Technical University of Lodz
Departament of Microelecronics and Computer Science

MSc Thesis

Distributed System for Designing Reliable Digital Systems Using Genetic Algorithms

Tomasz Norek

Student number: 106121

Supervisor:
Grzegorz Jabłoński, PhD

Auxiliary Supervisor:
Dariusz Makowski, MSc

Łódź, 2005

Streszczenie

Układy programowalne stają się coraz ważniejsze w dzisiejszych zastosowaniach elektronicznych. Ich wszechstronność jest coraz częściej wykorzystywana już nie tylko do testowania prototypowych rozwiązań, ale nawet do produkcji urządzeń w małych seriach, gdzie koszt wytworzenia specyficznego układu krzemowego znacząco wpłynąłby na cenę ostatecznego produktu. Ta uniwersalność jest także źródłem słabości układów programowalnych, ponieważ stają się one wrażliwe na zjawiska zachodzące w krzemie pod wpływem promieniowania. Praca ta opisuje jedną z technik zapobiegania negatywnym skutkom takich zjawisk – użycie algorytmów genetycznych do zaprojektowania takiej konfiguracji układu programowalnego, która mimo zmian powodowanych przez promieniowanie w niej samej, zachowa prawidłową funkcjonalność układu. Ponieważ symulacja układów programowalnych jest zadaniem wymagającym dużych mocy obliczeniowych, praktyczna część pracy obejmuje budowę rozproszonego systemu do obliczeń genetycznych i użycie tego systemu do przeprowadzenia symulacji weryfikujących przydatność wyżej wspomnianej techniki. System rozproszony okazał się bardzo dobrze spełniać swoje zadanie – dał możliwość wykorzystania dużej mocy obliczeniowej bez żadnych dodatkowych kosztów. Symulacje przeprowadzone przy użyciu systemu pozwoliły na zmniejszenie prawdopodobieństwa powstania wadliwej konfiguracji przykładowego układu ponad 50-krotnie. Praca zawiera również wyjaśnienie mechanizmów oddziaływania różnych typów promieniowania z układami krzemowymi, opis technik zapobiegania negatywnym skutkom takiego oddziaływania, opis typów układów programowalnych, opis możliwych skutków jakie może wywołać promieniowanie w tych układach (rozdział 2), wyjaśnienie zasady działania algorytmów genetycznych (rozdział 3). Struktura systemu rozproszonego użytego do symulacji wraz z krótką charakterystyką środowisk używanych do budowy systemów rozproszonych została przedstawiona w rozdziale 4. Rozdział 5 prezentuje wyniki uzyskane podczas symulacji. Rozdział 6 podsumowuje wyniki, zawiera wnioski dotyczące praktycznego użycia zaprezentowanego rozwiązania.

Table of Contents

1. Introduction

The programmable circuits are becoming more and more important in contemporary electronic applications. Application Specific Integrated Circuits (ASICs) usage is justified only in high-volume projects. In most of the other tasks programmable circuits of many kinds are satisfactory and cost-effective solution. Their possibility of changing the configuration by the user is a great advantage and results in circuit flexibility, but can also become a curse when the device is placed in the high radiation environment. The device configuration can be changed by the radiation particle and result in functional failure. Chapter 2 describes mechanisms of radiation-matter interaction, radiation effects mitigation techniques, the types of programmable circuits and potential effect of radiation on those circuits.

One of the techniques, that can improve circuit reliability, is a formulation of special fault-tolerant configuration, which despite some changes in its contents retains the functionality of the circuit. However, there are no widely available tools for the design of such configurations. Possibly the genetic algorithms can be used for that purpose. Chapter 3 explains the idea of genetic algorithms and presents the details of potential evolution application.

The simulation of the programmable circuit requires big amount of computational power, which is not easily accessible. This problem can be solved by the usage of distributed system, which will distribute tasks to many ordinary computers, thus increasing the available computational power. Chapter 4 describes the structure and functioning of the developed system.

Chapter 5 presents the results of simulations performed using the distributed system.

Chapter 6 contains conclusions drawn from the simulation results and summary of the thesis achievements.

1.1. Project Goals

The goal of the project is to design the distributed system for the circuit design using the genetic algorithms. The system should give access to the computational power enough for genetic simulations with no additional costs and at reasonable resources usage. Another goal is to verify the hypothesis that the genetic algorithms can be used effectively to improve the fault-tolerance of the programmable circuits by the design of special device configuration.

2. Impact of Radiation on Programmable Circuits

2.1. Radiation and Matter Interaction

There are several types of radiation, where different particles of different energies act. The quantum mechanics laws and theories description is far beyond the topic of this thesis, but at least brief explanation of terms used later throughout the chapter is essential. Nowadays, particles are thought to consist of **quarks** (which come in 6 flavours: up and down, charm and strange, top and bottom) and **leptons**. Quarks cannot be isolated; they are confined in particles called **hadrons** and held together by **gluons**. Furthermore, hadrons are divided into **mesons** (made of quark anti-quark pair) and **baryons** (made of three quarks). Hadrons interact via *strong interaction* and leptons via *photons*. The particles, which before 1970s were thought to be basic ones like proton, neutron and electron are accounted to the following groups: proton and neutron are baryons, electron is a lepton. Proton is build up from two up and one down quark and neutron is build up from one up and two down quarks. For full list of quantum particles, their properties and interactions between them please refer to [1].

Generally radiation interactions can be divided into two groups: ionising and non-ionising. Charged hadrons and leptons, heavy ions and photons are ionising particles, as they can ionise an atom. Neutrons and neutral hadrons cannot ionise atom directly, therefore are considered as non-ionising. However, neutrons can indirectly ionise atoms through the nuclear reactions. High-energy neutrons can excite an atom, which then emits gamma or x-ray radiation (photons with high energy). In the particle accelerator environment, which is the potential destination of the fault-tolerant programmable device, the presence of neutrons, electrons and gamma radiation is the basic danger.

2.2. Radiation Effects in Silicon

Under normal conditions radiation effects are not of a big concern. However, even at ground level, some space radiation particles can hit sensitive elements and cause failure of the system (in huge memory systems especially). They start to count seriously or become even critical when it comes to some special applications like space, military, avionics, nuclear power plants, High Energy Physics (HEP). Nowadays silicon integrated circuits are the most popular ones. They provide good speed, are cheap in production, but due to small elements size (for example gates of the transistors) and high packing density of the elements they are sensitive to radiation effects. Of course, one could argue that vacuum tubes are immune to radiation, but requirements like: circuit complexity, weight of the circuit, limited power consumption, mechanical resistance cannot be met by vacuum tube circuitry. In next few sections main radiation induced effects that occur in silicon circuits are presented.

2.2.1. Cumulative Effects

Cumulative effects alter semiconductor devices permanently. There are two main mechanisms accounted to the cumulative effects:

- Displacement damage
- Ionisation damage

Displacement damage occurs when incident quanta of sufficient energy hits semiconductor material, transfers momentum to the material atom, which in turn changes its place in the lattice. Such lattice defects have influence in the properties of the semiconductor material. They create so-called mid-gap states, which can result in generation of dark current (when electron from valence band goes to the conduction band via mid-gap state) in reverse-biased pn-junctions (shot noise), or recombination of the electrons from the conduction band with holes from valence band in forward-biased pn-junctions (reduction of signal or gain). When such mid-gap state is situated close to the edge of one of the bands, it can trap charge and release it after some time. Devices sensitive to this type of damage are bipolar transistors, optocouplers, and optical detectors.
Displacement damage does not depend on the total absorbed energy, but on the non-ionising energy loss (NIEL), which refers to the mass and energy of the incident quanta. So it is important to take into account what type and energy of radiation particles is. The table 2.1. shows a comparison of relative displacement damage for different types of radiation [2]:

Table 2.1. Comparison of relative displacement damage for different types of radiation

Particle	**Proton**	**Proton**	**Neutron**	**Electron**	**Electron**
Energy	*1 GeV*	*50 MeV*	*1 MeV*	*1 MeV*	*1 GeV*
Relative damage	1	2	2	0.01	0.1

Ionisation damage occurs due to liberation of charge carriers from insulating layers (for example SiO_2 used widely in silicon circuits for insulation) by ionisation mechanism. These free carriers drift or diffuse to other layers, where they can be trapped and contribute

to parasitic fields. Freed electrons are much more mobile than holes, thus the latter are more probable to be trapped. Holes trapped in oxide layer contribute to positive charge build-up; on the other hand holes trapped at the silicon-oxide interface may result in electron trapping. For example, in NMOS transistor, holes produced by irradiation in the gate oxide cumulate in the oxide and build up positive charge. Therefore, threshold voltage decreases. However, this is true only at low radiation level, at higher level the threshold voltage increases and can even pass pre-radiation value. This is effect of formation of negatively charged acceptor interface traps. The change in PMOS transistor is smaller than in NMOS transistor, but positive charge trapped in the gate and lateral oxide decrease the threshold voltage. Such changes of the device characteristics can severely affect functioning of analogue circuits, because operation points change, but also of digital circuits, because switching times are affected.

Ionisation damages are independent on the type of radiation, but rather on the total absorbed ionising energy (Total Ionising Dose – TID). Typically the ionisation mechanism is the main absorption mechanism (for gamma radiation, hadrons, electrons and ions), therefore TID is usually expressed in terms of total energy absorbed per unit volume (1 rad = 100 erg/g or 1 Gray = 100 rads). The same TID causes ionisation damages of different scale in different materials; therefore, beside TID also absorbent material has to be stated.

2.2.2. Single Event Effects

Single Event Effects (SEEs) are result of hitting sensitive circuit elements by a single energetic radiation particle. Here we have to move to a statistical domain, since we cannot say exactly when such effect occurs, but we can just estimate probability of such event. SEE can be divided into permanent effects and transient effect. Permanent effects are those, which permanently change the structure of the device, so called “hard errors” for example Single Event LatchUp (SEL) in CMOS ICs, which turns on parasitic transistors in the circuits what destroys power lines if power is not turned off fast enough. Usually manufacturers care about latch up that can occur due to improper powering sequence, but do not take into account effects caused by radiation. Permanent effects similar to SEL occur also in power MOSFETs (Single Event Gate Rupture) and BJTs and power diodes (Single Event Burnout).

Transient effects affect functioning of the devices only temporarily, therefore, they have biggest impact on digital circuitry, and such effects in analogue circuits can even pass unnoticed. Examples of transient SEEs are Single Event Upsets (SEUs), which change contents of the memory and Single Event Transients (SETs) that are transient changes of the signals on the lines. When SEU changes bit in the register responsible for the functioning of the whole device, it can be called Single Event Functional Interrupt (SEFI).

The single high-energy ionising particle can leave behind an ionised path of electron-hole pairs. When carriers are liberated in the depletion region electric field puts them in systematic motion and current spike occurs. Figure 2.1. illustrates the phenomenon.

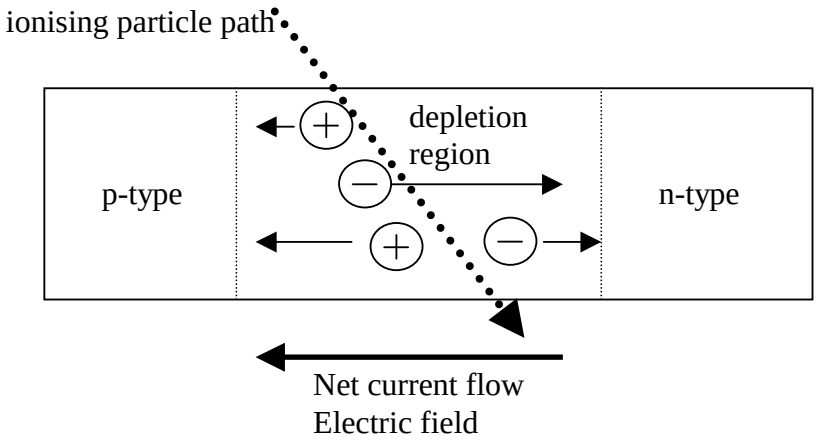


Figure 2.1. Mechanism of charge deposition in depletion layer by high energy ionising particle

But the interaction of the ionising particle does not end just on the surface of the silicon device. It goes further and can deposit lot of charge in the bulk of device, which when collected can contribute to even higher current flow. This phenomenon is so called funnelling.

Current spike can produce flipped state on signal line, we have SET – a small glitch on the line, which can result in wrong output. But if such glitch occurs in memory cell, it can change its contents.

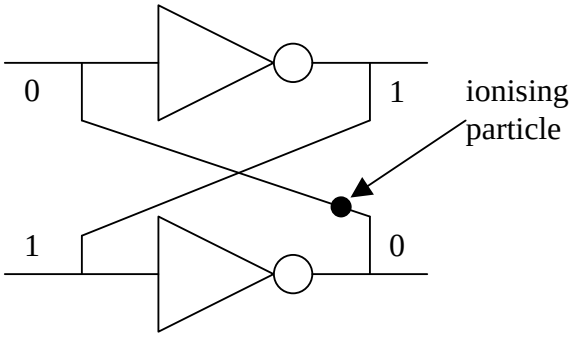


Figure 2.2. Bit-flip mechanism in Static RAM cell

Figure 2.2. presents SRAM cell or just a latch affected by ionising particle. If such particle changes state on any of the line (here we suppose it changes state from 0 to 1), the contents of the cell will be changed. In very sensitive devices, such single particle hit can even cause Multiple Bit Upset (MBU). SEUs are so called “soft errors”, they do not impair the physical structure of the device, therefore, can be removed by reprogramming of the device. The rate at which SEUs occur depends strongly on the type of particles (type of radiation), energy of the radiation particles, device sensitivity and Linear Energy Transfer (LET). Device sensitivity, in turn, depends on the Sensitive Volume (SV) because not all elements of the device are sensitive to glitches on the lines and on the critical energy (E_{crit}) – energy needed to change a state of the line. LET is the energy that can be transferred by given particle to the matter penetrated by the radiation on certain distance and. LET is defined in (2.1).

$$LET = \frac{dE}{dx} \tag{2.1}$$

For example heavy ions are high LET particles and hadrons are low LET particles.

The probability of SEE is very hard to estimate, because not only heavy ions can deposit energy bigger than critical energy. Hadrons that are not able to deposit sufficient energy

in sensitive volume can by nuclear interaction produce heavy ions and thus deposit energy bigger than E_{crit}.

2.3. Mitigation Techniques

Mitigation in case of effects connected with cumulative effects can be done on the physical structure level. As we can imagine, the number of carriers liberated in the insulation layer depends strongly on the thickness of this layer. Therefore, in order to minimise the effect of ionisation damage or displacement damage, we have to reduce the thickness of the insulating layer, what in practice means, use the device produced in smaller gate length technology. Not only the thickness of the gate oxide is the problem, but also oxide insulating two adjacent transistors in the IC - field oxide (which is usually much thicker than gate oxide and unfortunately does not scale down with gate length) has to be taken into account. Trapping of the charges at the oxide-silicon interface can lead to leakage current (holes trapped in the field oxide-silicon interface can create inversion layer) between two transistors what produces increased power consumption and can lead to IC malfunction. Thick oxide near source and drain edges can be a problem. Parasitic transistors can be created there, because whole device is covered with thick oxide layer (except the area under the gate). The figure below explains the problem.

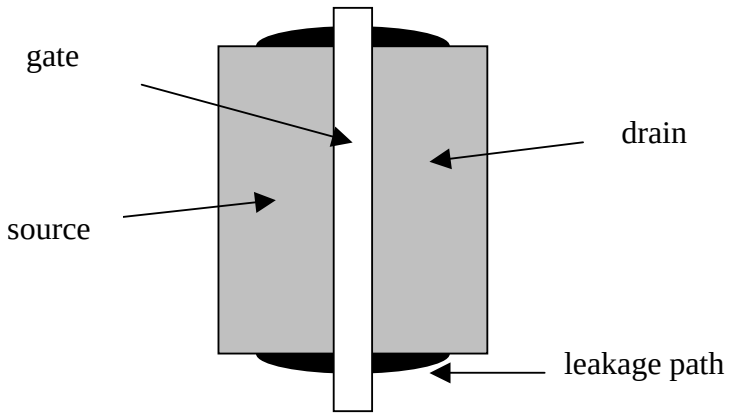


Figure 2.3. Parasitic leakage paths in MOS transistor

The solution to those two problems is device guarding. This is technology-hardening method. The idea is explained in the figure 2.4.

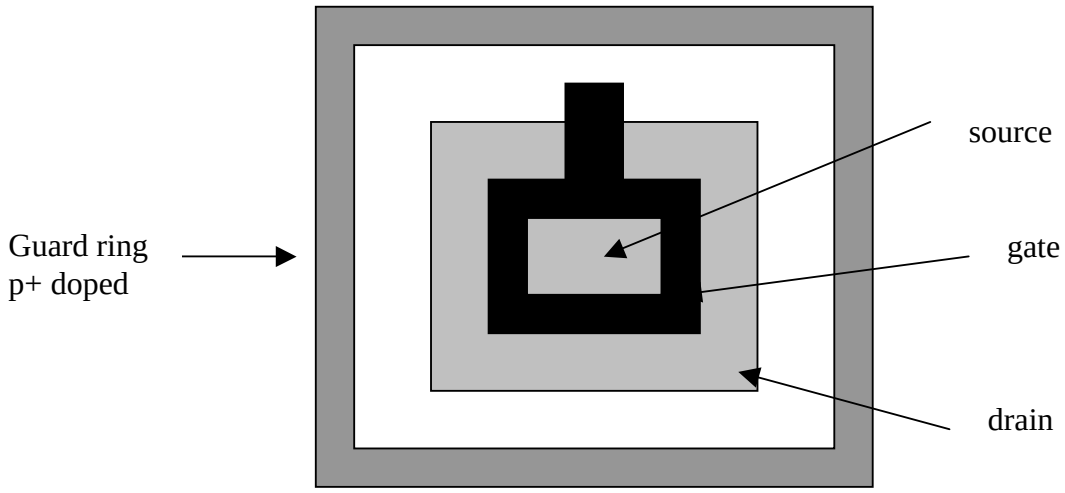


Figure 2.4. Guard ring and gate surrounding the source

The gate totally encloses source, so parasitic transistors do not create because gate oxide is thin enough to restrict or even eliminate cumulative radiation effects. Heavily doped p+ guard ring cuts the leakage current between two adjacent devices, because holes trapped at the oxide-silicon interface will not be able to create inversion layer.
Unfortunately every solution has its drawbacks. This solution increases size of the IC elements, what in turn results in decreased circuit speed, worsens packing density and increases power consumption.
It is also worth mentioning, that cumulative effects of radiation can be removed by device annealing.

The technology goes forward very quickly, the devices are scaled down what reduces consumed power, increases switching speed and increases number of devices per area unit. When scaling down V_{DD} is decreased and capacitance of the individual devices decreases, thus critical energy is also smaller. But we have to remember, that sensitive volume also decreases due to smaller feature size. In DRAM cells the cell area scaling seems more important than decrease of capacitance, and therefore DRAM sensitivity decreases when scaling down. For SRAM answer is not so clear, it seems that both phenomena cancel each other [4], but experimental results show, that error rates increase with scaling [5].
There are different ways to eliminate or limit SEEs. One of the approaches is to change physical structure of the device to make SEE less probable. It is so called “hardware hardening”. One of the ideas is Dual Interlocked CEll (DICE) structure [6], which adds

some redundancy to the circuit. The latch in this structure requires voltage change on two nodes in order to change the information stored in a cell. But technology is successively scaled down and it becomes probable, that one incoming ion can produce MBU and cause bit flip in the DICE cell. The Triple Interlocked CEll (TICE) is under investigation and should be much more immune to SEUs. Other circuit level idea of hardening the device is resistor-decoupling technique, where resistor is put in series with each inverter gate. The resistor, together with inverter gate capacitance, forms RC low-pass filtering circuit. This filtering circuit can filter out high frequency components and thus eliminate current spike at signal line. But again scaling down of a feature size becomes a problem. With smaller gate length, the capacitance of the gate decreases and to keep filtering circuit cut-off frequency at desired level, we have to increase the resistance. With large resistance values (order of MΩ in technology 0.25 μm [6]) the technological aspect starts to play significant role. Resistors that can provide such resistance are strongly thermally dependent and can change cell characteristics considerably over operating temperature interval. The solution to this issue is additional capacitor inserted in parallel to the signal line. This leads to reduction of the required resistance.

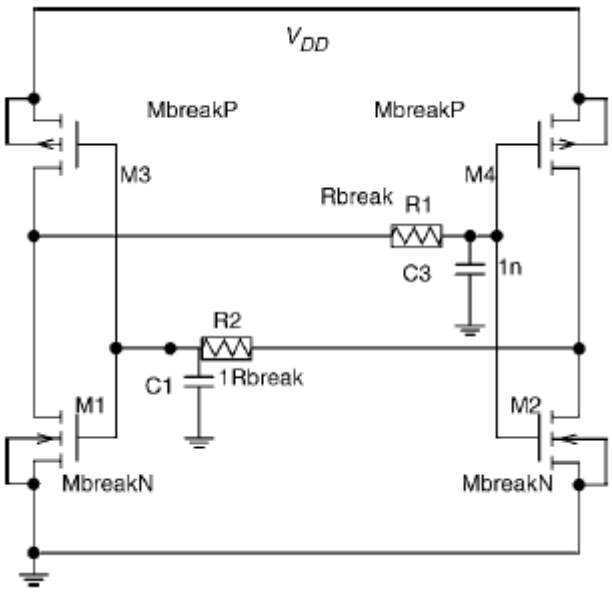


Figure 2.5. RC radiation hardened SRAM memory cell [6]

Above presented methods are used for “hardware hardening” of SRAM cells, so volatile memory cells. In case of non-volatile memories, like EPROMS, EEPROMS and FLASH memories other methods have to be used. Usually such memories are built using floating

gates. Floating gate is similar to the normal transistor, but it has two gates (selection gate and floating gate). The structure is shown in figure 2.6.

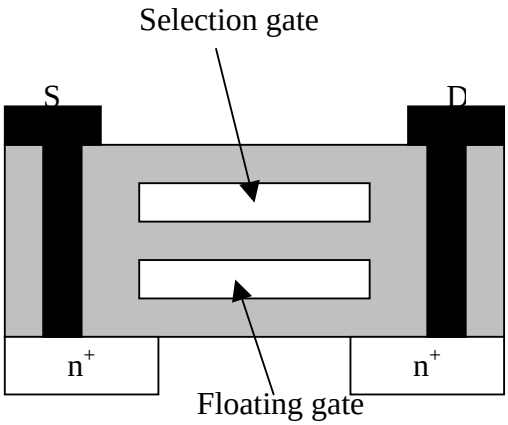
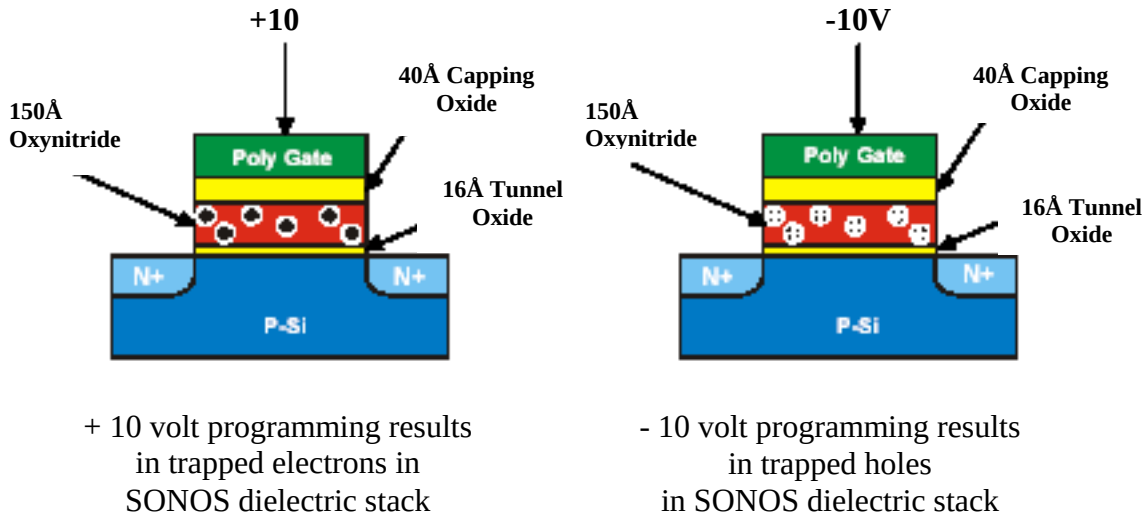


Figure 2.6. Floating gate transistor structure

The floating gate is isolated from the substrate with thin tunnel oxide (t_{ox} < 100 Å). When high voltage is applied (about 20 V) between Source and Selection gate – Drain, high electric field is created, which causes avalanche injection of the electrons to the floating gate, where they get trapped. Accumulation of the charge in the floating gate changes the threshold voltage of the device towards the higher voltages. This process is self-limiting, the charge build-up in the floating gate effectively lowers the electric field and stops avalanche injection. The state of the device (stored information) is checked using Selection Gate. The low voltage (just enough to overcome the threshold voltage of non-programmed device) is applied to the Selection Gate. If the floating gate does not hold accumulated charge – the inversion layer is created and current can flow between source and drain (device stores “0”), but if it holds some charge, inversion layer cannot be formed and there is no conduction path between source and drain (device stores “1”). Silicon dioxide isolates floating gate from any conducting parts of the circuit and once programmed, device can hold information for decades, because leakage of the charge is very, very small. In EPROM data can be erased by strong UV irradiation (what generates some electron-hole pairs) in the oxide and allows charge to leak from the floating gate. UV erasure is very slow; therefore EEPROM is in wider use. In EEPROM tunnelling mechanism is reversible by high negative voltage.

Memories built using floating gates differ from the volatile memories mentioned earlier and have different radiation sensitivity. Experimental data proves that TID effects are main

reasons of data loss in floating gate devices [7]. The ionising damage puts some variation on threshold voltage of the devices, moreover, it can damage the tunnelling oxide and cause charge leakage. But not all experimental results can be justified by cumulative radiation effects. Heavy ion can discharge the floating gate devices [8]. The solution to this problem is the Silicon-Oxide-Nitride-Oxide-Silicon (SONOS) device. The technology has been developed by Northrop Grumman Corporation (NGC), which is involved in space applications of non-volatile memories for over 30 years. The structure of such device is shown in the figure 2.7.



+ 10 volt programming results in trapped electrons in SONOS dielectric stack

- 10 volt programming results in trapped holes in SONOS dielectric stack

Figure 2.7. SONOS transistor stack [8]

These are stacked transistors. First there is 15 Å of thermal oxide, than 150 Å of silicon nitride, 40 Å of blocking oxide and phosphorous-doped polysilicon gate. Positive voltage of 10 V applied to the gate results in electrons trapped in nitride layer, negative voltage results in holes accumulation. The charge is stored in traps in silicon nitride layer, therefore it cannot be removed as easy as in case of conducting polysilicon floating gate. The retention of the data depends on the width of programming pulse, but is between 10 to 100 years! The number of reprogramming cycles is estimated to be 100,000 times.

We have to keep in mind, that while such methods (on the hardware level) are very effective and do not require changes on the higher levels, they are also very expensive. The radiation-hardened devices are not widely used, they are suitable only for certain applications, therefore are produced in small series, what increases the price. Moreover research and development of such circuits requires many expensive tests and experiments

may require change of the technology in the silicon foundry. Sometimes, when we deal with environment in which radiation effects are not so severe and often, it is better to use commercial ICs, but use radiation hardening on the system design level ("software hardening").

Obvious hardening method implemented at system design stage is data duplication. Of course, we can use two separate places to store the same data; everything is fine until no data difference occurs. When data from one storage is different than from the second one, we have to decide somehow, which is the true value. We have implement at least some error detecting coding scheme. This approach has its drawbacks: double memory needed to store data, slow memory access due to coding/decoding and double copying. But if we have to implement at least error detecting coding, maybe we could sacrifice some speed, but use only one storage place to store data with Error Detection And Correction (EDAC).

Coding schemes which allow error detection or detection and correction need some redundancy. We have to add some bits to the information to facilitate detection and correction. Some memory ICs have even dedicated EDAC circuitry, but due to costs this is not true in commercial ones. In such a case, we have to implement EDAC entirely in software. Of course the problem is that usually code and data is stored in SRAM and therefore is sensitive to SEEs. When error occurs in data, this is not a problem, the EDAC would correct the single error, but error in code can lead to IC malfunction. The code memory segment can be also protected with some redundant bits, but if error occurs in the instruction that is going to be executed, the unpredictable behaviour can be the result. Coming back to the protecting coding schemes. There are two type of coding schemes: systematic (separable) and non-systematic (non-separable). Separable codes keep protected data intact, but add some check bits. Non-separable codes mix check bits with data bits. In our case it seems reasonable to use systematic coding to keep data in memory as is, but keep some additional data to be able to detect and correct errors.

One of the most popular and simple coding scheme is parity checking. This code adds one bit to the information and is able to detect odd number of errors, but cannot correct any errors. Usually even parity is used. It means that extra bit keeps number of "1"s in the protected information always even. Suppose, the 8-bit information is protected. Table 2.2. presents the idea of even parity checking.

Table 2.2. Even parity checking can find only odd number of errors

	Bit 7	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**	**No of "1"s**	**Parity bit**	**Check result**
No error	0	1	1	0	1	1	0	1	5	1	OK
1 error	0	1	1	***1***	1	1	0	1	6	1	**Error**
2 errors	0	1	1	***1***	1	1	***1***	1	7	1	OK
3 errors	***1***	1	1	***1***	1	1	***1***	1	8	1	**Error**

As we can see parity checking is not very suitable in case of SRAM protection against errors resulting from SEEs. It only detects errors, but cannot correct any, what could only help in case of doubled memory storage. Method can be modified to enable error correction. Data is stored in the memory usually in 8-bit addressable cells. Apart from one extra parity bit for each cell as horizontal protection, one cell is devoted to vertical protection. Table 2.3. presents 3 memory cells protected with 8 vertical and 3 horizontal parity bits and Table 2.4. same memory with single error.

Table 2.3. Three memory cells protected with vertical and horizontal parity

Address	**Bit 7**	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**	**Horizontal Parity bit**
0x0000	1	0	0	1	1	1	0	0	0
0x0001	1	0	1	0	1	0	1	0	0
0x0002	1	1	1	1	0	0	0	0	0
Vertical Parity Bit	1	1	0	0	0	1	1	0	

Table 2.4. Three memory cells protected with vertical and horizontal parity with one error

Address	**Bit 7**	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**	**Horizontal Parity bit**
0x0000	1	0	0	1	1	1	0	0	0
0x0001	1	0	1	***1***	1	0	1	0	***0***
0x0002	1	1	1	1	0	0	0	0	0
Vertical Parity Bit	1	1	0	***0***	0	1	1	0	

Single error position can be precisely determined and corrected. More errors can be corrected, but only when no two errors occur simultaneously in the same row or column. In such a case, only odd number of errors can be detected in the row or column. It has to be kept in mind, that every protection method adds significant overhead to the read/write operations and needs some additional memory. In this case, every 8 bytes of protected memory require 10 bytes of storage, 1 byte for horizontal parity bits, 1 byte for vertical parity bits. Of course vertical parity byte does not have to be inserted after every 8 bytes of memory, it can be put more often or less often. This strongly depends on the probability of double errors in one row or column in smallest cross-protected area.

Another frequently used protection code is Hamming code. Is also based on additional parity bits, but it allows to correct single error or detect up to two errors, but not both simultaneously. The number of required additional bits is determined by “Hamming rule”:

$$2^m \geq m + d + 1 \tag{2.2}$$

Where d is the number of data bits to protect and m is the number of parity bits. A code, which is constructed in such a way, that equality sign can be used in (2.2) is called a perfect code. Codes are denoted as (m+d, d) Hamming codes.

Suppose, the (7,4) Hamming code is constructed. There are 4 data bits and 3 parity (check) bits in this code, which can indicate 7 positions of an error, position 000 indicates no error. In such a code parity bits are also protected. Table 2.5. shows positions of errors and corresponding parity bits values.

Table 2.5. (7,4) Perfect Hamming code parity bits values

Position	**Bit 2**	**Bit 1**	**Bit 0**
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

Bit 0 is responsible for parity check on positions 1, 3, 5, 7. Bit 1 on positions 2, 3, 6, 7. Bit 2 on positions 4, 5, 6, 7. The rule is that code should be constructed in such a way, that

no parity bit checks the other parity bit. This can be achieved by placing parity bits on positions 1, 2, 4, since these positions contain only one "1" in binary representation and are checked only once. Parity bits are determined exactly in the same way, as in even parity coding, for example 1 Parity bit is 0 since bits on positions 3, 5, 7 are 1, 0, 1 (two "1"s in total). As an example consider the following data: 1011. The resulting Hamming code looks as in Table 2.6.

Table 2.6. Hamming code for 1011 data

Position	***1***	***2***	***3***	***4***	***5***	***6***	***7***
Function	***Parity 0***	***Parity 1***	***Data 3***	***Parity 2***	***Data 2***	***Data 1***	***Data 0***
	0	*1*	*1*	*0*	*0*	*1*	*1*

Decoding is a matrix multiplication of matrix containing possible combinations of parity bits and received data. Modulo 2 arithmetic has to be used to calculate the multiplication. In this case:

$$\begin{bmatrix} 0&0&0&1&1&1&1 \\ 0&1&1&0&0&1&1 \\ 1&0&1&0&1&0&1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \tag{2.2}$$

What indicates that there is no error (error position 000). The erroneous data 0010011 would give:

$$\begin{bmatrix} 0&0&0&1&1&1&1 \\ 0&1&1&0&0&1&1 \\ 1&0&1&0&1&0&1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \tag{2.3}$$

what indicates error on position 010 = 2. The mechanism of error detection and correction by Hamming code is based on Hamming distance. Hamming distance is the number of bits at which two code words are different and Hamming rule ensures that that distance

between any two valid code words is at least 3. Therefore, to change from one valid code word, to another code word, at least 3 bits have to change. When only one bit changes, it is possible to decide which was the correct valid code word and correct the error, because this is the only one with Hamming distance of 1 from the code word read with error. When two bits change, it is only possible to detect that double error occurred, because read code word is not valid. Unfortunately error cannot be corrected since it is close (Hamming distance = 1) to another valid code word. That is why perfect Hamming code is able to correct single error and detect double error, but cannot do both functions simultaneously. Of course, code with bigger Hamming distance between any two valid code words can be selected. For example code (7,3) is able to correct single error and detect double error simultaneously (minimum Hamming distance is 4). One could say that Hamming code is non-separable code and is hard to implement in hardware. But we can make it separable by moving parity bits to the end of the data. Implementation is also simple. XORing appropriate bits can do the coding. For example bits on positions 3, 5, 7 in the above considerations were 1, 0, 1. Therefore Parity bit 0 is:

$$p_0 = b_3 \oplus b_5 \oplus b_7 = 1 \oplus 0 \oplus 1 = 0 \tag{2.4}$$

Decoding is also straightforward. First the error position has to be determined. For example for received data r = [0010011]:

$$\begin{aligned} e_0 &= r_1 \oplus r_3 \oplus r_5 \oplus r_7 = 0 \oplus 1 \oplus 0 \oplus 1 = 0 \\ e_1 &= r_2 \oplus r_3 \oplus r_6 \oplus r_7 = 0 \oplus 1 \oplus 1 \oplus 1 = 1 \\ e_2 &= r_4 \oplus r_5 \oplus r_6 \oplus r_7 = 0 \oplus 0 \oplus 1 \oplus 1 = 0 \end{aligned} \tag{2.5}$$

If the position of error is different than 000, the bit has to be flipped.

