021.htm>; <http://www.atcorp.com/systemdevelopment/quantcomp.html>
- [112] Andrew Lane and Paul E. Black. (2008) Quantum Compiling and Simulation. [Online]. <http://hissa.nist.gov/~black/Quantum/qcsim.html>
- [113] **Cristian Ruican**, Mihai Udrescu, Lucian Prodan, and Mircea Vladutiu, "Automatic Synthesis for Quantum Circuits using Genetic Algorithms," in *International Conference on Adaptive and Natural Computing Algorithms*,

- vol. LNCS 4431, Warsaw, 2007, pp. 174-183, <http://www.springerlink.com/content/e57868200314g377/>.
- [114] Riccardo Poli, William B. Langdon, and Nicholas F. McPhee, *A Field Guide to Genetic Programming.*, March 2008.
- [115] Francisco Herrera and Manuel Lozano, "Fuzzy adaptive genetic algorithms: design, taxonomy, and future directions," *Evolutionary Intelligence*, pp. 27-46, 2008, <http://sci2s.ugr.es/publications/ficheros/GFS-Taxonomy-current%20research%20trends%20and%20prospects.pdf>.
- [116] Andreas Fink and Stefan Voss, "Reusable metaheuristic software components and their application via software generators," in *Metaheuristics International Conference*, 2001, p. 6, <http://citeseer.ist.psu.edu/fink01reusable.html>.
- [117] Ovidiu Gheorghies, Henri Luchian, and Adriana Gheorghies, "A Study of Adaption and Random Search in Genetic Algorithms," in *IEEE Congress on Evolutionary Computation (CEC)*, 2006, pp. 2103-2110, <http://ieeexplore.ieee.org/Xplore/login.jsp?url=/iel5/11108/35623/01688566.pdf?isnumber=35623&prod=CNF&arnumber=1688566&arSt=2103&ared=2110&arAuthor=Gheorghies%2C+O.%3B+Luchian%2C+H.%3B+Gheorghies%2C+A>.
- [118] Georges R. Harik and Fernando G. Lobo, "A Parameter-Less Genetic Algorithm," in *Genetic and Evolutionary Computation Conference*, 1999, p. 8, <http://citeseer.ist.psu.edu/cache/papers/cs/2445/ftp:zSzzSzftp-illigal.ge.uiuc.eduzSzpubzSzpaperszSzIlliGALszSz99009.pdf/harik99parameterless.pdf>.
- [119] Deb Kalyanmoy and Hans-Georg Beyer, "Self-Adaptive Genetic Algorithms with Simulated Binary Crossover," University of Dortmund, 1999.
- [120] Lam N. Law and Kwok Y. Szeto, "Adaptive Genetic Algorithm with Mutation and Crossover Matrices," in *20th International Joint Conference on Artificial Intelligence*, Hyderabad, India, 2007, pp. 2330-2333, <http://www.ijcai.org/papers07/Papers/IJCAI07-375.pdf>.
- [121] Fernando G. Lobo and David E. Goldberg, "The parameter-less genetic algorithm in practice," *Information Sciences*, vol. 167, no. 1-4, pp. 217-232, December 2004, <http://citeseer.ist.psu.edu/cache/papers/cs/22142/ftp:zSzzSzftp-illigal.ge.uiuc.eduzSzpubzSzpaperszSzIlliGALszSz2001022.pdf/lobo01parameterless.pdf>.
- [122] Sonja Novkovic and Davor Sverko, "Towards a Genetic Algorithm for Function Optimization," Saint Mary's University, Canada, Halifax,.
- [123] Hartmut Pohlheim, "Visualization of Evolutionary Algorithms Set of Standard Techniques and Multidimensional Visualization," in *Genetic and Evolutionary Computation Conference*, 1999, p. 8, <http://citeseer.ist.psu.edu/cache/papers/cs/25740/http:zSzzSzwww.pohlhei>

m.comzSzPaperszSzconf\_gecco1999zSzPohlheimH\_VisualizationEA\_GECCO1999.pdf/pohlheim99visualization.pdf.

- [124] Daisuke Yamashiro, Tomohiro Yoshikawa, and Takeshi Furuhashi, "Visualization of Search Process and Improvement of Search Performance in Multi-Objective Genetic Algorithm," in *IEEE Congress on Evolutionary Computation*, 2006, pp. 1151-1156, <http://ieeexplore.ieee.org/Xplore/login.jsp?url=/iel5/11108/35623/01688439.pdf?tp=&isnumber=&arnumber=1688439>.
- [125] E. Agoston Eiben, Zbigniew Michalewicz, Marc Schoenauer, and E. James Smith, "Parameter Control in Evolutionary Algorithms," *Parameter Setting in Evolutionary Algorithms*, vol. 54, pp. 19-46, 2007, <http://www.informatik.uni-trier.de/~ley/db/series/sci/sci54.html#EibenMSS07>.
- [126] O. Gheorghies, H. Luchian, and A. Gheorghies, "Walking the Royal Road with Integrated-Adaptive Genetic Algorithms," University Alexandru Ioan Cuza of Iasi, 2005.
- [127] (2007) Electronic Statistics Textbook.
- [128] Singh Prashant, "Evolving Quantum Circuits using Genetic Algorithm," 2005, revised 2007, <http://arxiv.org/abs/quant-ph/0511036v2>.
- [129] M. Mohammadi, M. Eshghi, and K. Navi, "Optimizing the Reversible Full Adder Circuit," in *IEEE EAST-WEST DESIGN&TEST*, Yerevan, 2008, [http://ewdtest.com/conf/EWDTS%2707\\_Presentations/Mohammadi,Eshghi,Navi.pdf](http://ewdtest.com/conf/EWDTS%2707_Presentations/Mohammadi,Eshghi,Navi.pdf).
- [130] Rodney Van Meter, W. J. Munro, Kae Nemoto, and Kohei M. Itoh, "Arithmetic on a Distributed-Memory Quantum Multicomputer," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 3, no. 4, 2007.
- [131] Vivek Shende, Stephen S. Bullock, and Igor L. Markov, "Synthesis of Quantum Logic Circuits," *IEEE Transactions on Computer-Aided Design*, vol. 25, no. 6, pp. 1000-1010, June 2006.
- [132] Ding Shengchao, Jin Zhi, and Yang Qing, "Evolving Quantum Oracles with Hybrid Quantum-inspired Evolutionary Algorithm," 2006, <http://arxiv.org/abs/quant-ph/0610105v1>.
- [133] Tim Reid, "On the Evolutionary Design of Quantum Circuits," 2005.
- [134] Krista Svore, Andrew Cross, A. Aho, I. Chuang, and Igor Markov, "Toward a Software Architecture for Quantum Computing Design Tools," 2006, <http://www.eecs.umich.edu/~imarkov/pubs/misc/qpl04-tools.pdf>.
- [135] SCI. Understand. [Online]. <http://www.scitools.com/>