There are other simple and more sophisticated coding schemes [9], which could be employed for memory protection against SEE induced errors, but there is no point in describing them here.

Until now, the memory protection techniques were mentioned, but systems as a whole are also subject to SEEs. For example SETs can change output of a device for a short time. It is not important, whether it is combinatorial circuit or sequential circuit; it produces bad output for a while. This can lead to malfunction of other connected systems. One of the techniques, which alleviate this problem, is voting circuit. This technique has been used for many years in electronic devices working in space. The idea is simple. There are three

modules, which perform the same function and are subject to SEEs. Their outputs are connected to the inputs of voting circuit. If two of them have the same output and the third has different or all of the three have the same outputs, the output of the voting circuit is set to the value that is present on the majority of inputs of the voting circuit. The figure 2.8. presents example of voting circuit implementation.

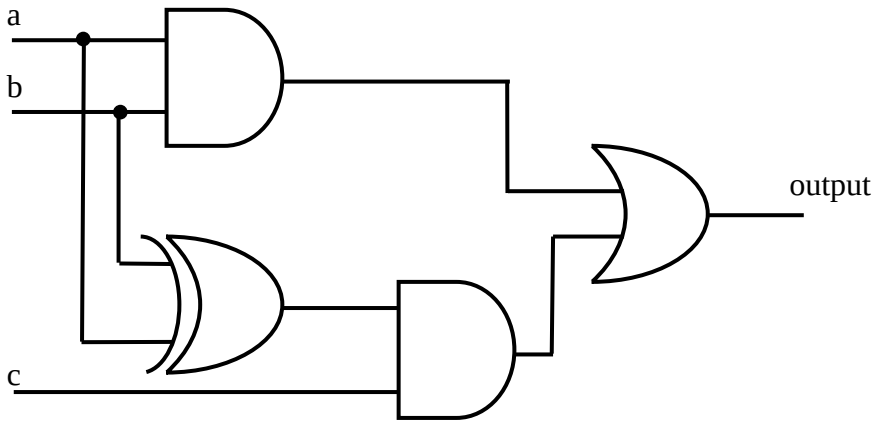


Figure 2.8. Voting circuit implementation

When implemented in programmable circuit, which stores its configuration in memory insensitive to radiation or built from the discrete elements the voting circuit is only a subject to SET. However, it is very simple and small and probability of SET is usually small. Of course, when voting circuit is implemented in the programmable circuit, which stores its configuration in memory sensitive to radiation, voting circuit also becomes a subject to SEUs.

2.4. Programmable Circuits

Mass-produced Integrated Circuits (ICs) are usually relatively cheap, fast and widely available, moreover the selection of IC types and manufacturers is so big, that many projects can be realised using only "stock" ICs. But in some cases an Application Specific Integrated Circuit (ASIC) is needed because of project requirements, which can be for example device size, speed or power consumption. However, because of low quantity production ASIC manufacture process is usually long and expensive. Therefore, many prototypes and even many final products are built using programmable circuits. Most of the combinatorial or sequential circuits can be realised using these Field-Programmable Devices (FPDs). They are produced in large quantities, what decreases the unit cost, but are flexible and can be programmed to realise user defined functions, what in turn lowers the start-up cost and the financial risk of the project.
Taking into account the complexity of those circuits, they can be divided into three groups: Simple Programmable Logic Devices (SPLDs), Complex Programmable Logic Devices (CPLDs) and Field-Programmable Gate Arrays (FPGAs).

SPLDs

SPLDs are usually devices with programmable AND array followed by fixed or programmable OR array. Programmable Logic Array (PLA) devices have programmable AND and OR arrays, but this introduces significant propagation delays and the need for fast programmable circuits led to Programmable Array Logic (PAL) devices developed by Advanced Micro Devices. In PALs only AND array can be programmed and OR array is fixed. Architectures of programmable devices vary from vendor to vendor what is reflected in vendor-specific device names, like Generic Array Logic (GAL – Lattice Semiconductors trademark) which is a variation of PAL architecture with some additional features. GAL device is described later as an example. Array outputs can be registered to facilitate sequential circuits functions. Figure 2.9. presents the simplified structure of PAL device. In the figure, the short notation for AND matrix connections is used. Horizontal lines do not represent a single connection to the AND gate, but connections to all vertical lines that cross them. Moreover, usually there are more than 4 inputs and 2 outputs. Different configurations of PAL devices are available. The configuration of the device can be identified by the device name. For example 16R8

indicates maximum of 16 inputs and maximum of 8 outputs. The letter “R” means that the outputs are registered, other commonly used letter “V” means “versatile” and indicates that the outputs can be configured in various ways.

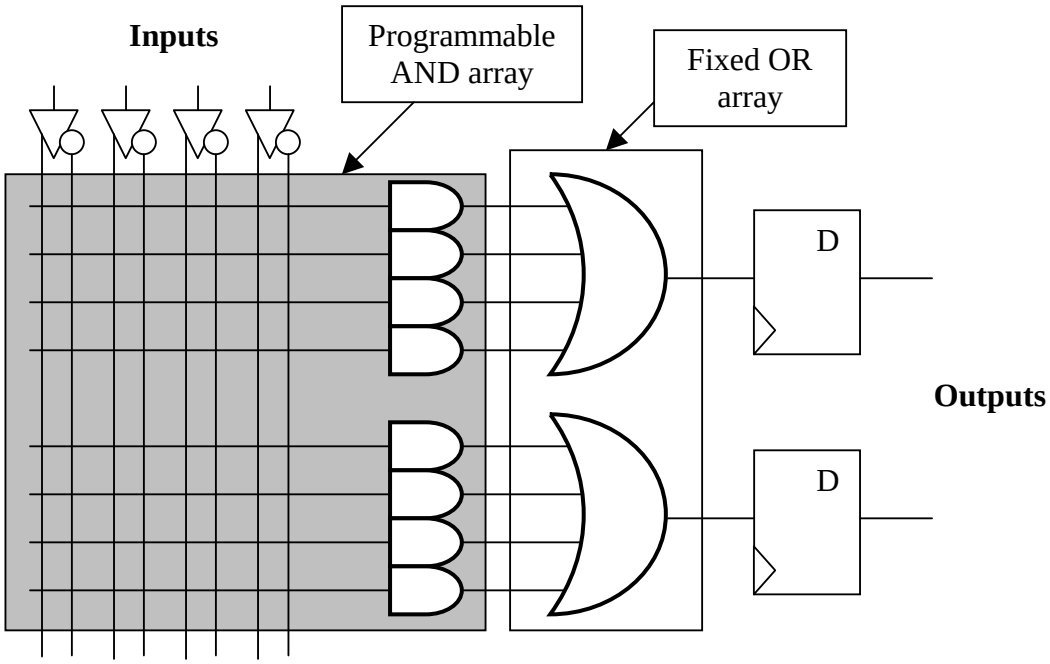


Figure 2.9. Simplified PAL structure

Good example of PAL device is GAL16V8 from Lattice Semiconductors, which is used in simulations throughout this thesis. Its functional block diagram is shown in the figure 2.10.

The Output Logic Macro Cell (OLMC) is the generic feature of this device, making it different from the standard PAL structure. The OLMCs can be configured with 2 global and 16 individual configuration bits into three operation modes: simple, complex and registered. The figure 2.11. shows internal OLMC structure in registered configuration for registered mode. In this mode all macrocells share common clock (CLK) and output enable (OE) control pins, any of the macrocells can be configured as registered output or input/output. XOR controls the polarity of the output. The output can be fed back to the AND matrix. In other modes register or feedback can be disabled. The configuration of the device is kept in the EEPROM memory, thus it is immune to SEUs. For further information, please refer to [10].

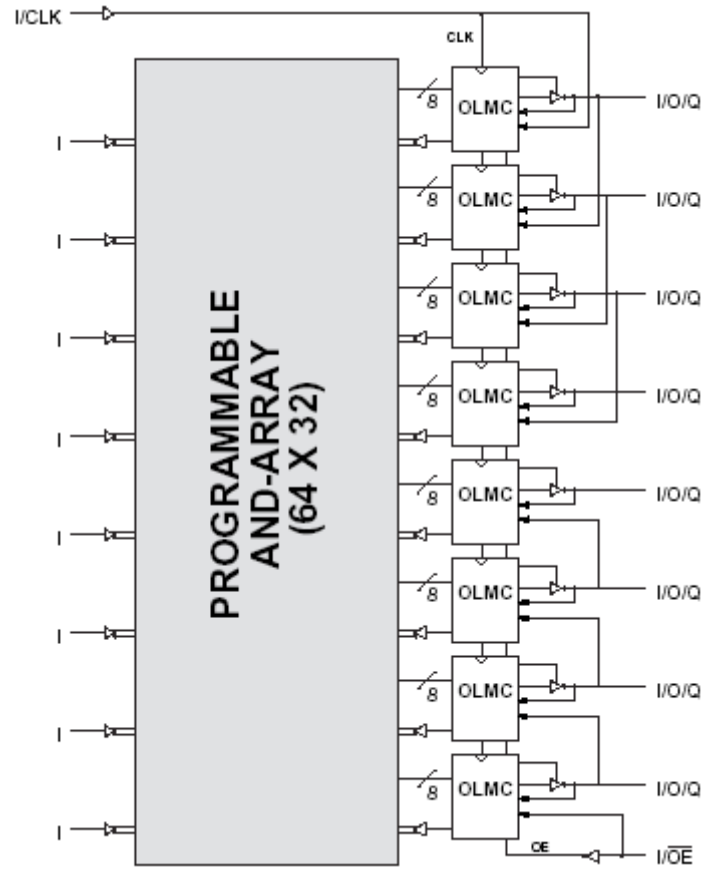


Figure 2.10. GAL16V8 Functional block diagram [10]

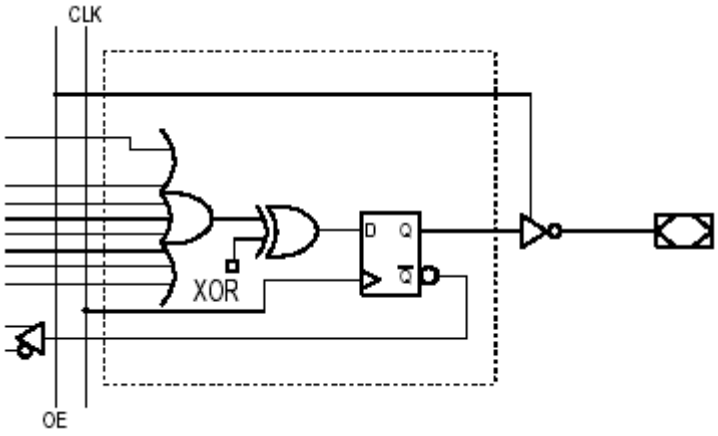


Figure 2.11. OLMC structure in registered configuration for registered mode [10]

CPLDs

CPLDs are the next step in programmable circuits evolution. SPLDs capacity cannot be easily increased because the programmable matrices take too much silicon area when

number of inputs and outputs is increased. Therefore, CPLDs are based on many SPLD-like blocks connected together to increase the programmable circuit resources.
The good example of CPLD is Altera MAX 3000 family. These devices contain from 32 to 512 macrocells. The device structure is shown in the figure 2.12.

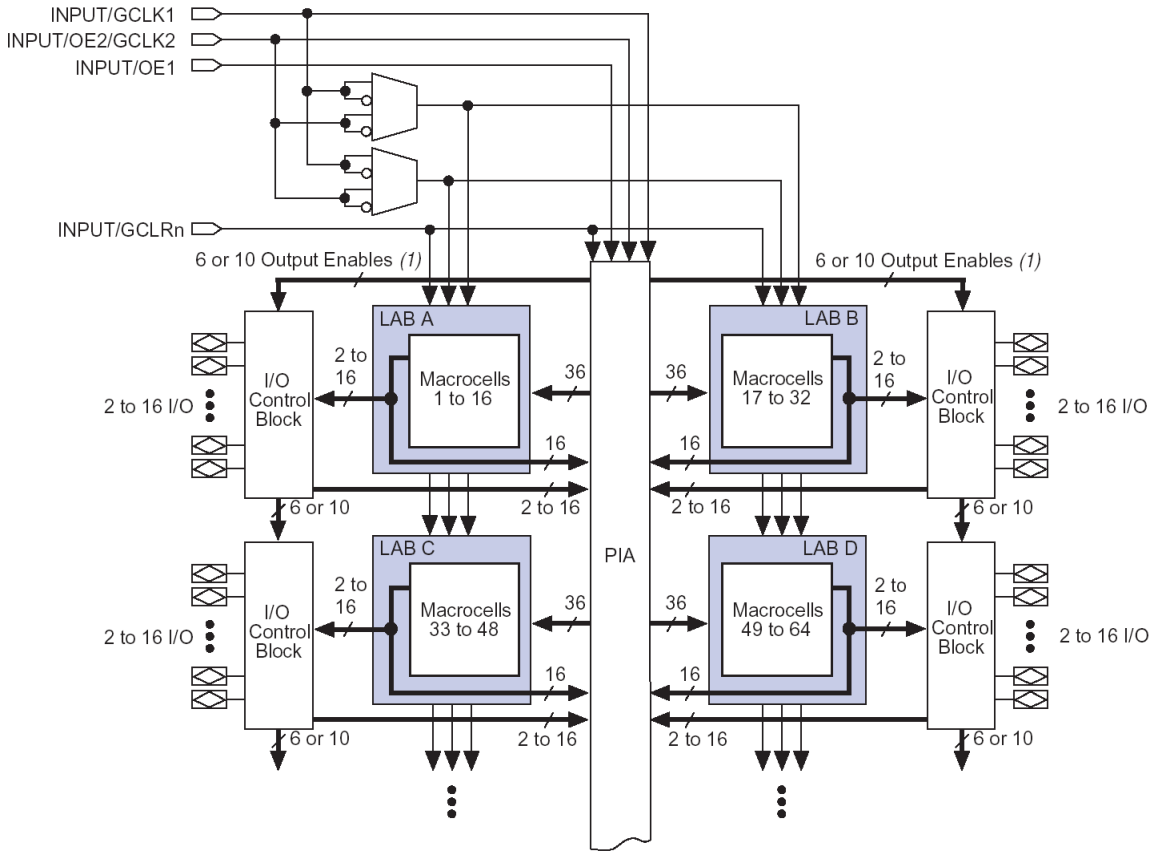


Figure 2.12. Altera MAX3000A device block diagram [11]

16 macrocells form a Logic Array Block (LAB). LABs are interconnected via Programmable Interconnect Array (PIA). PIA is fed by I/O pins, input pins and macrocells. The macrocell structure is shown in the figure 2.13. Each macrocell is similar to the PAL device. The Product Term Select Matrix (AND matrix) is programmable and product terms are directed to the OR and XOR gate. This part realises combinatorial logic functions. Register at the output of the macrocell can be used for sequential circuits. Two types of logic expanders are present in the macrocells. The Shared Logic Expanders enable inverted product terms to be fed back into the logic array. The Parallel Logic Expanders enable product terms from adjacent macrocell to be borrowed.

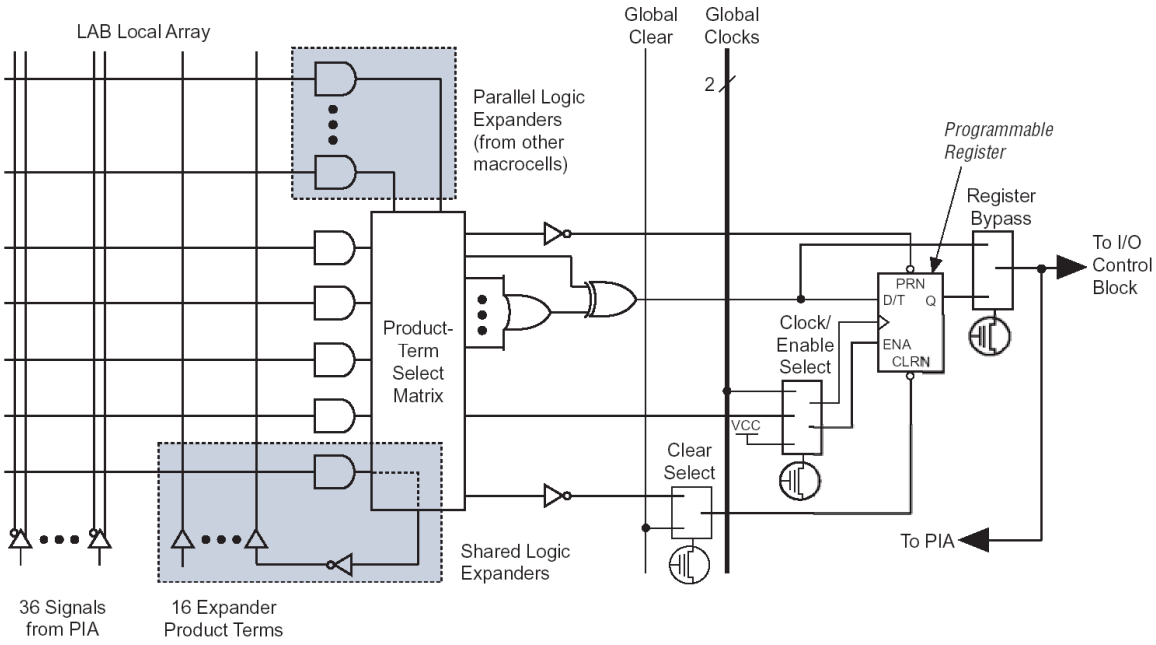


Figure 2.13. Altera MAX 3000A macrocell structure [11]

The configuration of the device is kept in EEPROM memory and the device is In-System Programmable (ISP), what means, that it does not have to be programmed in separate programming device, but its configuration can be changed in the target system. For further information, please refer to [11].

The Altera MAX programmable logic devices family is only one of many available on the market. Each vendor uses different device architecture, therefore the device type and vendor must be selected carefully, with all project requirements in mind.

FPGAs

Because some projects require more resources (more gates or registers) than CPLDs offer, the next group has been developed. The FPGA structure is not an extension of the CPLD architecture, but employs different approach. There is no longer product term selection matrix, but the combinatorial functions are realised using Look-Up Tables (LUTs). The typical FPGA structure is shown in the figure 2.14.

The Programmable Logic Blocks (PLBs) are interconnected using Programmable Interconnect. I/O Blocks control the pin functions.

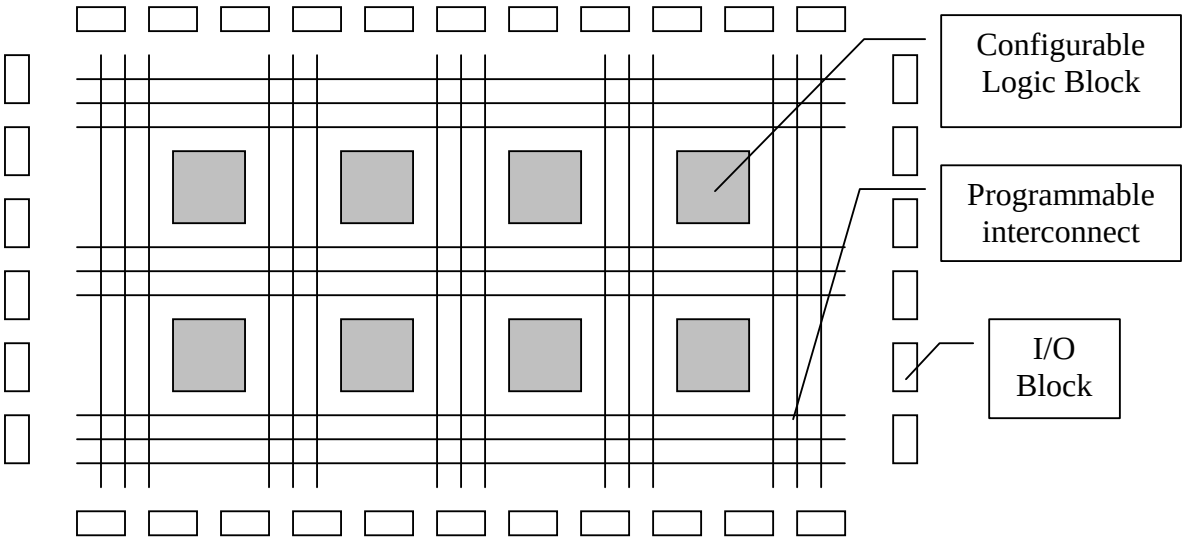


Figure 2.14. Typical FPGA structure[12]

It is good to describe the details of architecture using some exemplary programmable logic device. The good example is Xilinx XC4000 FPGA family. The figure 2.15. shows the structure of Xilinx XC4000 Configurable Logic Block (CLB).

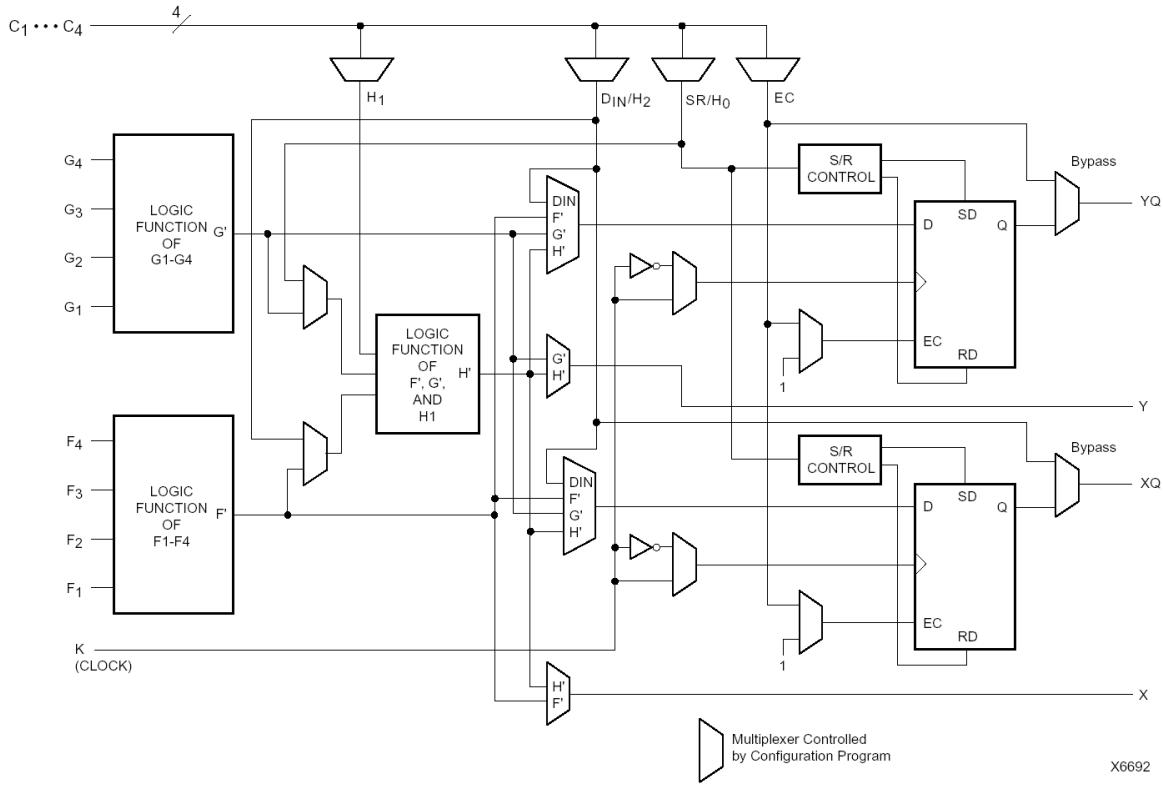


Figure 2.15. Block diagram of Xilinx XC4000 CLB [13]

The three logic function generators, built using LUTs, are provided. The G and F generators have 4 inputs, the third generator (H) has three inputs. The inputs of the H generator can be the outputs of G and F generators or inputs from outside the CLB. The sequential circuits are realised using two D flip-flops. The flip-flops can be fed by the logic function generators outputs or external signal. Moreover, combinatorial logic and the storage elements can be used independently, because there are separate, non registered outputs available (X, Y) for combinatorial logic. The LUTs used for logic function generators are simply the 16x1 bit SRAM memory areas. The inputs are address lines of the memory and the stored value is a function value. In this family of programmable logic devices, the function generators can be used also as high speed RAM, the single CLB can be configured as 16x1, 16x2 or 32x1 bit array. LUT usage in logic function generators makes the propagation time independent on the function implemented.

Figure 2.16. shows the I/O block structure.

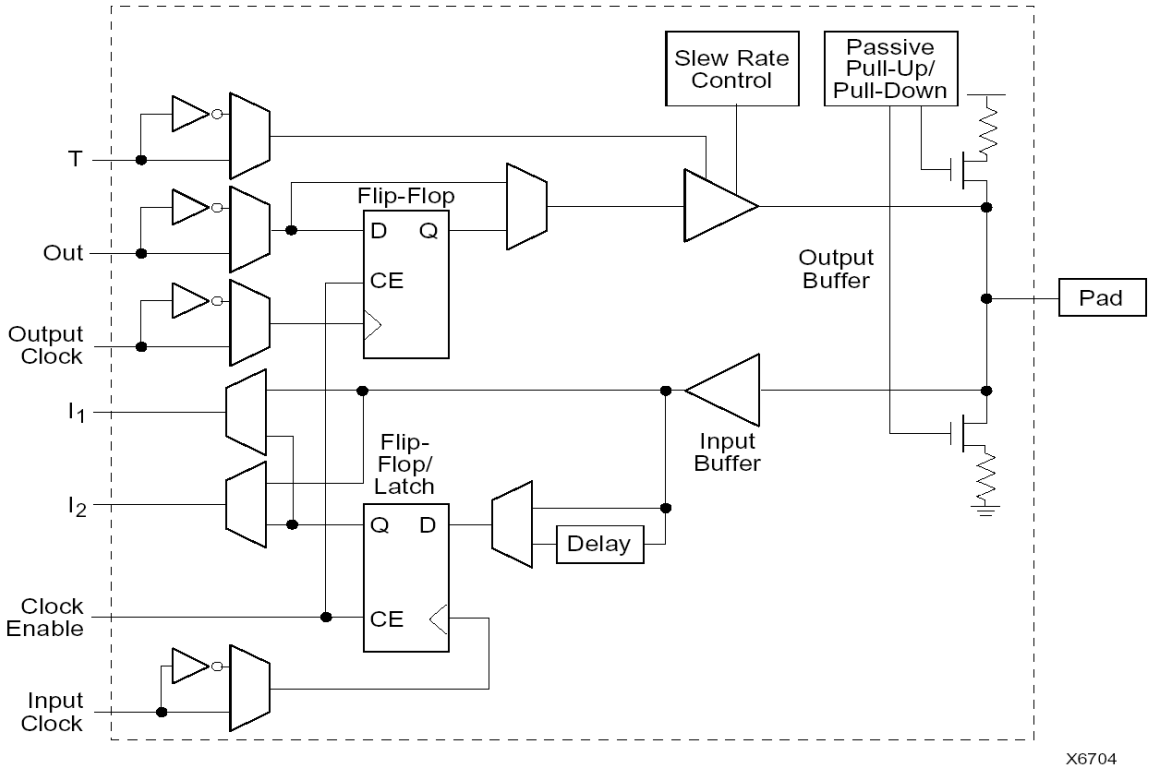


Figure 2.16. I/O Block structure [13]

The I/O blocks are interfaces between external device package pins and the internal connections. Each pin has its dedicated I/O block. The I_1 and I_2 inputs can be connected

to the pad directly (via the buffer) or through the D flip-flop, which can also be configured as latch. The Out output can also be connected directly to the pad (via the buffer) or through the D flip-flop. The input and output storage elements have common Clock Enable signal (CE), but use different clock signals. The output signal can be inverted in the I/O Block. The output buffer can be configured in the high-impedance state (using T signal).

The last detail of Xilinx XC4000 family architecture that needs explanation is the programmable connection between CLBs. These devices use hierarchical wiring structure. Namely, each CLB is connected to the five types of interconnects: length-1 lines, length-2 lines, length-4 lines, length-8 lines (only in XC4000X) and long lines. This feature simplifies the routing procedure done by the design software and enables better device resources usage. The wire segments are placed horizontally and vertically around the CLB. The cross points of vertical and horizontal wiring lines are called Programmable Switch Matrices (PSMs). The figure 2.17. shows single and double-length lines in the device and figure 2.18. shows internal structure of PSM.

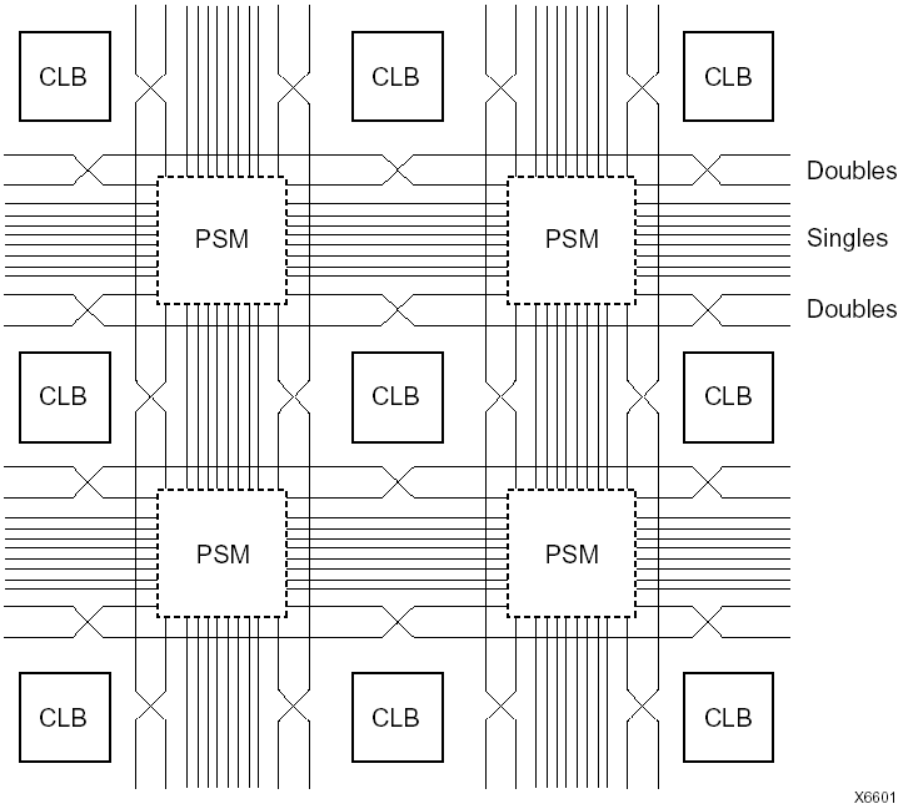


Figure 2.17. Single and double-length lines [13]

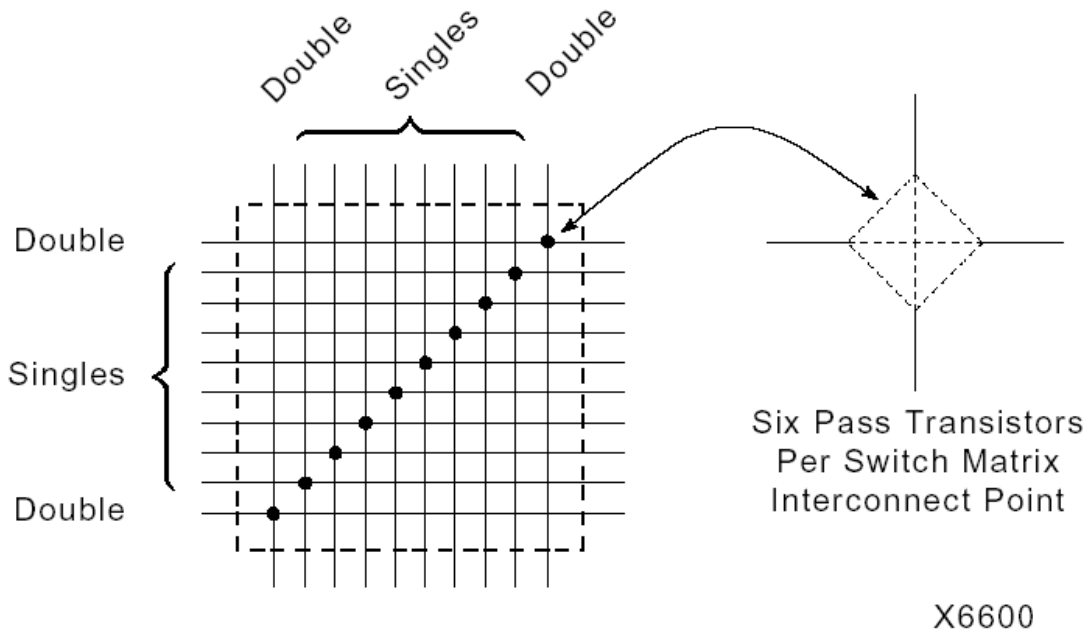


Figure 2.18. Programmable Switch Matrix

Additional wiring is placed in the outer device ring formed by I/O blocks routing.

The configuration of the Xilinx XC4000 family devices is stored in SRAM and therefore is a subject to SEUs. Xilinx XC4000s are mature products, there are much more powerful devices nowadays, but the family has been chosen as an example because of its relatively simple inner structure. For example Xilinx's Virtex-4 family of the FPGA devices belongs to the one of the most powerful on the market. The members of that family have many additional features beyond the programmable logic. These are: PowerPC 405 processor core available (PowerPC is an IBM trademark), an interface for user coprocessor, 622 Mb/s to 10 Gb/s serial transceivers, Digital Signal Processing slices (which can act as a simple DSP processors). The programmable logic is based on Advanced Silicon Modular Blocks (ASMBLs). For further information please refer to [14].

Radiation influence on the programmable circuits

In order to describe the influence of the radiation on the programmable circuits, it is advisable to describe the user-programmable switch technologies. The programmable switches are the key elements that enable different device configuration options.

EPROM or EEPROM based devices use structure presented in the figure 2.19.

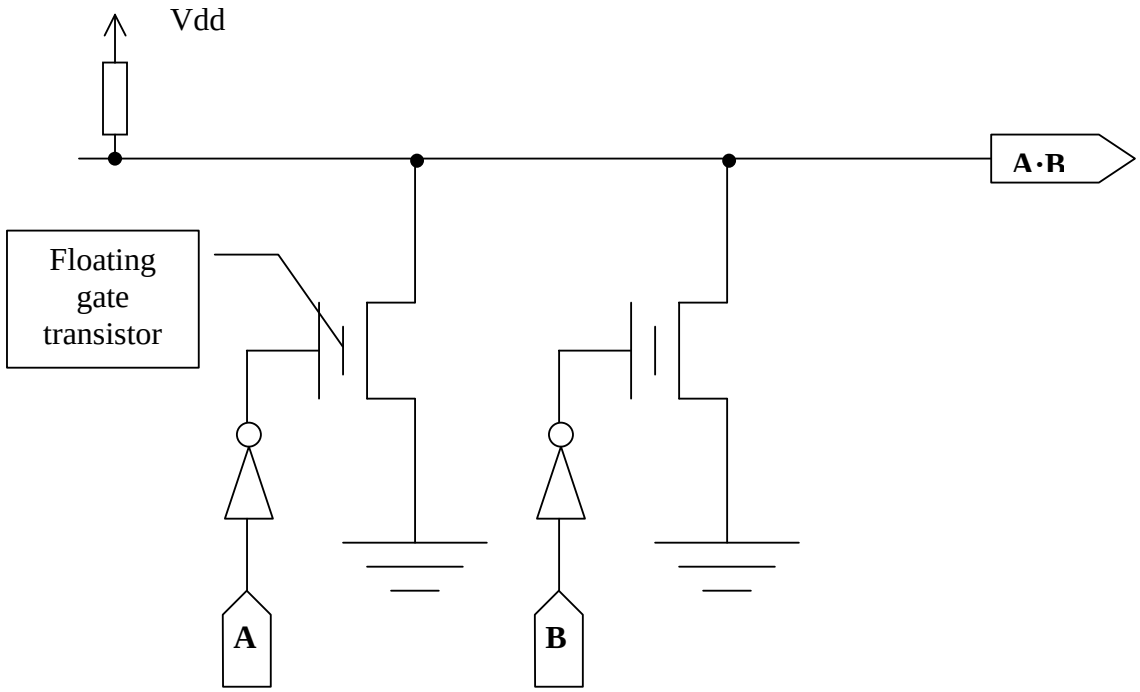
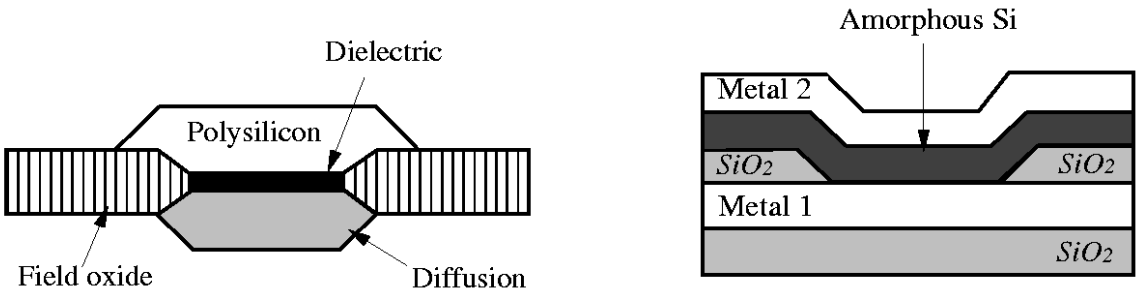


Figure 2.19. EPROM programmable switches realising AND function

These switches are build using floating gates. When the gate is charged (stores “1”) the input signal cannot change the state of product wire (the conducting path to ground cannot be formed). Therefore, only signals lines, where switch stores “0” contribute to the product. The arrangement shown above with inverters put on the inputs realises AND function if two floating gates are unprogrammed (store “0”).

SRAM based switches are simply pass-transistors with gates controlled by appropriate SRAM bits. The pass transistors are used in PSM as shown in the figure 2.18.

FPGA circuits employ also another programming technology – antifuse technology. This is one time programmable switch, which is build using CMOS antifuses. Antifuse is a device, that functions in an opposite way than the fuse does. It is composed of and insulating layer sandwiched between two conducting layers. Antifuse is initially non-conducting, because conducting layers are separated by the insulator. When current is passed through the antifuse (~5mA), the insulating layer melts and the conducting path is created. Figure 2.20. shows two commonly used antifuse technologies.



a) *b)*

Figure 2.20. Antifuse technologies: a)ONO antifuse b) amorphous antifuse [15]

The Actel's antifuse uses Oxygene-Nitrogen-Oxygene (ONO) dielectric layer put between polysilicon and n^+ diffusion layers. Quicklogic's antifuse employs amorphous silicon between two metallization layers.

Above considerations lead to the conclusion that the only switches that are subject to SEUs are SRAM based switches. Therefore SPLDs and CPLDs using EEPROM memory are only subject to SETs. FPGAs using SRAM for device configuration are a subject to SETs and SEUs. SEUs can affect interconnections between CLBs and LUTs as well. There are FPGAs using antifuse technology for interconnections programming, but this still does not alleviate problem of LUTs stored in SRAM. All devices are subject to cumulative radiation effects.

3. Application of Genetic Algorithms in Fault-Tolerant Circuit Design

As described in chapter 2 techniques mitigating the influence of radiation on the programmable circuits are implemented at different levels of circuit design process. The technique proposed below is implemented on the system design level. System is to be designed in a way, which minimises negative effects of bit flips in configuration data of programmable circuit. The ideal “radiation tolerant” configuration would provide proper device functioning with any of its bits flipped. Due to complexity of this task, only single error tolerance is assumed. Length of programmable circuits configurations depend on the type of the designed system and type of target device. Simplest configurations realizing simple combinatorial circuits in GALs have lengths of hundreds of bits. This gives enormous solution space, which cannot be searched in reasonable time using “check all” method. Search has to be directed somehow towards right solution. Genetic Algorithms (GAs) can be helpful in this case.

3.1. Genetic Algorithms

3.1.1. Idea of GAs

There are problems, where solution is contained in large search space. This is the case, when there are many parameters that have to be optimised simultaneously to find the right combination. Solution space size depends on the number of parameters and possible parameter values. Optimisation problems may be solved using analytical or numerical methods. However, former approach is applicable only to the class of problems, which can be described as analytical function. Usually real-life problems are complex and are difficult or impossible to describe analytically and can be solved only numerically, by searching in solution space. When the number of possible solutions is not too large, we can check all possible combinations and choose the best one. This approach is always successful, it always gives the best solution, or in other words global extremum of the function. But application of this algorithm is limited by the computational power of the computer used for program execution. Random searching could be employed, but this approach requires also much of CPU power, because the longer the algorithm runs, the better final solution we get. Moreover, the quality of the final solution depends on luck and does not guarantee that satisfying solution will ever be found. In cases of huge search space some Artificial Intelligence (AI) should be employed to direct search toward the places in the space with better solutions. Genetic Algorithms employ some sort of AI for the solving process, it is a compilation of random and intelligent search. These are stochastic algorithms, where search process works similarly to the processes that are responsible for evolution: inheritance and natural selection.

As an example, consider population of mice. There are fast, smart, slow and silly among them. Faster and smarter mice usually escape from the cat. Therefore, after some time, population of mice has majority of smart and fast, because other died. Of course some silly and slow mice also survive, because they are simply lucky. This population has an offspring. Next generation inherits genetic material from the parents and the children abilities are the mixture of fast, smart, slow and silly mice abilities. Mice become on average faster and smarter in every evolution step. Additionally nature introduces mutation of genes during reproduction, what results in mice with abilities,

which were not present earlier in the population. Evolution in natural environment goes very slowly, because individuals from one generation need some time to be ready to contribute to the next generation and it is hard to notice any improvements in our surrounding. In computer memory individuals may be represented by binary chromosomes. These chromosomes may exchange information or undergo mutation. In this virtual world it is possible to speed up the process and use evolution to find problem solution. GAs description uses some terms straight from natural genetics [16].

GA processes the population of **individuals**. Each of them is simply one of the possible solutions. Each individual is judged on the basis of **fitness function** value, which is the measure of goodness of the solution. The fitness function is constructed accordingly to the problem to be solved, it describes **environment** in which individuals are placed. The abilities, or in other words attributes of every individual are coded in its **genotype**, which in turn consists of **chromosome** or chromosomes. A chromosome consists of the elementary genetic units called **genes**. Values possible to represent by each gene are called **alleles.** Algorithm proceeds in iterations, creating new generation each time. Figure 3.1. presents basic GA loop.

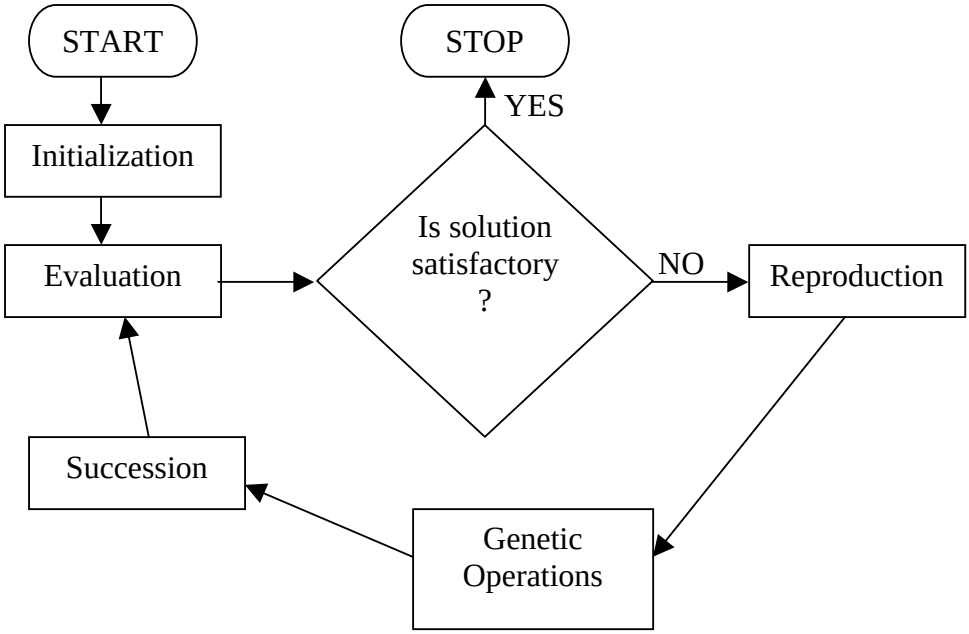


Figure 3.1. General Genetic Algorithm schematic

Initialisation step involves creation of initial population of individuals. This is done by picking chromosomes at random. Each gene in each chromosome is selected at random from the alleles of the gene. For the binary chromosome alleles of the gene are “0” and “1”.

Evaluation step involves calculation of fitness function value for each individual. After this step algorithm checks if best individual fulfils the solution requirements, if not, program goes on.

Reproduction step involves selection of individuals, which should contribute to the next generation. The intermediate generation is created. Selection can be done in variety ways, but should take into account value of fitness function of the individuals. In classic genetic algorithm, so called “Goldberg algorithm”, roulette wheel selection mechanism is used, where each individual occupies space on the wheel proportional to its fitness. The probability of selecting the individual to the intermediate generation is proportional to the space occupied by the individual on the wheel. Equation (3.1) presents probability p_i of selecting v_i from pop_size of population members.

$$p_i = \frac{fit(v_i)}{\sum_{j=1}^{pop\_size} fit(v_j)} \tag{3.1}$$

Therefore, best individuals usually place more than one copy in the intermediate population and worse have small chance to place any. This is similar to the natural selection.

Genetic Operations step involves crossing-over and mutation. Crossing-over is genetic operator that usually creates two children from two parents, but there are different crossing operators and algorithm designer can invent new ones. The simplest one is one point crossing-over as presented in the Figure 3.2.

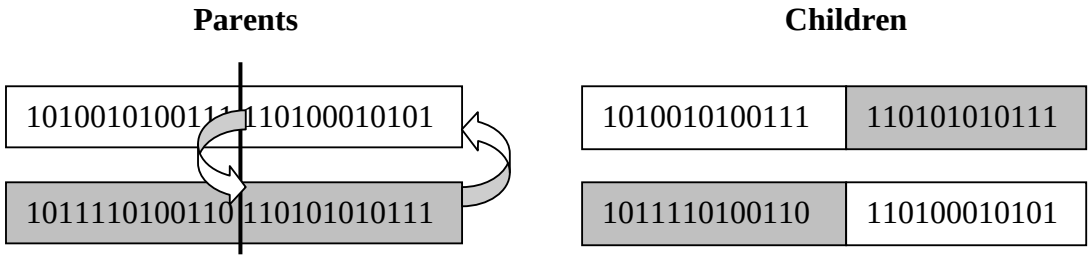


Figure 3.2. One point cross-over operation

The crossover position is picked at random, chromosomes are broken at this position and parts are swapped. One could think of two-point crossover, where there are two cross positions chosen randomly and parts between these positions are swapped or uniform crossover, where every gene in the child chromosome is taken from one or the other parent or even more complicated ones. Some exemplary crossover operators are described in section 3.1.4.

Next genetic operator is mutation. Mutation changes randomly chosen genes in the chromosome to other values chosen from possible alleles. In case of binary chromosome, this is simply a bit flip in the chromosome. It is worth mentioning that crossing-over does not necessarily need to be done for every pair of individuals in the intermediate population, it is done with some probability. The same applies to the mutation. The crossover probability and mutation probability are the basic parameters of the algorithm, which have to be wisely chosen, what requires some experience with GAs. Mutation probability is usually set to small value; the order of 1% or even less seems reasonable. Too much mutation may prevent algorithm from converging to the satisfactory solution, because each mutation introduces element of randomness, what usually worsens the solution, when algorithm approaches the right one. On the other hand, too small probability of mutation may let algorithm stuck in so-called evolution trap, which is simply local extremum of the optimised function.

Succession involves replacement of old population by a new one. There are different methods to do that. It can be simple replacement: all new replace all old or more sophisticated like: few best old stay and the rest is chosen from the best new - so called "elitism". Here again, like in case of crossover operator, the imagination of the algorithm designer is the only limit.

The explanation of GAs functioning bases on the representation of the solutions by chromosomes and on schemata [17]. Schemata are build using "don't care" symbol (usually *) to the gene alphabet. Schema represents all chromosomes, which conform to it in all positions everywhere except * positions, in other words it represents hyperplane in the search space. For example schema 10*00 matches two strings 10000 and 10100, and schema 1*0*0 matches four strings: 10000, 10010, 11000, 11010. There are two terms connected with schemata, which need to be defined at this point, namely ***schema order*** and ***defining length***. Schema order *o* is the number of symbols other than * in the schema. For example schema 100*0* is of order 4, and schema 1***** of order 1. Schemata of higher orders are more specific; there are less strings that match them. Defining length Δ is the distance between first and last specified symbol (not * symbol).
For example Δ(1***0) = 5-1 = 4, Δ(****0) = 0, Δ(**11*) = 4-3 = 1. Defining length is useful in calculations of probability of survival of the schema after crossing-over. It is easier to understand hyperplanes by drawing them in 3D space. As an example consider problem encoded with 3-bit chromosomes. The possible solutions form hypercube as shown in the figure 3.3 [18].

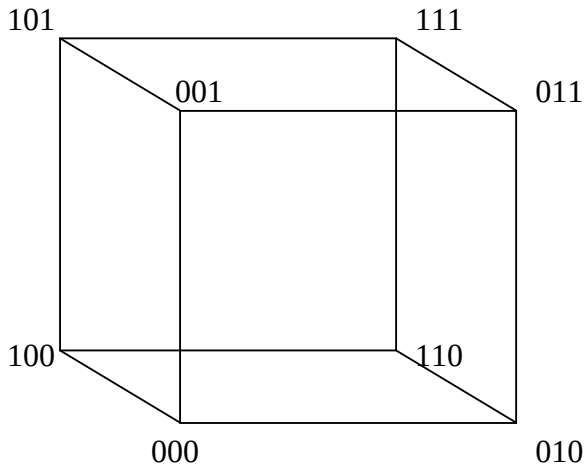


Figure 3.3. 3-bit Hypercube

The corners are labelled by 3-bit chromosomes or solutions, where all adjacent corners differ by 1 bit. Every plane of the hypercube can be represented by a schema. For example front plane 0**, back plane 1**, top plane **1, and so on. Assuming that binary chromosome encoding of the length m is used, every corner of the hypercube is a member of $2^m - 1$ hyperplanes, ***…*** represents entire search space and is not counted as a hyperplane. The $3^m - 1$ hyperplanes can be defined in the search space, because the 0, 1 or * can be placed in m positions. Fact that one solution is a member of $2^m - 1$ hyperplanes is the key part of GAs, because by examination of the single solution, many hyperplanes are sampled at the same time. The idea of searching for the perfect solution by means of population of possible points gives huge potential to the GAs, because in the population of n members $n \cdot (2^m - 1)$ schemata can be represented. Thus, usually n members of population provide information on more hyperplanes than n. Furthermore, it is clear that hyperplanes represented by low order schemata are sampled by more solution points than high order. All above mention facts contribute to so called ***implicit parallelism***, which is the true power of GAs [18]. Implicit parallelism means that upon evaluation of the population of chromosomes many hyperplanes are sampled at the same time. Every hyperplane contributes to the fitness function of the solution points, which lie on it. Chromosomes with higher fitness function value have higher probability of being selected to the next population and to enlarge its presence in the population, while those with lower value of fitness function usually are removed from the population. Therefore, more

promising hyperplanes are sampled more and more precisely while less promising are forgotten.

An example can help to understand the hyperplane sampling. Consider 4-bit chromosomes. The fitness function value is simply the number of "1" in the chromosome. Fitness function value for the schema is defined as average of fitness values of all chromosomes matching given schema. Equation (3.2) defines fitness function value for the schema S, where t is current population time, v_{Sj} is the j-th string matching schema S, match function value is the number of all chromosomes matching schema S at time t.

$$fit(S,t) = \frac{\sum_{j=1}^{m} fit(v_{Sj})}{match(S,t)} \tag{3.2}$$

Table 3.1. presents fitness values for 1*** and 0*** hyperplanes.

*Table 3.1. Calculation of fitness function values for 1*** and 0*** hyperplanes*

Schema	**Fitness**	**Schema**	**Fitness**
1000	1	0000	0
1001	2	0001	1
1010	2	0010	1
1011	3	0011	2
1100	2	0100	1
1101	3	0101	2
1110	3	0110	2
1111	4	0111	3
1***	**20 / 8 = 2.5**	**0*****	**12 / 8 =1.5**

The hyperplane 1*** seems more promising, because of higher fitness and therefore will increase its presence in the subsequent generations. Of course hyperplanes *1**, **1* and ***1 have the same fitness and will also increase its presence. But this is just explanation of the idea and for the sake of simplicity, the consideration is limited to the hyperplane 1***. In the next step schemata of higher order, but representing hyperplane 1***, have to be considered. The fitness for hyperplane 11** = 12 / 4 =3, and for hyperplane 10** = 8 / 4 =2. Again going to the schemata of higher order, but representing hyperplane

11** we get: fitness for 111* = 7 / 2 = 3.5, and for 110* = 5 / 2 = 2.5. Finally 1111 = 4 and 1110 = 3. The final solution is 1111 with maximal fitness function value. In this example, the fitness function values for hyperplanes were determined precisely by calculating average of fitness function values for all solution points belonging the plane, what is obviously not the case in the real GA run, this would be pointless. In the real case only solution points present in the population sample the hyperspace and by selection of better chromosomes better hyperplanes are promoted. For example chromosome 1010 contains in its fitness function value contributions of the following hyperplanes: 1***, *0**, **1*, ***0, 10**, *01*, **10, 1*1*, 1**0, *0*0, 101*, *010, 1*10, 10*0.

After selection we expect that *match(S,t+1)* chromosomes matches schema S. The probability of selecting the average chromosome matching schema S at time t is

$$p_S = \frac{fit(S,t)}{\frac{\sum_{j=1}^{pop\_size} fit(v_j)}{pop\_size}} \qquad .(3.3)$$

Equation (3.3) is similar to the equation (3.1) for the probability of selecting a chromosome, but chromosome fitness is replaced with schema fitness. The number of chromosomes matching schema S is *match(S,t)*. Number of opportunities (selections) is *pop_size.* Collecting above-mentioned facts together, we can define the number of chromosomes matching schema S at time t+1 as

$$match(S,t+1) = match(S,t) \cdot pop\_size \cdot p_S = match(S,t) \cdot \frac{fit(S,t)}{\frac{\sum_{j=1}^{pop\_size} fit(v_j)}{pop\_size}}. \qquad (3.4)$$

(3.4) proves that number of chromosomes matching given schema changes proportionally to the ratio of the schemata fitness function value and the average population fitness function value. This means, that individuals evaluated above the average increase their presence, the average individuals do not change their number and individuals below the average will decrease their presence. This comes from the fact that ε>1 in (3.5) for schemata evaluated above average fitness function value, ε=1 for average, ε<1 for worse than average.

$$fit(S,t) = \varepsilon \frac{\sum_{j=1}^{pop\_size} fit(v_j)}{pop\_size} \qquad (3.5)$$

By substitution in (3.4) by fit(S,t) from (3.5) :

$$match(S,t+1) = match(S,t) \cdot \varepsilon$$
$$match(S,t) = match(S,0) \cdot \varepsilon^{t} \tag{3.6}$$

Now it is clear that schemata above the average gain more and more place in population, what is more, it is exponential gain [17].

Selection mechanism does not produce any new schemata or solution points; it just copies chromosomes to form intermediate population. This is done in the next step of evolution using crossover and mutation genetic operators. These operators obviously interfere with schemata copying, and the equation (3.4) has to be slightly modified to reflect the real behaviour. For simplicity, only one point crossover operator will be considered here. For example there are two schemata of length 10:

$$S_0 = ***101****$$
$$S_1 = 11*******0$$

Assume, the crossover position is 7. Schema S_0 survives the operation and S_1 is destroyed. As mentioned earlier in this chapter, defining length is the parameter, which helps in probability of survival after crossing-over. $\Delta(S_0)$ = 6-4=2 and $\Delta(S_1)$=10-1=9. The crossover position can be selected from *m-1* possibilities, thus the probability of schema destruction is:

$$p_d(S) = \frac{\Delta(S)}{(m-1)} \tag{3.7}$$

Thus, the probability of survival

$$p_s(S) = 1 - p_d(S) = 1 - \frac{\Delta(S)}{(m-1)} \tag{3.8}$$

However, not all chromosomes undergo crossover, but the crossover probability is p_c, therefore

$$p_s(S) = 1 - p_c \frac{\Delta(S)}{(m-1)} \tag{3.9}$$

In fact equation (3.9) should be inequality, because there is very small chance, that even though the crossover position is inside the schema like S_1, it will survive. This can be the case when the child inherits genes, which match the schema. Thus, after collecting selection and crossover mechanisms together equation (3.4) changes into:

$$match(S,t+1) \geq match(S,t) \cdot \frac{fit(S,t)}{\frac{\sum_{j=1}^{pop\_size} fit(v_j)}{pop\_size}} \cdot \left(1 - p_c \frac{\Delta(S)}{(m-1)}\right) \tag{3.10}$$

Mutation also can alter the schemata. The probability of changing single bit in the chromosome is p_m, thus probability of bit survival is $1 - p_m$. The number of positions in the chromosome relevant to the schema is the order of schema, so the probability of schema survival is:

$$p_s = (1 - p_m)^{o(S)} \tag{3.11}$$

since p_m is usually much, much smaller than 1 (3.11) can be approximated as:

$$p_s(S) \approx 1 - o(S) \cdot p_m \tag{3.12}$$

Thus, after collecting selection, crossover and mutation influence equation (3.4) becomes:

$$match(S,t+1) \geq match(S,t) \cdot \frac{fit(S,t)}{\frac{\sum_{j=1}^{pop\_size} fit(v_j)}{pop\_size}} \cdot \left(1 - p_c \frac{\Delta(S)}{(m-1)} - o(S) \cdot p_m\right) \tag{3.13}$$

Again, the conclusion is that the number of chromosomes representing schema evaluated above the population average is rising exponentially. But this has to be schema of small defining length and low order, because only then the destructive effect of mutation and crossover is not relevant. However, crossover and mutation operators are essential, because selection as such, does not introduce any new schemata into the population.

This is the basis for the **Schemata Theorem**.

Short, low order schemata evaluated above the average get exponentially raising representation in the population. [17]

3.1.2. Pros and Cons of GAs

This section contains short summary of GAs properties divided into advantages and disadvantages.

Advantages:

- **Easy to understand [19]**

The basic concept of GAs is easy to understand, because it is based on natural selection and inheritance laws, which are easily explainable due to correspondence to real life situations

- **Chromosome abstraction**

The algorithm designer does not have to deal with parameters of the optimised function, does not have to change them directly or analyse relations between them. The basic idea is always the same: evaluation, selection, genetic operations and so on. Program works on chromosomes, which code the solution and thus provide solution abstraction.

- **Multiparameter optimisation [19]**

Programs based on GAs are capable of optimisation of many parameters at once. The possible combination of parameter values is represented as single fitness function value, which is optimised.

- **Discrete functions optimisation**

In case of discrete functions, analytical methods cannot be used, only stochastic ones are able to find the solution. GAs handle discrete and continuous functions without problems because of above-mentioned chromosome coding abstraction.

- **Always provides the solution [19]**

This statement at first seems questionable, but in fact from the first population the solution is available. It is not the best possible, but becomes better and better with time. This is not the truth when chromosome coding is chosen in a way, which allows for coding of individuals that are out of possible solutions set. This issue is discussed further in the section 3.2.3.

- **Solution is sought in whole search space**

The search for the best solution is started from many randomly chosen points in the search space and is then biased towards promising space points.

- **Relatively easy in implementation**

The implementation requires only the following elements:

  - Population container – the container for individuals, together with their genetic material
  - Genetic operations – selection, cross-over and mutation
  - Fitness function implementation

Moreover, implementation does not consume memory for history keeping, like in ordinary stochastic algorithms. The "wisdom" of the population is incorporated into the genetic material of the individuals.

- **Flexibility**

Once implemented, the program can be easily changed for totally different problem. As long as the physical structure of the chromosome stays the same the only element that must change is fitness function, because this is the only part that deals with logical chromosome structure.

- **Easy to distribute [19]**

Selection is the only step, which needs knowledge of the whole population, all other operations require only information that they directly act on (usually chromosomes). Therefore, the program is easily distributable, many computers may work on the solution of single problem.

- **May be improved as knowledge on the problem is gained**

As knowledge on the problem domain is gained, chromosome coding and fitness function can be easily adjusted to reflect the problem more precisely and speed up the solution search.

Disadvantages:

- **Choice of representation and fitness function is critical [19]**

The hardest task for the algorithm designer is to choose appropriate chromosome-solution correspondence and fitness function. Chromosome should be able to code all possible solutions of the problem, but prevent coding of solution that lies outside the acceptable set. Fitness function has to take into account all of the goals algorithm has to achieve, and find good balance between them, what leads to simultaneous optimisation of all parameters. In fact determination of chromosome coding and fitness function are critical parts of the whole application. There are some attempts to theoretically describe the right way

of doing that, but still right choice of those parameters of the algorithm depends mainly on the experience and feeling of the designer. More on that can be found in section 3.2.3.

- **Genetic operations and parameters have to be wisely chosen**

As in case of chromosome abstraction and fitness function, the type of population maintenance algorithm, selection scheme, cross-over type, mutation type, cross-over probability and mutation probability have to be adjusted with care and feeling. Some trials need to be done to get to know the problem, see how different algorithms behave and choose the right one. The imagination of the designer is the limit, therefore the algorithm can be tailored to the problem, but here the theory is more helpful than in former case. There are many well-investigated and described algorithms, which can suit the problem.

- **Finish criterion problem [19]**

There has to be some finish criterion set. This can be maximum number of generations or minimum required fitness function value. But it is hard to estimate how many generations are needed to arrive at satisfactory solution, therefore another disadvantage is that application run time is hard to estimate. On the other hand, the algorithm may be unable to arrive at minimum required fitness function value in satisfactory time. However, as mentioned in advantages section, in case of genetic algorithms, there is always a solution and application can be stopped at any time.

- **Random number generator dependence**

The success or failure of the genetic algorithm may be dependent on the type of random number generator used. This issue is further explained in the section 3.1.3.

3.1.3. Genetic Algorithms Issues

There are couple of issues that are crucial for functioning of GAs. One of them is mentioned in the previous section, namely chromosome coding. Some problems have limitations, which decrease the set of valid solutions. The chromosome coding should be designed in a way, which does not allow for invalid solutions coding (search space equals set of valid solutions). In such a case algorithm does not waste time and resources for maintenance, evaluation or repair of individuals, which represent solutions wrong from the point of view of the problem. But sometimes it is hard to design such chromosome coding, which guarantee only valid solutions. Moreover, complex decoders and coders can have large computational power requirements. Another approach is to change problem into one without limits. The search is carried out in the whole solution space. Next invalid

solutions are punished for passing limits by decrease of fitness function value. Usually the penalties are incorporated into fitness function. The penalties can be constant or depend on the degree of limit violation. There are extreme versions of punishment, which remove invalid solutions from the population. Sometimes, this method can be successful, but has its drawbacks. In some problems, the probability of generation of valid solution at random is relatively small and algorithm does not move forward, because too many individuals die just after birth. Furthermore, chromosomes representing invalid solutions may posses genes, which after couple of generations and genetic operations may result in good, valid solution. After death they no longer contribute to population genetic variety. Another method is based on chromosome repair. After individual is identified as invalid solution, special operations are carried out on the chromosome, to put it back into the set of valid solutions. However, this approach in some cases requires complex processing, which consumes much of processing power.

As an example consider packing problem [17]. The set of articles to be packed is given. Every article has its weight W(i), its value P(i). The choice has to be made of one or more disjoint sets of articles, where sum of weights does not violate the limit of rucksack capacity C and the sum of article values is maximal. The problem is in choosing a binary vector x=<x[1], …., x[n]> such that:

$$\sum_{i=1}^{n} x[i] \cdot W[i] \leq C \tag{3.14}$$

and

$$P(x) = \sum_{i=1}^{n} x[i] \cdot P[i] \tag{3.15}$$

is maximal.

When vector is coded directly as binary string in the chromosome, it is obvious that invalid solutions can occur in the population. In such a case, methods described above can be useful. At first, consider special coding/decoding approach. The chromosome can be string of n integer numbers, where number at i-th position is from the range 1 to n-i+1. The number is a vector, which describes position of selected article on the list of available articles L. For example for L = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10), chromosome <3, 2, 2, 3, 2, 1, 2, 3, 1, 1> results in the following list of items selected: 3, 2, 4, 6, 5, 1, 8, 10, 7, 9. Such coding has a big advantage, the crossover of two parents gives valid children. Mutation

at i-th position of the chromosome changes its value to the one in the range [1, n-i+1]. Figure 3.4. presents possible decoding algorithm, which guarantees valid solution.

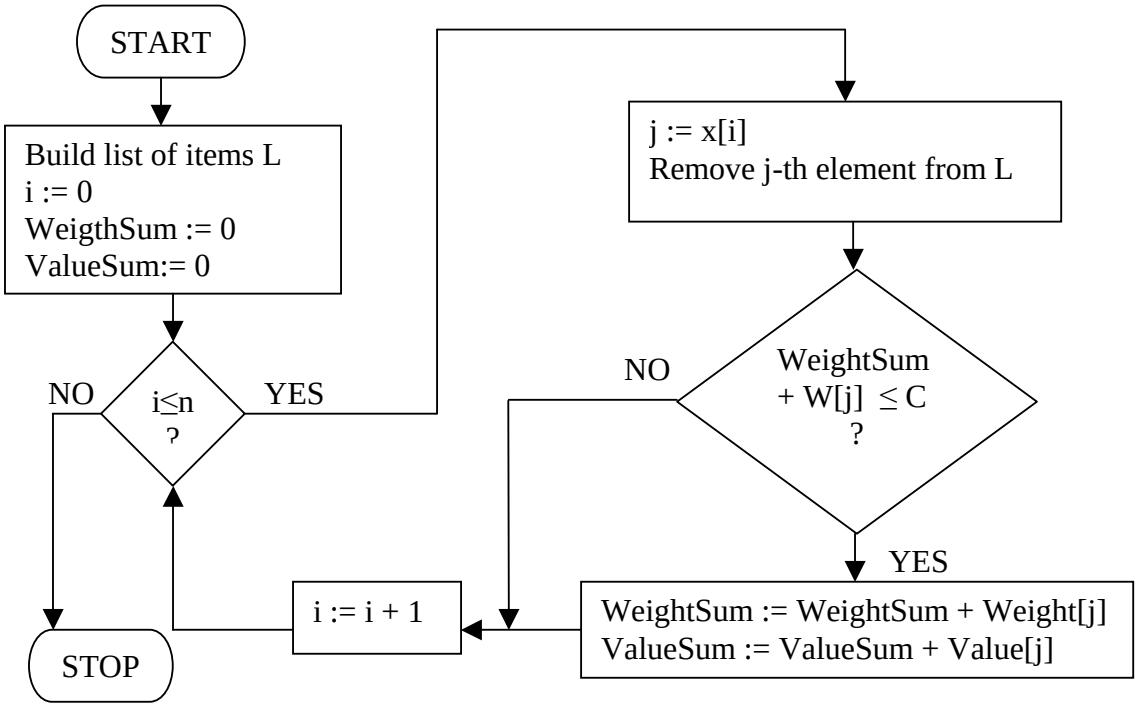


Figure 3.4. Decoding algorithm for rucksack problem [17]

While chromosome is decoded, only those items are taken, that do not violate the capacity of the rucksack limitation, therefore only valid solutions are generated. There can be two flavours of the decoding algorithm. One can build list L at random, second can build list as ordered list, for example in descending order of value to weight ratio.

Next, consider punishment for violation of limitation method. Fitness function of the individual is decreased by the value of the penalty function Pen(x) such that for every valid solution, that is fulfiling inequality (3.14), function equals 0 and for any other is greater than 0. This function can be defined in variety of ways. Equations (3.16 – 3.18) present examples of such definitions [17].

$$\rho = \max_{i=1..n}\{P[i]/W[i]\}$$

$$Pen(x) = \log_2(1+\rho\cdot(\sum_{i=1}^{n} x[i]\cdot W[i]-C)) \tag{3.16}$$

$$Pen(x) = \rho\cdot(\sum_{i=1}^{n} x[i]\cdot W[i]-C) \tag{3.17}$$

$$Pen(x) = (\rho \cdot (\sum_{i=1}^{n} x[i] \cdot W[i] - C))^2 \quad (3.18)$$

The Pen(x) function value is dependent on the degree of the limitation violation. In (3.16) this is logarithmic dependence, in (3.17) linear, in (3.18) square dependence.

Last method of dealing with invalid solutions is to repair chromosome in a way, which brings it back to the valid solutions set. Assuming standard chromosome coding, i-th item is put to the rucksack, when **x**[i] = 1. This can produce solutions, which violate limitation (3.14). The repair procedure creates chromosome **x'** which is repaired version of chromosome **x**. What is interesting, repaired chromosomes can replace only some part of original chromosomes, what leaves some invalid solutions, but ensures variety of genetic material in the population. Figure 3.5. presents proposed repair algorithm.

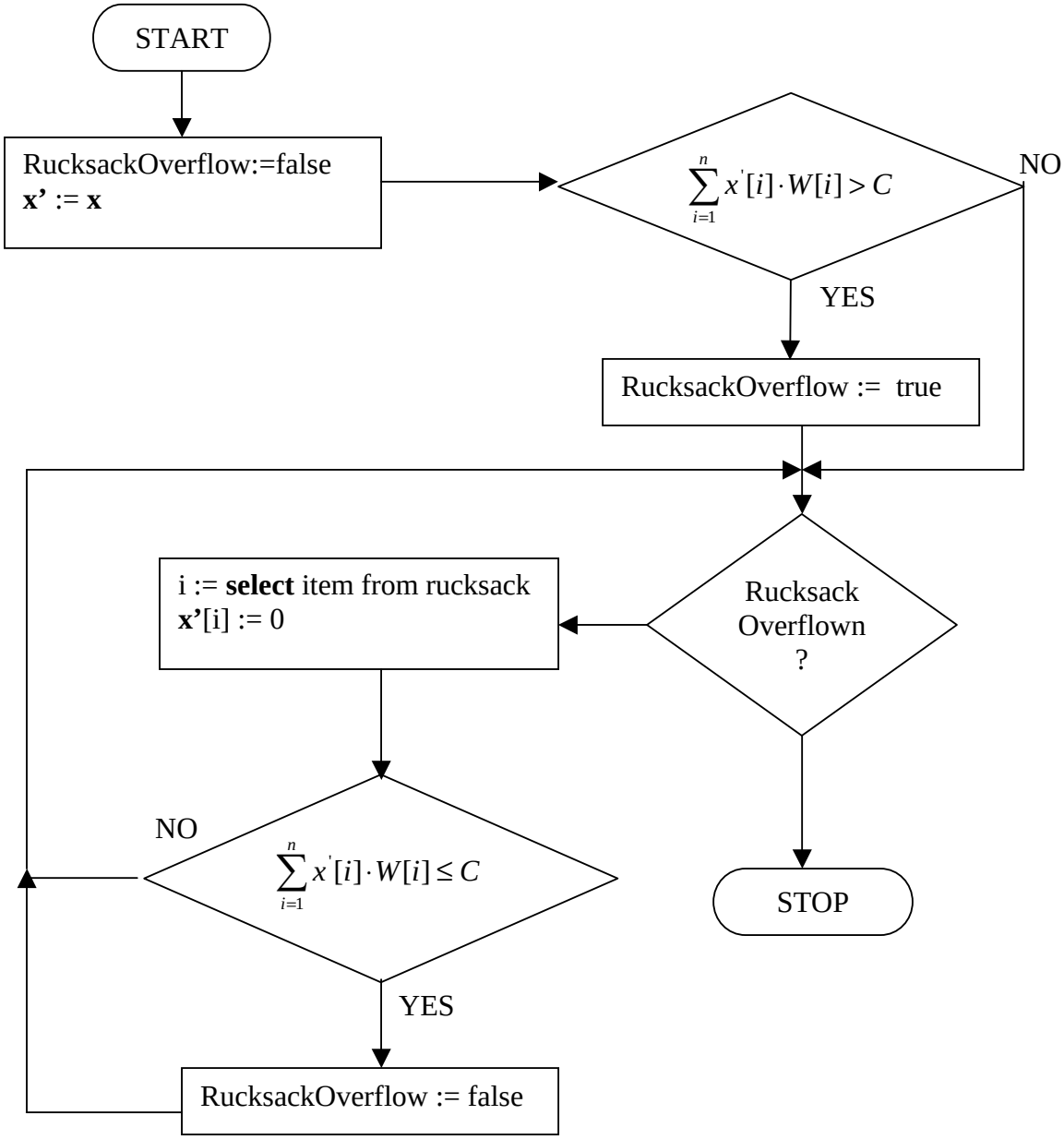


Figure 3.5. Chromosome repair algorithm [17]

Selection of the item to remove by **select** function can be done in variety of ways. For example, item can be picked at random or the item with smallest value to weight ratio.

Another issue, important at the stage of algorithm design is **Random Number Generator** (RNG). Genetic algorithms belong to the class of stochastic algorithms, therefore they use randomly generated numbers intensively. Random numbers are needed everywhere, during selection, mutation, crossover, etc. The conclusion is that random number generator does have influence on application functioning. The more ideal generator is, the smaller is its negative impact on the genetic process, because low quality RNG can interfere with statistical process. Such interference may be hard to isolate, because usually programmers look for the bugs in code, and not in statistics.

RNGs can be divided into two groups: software and hardware generators. Hardware generators require some hardware connected to the computer, which generates numbers or voltages and then transmits them to the computer (via serial link for example). Some of them employ physical properties of the matter, like thermal noise or radioactive decay. Shot noise from the resistor or signal from the radiation detector can be amplified and converted into bits, bits into bytes and then transferred to the computer. Some use access times to the hardware like hard disks, keyboard or mouse as basis for number generation. Numbers generated by hardware generators are purely random, because abovementioned events cannot be predicted. Therefore they are widely used for cryptography - key generation, lottery or simulations of physical phenomena. However, hardware RNGs are not portable as they require additional hardware, drivers, etc. Some commercially available motherboards have hardware RNG incorporated. For example, Intel i8xx based motherboards are equipped with 82802 Firmware Hub, which contains hardware RNG. This RNG uses the thermal sensor to convert thermal noise generated by the system to produce random numbers. For further information, please refer to [20]. In most of the stochastic applications software RNGs can be used. They generate pseudo-random numbers, because it is not possible to generate truly random numbers with arithmetic algorithms, as they are repeatable and predictable. Almost all RNGs are based on the sequences of numbers, therefore they work in cycles or in other words are periodic. They use sequences to generate numbers. Consider, as example, **Linear Congruential Generator** (LCG), which is widely used by RNGs [21]. Every integer is generated using previous generated value. This is usually done like in (3.19).

$$x_{n+1} = a \cdot x_n + c \quad \text{(modulo m arithmetic)} \tag{3.19}$$

a and *c* are constant integers, such generator is denoted as LCG(a, c, m, x_0) The randomness effect is caused by the use of modulo m arithmetic. In modulo m arithmetic, integer must be smaller than m, it can be regarded as the remainder part of division by m. For example integers 0, 1, 2, 3, 4, 5, 6, 7 converted to modulo 4 arithmetic become 0, 1, 2, 3, 0, 1, 2, 3. Generator needs some starting point referred to as **seed**, which is x_0 value. As an example consider LCG (5, 1, 8, 1). The numbers generated are presented in the table 3.2.

Table 3.2. LCG(5, 1, 8, 1) generated numbers

Number	Value	Binary value
x_0	1	001
x_1	6	110
x_2	7	111
x_3	4	100
x_4	5	101
x_5	2	010
x_6	3	011
x_7	0	000
x_8	1	001

The following properties can be distinguished:

- maximum period of the generator is equal to the modulus m,
- the distribution is uniform (all possible integers are used),
- any seed results in the rotated, but identical sequence of numbers
- numbers are not random; serial correlation between them is obvious. They form alternating sequence of even and odd numbers, therefore the least significant bit forms sequence of alternating zeroes and ones

Next consider LCG(5, 0, 8, 1). The table 3.3. summarises the results.

Table 3.3. LCG(5, 0, 8, 1) generated numbers

Number	Value	Binary value
x_0	1	001
x_1	5	101
x_2	1	001

The following properties can be distinguished:

- period of the generator is 2
- the distribution is uniform for small granularity (up to 2 ranges), namely for ranges [0, 4) and [4,8), but distribution is no longer uniform for larger number of ranges
- seeds 1 and 5 result in the same sequence, while other seeds result in other sequences, but with the same period
- numbers are not random; serial correlation between them is obvious. They form alternating sequence of 1 and 5, therefore 2 least significant bits are always 01.

Above-mentioned LCGs have many disadvantages, but their unquestionable advantage is their speed and small resource consumption. The choice of the RNG suitable for application depends on the granularity of the problem, the number of random numbers needed, the speed of generation. The following RNG properties should be taken into account upon selection:

- **Period** Generally, the larger the period, the better. Ideally sequences generated by the RNG should not repeat, but in practice repetition after generation of very large set of random numbers is acceptable in some applications. This is very important especially in cryptographic applications, because RNG with large period is much more secure, than one with short period (many generated keys have to be collected to enable prediction of the next key). Generally cryptographers stay away from the linear RNGs because of their predictability.
- **Uniformity** Numbers generated by the RNG should be distributed uniformly in whole generation range. The lack of uniformity can severely affect the application, which expects uniformly distributed random numbers.
- **Correlation** Ideally, there should be no correlation between the consecutive numbers, they should be independent. However, in linear generators every number depends on the previous one, therefore correlation is unavoidable.

- **Speed** In applications, where huge sets of random numbers are needed, speed is critical aspect. Usually there is a trade-off between speed and randomness of the RNG, therefore stochastic simulations usually employ linear generators with large periods, and cryptography employs more complex and slow, but less predictable generators.

3.1.4. Genetic Algorithm for Circuit Design

This section presents the ideas for the application, which can be useful for circuit design.

Random Number Generator

The first thing that should be set up for the application is the proper random number generator. Genetic algorithms belong to the group of stochastic algorithms, therefore good linear RNG should be satisfactory. RNG for this purpose should:

- **Be fast** - speed is one of the greatest concerns in this case, because huge sets of random numbers will be needed. Slow RNG would affect the application performance severely.
- **Have large period** – this is not a cryptographic application, but as it was mentioned in the previous section, with large periods more randomness is incorporated into the generated numbers and the influence of the RNG on the process is smaller.
- **Have uniform distribution** – application expects RNG to generate numbers, which are uniformly distributed within the generator range, therefore any non-uniformity may result in reduced speed of convergence or non-optimal solution.

The RNG usually shipped with C or C++ compiler, namely the implementation of the rand() function form the C standard library may or may not be a good choice. The properties of this generator depend on the implementation, can be different for every compiler or platform. Therefore, this RNG cannot be counted on.

Mersenne Twister (MT) generator has properties, which should be satisfactory for the genetic algorithms. It is not cryptographically secure, because it is based on linear recursion. However, its other advantages make it ideal for stochastic simulations despite generator linearity [22]:

- **C-code** - generator is coded in C language, therefore is portable to any platform and is fast enough. It is even faster than some implementations of ANSI-C rand() function.
- **Speed** – generation of 100,000,000 random numbers took MT generator 2557 ms and standard rand() 3943 ms
- **Large period** – the proved period of this generator is 2^{19937}-1
- **Uniform distribution** – 623-dimension equidistribution is assured
- **Small memory consumption** – it uses only 624 words of memory

Every generator needs a starting point, a seed. The seed is usually chosen to be the time value at the moment of application start, what initialises the generator with different value at every execution.

Chromosome representation

The aim of the algorithm is to design such circuit, that its implementation in the programmable device will remain functioning despite single error on any of configuration bits. Circuit at the logical level can be designed without the knowledge on its physical implementation, but in order to design a circuit that is radiation tolerant, its internal, physical structure must be known. That is because radiation interacts directly with the silicon structures. In this case the target programmable circuit has to be chosen. The GAL16v8 described in chapter 2 seems to be a reasonable choice, because it is small, simple device, with relatively short configuration. The additional important feature is that the internal structure of GAL is well documented, what is not the case in vast majority of other programmable circuits. These properties make this device easy to simulate, what is also very important in purely software-implemented algorithm. Moreover, not all 2048 bits of configuration have to be used, it can be shortened to speed up the calculations, when not all 8 inputs are needed. The figure 3.6. shows how shortened configuration is created.

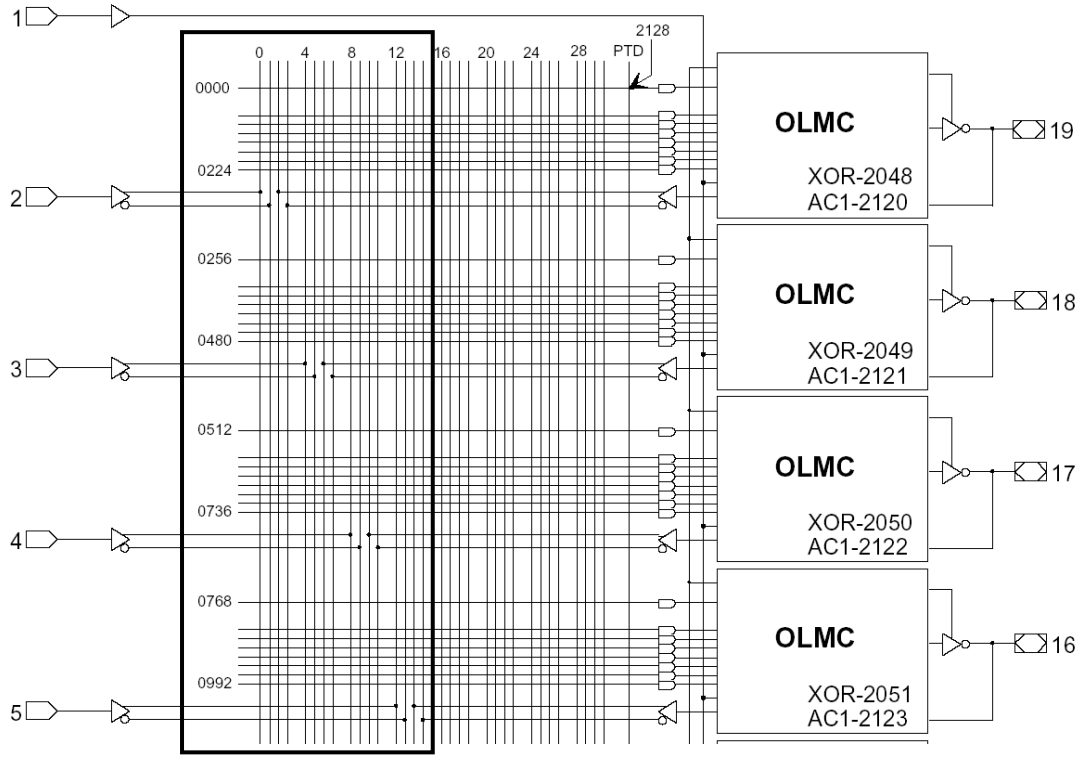


Figure 3.6. Only part of the GAL16v8 is used [10]

In figure 3.6. only inputs 2, 3, 4, 5 are used, therefore only 16*7*4 = 448 bits are needed. One could argue, that the rest of the configuration is also a subject to SEU and that application should deal with 2048-bit configuration. Moreover, there are configuration bits, which cannot be made immune by means of circuit design. The change of any of the configuration bits puts device into different operational mode and thus produces bad output. Finally, one could notice that nowadays GAL16v8 configuration is held in EEPROM memory, which is build using floating gates and is not a subject to SEUs. All above mentioned arguments are true, but GAL is chosen here because of its simplicity. It serves as a basis for simulations, which are needed to check if the method of using GAs for fault tolerant circuits design can be successful. The improved version of the application should work on programmable circuits, which hold their configuration in memory susceptible to SEUs. Another difficulty is that beside SEUs, SETs may occur in the circuit, but the designed system is not going to simulate SETs, therefore configuration is not prepared for dealing with this type of SEEs.

Chromosomes have to code configuration somehow. The straightforward idea is to put configuration bits directly into the chromosome. In fact this seems to be the only coding we can use, because the main part of the algorithm is simulation of SEEs by configuration bit changes, what imposes the usage of the direct configuration. Such coding has its disadvantages, namely the chromosome with configuration representing valid circuit (fulfilling the truth-table) can become invalid after genetic operations. Therefore some of the techniques described in section 3.1.3. should be used. Special coding/decoding scheme seems to be not a very good choice, because as mentioned earlier, the chromosome must correspond to the physical structure of the device. Coding at higher level of abstraction (the level of gates for example) no longer describes the physical interconnections in or between cells, thus cannot be evaluated for fault tolerance. Repair algorithm is also hard to think of, because having a configuration from the chromosome and its fitness function value, one is unable to say how to change the configuration to obtain better results. The only way of doing that is to change the configuration bits, evaluate it and check the result. But this is what the genetic algorithm is supposed to do, not the fitness function or repair algorithm. Therefore, punishment method seems to be the right choice in described case.

Fitness function

There are couple of way of fitness function value determination that can be used. There are two groups of functions needed, namely basic functionality evaluation and radiation tolerance. For simplicity, in the further part of the text, former function will be called “short evaluation” and the latter “radiation evaluation”.

Short evaluation should give a measure of how good is the GAL with configuration contained in the chromosome at performing basic functions needed. Combinatorial circuits are the only ones we are dealing with, therefore for the desired circuit and every evaluated circuit, the truth-table can be formulated. Such table contains all possible combinations of inputs with corresponding values of outputs. Usually input values are put in rows and output values are put in columns. The short evaluation can be simply the total number of output values of the circuit evaluated, which agree with the output values of the needed circuit in the same truth-table row. Tables 3.4. and 3.5. show exemplary truth-tables.

Table 3.4. Full-adder truth-table

Inputs			Outputs	
A	**B**	**C_{in}**	**Y**	**C_{out}**
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Table 3.5. Randomly chosen circuit Truth-table

Inputs			Outputs	
A	**B**	**C_{in}**	**Y**	**C_{out}**
0	0	0	0	0
0	0	1	0	0
0	1	0	1	0
0	1	1	0	1
1	0	0	0	0
1	0	1	0	0
1	1	0	0	0
1	1	1	0	U

Full-adder is described in chapter 5. Table 3.5. contains the truth-table of the real chromosome generated in the early stage of the genetic algorithm. It does not function like the full-adder, however it does have proper output values for some inputs. The short evaluation will give the result 10 in this case, because there are 10 places, where outputs agree. The **U** symbol in the table 3.5. denotes unknown state of the output. Unknown state of the output may happen for certain input combinations, when configuration contains feedback connection. Figure 3.7. shows an exemplar circuit, which produces U value at the output. Table 3.6. contains its truth-table.

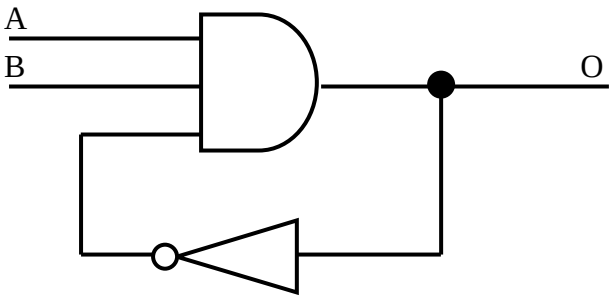


Figure 3.7. Circuit with feedback

A	B	O
0	0	0
0	1	0
1	0	0
1	1	***U***

Table 3.6. Truth-table for the circuit from fig. 3.7.

Radiation tolerance evaluation should give a measure of how good is the GAL with configuration contained in the chromosome at performing functions needed, when there is a single fault in configuration.

Version 1 ("normal")

In the simplest case, this can be the average of short evaluation results for every bit flipped in the configuration. Equation (3.20) shows the definition of this fitness function.

$$radFit_1(C) = \frac{\sum_{i=1}^{n} shortFit(\, flip(C,i))}{n} \tag{3.20}$$

where:

n – the length of the chromosome C

$radFit_1(C)$ - radiation tolerance evaluation result for chromosome C

$shortFit(C)$ – short evaluation result for chromosome C

$flip(C,i)$ – function which returns chromosome C with i-th bit flipped

This version of the evaluation function has serious drawback. It does not take into account the fact whether the chromosome is valid without any bit flipped. This predestines this function to the algorithm with removes invalid chromosomes in the evolution loop.

Version 2 ("added")

In order to alleviate the drawback described above, the following radiation tolerance evaluation function can be used:

$$radFit_2(C) = \frac{\left(\sum_{i=1}^{n} shortFit(\, flip(C,i))\right) + shortFit(C)}{n+1} \tag{3.21}$$

Here the validity of the chromosome without any faults is checked, moreover the individual with the fitness corresponding to the maximum possible value (as defined by (3.24)) guarantees the desired functionality without any faults and with single fault at any position in the configuration. Short evaluation of non-altered configuration can be regarded as a penalty, because it lowers the fitness function slightly. However, this approach has also a serious drawback. Namely, only the ideal solution is sure to be valid for configuration without a fault. The solution that gives evaluation result smaller than the maximum possible does not perform as desired for some configurations, one of them may be unaltered one. This results in a big limitation - non-ideal solution may be useless. Since chromosome, which is sensitive to faults on 20 bits is much better than one sensitive on 448 bits, it would be advantageous to have non-ideal solution, which guarantees validity with non-altered configuration.

Version 3

In order to alleviate the drawback described above, the following radiation tolerance evaluation function can be used:

$$radFit_3(C) = Pen(C) \cdot \frac{\sum_{i=1}^{n} shortFit(\,flip(C,i))}{n} \tag{3.22}$$

where:

Pen(C) – a penalty function for the chromosome C. This function takes the values from the range [0, 1]. 0 for maximum penalty, 1 for no penalty.

The penalty function can have different definitions. Figure 3.8. shows three possible penalty functions.

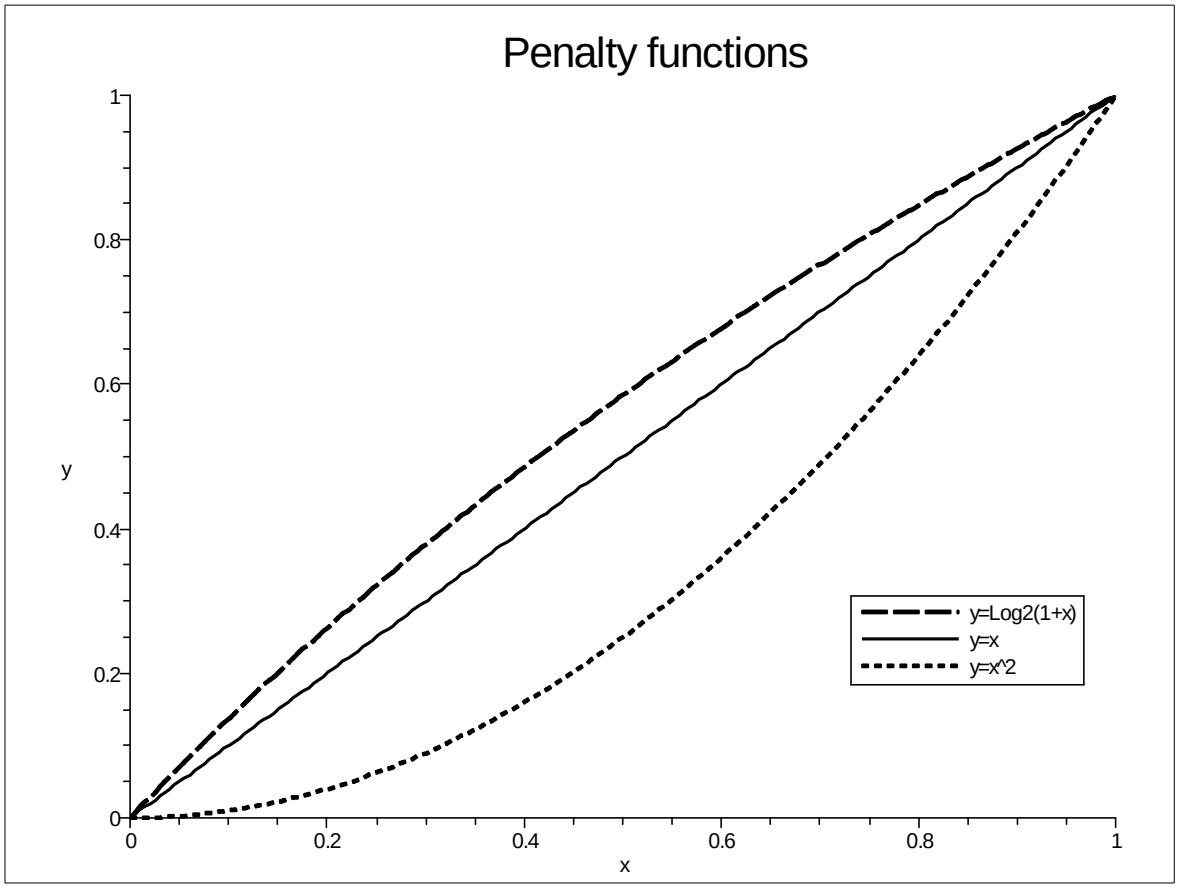


Figure 3.8. Possible penalty functions

Version 3a ("log2")

Logarithmic function can be used for penalty function definition:

$$Pen_{3a}(C) = \left(1 + \log_2\left(\frac{shortFit(C)}{maxFit}\right)\right) \tag{3.23}$$

where:

maxFit – maximum possible value of the shortFit(C) defined as:

$$maxFit = outputs \cdot 2^{inputs} \tag{3.24}$$

outputs, *inputs* – number of outputs and inputs defined in reference circuit (in case of full-adder maxFit = 16).

This type of penalty function slightly departs from the punishment proportional to the number of wrong output values. The larger penalty is put on the individuals, which produce many or little number of wrong outputs. The penalty is slightly smaller for the individuals that have ca. half of the outputs wrong, thus this function can be more

effective at the intermediate stage of simulation. It guarantees that the individual is valid with non-faulty configuration, when fitness function is above the limit defined in (3.25)

$$\left(1+\log_2\left(\frac{maxFit-1}{maxFit}\right)\right)\cdot maxFit \tag{3.25}$$

In case of full-adder, when maxFit = 16, this limit is ≈14.51.

Version 3b ("proportional")

Linear function can be used for penalty function definition:

$$Pen_{3b}(C)=\frac{shortFit(C)}{maxFit} \tag{3.26}$$

This type of penalty function punishes every individual proportionally to the number of wrong output values. It guarantees that the individual is valid with non-faulty configuration, when fitness function is above the limit defined in (3.27)

$$\frac{maxFit-1}{maxFit}\cdot maxFit = maxFit-1 \tag{3.27}$$

In case of full-adder, when maxFit = 16, this limit is 15.0 .

Version 3c ("square")

Square function can be used for penalty function definition:

$$Pen_{3c}(C)=\left(\frac{shortFit(C)}{maxFit}\right)^2 \tag{3.28}$$

This type of penalty function puts more punishment on individuals, which produce more wrong output values. The more proper outputs produced, the smaller penalty. Moreover, penalty function, by its non-linearity strongly promotes better individuals. Thus, the performance of the algorithm employing this function may be low in the initial stage, but then it should be higher. It guarantees that the individual is valid with non-faulty configuration, when fitness function is above the limit defined in (3.29)

$$\left(\frac{maxFit-1}{maxFit}\right)^2\cdot maxFit \tag{3.29}$$

In case of full-adder, when maxFit = 16, this limit is 14.0625 .

Fitness scalers

Additionally to above mentioned implementations of fitness function, fitness scalers can be used. Scalers enable control of selection pressure. Selection pressure and genetic diversity are two factors, which severely affect evolution process. These two factors are strongly correlated. Larger selection pressure decreases diversity of genetic material in the population. In the other words, too large selection pressure supports pre-mature convergence of the genetic algorithm and too small selection pressure may result in inefficient search. Therefore, it is very important to keep balance between those two factors. Fitness scalers can be used when individuals are selected to the intermediate population. For example in roulette wheel selection, fitness of each individual could be an argument of some scaling function. Square or cubic function are suitable for this purpose, because they give more chance of placing a copy in the intermediate population to the individuals with better fitness function value.

Chromosome cross-over operators

As described in section 3.1.1. one point cross-over operator is not the only way of selecting genes from parent chromosomes. The other ideas listed in this section just mentioned are presented below in details.

Two point cross-over

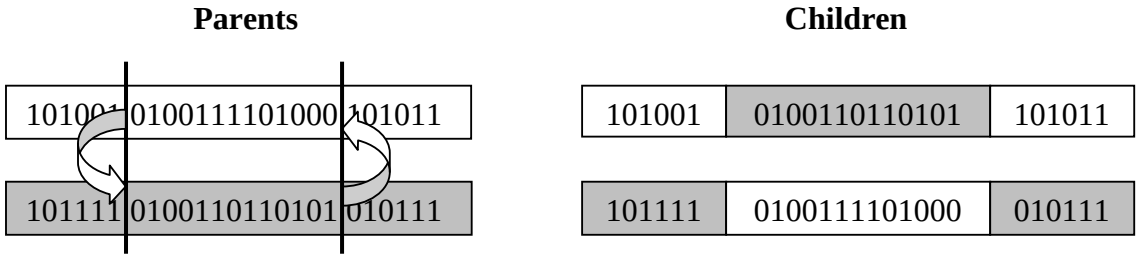


Figure 3.9. Two point cross-over operation

Two cross-over positions are picked at random and part of the chromosome contained between them is exchanged.

Operator exchanging segment of constant length

This is operator identical to the one described above, but it picks only one cross-over position. The position is chosen from 0 to L-S, where L is the length of the chromosome and S is the length of the exchanged segment. The second position is the first position moved by S. This operator may be useful in situations where adjacent genes are somehow correlated and should be inherited together.

Operator exchanging segment of constant length with displacement

This is operator identical to the one described above, but cross-over positions are chosen independently for every parent. This results in displaced segment of genes. Operator affects the schemata severely, thus introducing genetic diversity. However, the diversity is different than one caused by mutation, because large groups of adjacent genes are changed.

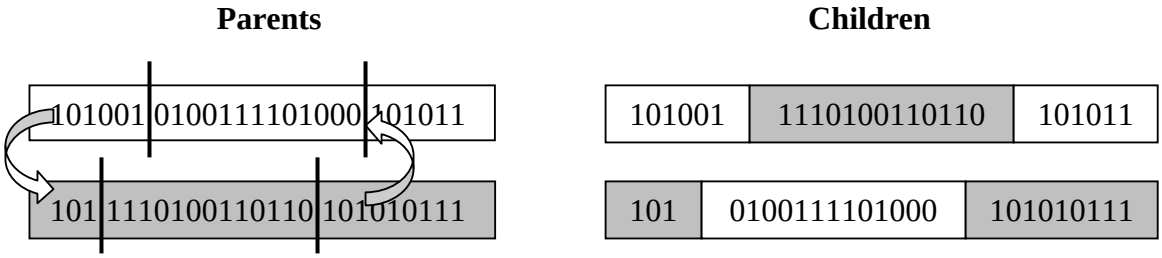


Figure 3.10. Operator exchanging segment of constant length with displacement

Uniform cross-over operator

In this operator one child receives a gene from randomly selected parent and the other child from the other parent. Every gene is inherited independently. This exchanges the single genes instead of groups of genes, thus the different properties of the parents may be inherited disregarding their relative position.

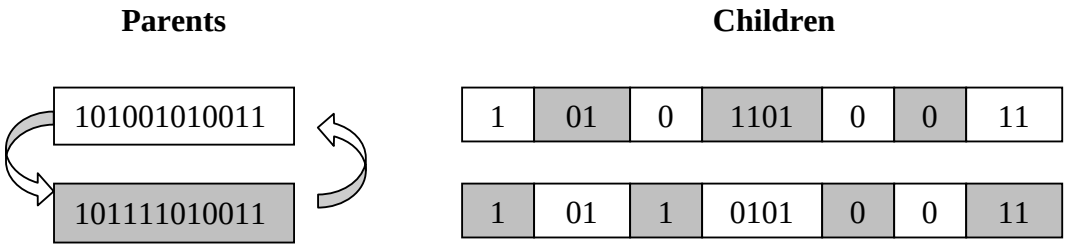


Figure 3.11. Uniform cross-over operator

Evolution program

Up to this point, the elements of evolution programs were described. The elements, which can be used almost in any program, in any genetic algorithm. However, they cannot be used without the evolution program itself. This section presents possible implementation of the evolution program.

Main loop of the program is usually constructed as in figure 3.12.

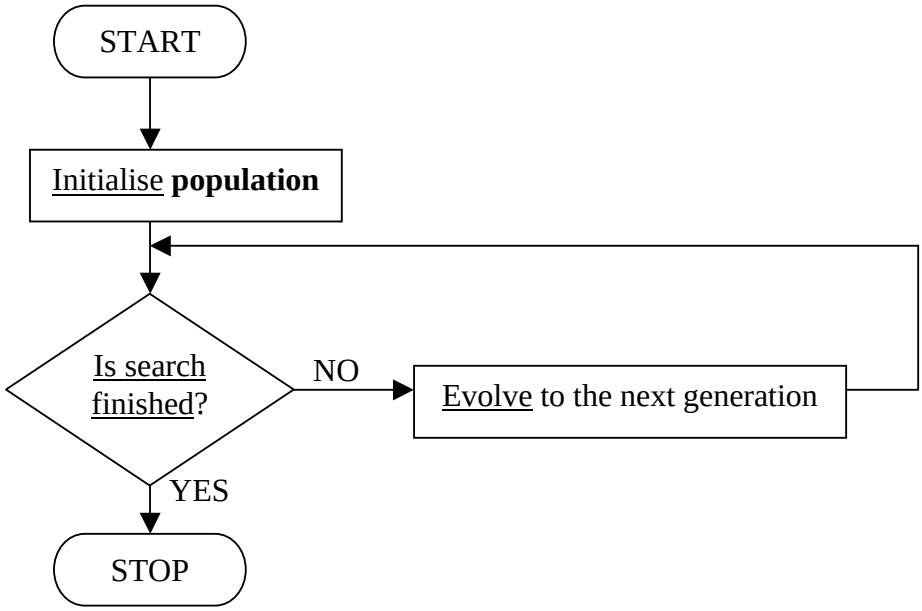


Figure 3.12. Main loop of the evolution program

Initialise function

This function simply fills chromosomes of the population individuals with randomly generated genes. Usually genes are selected from the alleles with equal probability.

Is search finished? function

This function returns boolean value that informs the program whether the evolution has led to the solution required or not. Usually maximum generation number or minimum required fitness or both just mentioned requirements are used as finish conditions.

Evolve function

This function is the main part of the program. It takes the population to the next generation. This can be done in variety of ways, there are no rules limiting the designer. Usually couple of them need to be tested before the algorithm suitable for the problem is found. Three ideas for this function are presented below.

Version 1 ("classic with overlapping")

This is the realisation of classic Goldberg algorithm, but supports population overlapping. It requires the following parameters:

- **Population size (PopSize)** – the size of the population used by the algorithm
- **Overlap size (Overlap)** – the number of best individuals, which are copied directly (without cross-over or mutation) from the current population to the offspring population
- **Cross-over probability (CrossProb)** – the probability of cross-over between individuals in the intermediate population
- **Mutation probability (MutProb)** – the probability of gene mutation in the chromosome
- **Fitness scaler (FitScaler)** – scaling function used for selection pressure control

Figure 3.13. presents the block diagram of the algorithm.

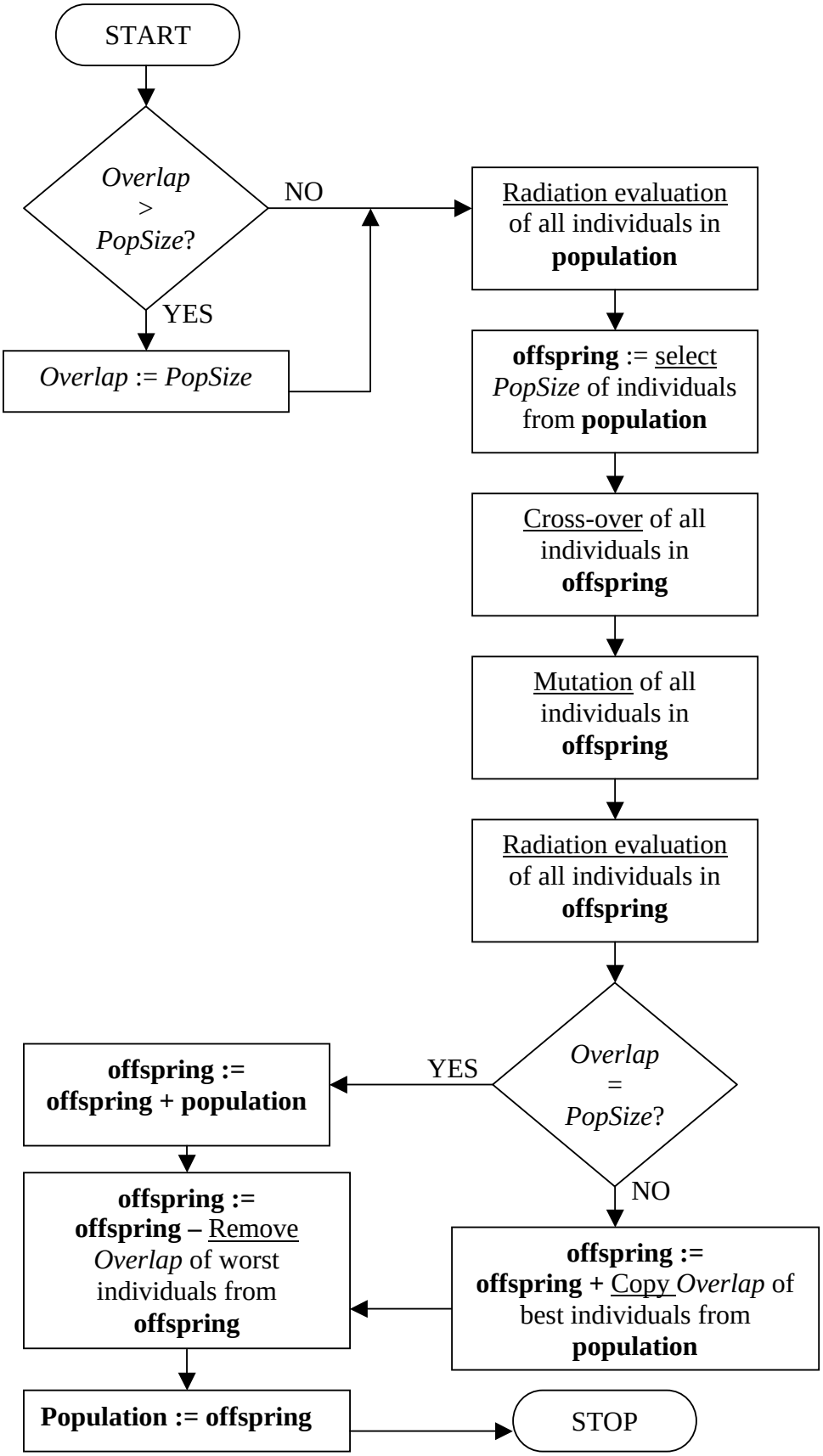


Figure 3.13. Classic genetic algorithm with overlapping of poulations

Radiation evaluation function

This function simply determines the fitness function value for each of the individuals using radiation tolerance evaluation function.

Select function

This function selects individuals from the population. This can be done for example using roulette wheel selection. Additionally, fitness scalers can be used to control the selection pressure.

Cross-over function

This function performs cross-over of pairs of individuals in the population using one of the cross-over operators. Pairs can be selected in the order of appearance in the population: 1-2, 3-4, 5-6 and so on. However, this approach is in agreement with statistics only with even *PopSize*, because in the case of odd *PopSize* the last individual is never crossed-over. Thus, the order is relevant and the whole operation is unfair. Cross-over is performed for every pair with probability equal to *CrossProb.*

Mutation function

This function mutates every gene in every chromosome with probability *MutProb*. This implementation is dependent on the length of the chromosome. The longer the chromosome, the more mutated genes expected.

Copy best function

This function copies n best individuals from the population. When there are more than n individuals with largest fitness value, the choice which of them to copy has to be made at random. Choice made on a basis of individuals position is unfair.

Remove worst function

This function removes n worst individuals from the population. When there are more than n individuals with smallest fitness value, the choice which of them to remove has to be made at random. Choice made on a basis of individuals position is unfair.

Version 2 ("classic with overlapping and strong start")

In order to accelerate the search, initial set of randomly generated chromosomes may be modified before the main evolution loop. For example the minimal number of chromosomes performing the required function with non-altered configuration (strong individuals) may be required in the initial population. The *Strong* function presented in figure 3.14. may be put after *Initialise* function in the main evolution program loop.

Additional parameters needed by this function are:

- **Strong treshold (StrongTreshold)** – the factor (0 to 1.0), which indicates when the individual is considered as strong. With factor 1.0 only those with fitness equal to *maxFit* are considered as strong.

- **Strong ratio (StrongRatio)** – the factor (0 to 1.0), which controls the number of strong individuals required in the population to stop the Strong function. Factor 1.0 sets the minimum required number to *PopSize*.

The Strong function is presented in the figure 3.14.

This function provides start point with higher fitness, what results in faster convergence, but it decreases the diversity in the population and consumes the processing power, what may result in pre-mature convergence.

Strong function

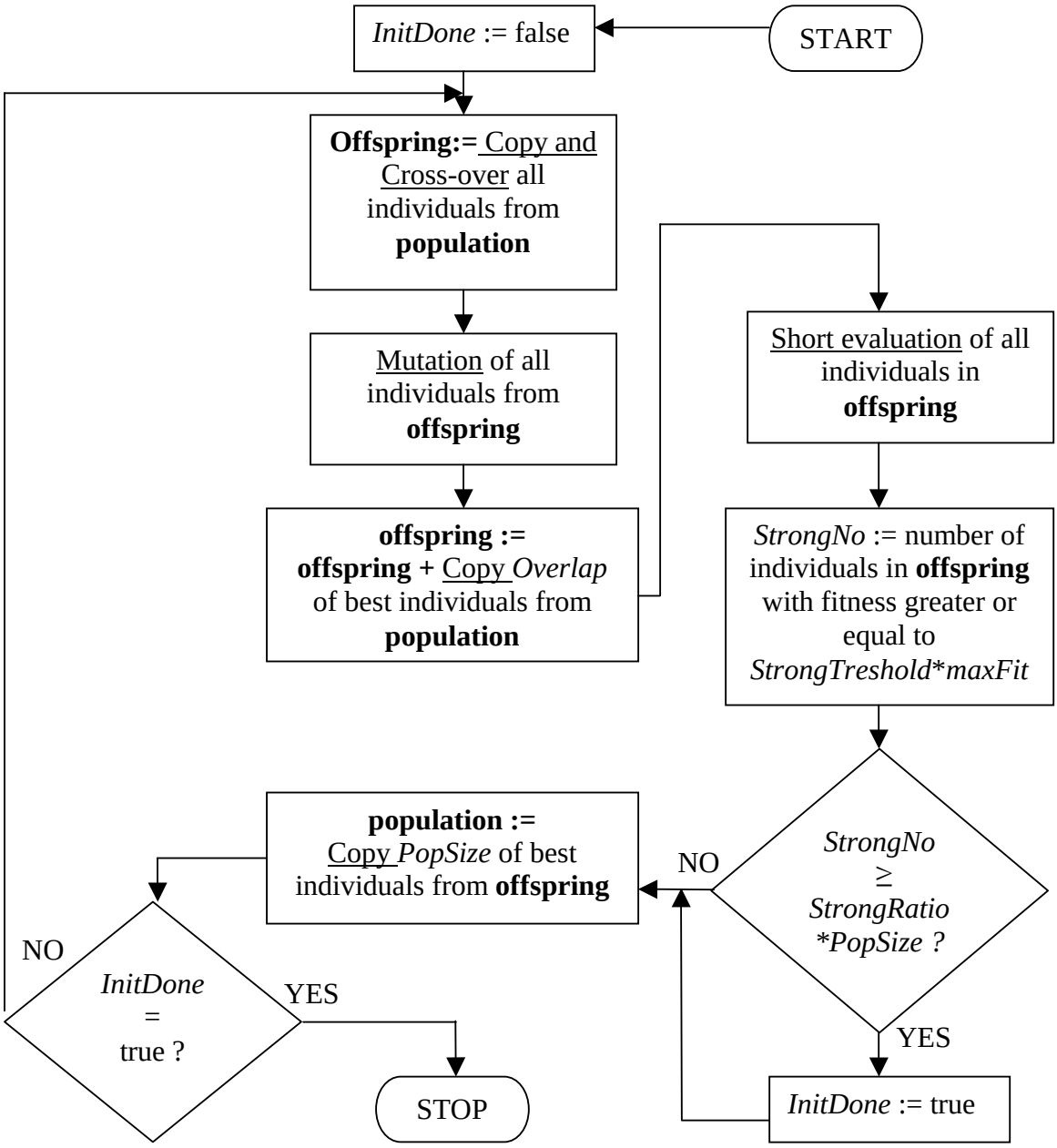


Figure 3.14. Strong function

Copy and cross-over all individuals

This function copies every possible pair combinations from the population. Every pair copied is crossed-over. This function produces many individuals, but provides wide selection possibilities. For example population of 100 individuals produces 10000 individuals for short evaluation.

Version 3 ("greedy for processing power")

The idea put in the *Strong* function can be used in the whole algorithm. All possible pair combinations may be used to search the solution space for the right one. Moreover, the individuals, which do not meet the requirements can be removed from the population. The *Strong* function is no longer used here, the *Evolve* function is severely changed, as in figure 3.15. Parameters required in this algorithm and not yet described are:

- **Strong required (StrongRequired)** – the number of strong individuals required in the population. This algorithm no longer uses *StrongRatio*.

- **Required short fitness (ReqShortFit)** – the minimum fitness function value, which is required for individuals after short evaluation is done. This is taken into account in selection which individuals to remove.

- **Required radiation fitness (ReqRadFit)** – the minimum fitness function value, which is required for individuals after radiation tolerance evaluation is done. This is taken into account in selection which individuals to remove.

- **Maximum population size (MaxPopSize)** – this is the maximum size of the population after the evolution step.

This function produces many individuals, even at the radiation tolerance evaluation step. Therefore, it needs much of processing power, because radiation tolerance evaluation is the most computationally intensive part of the program. The advantage of this algorithm is that it does a wide search, because many individuals are checked, but by removing non-satisfactory individuals it decreases genetic diversity in the population, what may lead to pre-mature convergence.

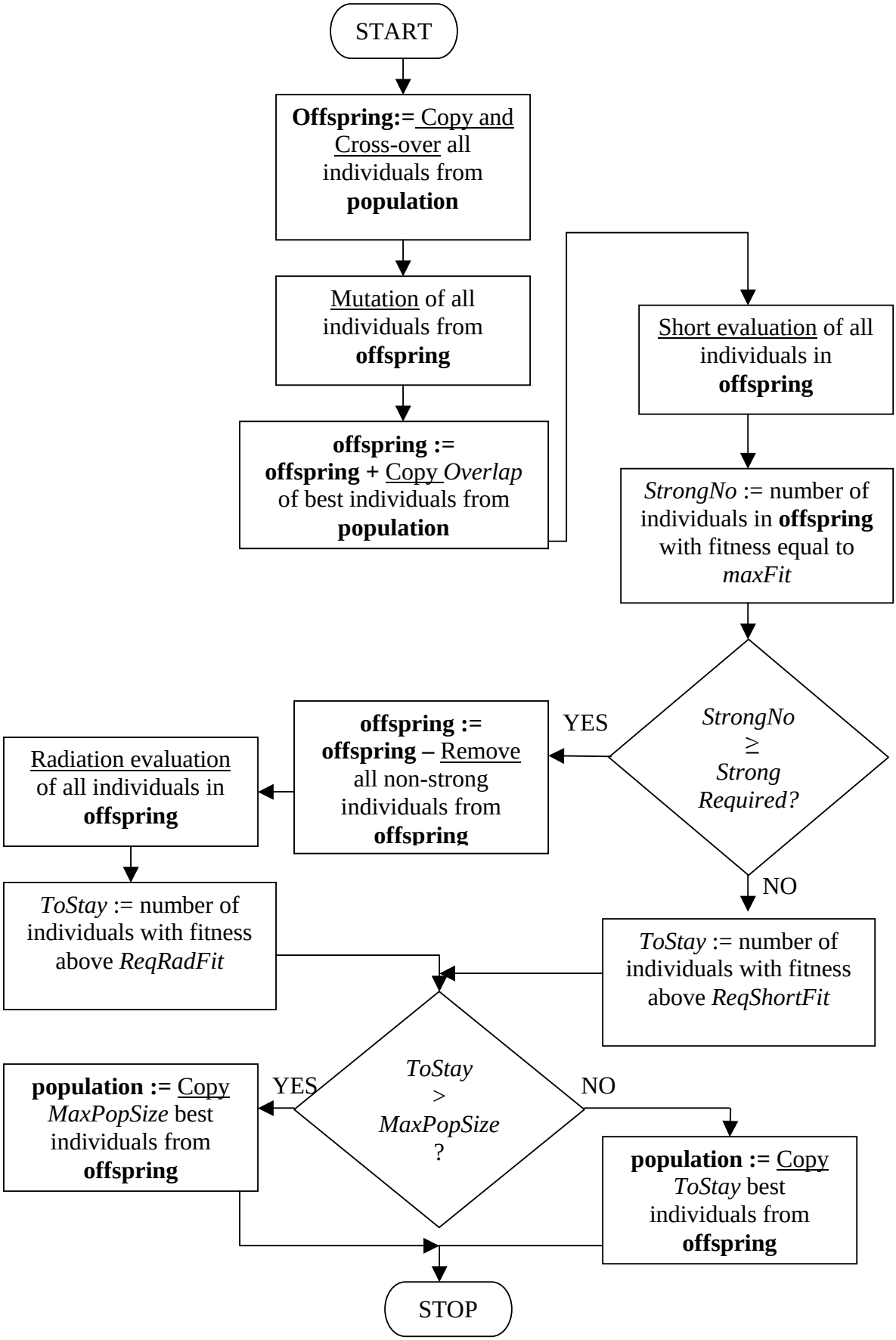


Figure 3.15. Greedy evolution algorithm

4. Distributed System

All evolution program implementations described in the previous chapter require large processing power resources. Usually such applications are run on supercomputers, which use many processors to finish computationally intensive tasks in satisfactory time. However, supercomputers are not widely available, not many scientific or even military institutions can afford these machines. Therefore, their usage is very limited. When large computing power is needed, but is not available in single machine, the distributed system can be used. Such system distributes the parts of the problem to many machines; they work simultaneously on the parts provided and return the result. This approach is useful, when there are many unused ordinary computers available, which can devote their resources to the system. This idea reminds computer farms, often created in academic centres. But computer farms are much bigger than the designed system and usually cooperate using WAN connections. There are several well known distributed systems utilising Internet as connection medium between elements of the system. For example Search for Extraterrestrial Intelligence at home (SETI@Home) [23]. It is a scientific project seeking for the intelligent life outside Earth. It uses the processing power of participating computers to analyse the signals coming from the UC Berkeley telescope. The client program comes in a form of a screen saver. Normally computers run screen savers, when they are not used, but stay powered on. Their resources are wasted. Here comes the SETI system, which utilises the power of thousands of such computers around the world. This provides large potential, which can be used to analyse data from the telescope. The purpose of this thesis is to create the similar system for circuit design using genetic algorithms.

4.1. System Structure

Usually distributed systems use client/server architecture. The designed system has one, central server and many clients connected. This reminds the star topology. Figure 4.1. presents the elements of the system.

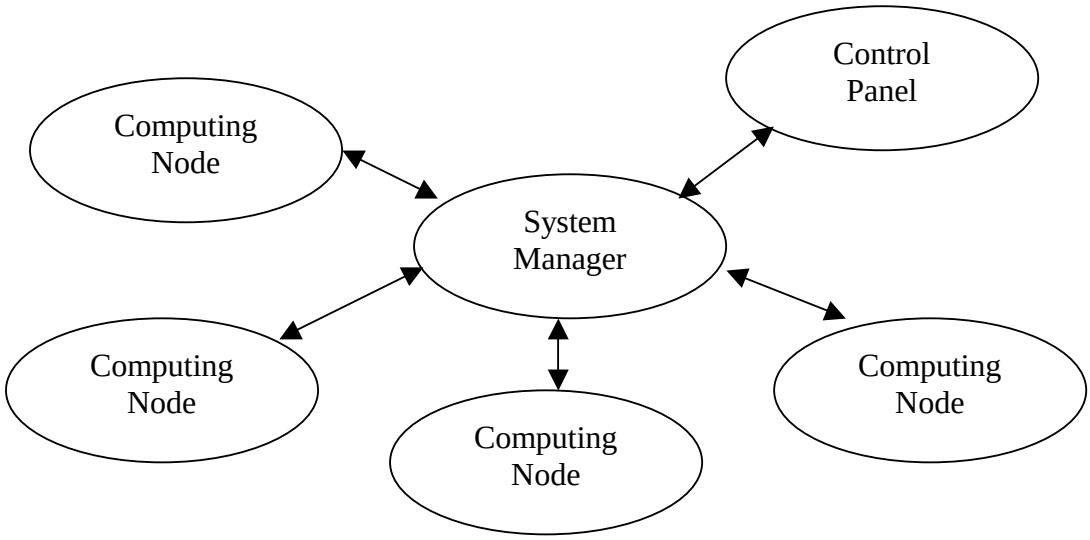


Figure 4.1. Distributed computing system

Every element presented in the figure 4.1. can be executed on separate computers or on one machine for local computations. Double arrows indicate communication between elements of the system, what can be achieved using a distributed environment. The most computationally intensive part of the evolution program is the evaluation function value calculation; therefore this is the part of the problem, which should be delegated to many machines. Description of possible distributed environment choices is presented in the section 4.2.
The functions and internal structure of each element are presented below.

System Manager

This is the main part of the system. By assumption, there can be only one main node. It prepares the parts of the problem that can be delegated to the computing nodes and than collects the results. The internal structure of the System Manager is shown in the figure 4.2.

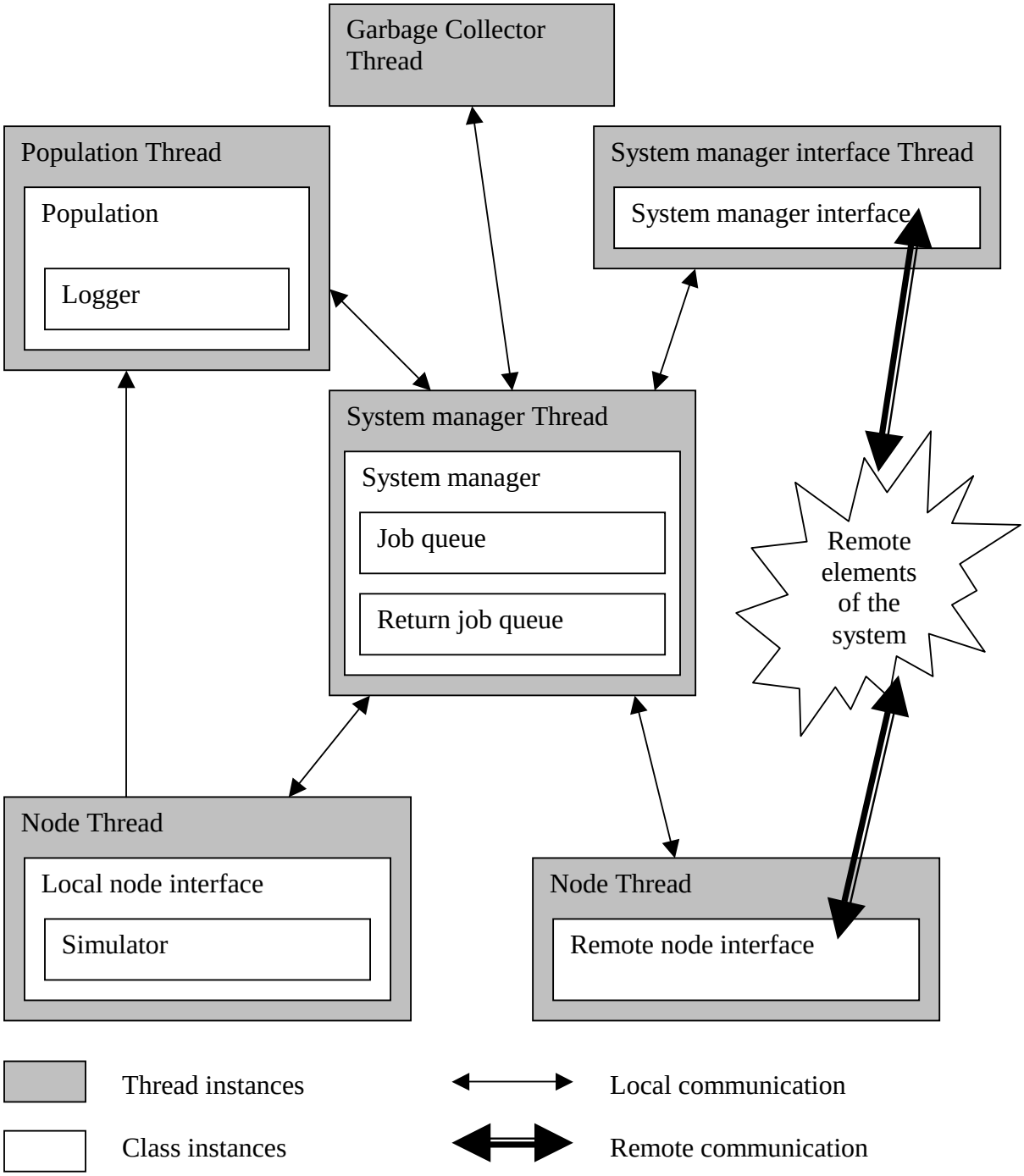


Figure 4.2. System manager internal structure

Population thread

Every population thread together with population class instance represents an evolution program. Population thread provides main evolution program loop processing. Population class contains population parameters, the chromosomes of population individuals and code describing how to prepare the population for evolution and how to evolve to the next generation. It also creates an instance of Logger class, which saves the most important

parameters of the population to the file. In fact, the Population class cannot be instantiated, because it is an abstract class. It needs to be inherited and descending class implements **evolve()** and **initialize()** methods. This way, program can be easily extended with new evolution algorithms. The figure 4.3. shows the inheritance diagram of classes implementing all algorithms described in section 3.1.4.

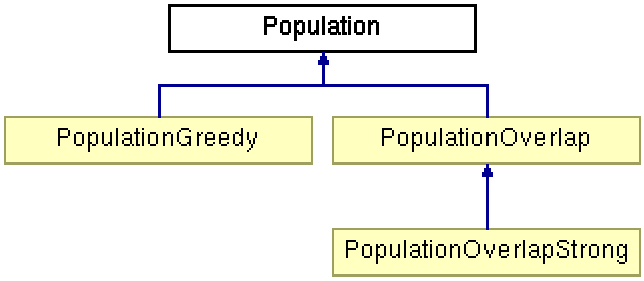


Figure 4.3. Population inheritance diagram

There can be many population threads running at once.

Node thread

Node thread is created for every connected node. The node thread can use local node interface instance, which has its own instance of programmable circuit simulator and uses the local machine processing power for computations. It can also use remote node interface, which communicates with remote machines using distributed environment and this way delegates the task.

Garbage collector thread

This thread checks at some interval for unused thread instances and frees the memory occupied by such objects. When remote machine disconnects, its node thread becomes needless, or when population evolution is finished, its population thread is needless. These are cases when garbage collector is in action.

System manager thread

System manager thread uses system manager object instance to communicate with all other objects. This is a “connection point” for most of the object instances. It has two queues, which are used for task delegation to the node threads. Job queue is normally used

as a container for chromosomes waiting for evaluation; return queue is used in case, when some node cannot finish chromosome evaluation.

It is important to know how tasks are distributed around the system. Figure 4.4. shows a communication sequence between the main threads.

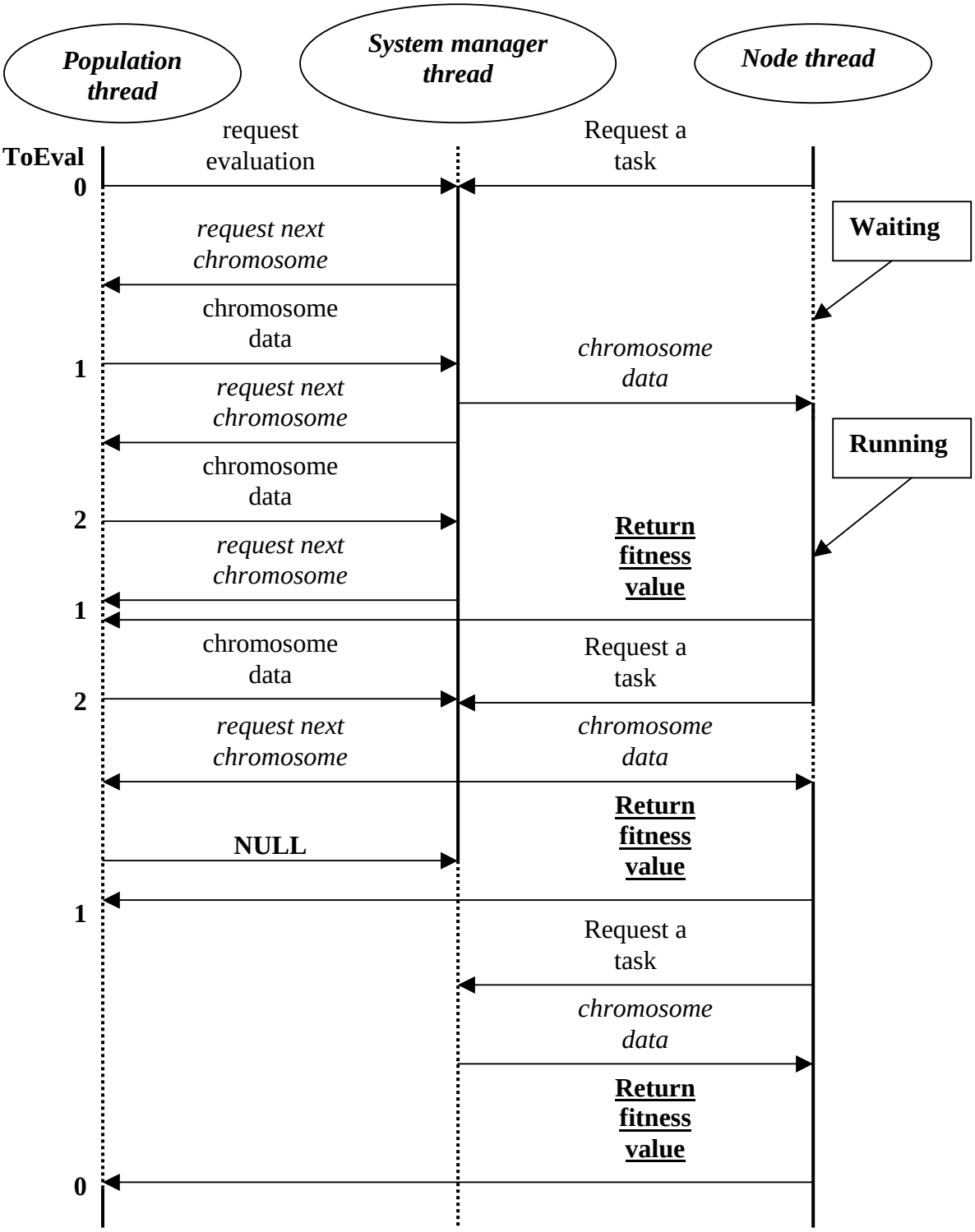


Figure 4.4. Communication between threads

The population thread requests evaluation of all individuals in the population by calling appropriate method of system manager class instance and goes into waiting state. The method called adds population to the list of populations ready for evaluation and wakes up system manager thread. The system manager thread takes the population from the list and requests chromosome data. Chromosome data is returned as a result of population class method called from the context of system manager thread; therefore population thread does not have to be woken up. Chromosome is put into the job queue and next chromosome is requested. When there are no more chromosomes to evaluate, NULL value is returned what indicates the system manager that all data is in the job queue. System manager thread removes the population from the list of populations to evaluate and goes into suspended state if there are no other populations waiting for evaluation. Simultaneously, the node thread requests a chromosome to evaluate. The job queue is a blocking queue; therefore when there are no jobs available, node thread is suspended. As soon as the chromosomes are put into the job queue, node thread wakes up and starts evaluation. Local node uses local machine resources for evaluation; remote node sends the task for evaluation to the remote machine. The fitness function value of the chromosome is returned directly to the population class instance by calling appropriate method; the counter of not evaluated chromosomes is decreased. When fitness function values of chromosomes are saved, the population thread resumes operation and continues evolution loop.

The return queue is used, when remote machine disconnects suddenly. Then, the node thread responsible for that computing node, returns not evaluated chromosome to the return queue and finishes operation. Redundant thread is removed later by garbage collector thread.

In fact, the sequence of operation presented in the figure 4.4. is simplified, because there is only one population and only one node present. Moreover, in real system, the node thread does not return fitness value just after the chromosome has been received. It collects couple of chromosomes and sends a packet for evaluation to the remote machine. This technique is used to decrease possibility of network congestion caused by frequent exchange of information between system manager and computing nodes. One could say, that there is no difference between sending chromosomes one by one or in a packet, because the amount of data is the same. This is not the truth, because there is some overhead produced by distributed environment whenever the connection is used.

Additionally, the reference data, like the reference circuit or programmable circuit architecture are sent only once.

Coming back to the description of the system elements presented in the figure 4.1. the computing node and control panel need to be described.

Computing node

The internal structure of this program is shown in the figure 4.5.

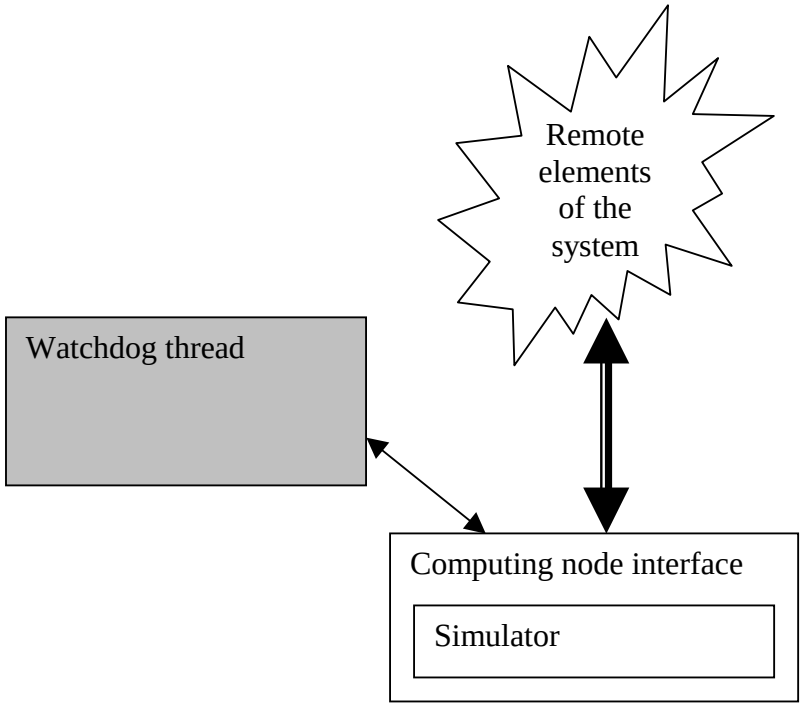


Figure 4.5. Internal structure of the remote computing node

After the node program is executed, it tries to contact with the system manager interface and register by sending its reference. System manager starts node thread for the computing node being registered. Started node thread uses the reference to the computing node interface for evaluation of the chromosomes. Computing node has a watchdog thread running simultaneously with the main thread. Watchdog thread has a counter, which is incremented at certain time intervals. When the limit of the counter is reached, program exits. Each evaluation method call resets the watchdog counter to 0. This feature is implemented to prevent node from occupying resources when there is no population simulated, system manager computer is down or some network problem broke the connection between system elements. The program may be started periodically

(with cron for example) to check, whether the system manager server is up and simulates any population.

Control panel

The control panel is simply a command line interface used for the control of system functioning. It can start new populations, stop populations, list nodes connected, list populations being simulated. System manager interface is used for communication with the system. Calling appropriate methods of this object does all above-mentioned actions. Upon start of the new population, the user answers series of questions concerning population parameters; the reference circuit and programmable circuit architecture files are loaded and sent to the system manager interface. The system manager interface starts new population thread with appropriate parameters.

4.2. Distributed Environments

The cooperation between program elements like object instances can be easily realised by calling methods and in non object-oriented programs by function calling. In the past, all of the programs were executed on a single machine. In other words, the program used single address space. Method calling in the same address space is relatively simple and natural process. However, sometimes it is necessary to distribute the elements of the program and form a distributed system to get bigger flexibility and scalability. This way, the resources that can be used by the program are limited only by the number of computers connected together, moreover users using different machines can easily exchange information, work simultaneously on the problem and share hardware. The latter advantage is especially important in case of special, expensive hardware, which must be connected to only one computer, but can be used by many machines. The machines in the distributed system are physically separated and require some interconnection. Connectivity is usually provided by a computer network, where network sockets can be used for inter-machine communication. But this approach is dependent on the underlying network interface and involves creation of own application layer protocol. The distributed environments are designed to overcome the above-mentioned difficulties, because they define the protocols and rules of communication (especially higher level protocols). They provide the programmer with possibility of calling remote methods or remote functions (executed on the remote machine) as if they were contained in the local address space.

Different environments use different network protocols, different programming languages or platforms. But all of them are able to make the remote object or function look as a local one. This is done with so-called client/server architecture, where the calling party is the client and the called party is the server side. Usually the client side is represented by a proxy or stub, which looks from the application point of view as a local object or function, but it uses a distributed environment mechanisms to redirect the request to the appropriate server. Server is represented by a skeleton, which is able to receive request from the client, perform desired processing and return the response. There are plenty of distributed environments types and vendors. The most important are presented below.

Remote Procedure Call (RPC)

This is non object-oriented environment, which enables a programmer to call the remote procedures. The local procedure calling process involves the following steps [24]:

- Prepare the call arguments, and place them in shared memory (a register or a stack)
- Execute procedure code
- Receive return value from the specified memory location

Remote procedure call is very similar, but instead of calling local procedure the client stub procedure is called. The client stub puts or marshals the parameters into the network packet. Than the client stub sends a RPC call to the server. Server executes its local implementation of the procedure and returns the result over the network to the client stub. The client stub returns the result to the calling process. During synchronous RPC the calling process is suspended for the time of the call. Figure 4.6. illustrates the sequence diagram of synchronous RPC [25].

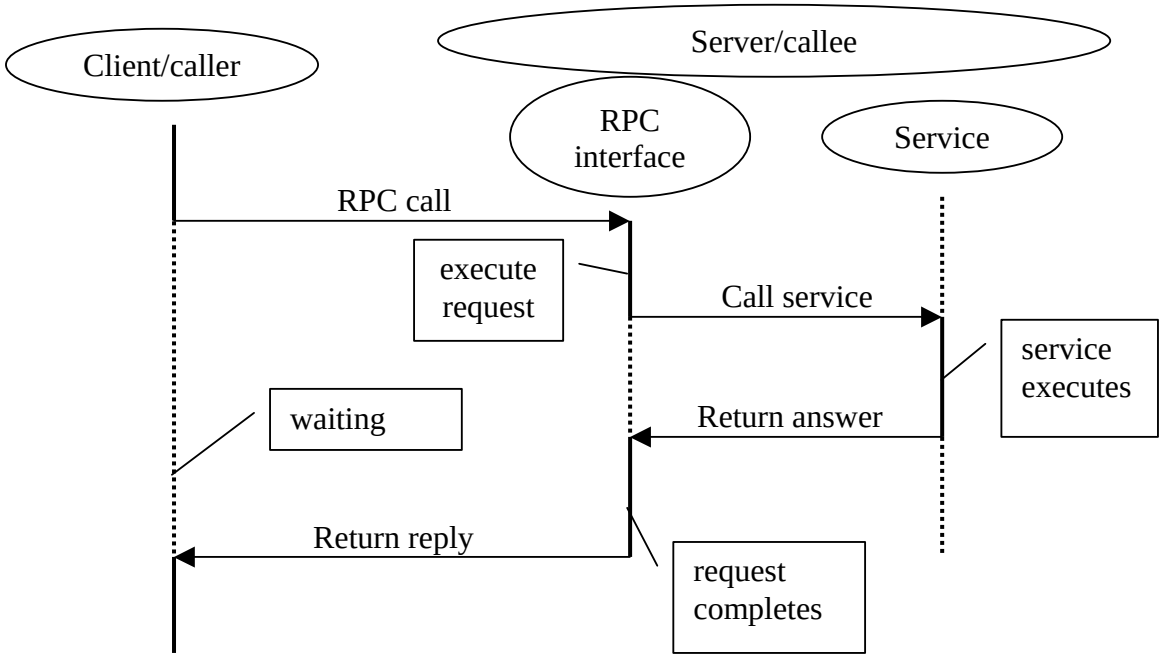


Figure 4.6. Synchronous remote procedure call

The asynchronous RPC calls can also be realised. The calling process is not suspended than, but continues execution. The result is then returned by use of callback functions. When server finished the procedure implementation execution, it calls client the callback function and gives a return value as a parameter. The good candidate for asynchronous RPC is a printing function.

The interface of the server (the procedures that server implements) is described in terms of Interface Definition Language (IDL). The appropriate code fragments like client stub and server skeleton that realise RPC are created automatically by the IDL compiler. Another file created is the external Data Representation (XDR), which is a data abstraction to facilitate the machine independent communication regardless the byte order and other machine details.

RPC provides the programmer's interface at different levels. At a very simple level, remote procedure call can be done using only one function. Three numbers uniquely identify the remote procedures: program number, version number and procedure number [25]. Program consists of a group of related procedures, which are all distinguished by the procedure number. Version number is used, when the interface is changed. Since already published interface may be used in many programs, every change needs a new program version number to keep the programs using old interface running. The biggest disadvantage of RPC is that object distribution is not supported what eliminates RPC as a possible distributed environment choice, because system structure described in section 4.1. is mainly built using objects, including the inter-element communication.

Distributed Component Object Model (DCOM)

DCOM is an extension of Component Object Model (COM) standard developed by Microsoft [26]. COM standard defines the binary representation of the component and the component creation process, what makes it compatible with many object-oriented languages. The standard is fully supporting the object-oriented programming with code and data encapsulation, polymorphism and inheritance. In a way, it is similar to the Dynamically Linked Library (DLL) standard, but is more versatile. The components provide their services to other components or applications by means of one or more interfaces. Interfaces are sets of functionally related methods, which are called using virtual tables. Virtual tables memory layout must obey the standard, which is identical to the C++ vtable used with abstract classes [27]. Each component interface represents a different view of the component. The client interacts with the component by acquiring the pointer to the one of the component's interfaces and calling the methods of the interface. DCOM is an extension, which distributes COM for more than one machine and allows the client to interact with the remote components as if the object

was present in the client address space. DCOM object interfaces are defined with the Microsoft Interface Definition Language (MIDL). It is an extension of RPC IDL mentioned in the previous section. As in the RPC case, IDL compiler creates client and server stub codes and header files, and type library (*.tlb file). In DCOM the client stub is called a proxy, and the server stub is called a stub. Each class receives a class ID (CLSID) and the interface receives universally unique ID (UUID) called the interface ID (IID) [26]. The correspondence between CLSID and the server code is registered with the system registry. This is exploited for static object invocation using vtable method. For dynamic object invocation, the object must implement IDispatch interface, which provides interfaces to query the description of object methods and their parameters in a form of type library.

In DCOM reference counting controls object life cycle [27]. This is realised by IUnknown interface, which is the mandatory ancestor of every interface. IUnknown interface implements three methods: QueryInterface(), which checks whether the object implements specified interface, AddRef(), which increments reference count and Release(), which decrements reference count. Each client should invoke AddRef() after acquiring the pointer to the interface and Release() after the interface has been used. When reference count is 0, the object server deletes itself and frees consumed resources.

In order to make a remote call, the client calls a client proxy. The client proxy marshals the parameters into the request message and invokes a wire protocol. In case of DCOM the protocol is Object Remote Procedure Call (ORPC) [28].

As mentioned before, DCOM is the standard described at the binary level and it supports components written in variety of languages. However, when platform aspect is taken into account the flexibility is much worse. As a Microsoft standard, DCOM is tailored for Windows platform. There are ports to Unix systems (developed by Software AG), but they at beta versions stage [29].

Common Object Request Broker Architecture (CORBA)

CORBA has been developed as a standard by Object Management Group (OMG) since 1989. The aim of CORBA is to provide the machine, programming language, platform and network technology independent standard of programming interfaces and models for object-oriented distributed applications. OMG does not provide any CORBA implementation; the independent vendors do this. Vendors have to implement

minimum required functionality in their products, but they often add proprietary extensions. Because of these extensions different CORBA products may not be fully interoperable, but provide interoperability at the basic level described by the standard. The central part of the CORBA system is the Object Request Broker (ORB), which provides a communication between objects, activates not running objects when needed and provides the interfaces that can be used by clients and objects. The figure 4.7. shows the interface categories distinguished by OMG.

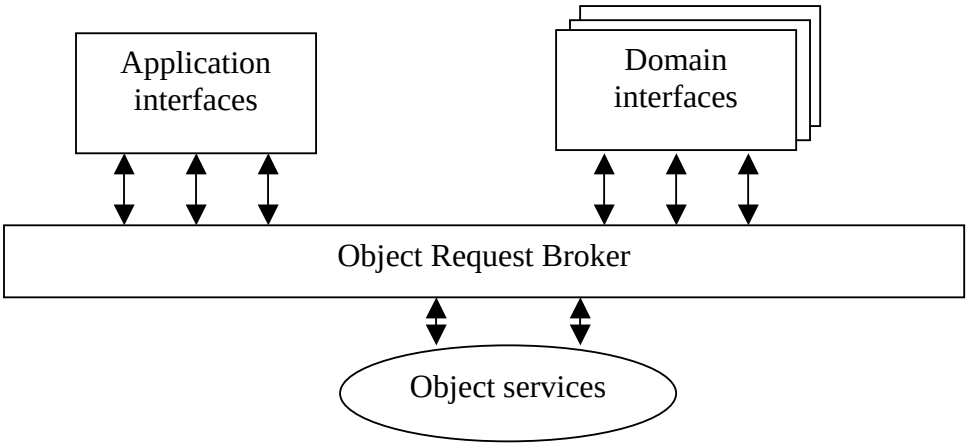


Figure 4.7. Interface categories defined by OMG [14]

- **Domain interfaces** are the interfaces by many distributed object applications. They are grouped into domains, because every domain is specific to some industry or field of application. There can be many domains define, as indicated by many boxes in the figure 4.7.
- **Application interfaces** are interfaces defined especially for the given application. They are not standardised by OMG.
- **Object services** these are interfaces that are commonly used by the applications. Usually these services are perceived as a part of core CORBA system. The Naming Service or Trading Service are examples of interfaces contained in this group. Both facilitate obtaining remote object reference by the client.

CORBA consist of elements shown in the figure 4.8.

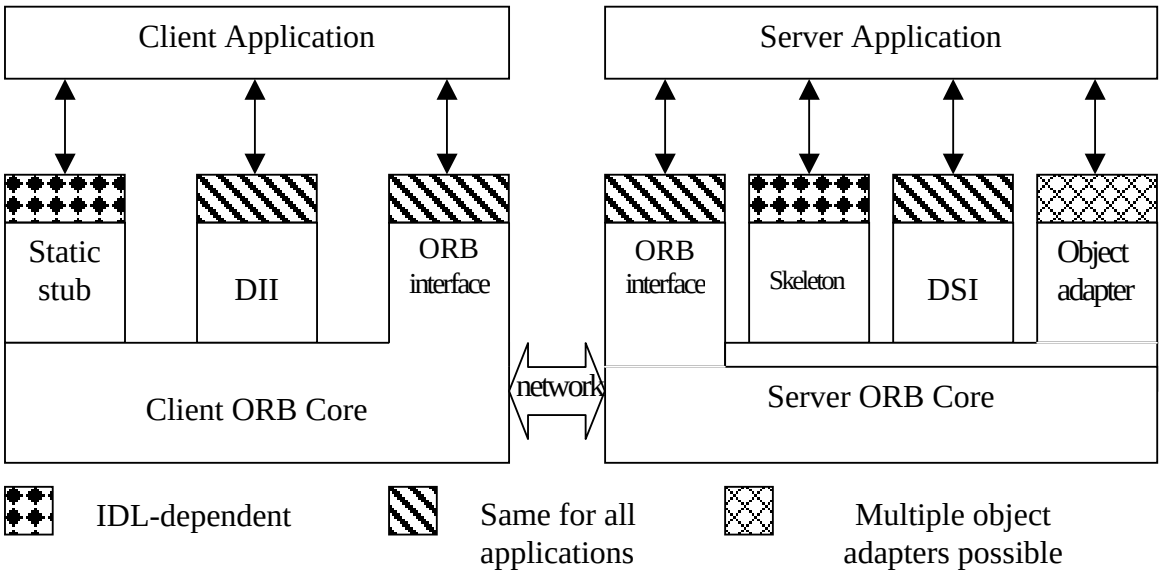


Figure 4.8. Common Object Request Broker Architecture [14]

As systems described previously, CORBA uses IDL to describe the interfaces. Every CORBA object has to be described in the IDL file. IDL compiler creates client stub, server skeleton and header files. These are elements of static object invocation. CORBA supports also dynamic object invocation with Dynamic Invocation Interface (DII) on the client side and Dynamic Skeleton Interface (DSI) on the server side. Objects are uniquely identified with Interoperable Object Reference (IOR). The client has to obtain a server IOR somehow. CORBA provides Naming Service and Trading Service for this purpose, which are accounted to the object services.

The request flow in CORBA system is as follows [30]:

- Client makes request using static or dynamic object invocation. The request is passed to the Client ORB Core
- Client ORB Core passes the request to the Server ORB Core utilising a wire protocol. The mandatory protocol for CORBA systems is Internet Inter-ORB Protocol (IIOP), which is a TCP/IP specialisation of General Inter-ORB Protocol (GIOP).
- Server relays the request to the object adapter that created the object. There are two types of object adapters: Basic Object Adapter (BOA), and Portable Object Adapter (POA). The former one is obsolete, as it is was implemented differently by different vendors. Object adapter assists ORB in object activation and request delivery.

- Object adapter, either statically or dynamically localises the servant implementing the object and executes appropriate method. Servant is a server-side class implementing the interface exposed to the ORB.
- After the method completion, the result is returned to the client

The object-life cycle is controlled using reference counting as in DCOM. Not used objects are deleted and all resources consumed by those objects are released.

CORBA supports multiple languages by so-called language mappings. They specify how IDL code is translated into the source code. The mappings are standardised for many languages: C/C++, Smalltalk, Cobol, Ada and Java. These mappings allow different elements of the system to be written in different languages, but stay interoperable.

CORBA ports are available to many platforms, what makes this architecture fully portable. Moreover, many implementations are available for free, what is additional advantage. The biggest CORBA disadvantage is the complexity of the architecture and lack of application development tools support what may increase application development time.

Remote Method Invocation (RMI)

This system has been developed for Java programming language. Its principle of operation is similar to the other distributed object environments. So, the stubs or proxies are created, which look for the programmer as local objects, but they facilitate invocation of the remote objects on other Java Virtual Machine (JVM). Remote objects implement methods exposed by local interfaces. However, the RMI follows the simplicity of Java language and distributed object model is integrated into Java programming language, it does not use any special language as other environments do. A remote interface must inherit (extend in Java terminology) *java.rmi.Remote* interface and all remote methods must declare throwing *java.rmi.RemoteException* exception. These features differentiate remote objects from local objects.

The call procedure is as follows [26]:

- local JVM initiates connection with JVM, which contains the remote object (client stub functionality)
- the parameters are marshalled and send to the remote JVM (client stub functionality)
- remote JVM unmarshalls the parameters, locates the interface implementation, executes the specified method (server skeleton functionality)

- remote JVM marshals the result and sends back to the local JVM (server skeleton functionality)
- local JVM unmarshalls the return value and sends back to the caller (client stub functionality)

Java wire protocol consists of two layers [31]. Upper layer is realised using object serialisation, which is used for marshalling and unmarshalling of parameters and return data. The lower layer is realised using HyperText Transport Protocol (HTTP). Serialisation encodes an object into byte-stream and facilitates the object reconstruction from this stream. Thus, parameters are not truncated during marshalling, private or transient data is protected and true polymorphism is supported. Serialisation also provides the information on the location of class definition files what enables dynamic class loading. The JVM, which unmarshalls the parameters tries to locate the classes by name in its local context. When the classes are not available, their definition files locations in form Uniform Resource Locator (URL) path are obtained from the serialised byte-stream.

The remote object reference may be obtained using simple naming service provided by *java.rmi.Naming* which uses URL path for object location purposes. The remote object reference can be also send as a parameter or a return value of a method call.

Object life cycle is controlled using reference counting, as in previous cases. The object, which is not referenced by any client, is referenced by RMI using weak reference. When there is only weak reference to the object and there are no local references, object is collected by JVM's garbage collector [31].

RMI support is provided for many platforms and operating systems with JVM implementation. Additionally, RMI is perceived by programmers as a simpler to use than other distributed object environments, because it is integrated into the Java language and does not use any special semantics. However, this feature limits RMI only to Java, it cannot be used with any other language. Moreover, use of object serialisation restricts interoperability with other distributed object environments. This is a severe limitation, which restricts use of RMI in systems written in other languages that need to be extended with distributed object support.

Choice of distributed object environment

In order to make a distributed environment choice, the basic system requirements need to be stated.

- **Operating system: Linux**

  The linux operating system is chosen because it is installed on all machines available for simulations, especially servers. Minimal linux installations consume small machine resources, because they do not provide graphical user interface, which is not needed in the case of distributed computing system. Additionally, there are many free implementations of distributed environments for the Linux.

- **Programming language: C++**

  The primary objective is system performance. C++ produces fast executable code and supports object-oriented programming what rises the programming comfort and shortens the application design time.

- **User interface: text**

  There is no need for graphical user interface (GUI), because the user actions are not complicated and can be easily realised using text interface. Text interface greatly simplifies the program and can be easily accessed via the Internet.

As mentioned earlier, RPC is not suitable to the distributed computing system purpose, because it does not support object-oriented programming. The RMI is excluded because of the Java language limitation. The DCOM could be used, but it is mainly targeted to the Windows environment. The best option is CORBA. It provides many free and well-designed linux ports and does support C++.

CORBA interfaces

System objects, which cross the boundaries of one address space need to be described in terms of IDL language and then compiled into client stubs and server skeletons. The IDL file describing the system functionality is presented in the listing 4.1. Some parts of the type definitions and parameter lists are omitted for clarity. The omitted parts are indicated with (…). Full version of IDL file for the system is provided in the appendix B.

Listing 4.1. IDL file describing distributed system interfaces

```
interface CORBANode{

      (...)

      long shortEvaluate(in Chromosomes question, out Fits answer);
      long radEvaluate(in Chromosomes question, out Fits answer, in
RadFitnessType radType, in double maxFit);
      void setArchitecture(in StringSeq arch);
      void setCircuit(in StringSeq circuit);
      void setResults(in StringSeq results);
      void reinitializeSimulator();
      void isAlive();
};

interface CORBAManager{

      (...)

      void registerNode(in string reference, in string name)
raises(CannotRegisterException);
      void startPopulation(...);
      void stopPopulation(in string name) raises
(CannotStopPopulationException);
      CORBANode::StringSeq listPopulations();
      CORBANode::StringSeq listNodes();
};
```

The **CORBANode** interface describes the interface of computing node, which is exposed to the system manager.

shortEvaluate() method performs the "short" evaluation of all of chromosomes in the supplied chromosome packet and returns the packet of fitness values

radEvaluate() method performs the "radiation" evaluation of all of chromosomes in the supplied chromosome packet and returns the packet of fitness values. The additional parameters determine fitness function type and maximum possible fit. Maximum possible fit is needed for these fitness function types, which scale the value obtained using the behaviour of the non-altered configuration as described in section 3.1.4.

setArchitecture() method changes the architecture of the programmable circuit used by the simulator

setCircuit() method changes the reference circuit used by the simulator. The truth-table reference use is disabled.

setResults() method changes the truth-table reference used by the simulator. The reference circuit use is disabled.

reinitializeSimulator() method reinitialises simulator after changes of the reference circuit

isAlive() method is used by the system manager to ping the computing node

The **CORBAManager** interface describes the interface of the system manager, which is exposed to the other elements of the system.

registerNode() method is used by the computing node for registering in the system manager. Computing node supplies its name and stringified reference.

startPopulation() method is used by the control panel to start a new population simulation. The parameter list is omitted in the listing, because of its length and complexity, which comes from the fact that every parameter of the started population must be determined.

stopPopulation() method is used by the control panel for population removal

listPopulations() method is used by the control panel to obtain the list of populations being simulated

listNodes() method is used by the control panel to obtain the list of the computing nodes connected to the system manager

5. Simulation Results

Distributed system efficiency

At first, the distributed system efficiency was tested. The test was carried out for classic algorithm, with population size 200, chromosome length 448 bits and full-adder reference circuit. The full-adder is described in the next paragraph. System consisted of 10 computers equipped with Celeron 1.2GHz CPU. The measure of system performance is the average computation time of one generation. Table 5.1. summarizes the results.

Table 5.1. Distributed system efficiency

Number of computers	Average time [s]	Total Power	Utilisation [%]
1	52.32	1.0	100
2	26.34	1.98	99
3	18.00	2.90	97
4	13.55	3.86	97
5	11.03	4.74	95
6	9.38	5.58	93
7	8.56	6.11	87
8	7.75	6.75	84
9	6.98	7.50	83
10	6.22	8.41	84

The distributed system utilises the total processing power of the computers quite efficiently, all nodes contributed over 80% of their potential to the system. The utilisation of the total computing power decreases with every new computer connected. Probably this comes from the fact that every node consumes some CPU resources of the main system node, moreover total time devoted to communication between nodes increases. The unexpected rise of the utilisation caused by 10-th connected computer and sudden fall of utilisation caused by 6-th connected computer possibly arises from slight differences in computer configurations. Figure 5.1. clearly depicts those anomalies.

The conclusion is that the distributed system is a powerful tool for any tasks requiring large processing power. An example is useful to pinpoint the importance of large processing power in genetic algorithm design, the simulations performed for this thesis needed 50 days to complete with 10 computers connected to the system, simulation on 1 computer would last 50 days*8.41 = 420.5 days = 14 months!

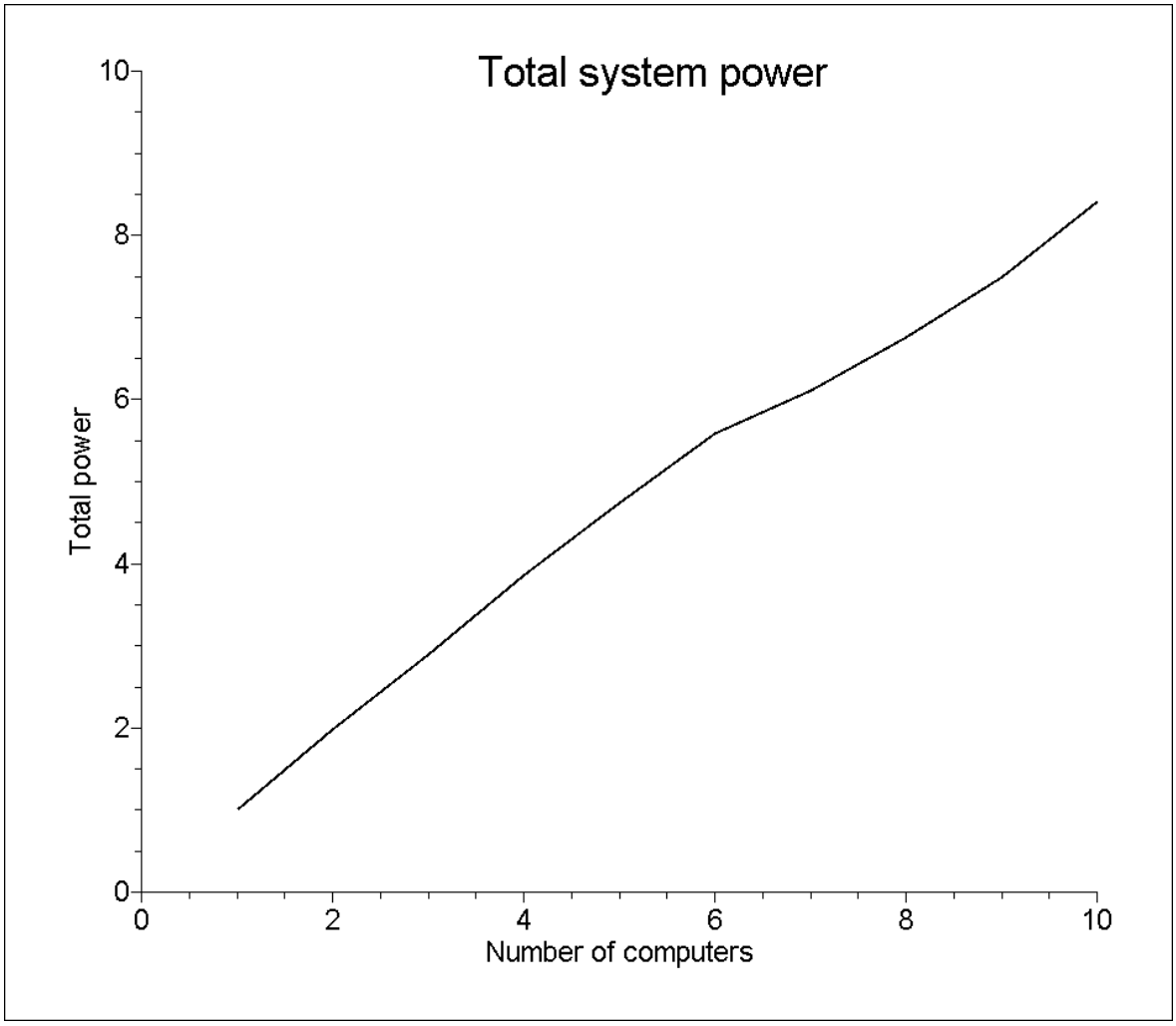


Figure 5.1. Total system power versus the number of connected computers

Simulation procedure

All below described tests are “one-shot” tests, what means that results come from single simulation. The best would be to run every simulation several times with the same parameters and then average the results, but despite the distributed system computing power, simulation times are quite long and force “one-shot” method. The negative aspect of this approach is that it is possible that “lucky” sequence of random numbers may promote one of the parameter sets, which in most of the simulations would not be so good. Usually the number of generations reaches tenths of thousands; therefore the influence of randomness incorporated in the genetic operations is small. Large population size should alleviate the problem with starting genetic material better than average. First simulations concentrate on the proper choice of parameters. For every parameter, several possibilities are tested and compared. Of course, there is some correlation between

them; therefore determination of the best value in isolation may not give the optimal set of parameters. However, due to limited processing power available, this is the only reasonable method. All simulations use full adder as reference circuit, that means that GAL loaded with the designed configuration is expected to function like full adder. Full adder implementation is shown in the figure 5.2.

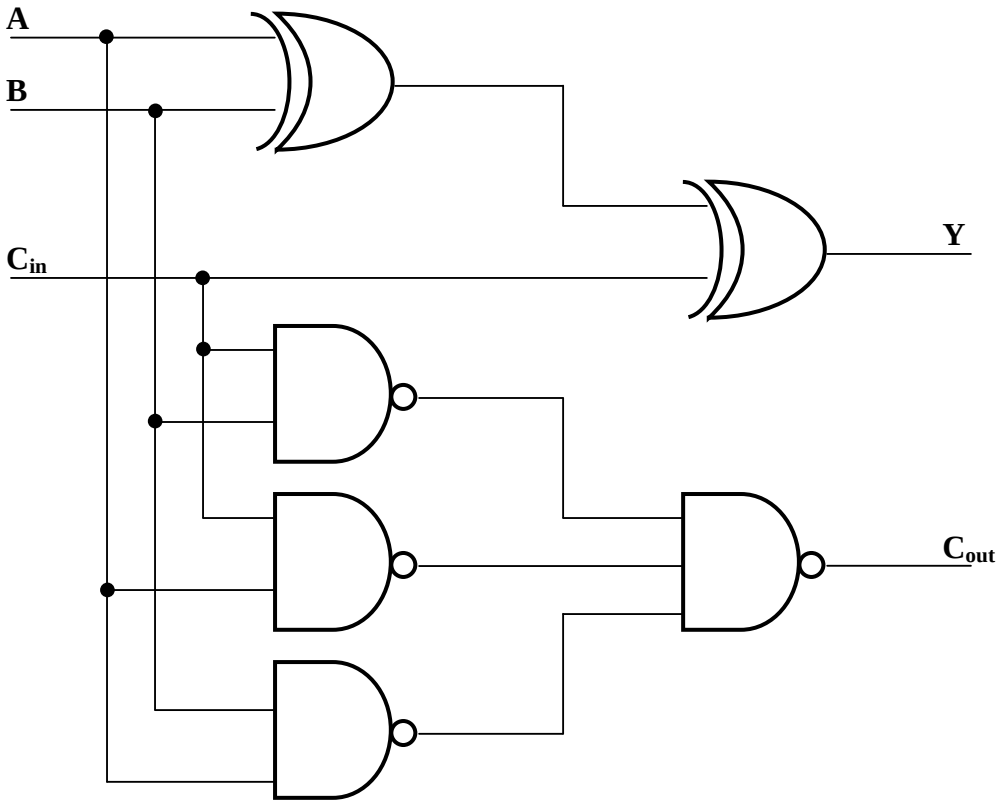


Figure 5.2. Full-adder reference circuit

Full-adder is a simple circuit; it can be easily implemented using discrete components, which are subject only to SETs. This simplicity is the main reason for using this circuit in the experiment at this early stage.

Crossover operators comparison

At first the crossover operators were compared. The simulation was done for classic evolution strategy, because it does not use any special improvements, like invalid chromosomes removal, included in other algorithms implemented in the application. Thus, the comparison made using this strategy is the most informative. The principle of operation of above-mentioned operators is described in section 3.2.4. Length of the chromosome was set to 448 what utilises 4 AND-matrix cells. This is enough because only 3 inputs

and 2 outputs are needed for full-adder. One not connected cell is provided for redundancy, to allow for additional connections. The initial segment length was set to 16 for both segment crossers, because this is the number of bits, which are needed to configure single line in the GAL with 4 cells used. Rest of the parameters were set as follows:

Strategy: classic with overlapping

Pop size: 100

Overlap size: 5

Crossover operator: 1 point, 2 point, uniform, constant segment (16), constant moving segment (16)

Cross probability: 0.3

Mutation probability: 0.01

Finish criterion: desired fitness value (16.0)

Fitness scaler: linear

RadFitness type: added

Chromosome length: 448

Figures 5.3. – 5.5. present the obtained results.

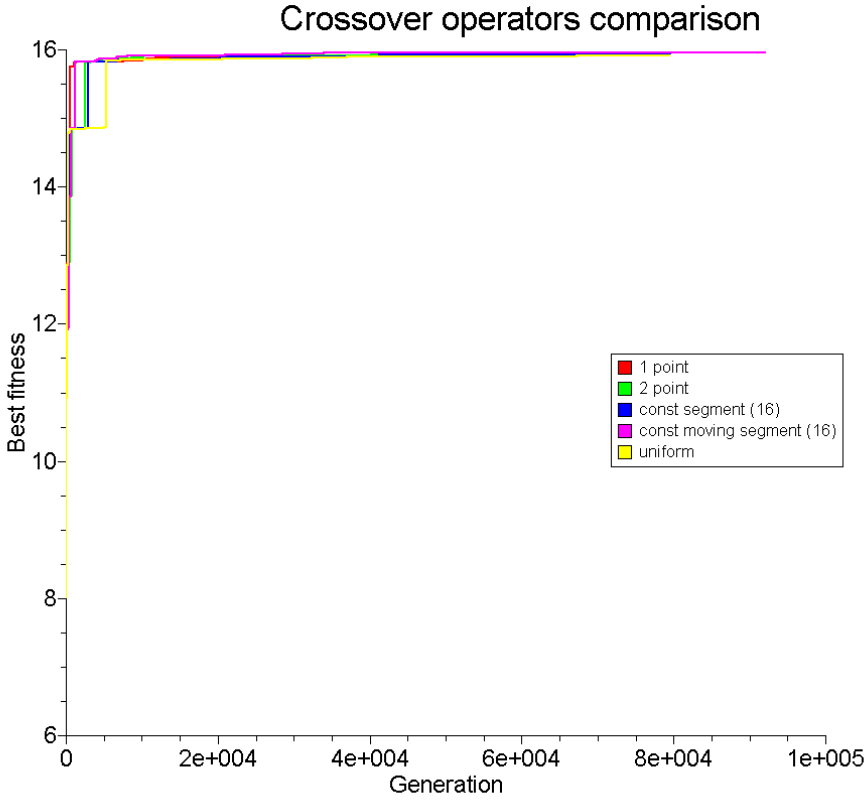


Figure 5.3. Crossover operators comparison

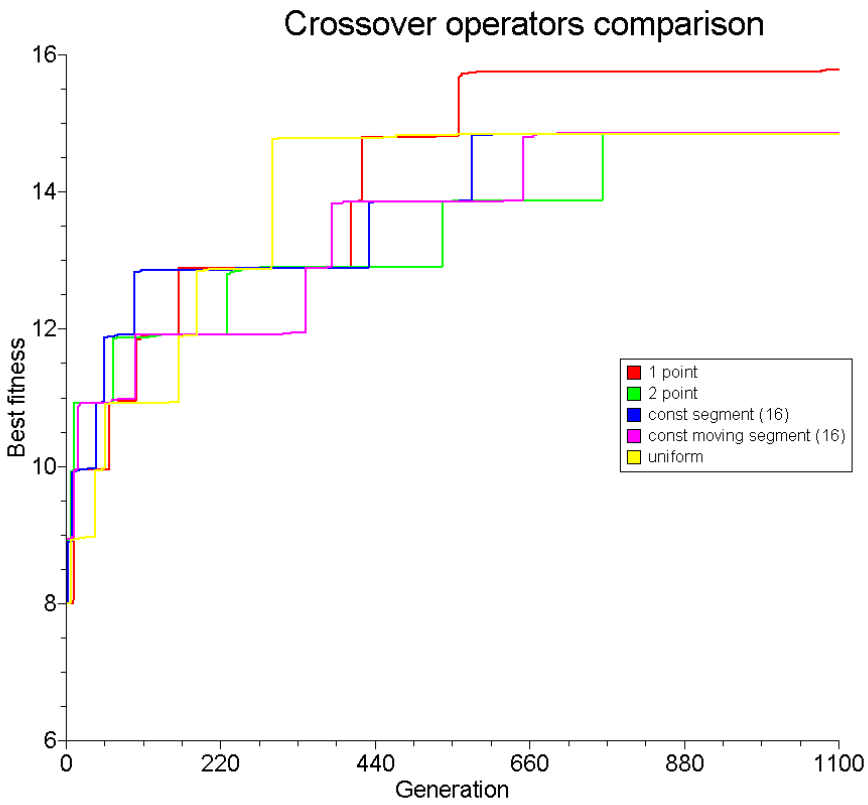


Figure 5.4. Crossover operators comparison – initial part zoomed

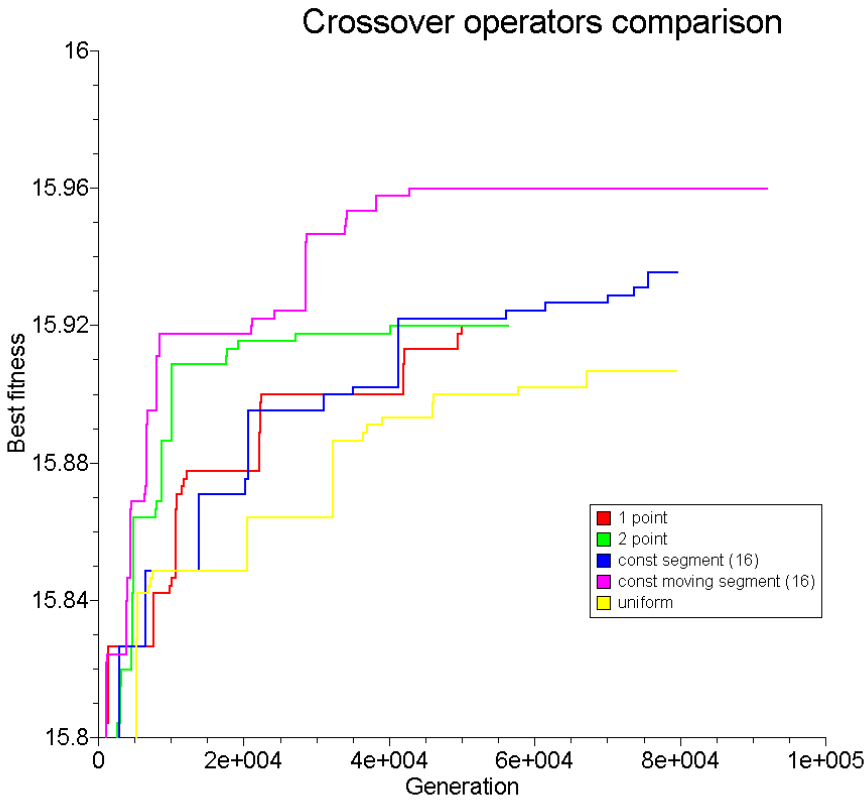


Figure 5.5. Crossover operators comparison – flat part zoomed

As presented in figure 5.4. initially 1 point crossover operator outperforms other. After 5000 of generations operator exchanging constant length segment with movement performs better than other ones. However, after 40000 of generations algorithm appears to be trapped in sub optimal point, because curve saturates at value 15.96. Finally after 90000 generations maximum fitness of the best individual stays at value 15.96, which corresponds to the configuration unsafe on 18 bits. Since it is "added" fitness function, the designed circuit is not guaranteed to perform desired operation with non-faulty configuration. Populations employing other crossover operators do not reach this value even after 80000 of generations.

Fitness scalers comparison

This simulation was performed similarly as the previous one, but the following parameters were used:

Strategy: classic with overlapping

Pop size: 100

Overlap size: 5

Crossover operator: uniform

Cross probability: 0.3

Mutation probability: 0.01

Finish criterion: desired fitness value (16.0)

Fitness scaler: linear, square, cubic

RadFitness type: added

Chromosome length: 448

Figures 5.6. – 5.8. present the results obtained.

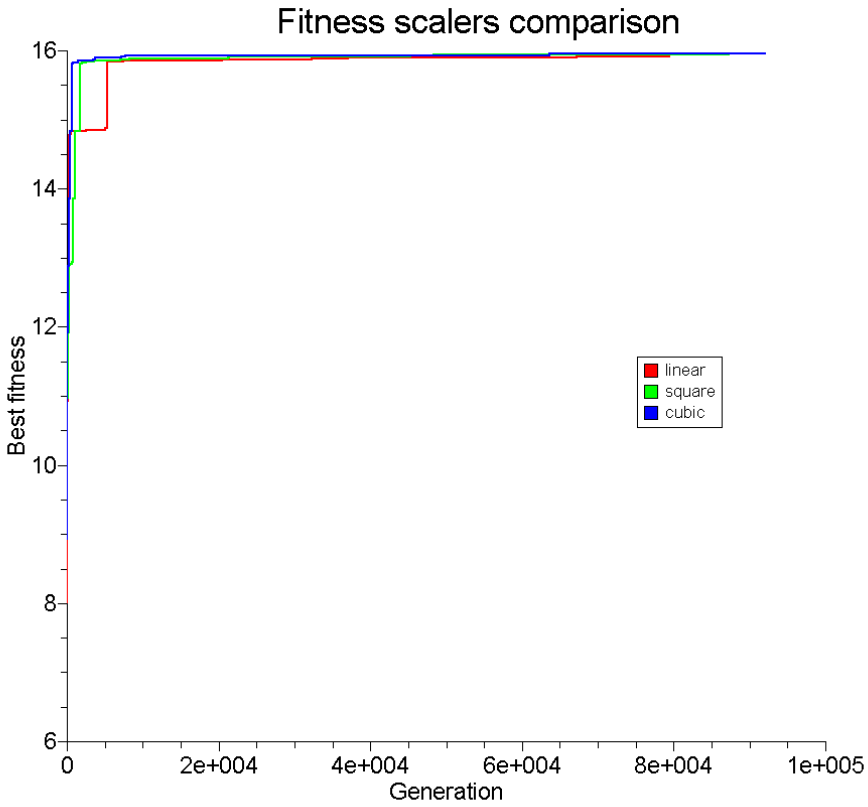


Figure 5.6. Fitness scalers comparison

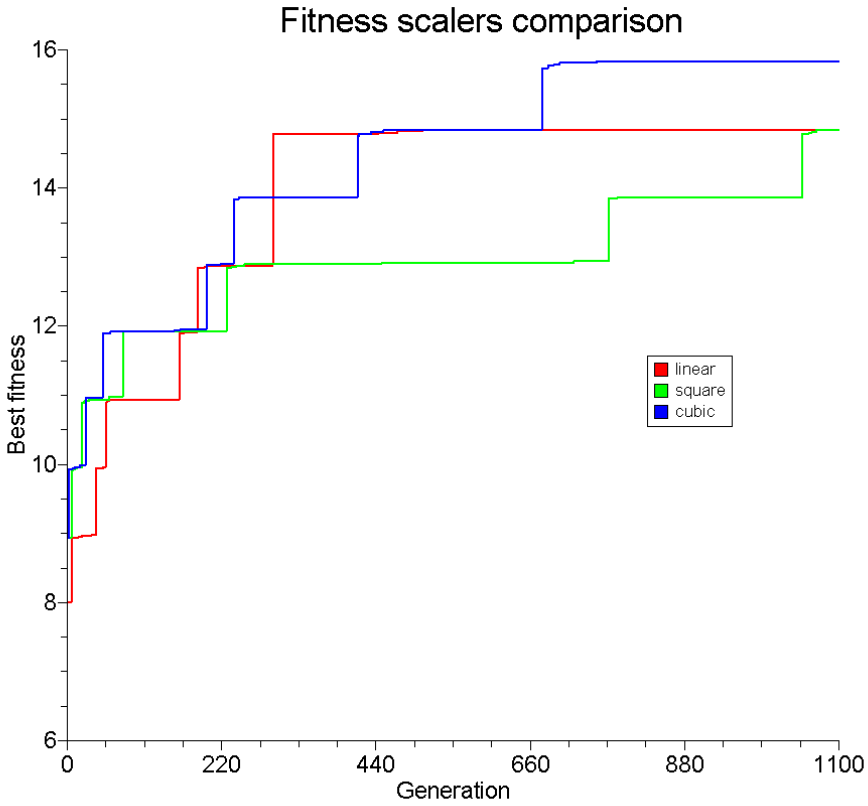


Figure 5.7. Fitness scalers comparison – initial part zoomed

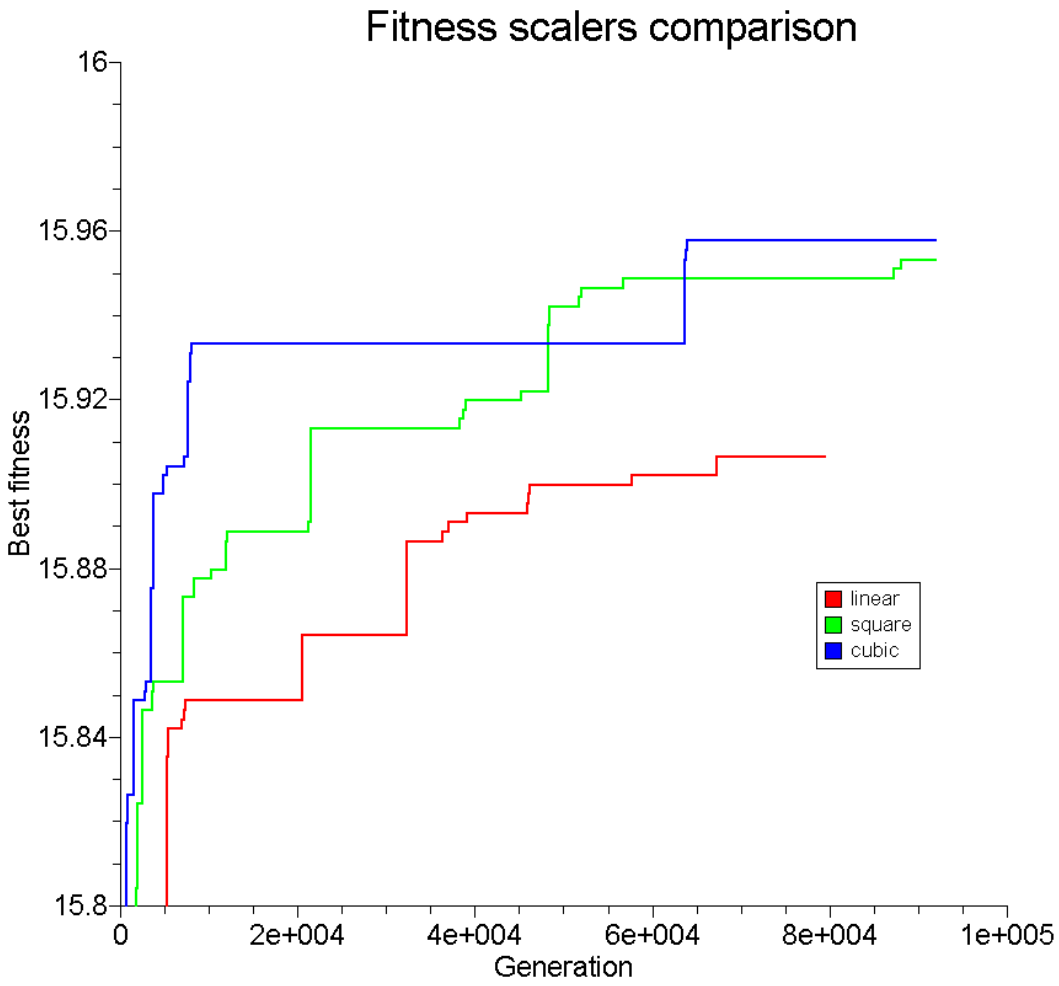


Figure 5.8. Fitness scalers comparison – flat part zoomed

As presented in figure 5.7. initially none of scalers outperforms other. After 700 generations cubic scaler starts to perform better than other ones. As in previous case, after 65000 generations curve saturates at value 15.96. Finally after 90000 generations maximum fitness of the best individual stays at value 15.9577, which corresponds to the configuration unsafe on 19 bits. Square scaler keeps close to the cubic scaler and even former outperforms the latter from 50000 to 60000 generations. Linear scaler performs much worse. It seems, that bigger selection pressure is advisable.

Mutation probability

This simulation was performed similarly as the previous one, but the following parameters were used:

Strategy: classic with overlapping

Pop size: 100

Overlap size: 5

Crossover operator: constant moving segment (16)

Cross probability: 0.3

Mutation probability: 0.001, 0.003, 0.01, 0.03

Finish criterion: desired fitness value (15.96)

Fitness scaler: cubic

RadFitness type: added

Chromosome length: 448

Figures 5.9. – 5.11. show the results obtained.

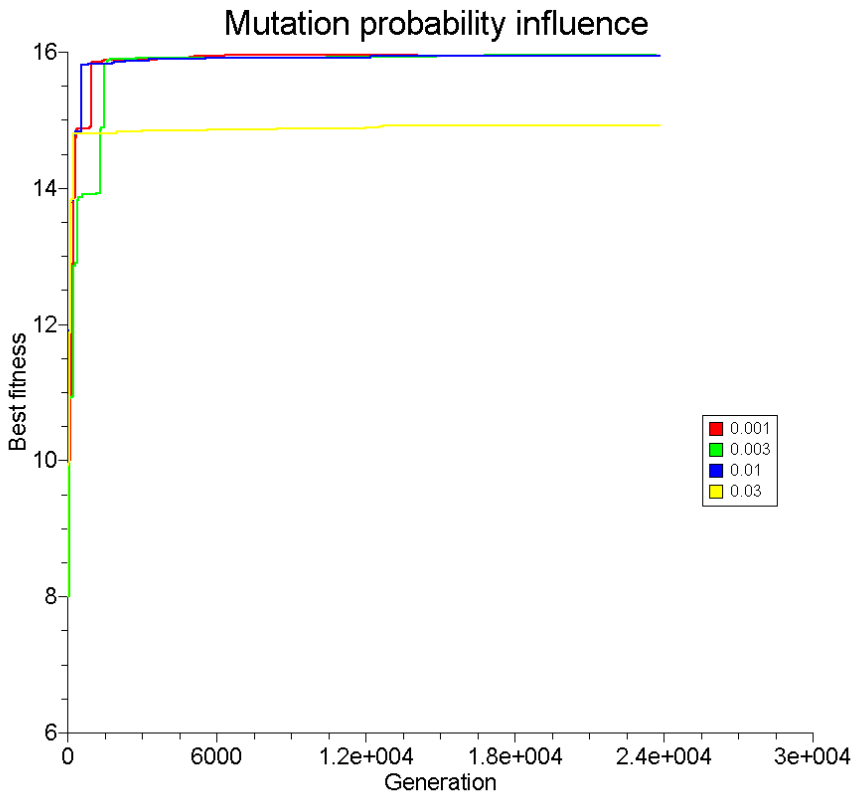


Figure 5.9. Mutation probability influence

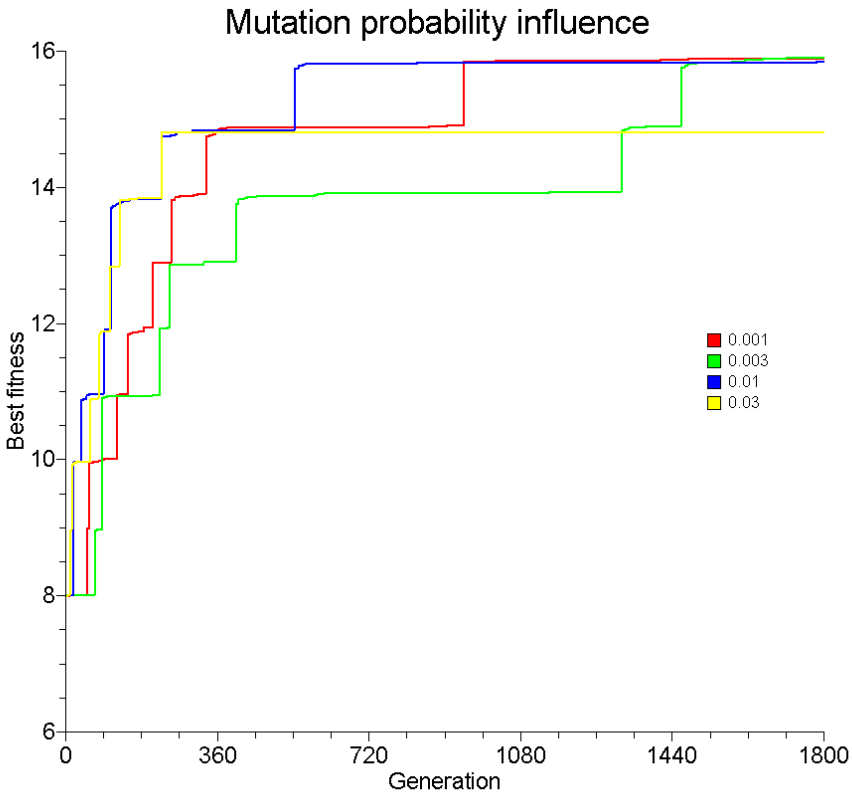


Figure 5.10. Mutation probability influence – initial part zoomed

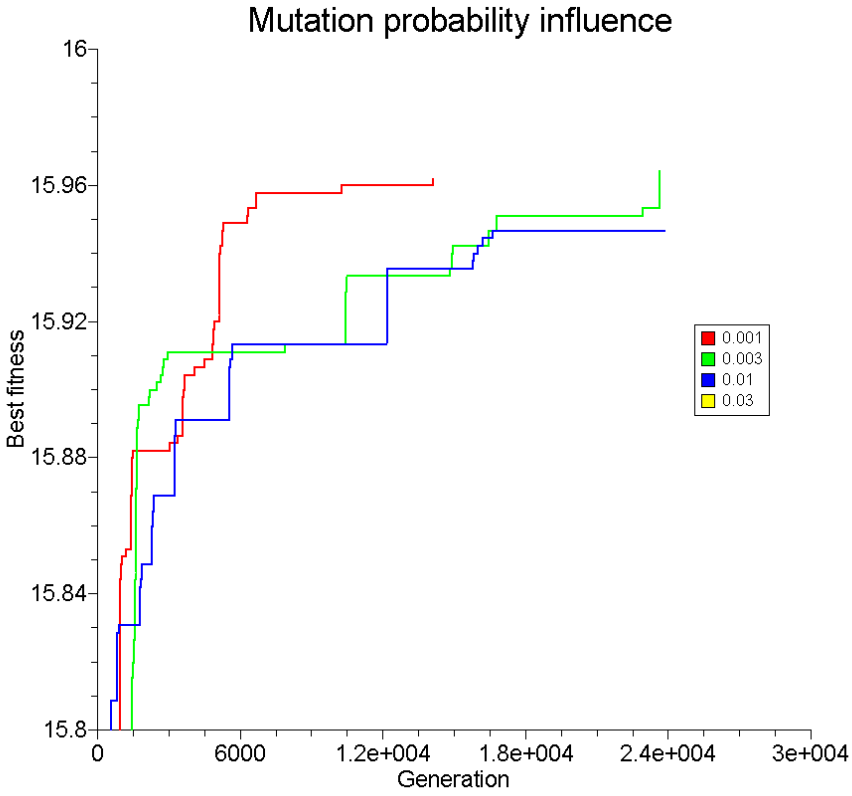


Figure 5.11. Mutation probability influence – flat part zoomed

As it can be seen in figure 5.9. and 5.10. when the probability of the mutation is large (0.03 in this case), initial steepness of the fitness function curve is large, but in later stage of simulation it damages too many good schemata and keeps the algorithm far from the maximum fitness value. In intermediate stage of simulation, no leading curve can be distinguished. In the figure 5.11. it is visible, that curve for mutation probability 0.001 reaches the fitness value 15.96 faster than other two (for probability 0.003 and 0.01). It seems that in this simulation, the smaller mutation probability, the better. However, one thing has to be kept in mind, the mutation is needed to get the algorithm out of the "evolution trap" – the local extremum of the function, so it cannot be set to 0.0 . Mutation influences the schemata in the population; it changes all schemata, in which the mutated bit is the relevant bit. Constant moving segment crossover operator also changes the schemata considerably. It is possible that by moving the segment from one place to another it provides enough schemata change, what enables very small mutation. In order to verify this hypothesis, next simulation was performed, but with uniform crossover operator, which also alters the schemata considerably, but does not move the fragments, genes stay at their position. The following parameters were used:

Strategy: classic with overlapping
Pop size: 100
Overlap size: 5
Crossover operator: uniform
Cross probability: 0.3
Mutation probability: 0.001, 0.003, 0.01, 0.03
Finish criterion: desired fitness value (15.96)
Fitness scaler: cubic
RadFitness type: added
Chromosome length: 448

Figures 5.12. – 5.14. depict the results obtained.

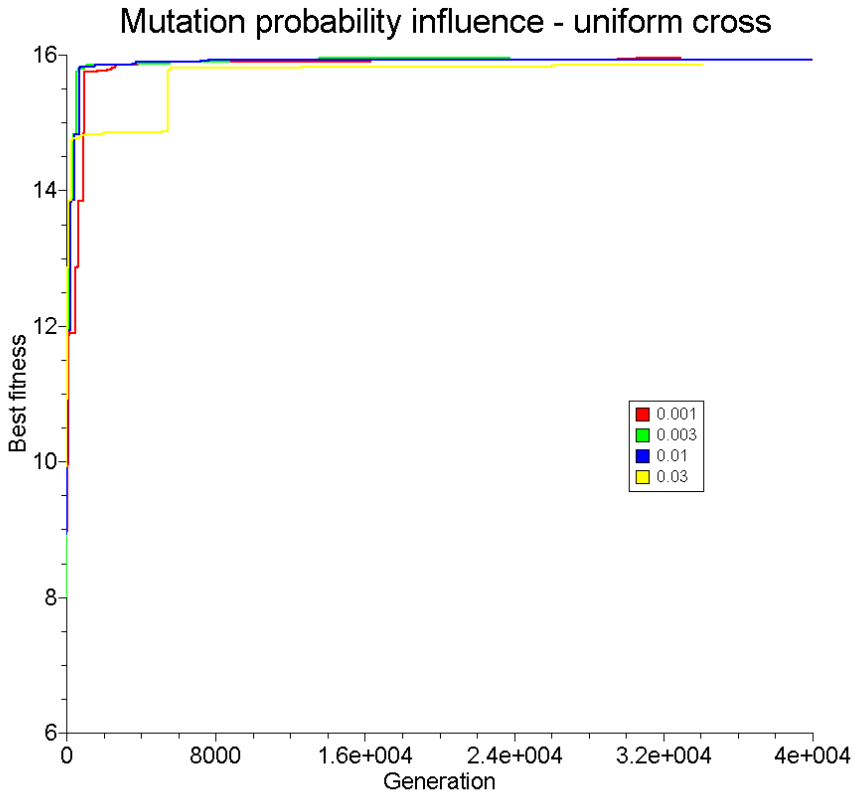


Figure 5.12. Mutation probability influence for uniform crossover operator

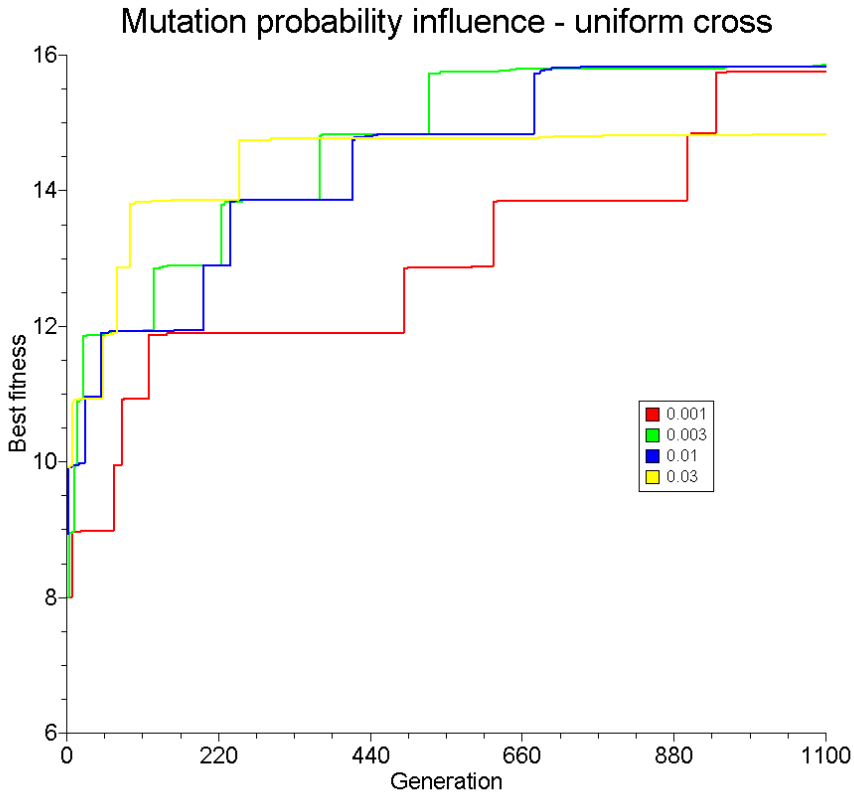


Figure 5.13. Mutation probability influence for uniform crosser – initial part zoomed

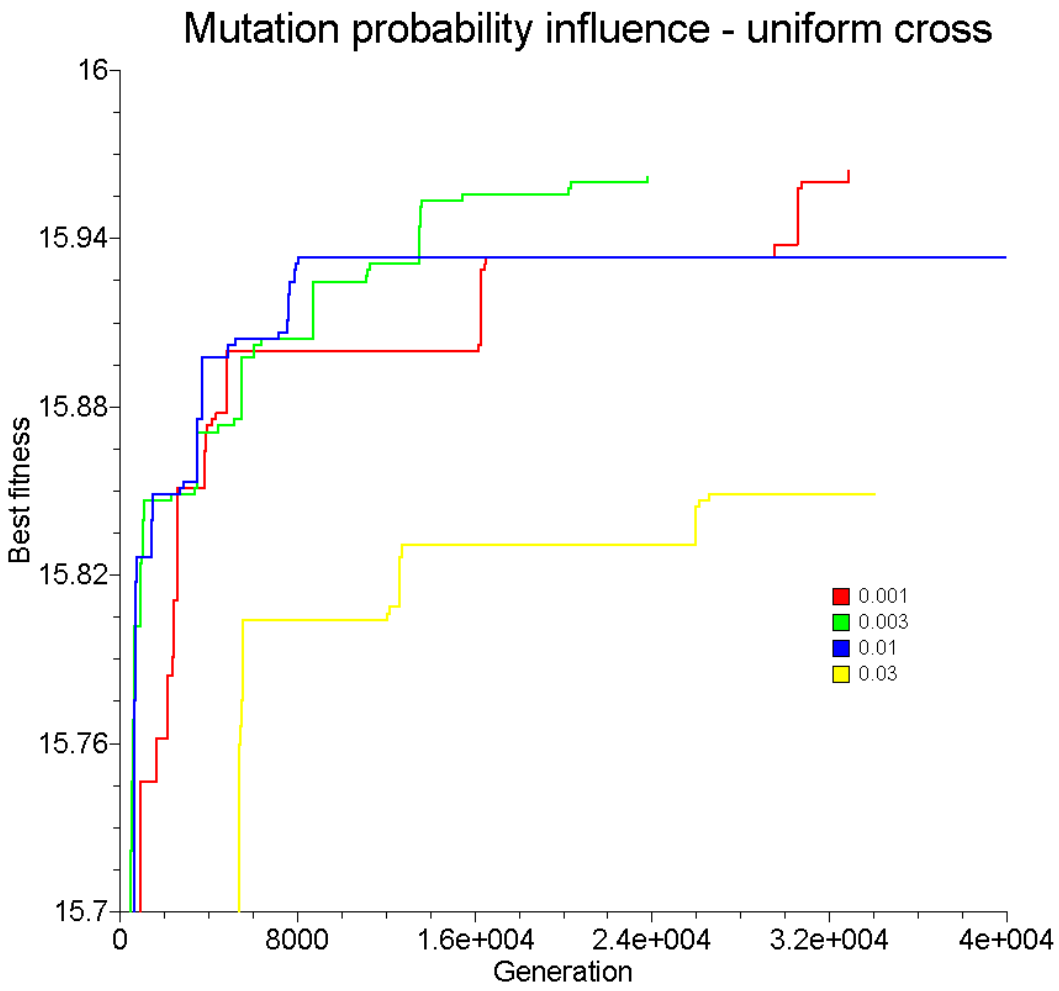


Figure 5.14. Mutation probability influence for uniform crosser – flat part zoomed

As depicted in the figure 5.12. there is no mutation probability that makes the algorithm to perform much considerably worse than other. Figure 5.13. shows, that initially large mutation probability speeds up the algorithm, and small slows it down. However, at later stage of simulation (figure 5.14.), as in previous case, large mutation probability (0.03) prevents algorithm from reaching the desired fitness. The hypothesis, that this crossover operator has smaller influence on schemata and therefore more mutation is needed seems to be true. In the case of the uniform crossover operator, the mutation probability 0.003 enables the algorithm to attain desired fitness in shortest time. This also proves, that determination of the parameters of the genetic algorithm one by one, in isolation, does not provide the best combination, because parameters are correlated.

Length of the segment

Crossover operators employing the idea of segment exchange are often used in the simulation, especially operator exchanging segment with movement. The length of the segment is also a parameter for the algorithm; therefore its influence on the performance was measured. This simulation was performed similarly as the previous one, but the following parameters were used:

Strategy: classic with overlapping

Pop size: 100

Overlap size: 5

Crossover operator: constant moving segment (4, 8, 16, 24)

Cross probability: 0.3

Mutation probability: 0.003

Finish criterion: maximum generation number (30000)

Fitness scaler: cubic

RadFitness type: added

Chromosome length: 448

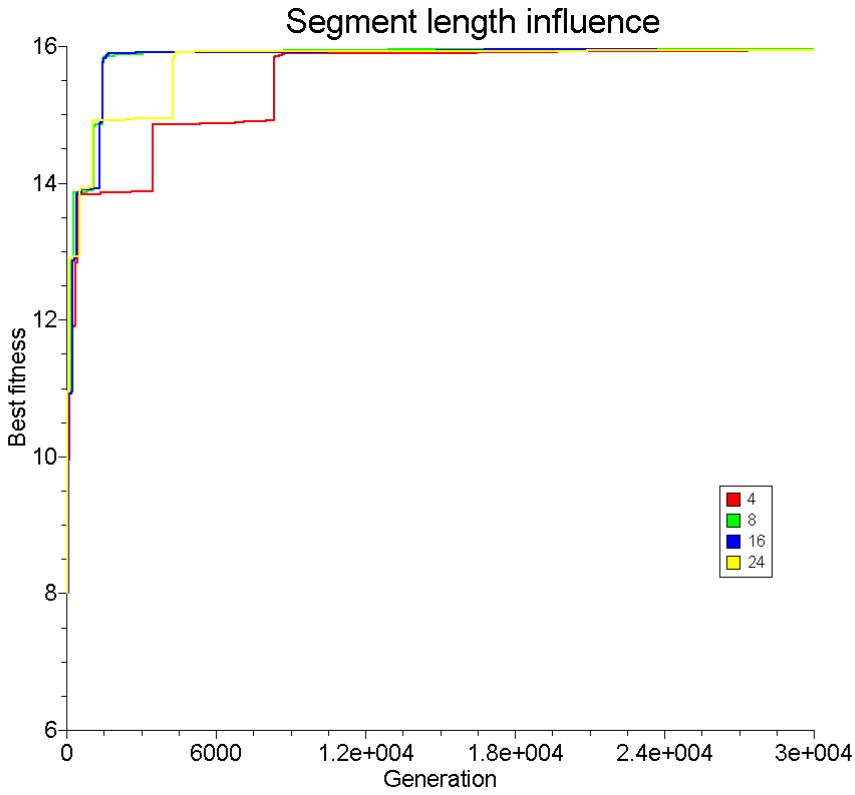


Figure 5.15. Length of the exchanged segment influence

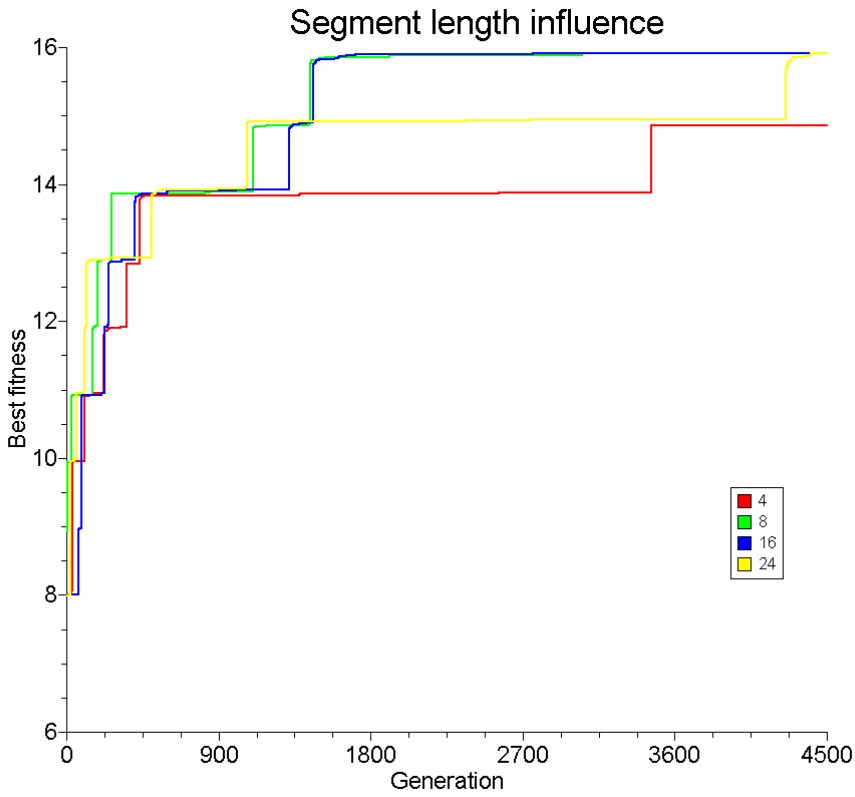


Figure 5.16. Length of the exchanged segment influence – initial part zoomed

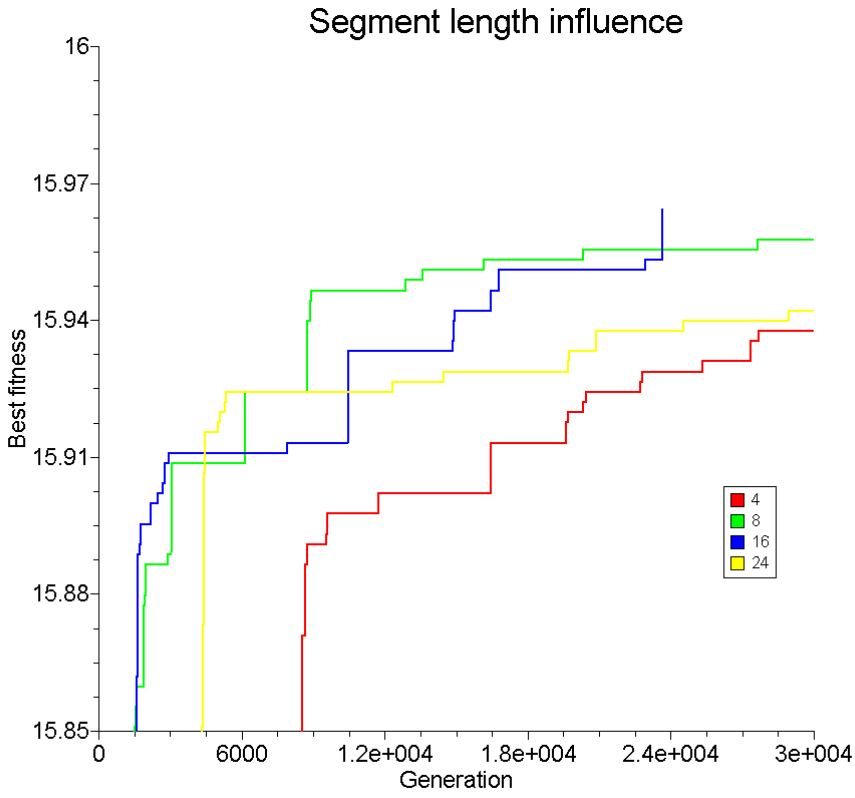


Figure 5.17. Length of the exchanged segment influence – flat part zoomed

Figure 5.15. reveals, that the difference between crossover operators employing different exchanged segment length is not as obvious, as in previous simulations. At the initial stage of simulation, results for the lengths 8 and 16 are better than other and very similar, while for 4 and 24 are worse then former ones. Figure 5.17. depicts zoomed flat part of the graph. Operator which exchanges segment of 4 bits performs worse than others, one exchanging 24 bits is a bit better at the beginning, but finally reaches similar fitness value. The operator with segment length of 16 bits attains the desired fitness in shortest time. However, operator using 8-bit segment is the best in ca. 50% of the time (from 8000-th generation to 24000-th generation) and should also be taken into account as an optimal setting.

Algorithms comparison (final simulation)

Since now, relatively short simulations were done, because these were trials of different parameter settings. In this simulations all three algorithms are used with parameters that seem the most appropriate from the previous experiments:

"Overlap"

Strategy: classic with overlapping

Pop size: 100

Overlap size: 5

Crossover operator: constant moving segment (16)

Cross probability: 0.3

Mutation probability: 0.003

Finish criterion: desired fitness (16.0)

Fitness scaler: cubic

RadFitness type: log2

Chromosome length: 448

"Overlap strong"

Strategy: classic with overlapping and strong start

Pop size: 100

Overlap size: 5

Crossover operator: constant moving segment (16)

Cross probability: 0.3

Mutation probability: 0.003

Strong ratio: 0.2

Strong treshold: 1.0

Finish criterion: desired fitness (16.0)

Fitness scaler: cubic

RadFitness type: log2

Chromosome length: 448

"Greedy"

Strategy: greedy

Pop size: 100

Overlap size: 5

Crossover operator: constant moving segment (16)

Cross probability: 1.0

Mutation expected: 1 (mutation probability: 1/448 = 0.002232)

Required Strong: 20

Maximum population size: 100

Minimum required short fit: 2

Minimum required radiation fit: 5

Finish criterion: desired fitness (16.0)

Fitness scaler: linear (strategy does not use selection, so scaler type is irrelevant)

RadFitness type: normal (strategy eliminates individuals which do not perform desired functions with non-altered configuration, therefore penalty function is not needed)

Chromosome length: 448

In this simulation problem not present in the previous ones appears. Since now, the generation number was used as a time reference for different populations. This could be done, because all of them used the same population size and the same algorithm, what implies the same number of time spent on each generation. But here generation number is no longer an appropriate time reference, because algorithms require different number of computing time for each generation. For example greedy algorithm with population size of 100 individuals may need couple of thousands of radiation tolerance evaluations per generation, while classic algorithm with the same population size requires always 100 radiation tolerance evaluations. The problem is solved using *logConverter* program included in tools subdirectory of the system. This program uses number of short evaluations and radiation tolerance evaluations saved in the log file to compute the simulation time instant for every generation. Further information on the files structure, tools usage and functioning can be found in the appendix A. LogConverter uses RadiationEvalsPerTimeUnit constant defined in its source code. This constant defines how many radiation tolerance evaluations can be done in the time unit used as a reference. In this simulation this constant was set to 180, because on average each of the computers

used in the system was able to carry out 3 radiation tolerance evaluations per second and 180 per minute. Time needed for short evaluation is calculated using the above mentioned constant and the chromosome length.

The simulation was the longest of all carried out before. The system worked simultaneously on the three populations for ca. 20 days using 17 computers. The time spent on each population expressed in reference time units was: greedy (53401 minutes = 37.08 days), overlap (119867 minutes = 83.24 days), overlap strong (118455 minutes = 82.26 days) what in total gives almost 7 months (203 days)! The effective total power of the system was: 203/20 = 10.15. What gives effective utilisation of almost 60%. This result seems quite satisfactory, especially when we take into account that some of the time was wasted because of network failures and power cuts and that this performance estimation method is not exact.

Figures 5.18. – 5.20. depict the obtained results.

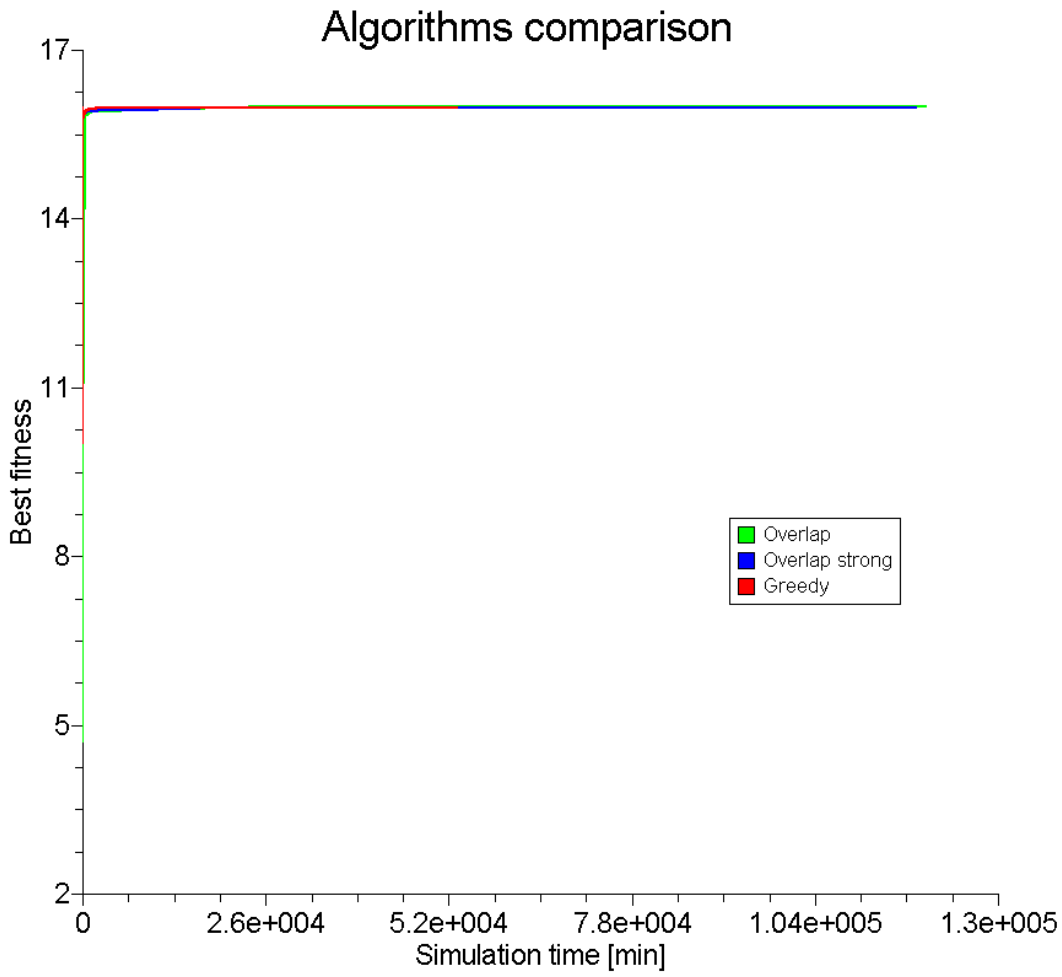


Figure 5.18. Algorithms comparison

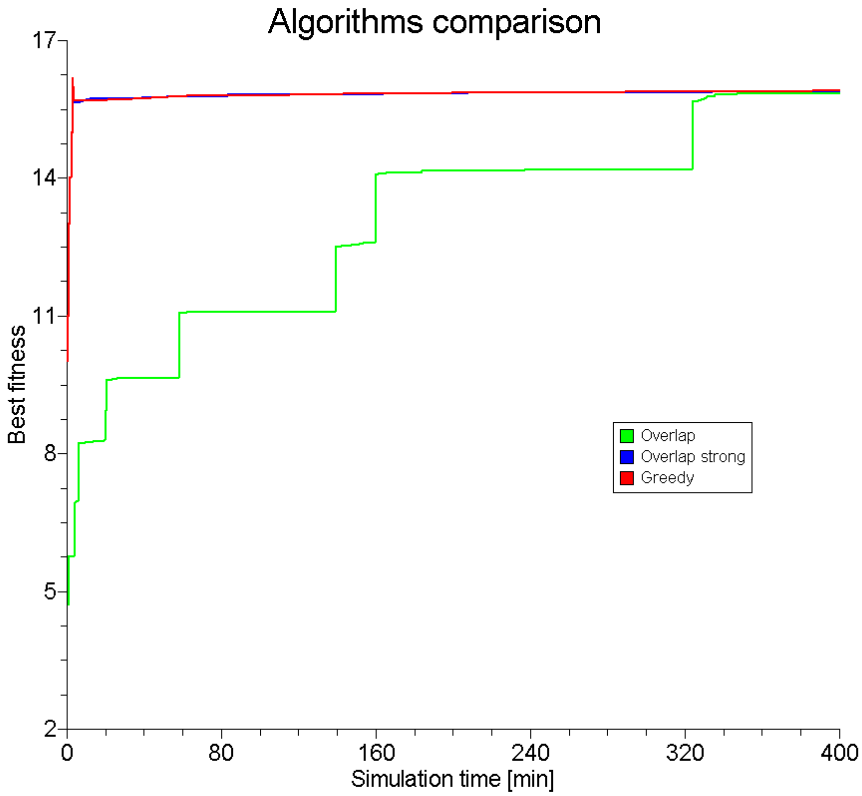


Figure 5.19. Algorithms comparison – initial part zoomed

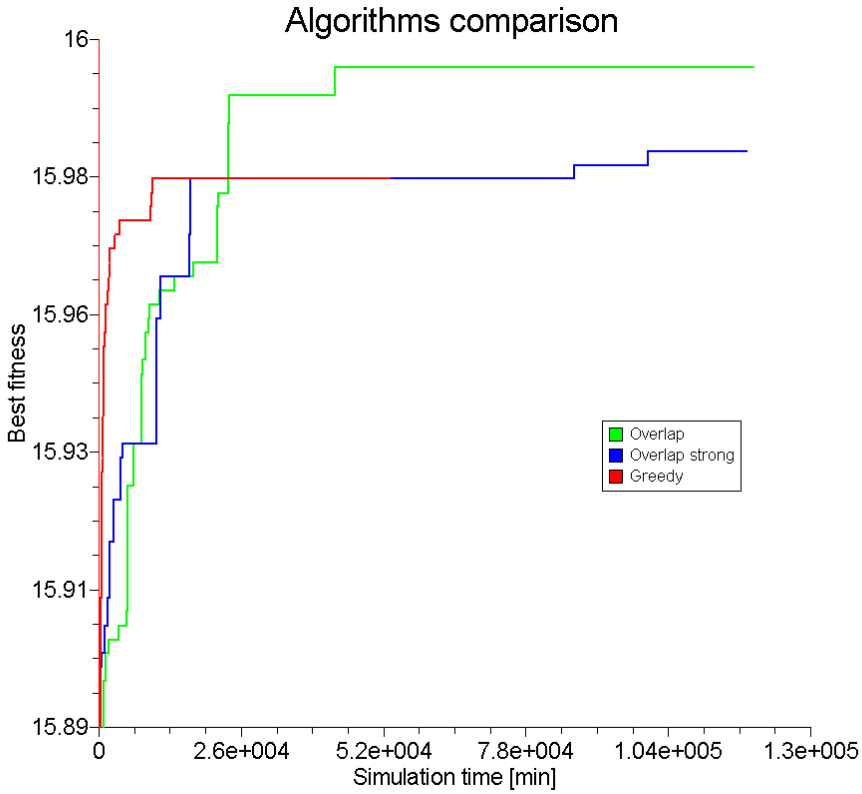


Figure 5.20. Algorithms comparison –flat part zoomed

Initially classic algorithm with overlapping ("overlap") performs worse than the other two. It becomes the best after ca. 26000 minutes of simulation. Steep initial part of the greedy algorithm curve reaching the value 16 is the period, where there are not enough strong chromosomes in the population and only short evaluation is done. Greedy and classic algorithm with overlapping and strong start ("overlap strong") quite fast attain high fitness value 15.9777 (10 wrong outputs of the device). The former after 10000 minutes and the latter after 16500 minutes. Both get stuck at that value for the very long time. Greedy simulation ends at 53401 minutes with no further improvement, the overlap strong algorithm manages to improve a bit at 87000 minutes and 100200 minutes and finally develops a chromosome with fitness 15.9821 (8 wrong outputs of the device). The overlap algorithm starts from the lower fitness values because it does not use strong start function. Moreover the log2 penalty function is used. After 26000 minutes it becomes unquestionable leader. After 43200 minutes its best chromosome fitness attains value of 15.9955 (2 wrong device outputs) and stays at that value till the end of the simulation. It appears that the algorithms (greedy and overlap strong) using strong start function perform better in the initial stage of simulation, because they start from the relatively good point in the promising area of the search space. However, at later stage they get stuck probably because the area of search is in fact suboptimal. The algorithm that starts the search from the random point and random area (overlap algorithm) gradually improves the solution and finally finds much better configuration than the other algorithms.

The conclusion that can be drawn from the obtained results is that when speed is the main concern, the greedy or overlap strong algorithm is a good choice, when quality of the solution is critical, the overlap algorithm should be used.

Summary

To sum up the simulation results analysis, the improvement obtained by the system should be estimated. Fitness value does not reveal the whole truth about the chromosome, because it indicates only the number of outputs that match desired outputs. It does not say how many bits of the configuration are sensitive to change (produce bad outputs when changed). Probably this should be incorporated somehow into the fitness function to promote individuals with the smallest number of sensitive configuration bits.

Full information on the chromosome is provided by the *chromInfo* tool, which is described in appendix A. Using this tool, the number of sensitive bits of the configuration can be checked.

The non-altered configuration of the device realising desired function is not sensitive on all of its 448 bits. The strong start function can be used to generate random configurations performing the full-adder function. The average of the number of sensitive bits in the configurations created in several function runs can be used as a reference for improvement estimation. The table 5.2. summarises the results.

Table 5.2. Results of 10 runs of strong start function

Run	Fitness	Wrong outputs	Sensitive bits
1	15.6451	159	102
2	15.7388	117	82
3	15.6875	140	88
4	15.5848	186	98
5	15.7054	132	95
6	15.5692	193	99
7	15.6987	135	85
8	15.6071	176	93
9	15.7411	116	82
10	15.7388	117	95
		Average:	**92.5**

On average non-optimised configurations performing the full-adder function have 92.5 sensitive bits. So, assuming that only 448 bits of configuration affect the device functionality (the configuration bits and the unused part of AND matrix are not taken into account) the probability that the configuration will be incorrect after a single bit change is 92.5/448 = 0.206. The final results obtained in the last simulation are as presented in the table 5.3.

Table 5.3. Final results summary

Algorithm	**Wrong outputs**	**Sensitive bits**	**Probability of incorrect configuration**	**Improvement**
"greedy"	10	10	0.022	9.4 times
"overlap strong"	8	7	0.016	12.9 times
"overlap"	2	2	0.004	**51.5 times**

The decrease in probability of incorrect configuration after a single bit change obtained using "overlap" algorithm is very significant (over 50 times!) and improves the radiation tolerance considerably. The best chromosome configuration obtained during the simulations is presented in the appendix C.

6. Summary and Conclusions

The main goals stated in the chapter 1 have been achieved. The distributed system has effectively enabled access to the computing power bigger than available without such system with relatively high utilisation of the resources (certainly above 60%). The fault-tolerance of the programmable circuit configuration has been increased over 50 times! However, these results need to be presented together with the assumptions and simplifications made in the thesis to show the full picture.

In the simulations, the PAL-like programmable device was used with simple reference circuit (full-adder). The PAL architecture is very simple, when compared to other contemporary programmable devices. Moreover, only 4 AND-matrix cells were used and taken into account. In real devices AND-matrix is bigger and bit change in the other part of matrix (not containing the simulated 448 bits of configuration) may result in device functional failure. In order to get real radiation-tolerance, the whole programmable area should be simulated, what would increase the simulation time, but possibly higher resources redundancy could increase the effectiveness. There are also usually some configuration bits in each device, which when changed affect the device functioning considerably (for example, combinatorial outputs may become registered). However, it is impossible to improve fault-tolerance in the configuration bits.

Nowadays, SPLDs usually hold the configuration in EEPROM memory, which is not a subject to SEUs. Therefore in real PAL device, it makes no sense to implement the fault-tolerant configuration, because such faults do not occur. The PAL device was used here because of its simplicity and well-known architecture, because the aim of the simulations was to check the effectiveness of the genetic approach towards the reliable circuit design. More complicated devices like CPLDs or FPGAs could be used with more complicated reference circuits, but simulation time would be considerably longer.

Tested algorithms proved to be quite effective when slight fault-tolerance improvement is needed. During the simulations the 10 times decrease of the incorrect configuration probability after a single bit change was attained relatively fast. When higher order of improvement is needed, they were not so effective. Totally fault-tolerant configuration has not been found. Probably the use of more advanced genetic algorithms (like aging

algorithm presented in [17]) or fitness function incorporating the number of sensitive bits could improve the effectiveness and solution quality.

The low effectiveness of the approach is the main and limiting disadvantage. Moreover, only single fault tolerance was assumed in the project, what may be insufficient in real applications. However, when other radiation mitigation techniques cannot be applied, the above presented technique may be helpful, especially for the fault-tolerance optimisation of some simple but critical parts of bigger systems.

References

[1] Contemporary Physics Education Project, "Standard model of fundamental particles and interactions", 2000, **http://cpepweb.org/**

[2] H. Spieler, „Introduction to Radiation-Resistant Semiconductor Devices and Circuits"

[3] F. Faccio, „Radiation effects in electronic devices and circuits", ELEC-2002 **http://humanresources.web.cern.ch/humanresources/external/trainig/tech/special/ELEC2002.asp**

[4] J.C. Pickel and J.T. Blandford, IEEE TNS, NS-29, 2049 (1982)

[5] R. Baumann, "Radecs Short Course", 2001

[6] W.Wang, „RC hardened FPGA configuration SRAM cell design", IEEE Electronics letters april 2004

[7] H. Chaohui, L. Yonghong, G. Bin, C. Xiaohua, Y. Hailiang, „Mechanisms of Radiation Effects in Floating Gate ROMs"

[8] D. A. Adams, D. Mavis, J. R. Murray, M. H. White, „SONOS Nonvolatile Semiconductor Memories for Space and Military Applications"

[9] P. P. Shirvani, N. R. Saxena, E. J. McCluskey, "Software-implemented EDAC Protection against SEUs" IEEE Transactions on reliability, vol.49, no. 3, september 2000

[10] Lattice Semiconductor Corp., "GAL16V8 datasheet", 2004

[11] Altera Corp., "MAX 3000A Programmable Logic Device Family Datasheet", 2003

[12] S.Brown, J. Rose, "Architecture of FPGAs and CPLDs: A Tutorial", Departament of Electrical and Computer Engineering, University of Toronto

[13] Xilinx Corp., "XC4000E and XC4000X Series Field Programmable Gate Arrays", 1999

[14] Xilinx Corp. , "Virtex-4 Family Overview", 2005

[15] Y. Chang, D.F. Wong, C. K. Wong, "Programmable Logic Devices"

[16] J. Arabas, "Wykłady z algorytmów ewolucyjnych", WNT Warszawa 2001

[17] Z. Michalewicz, "Algorytmy genetyczne + struktury danych = programy ewolucyjne", WNT Warszawa 1996

[18] D. Whitley, "A Genetic Algorithm Tutorial"

[19] W. Williams, „Metaheuristic algorithms, Genetic Algorithms: A Tutorial – slides"

[20] Intel Corp., "Intel® 82802 Firmware Hub: Random Number Generator. Programmer's reference manual", 1999

[21] Computational Science Education Project, "Random Number Generators", 1995, **http://csep1.phy.ornl.gov/rn/rn.html**

[22] Makoto Matsumoto, "Mersenne Twister Home Page", **http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html**

[23] **SETI@Home** project web page. **http://setiathome.ssl.berkeley.edu/**

[24] D. Marshall, "Remote Procedure Calls (RPC)", 1999

[25] DEC, “Programming with ONC RPC”, 1996

[26] Carl-Frederik Sørensen, “A Comparison of Distributed Object Technologies”, DIF8901 Object-Oriented Systems

[27] M. Orlikowski, “Rozproszone systemy obiektowe - slajdy”, DMCS TUL 2002

[28] P. E. Chung, Y. Huang, S. Yajnik, D. Liang, J. C. Shih, C Wang, Y. Wang, „DCOM and CORBA Side by Side, Step by Step, and Layer by Layer“, september 1997

[29] M. Horstman, M. Kirtland, „DCOM Architecutre“, MSDN Library 1997

[30] M. Henning, S. Vinosky, “Advanced CORBA programming with C++”, Addison Wesley 1999

[31] Sun Microsystems Inc., “Java Remote Method Invocation”, 2003

A. Appendix A
Distributed System User's Guide

A.1. System Requirements
The system requires omniORB 4.0.3 and ZThread library 2.3.2. installed in your system. The system itself does not have any additional requirements above the ones imposed by the omniORB and ZThread library.

A.2. System Installation
Unpack the archive in your destination directory using the command:

```
tar xzf genetic.tar.gz
```

system expands into genetic directory. Enter the directory and use commands:

```
make clean
make
```

Program should compile without errors. When errors occur, check the installation of required packets and availability of all include files needed. If the errors are connected with the ZThread library, try using modified include files from ZThread_patch.tar.gz.

Executable files are located in the *build* directory. Build directory contains also *tools* directory where system tools are located.

A.3. Using the System
OmniORB nameservice is required to be running before any of the system elements is used (except the system tools). Nameservice can be started using *omniNames* file. For omniORB configuration details please refer to the omniORB documentation.

A.3.1. Genetic Manager
At first the Genetic_Manager should be started. The program communicates any errors caused by improper installation or configuration of the CORBA environment. Particularly, it requires ORBInitRef NameService CORBA variable to be properly set as it gives a reference to the nameservice. If it is not set in the omniORB configuration file, it can be given as a parameter for the executable file.

```
./Genetic_Manager –ORBInitRef NameService=corbaname::machine.name:port_no
```

Genetic_Manager has a parameter –resume, which can be used when the system state should be resumed from the snapshot file. Snapshot files are described in the section A.3.

A.3.2. Genetic Node
When the Genetic_Manager is running and registered in the nameservice, the Genetic_Node program can be started at client computers. It also requires nameservice reference, which can be supplied as described in the previous section. When executed without arguments, the program asks for the node name. Node name does not have

to be unique, there can be many nodes in the system with the same name. However, it is good to give some unique names to the nodes. When executed with parameters, program uses first parameters as a node name and tries to connect to the system manager.

A.3.3. Genetic Panel

When the Genetic_Manager is running and registered in the nameservice, the Genetic_Panel program can be started at any computer, which can access the machine running nameservice and machine running the genetic manager program. Genetic panel is a Command Line Interface (CLI) for the distributed system. There are following commands available:

listnodes
the command displays a list of names of all nodes connected to the system manager

listpops
the command displays a list of names and start times of all populations being simulated at the moment of command issue.

help [command]
the command displays help text for the command given as an argument or displays a list of commands

newpop pop_name chromosome_length [snapshot_interval]
the command adds a new population to the list of simulated populations. After this command is issued, the program asks for the parameters of the population, used algorithm, etc. The terminology used in the parameter description is consistent with the one presented in chapters 3 and 4. The last optional command parameter *snapshot_interval* is used by the genetic manager. The snapshot files for the population are generated with the interval stated here. If not given, the interval is assumed to be *__SNAPSHOT_INTERVAL* defined in the */common/defaults.h*. Snapshot files and system resuming is described in section A.3.

delpop pop_name
the command removes pop_name from the simulated populations list and from the memory, log and snapshot files are left intact.

A.4. Log Files, Snapshot Files and System Resuming

Every time the population is added or removed using newpop or delpop command or the population evaluation ends because finish criterion is met, the system snapshot file is updated. Its name is set as a define *__SNAPSHOT_FILENAME* in the global definitions file */common/defaults.h* described in section A.4. System snapshot file contains data concerning the populations being simulated: population snapshot filename, population name, algorithm type, crossover operator type, fitness scaler, architecture file, reference file, etc. Part of the file is presented below:

```
2
new_overlap_Mon_Aug_29_12:44:30_2005.snp new_overlap 4 16 448 1 100 5 0.3
0.003 17 (...)
new_greedy_Mon_Aug_29_12:44:30_2005.snp new_greedy 4 16 448 3 100 5 1 1
20 100 2 5 17 (...)
```

(…) means that some parts of the file are omitted

The chromosome data is contained in the snapshot files that are created separately for every population. An example of snapshot file name is: *new_overlap_Mon_Aug_29_12:44:30_2005.snp*. The filename is created using population name and the population start date. Each population snapshot file is updated at the *snapshot_interval* generations.
Every population holds also its log file, where the data on the best population individuals is stored at each evolution step. Log file also holds data describing population parameters in human- readable form. Log filename is created similarly to snapshot file, but with .log extension.
System can be resumed using –resume argument to the Genetic_Manager executable. Manager looks for the *__SNAPSHOT_FILENAME* system snapshot file in its directory and resumes the populations contained in that file by adding the populations to the simulated populations and loading their chromosome data from the population snapshot files.

A.5. System Configuration

System is configured using the */common/defaults.h* file, where most important system options can be changed. The file contents together with default values are presented below:

//System
#define __NUMBER_OF_LOCAL_NODES 1
number of computing nodes started at the system manager machine. In fact only settings 0 and 1 make sense. 0 is useful, when system manager machine should not be slowed down by genetic evaluation. 1 can be set on fast machine, which can contribute its power to the system and simultaneously manage the system efficiently.

#define __GARBAGE_COLLECTOR_INTERVAL 10
[s] interval at which garbage collector thread is woken up.

#define __SNAPSHOT_FILENAME "pops_snapshot.snp"
function of this define is described in section A.3.

//Population
#define __SNAPSHOT_INTERVAL 10
the default snapshot_interval

//nodeTask.h
#define __ALIVE_CHECK_TIMEOUT 3000
[ms] interval at which the node is pinged

#define __LAST_RADTIME 500000
(µs) time needed by the node to evaluate one chromosome under radiation (this is default time, used when first packet is send to the node, because real evaluation time is unknown at the beginning).

#define __LAST_SHORTTIME 50000
(µs) time needed by the node to evaluate one chromosome without radiation (this is default time, used when first packet is send to the node, because real evaluation time is unknown at the beginning).

#define __MAX_COMPTIME 1000000
(µs) maximum computation time allowed for the packet (needed to determine how many chromosomes the packet should contain)

#define __MAX_PACKET_SIZE 50000
maximal number of chromosomes in one packet

#define __MAX_SHORTQUEUE_PACKET 22400
maximal number of shorteval chromosomes put in one turn into queue

#define __MAX_RADQUEUE_PACKET 50
maximal number of radeval chromosomes put in one turn into queue

The nodeTask defines usually do not have to be changed. The only that should be adjusted sometimes is *__MAX_COMPTIME*. For example, when there is only one population with 100 chromosomes and the system consists of 10 computers able to evaluate 20 chromosomes per second, the 100 chromosomes will be consumed by 5 computers and rest will stay unused. In this case the *__MAX_COMPTIME* should be set smaller or equal than 0.5 second. However, it cannot be set too small, because the packets for evaluation with only one chromosome may contribute to network congestion.

//sysManager
#define __MANAGER_QUEUE_LENGTH 100
length of the main manager job queue

#define __IDLE_CONDITION_WAIT 1000
time interval at which manager checks if there is anything to do

Again these define should be left unchanged

//CORBAnode
#define __WATCHDOG_INTERVAL 10
(s) interval at which watchdog counter is incremented

#define __WATCHDOG_TIMEOUT 6
number of watchdog intervals that node is allowed to stay unused (none evaluations a re done). When the time is exceeded the CORBANode program exits.

The watchdog feature of the node and resume option of a system manager can be used to make the system start automatically after power or network cut. The following start script can be executed from the crontab of the machine running system manager to make it resume nameservice and system manager program. Script makes 5 attempts, after that exits. The home directory is assumed to be `/home/students/norastom/`. `The runomni and runmgr` are scripts which start Genetic_Manager and omniNames respectively.

```
#!/bin/sh

counter=0

while [ $counter -le 5 ]; do
if [ "$(ps aux | grep [o]mniNames| wc -l)" -eq 0 ]; then
    echo starting omniNames;
    if [ "$counter" -eq 5 ]; then
      echo exit;
      exit -1;
    elif [ -x "/home/students/norastom/runomni" ]; then
      echo "/home/students/norastom/runomni"
      /home/students/norastom/runomni
    fi
      sleep 5;
elif [  "$(ps aux | grep [G]enetic_Manager| wc -l)" -eq 0 ]; then
      echo loop;
      if [ "$counter" -eq 5 ]; then
            echo exit;
            exit -1;
      elif [ -x "/home/students/norastom/runmgr" ]; then
            echo "/home/students/norastom/runmgr"
            nice -n 20 /home/students/norastom/runmgr
      fi
            sleep 5;
fi
counter=$(($counter+1));
done
```

Similar script can be used to resume Genetic_Node operation.

A.6. Extending the System Capabilities

User may need to add the custom evolution algorithm. There are several files that need to be changed.

In the */manager/population.h* and */manager/population.cpp* files, the new class has to be declared and defined. The class must inherit from the class Population or any of its descendants.

The population information must be added to the following methods (similarly to the information on other populations contained in those methods).

Population::prepareStringifiedDescription() (*/manager/population.cpp)*
GeneticSystem::startPopulation() (*/manager/geneticSystem.cpp)*
CORBAManager_i::startPopulation() (*/manager/CORBAManager.cpp)*

It also needs to be added to the ***new_pop()*** function in */panel/PanelMain.cpp*

The last file to modify is IDL description file */common/genetic.idl*, which must be recompiled.

A.7. System Tools

There are two tool programs available in */build/tools* directory: logconverter and chrominfo.

Logconverter
This tool converts logs created by the system manager to give results in "common time unit". Different populations have different evaluation times per generation, therefore their comparison can be made after conversion to common time unit. The reference unit is set in the *#define __RADEVALS_PER_UNIT* of the */tools/logconverter.cpp* file.

Usage:
logconverter input_file [output_file [column]]

input_file is a file to be converted

output_file is a destination filename

column is a number of a column, which is to be filtered out from the input_file

example:

logconverter pop.log pop.log.out 5

This command converts file *pop.log* into *pop.log.out* . Output file contains two columns: first with time instants converted to common time units and second with values from the 5-th column corresponding to those time instants.

Chrominfo
This tool gives full information on the chromosome. The fitness value, number of wrong outputs produced and number of sensitive inputs.

Usage:
chrominfo architecture_file [-rc reference_file] [-tt truth_table_file] chromosome_data

architecture_file is an architecture file of the simulated programmable circuit

reference_file or *truth_table_file* are reference files used in evaluation. The type of the file used is selected using –rc or –tt option.

chromosome_data is a chromosome data in a binary form.

A.8. Architecture and reference files

Architecture File
Architecture file describes the architecture of the programmable circuit used in the simulations.

It can contain the following lines:

input *input_node*
defines the input to the ciruit. Any *input_node* name can be used, but it has to be used consequently throught the file.

output *output_node*
defines the output from ciruit. Any *output_node* name can be used, but it has to be used consequently throught the file.

inv *output_node intput_node*
defines inverter in the circuit. Node names can be those defined with **input** and **output** lines, but can also be new ones.

andor *matrix_height matrix_width default_configuration output_node input_nodes*
defines andor matrix in the circuit. *matrix_height* is a number of inputs to the fixed OR matrix, *matrix_width* is the number of inputs to the programmable AND matrix. Every *matrix_width* of inputs produce one product term. *default_configuration* is a binary string of *matrix_height*matrix_width* space separated default configuration values given in rows. *output_node* can be any of the node names defined with **output** lines, *input_nodes* is a space separated list of *matrix_width* inputs to the matrix (selected from the names defined with **input, output** or **inv** lines).

Example
An example file describing a programmable circuit with two inputs, three and-or matrices (12x1) and one output. All outputs are fed back to the matrix.

```
input i1
input i2
output o1
inv _i1 i1
inv _i2 i2
inv _i3 i3
inv _o1 o1
inv _o2 o2
inv _o3 o3
andor 3 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 o1 i1 _i1 i2 _i2 i3 _i3 o1 _o1 o2 _o2 o3 _o3
andor 3 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 o1 i1 _i1 i2 _i2 i3 _i3 o1 _o1 o2 _o2 o3 _o3
andor 3 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 o1 i1 _i1 i2 _i2 i3 _i3 o1 _o1 o2 _o2 o3 _o3
```

Reference File
Reference file describes the architecture of the circuit that is used as a reference for the evaluation. It can be in two forms: circuit description file and truth-table file.

Circuit Description File
Can contain the following lines:

input *input_node*
defines the input to the ciruit. *input_node* name should correspond to the name used in architecture file.

output *output_node*
defines the output from ciruit. *output_node* name should correspond to the name used in the architecture file.

or *output_node input_nodes*
Defines the OR-gate. Output_node gives the name of the output of the gate and input_nodes is a space seprated list of inputs connected to the gate.

and *output_node input_nodes*
Defines the AND-gate. Output_node gives the name of the output of the gate and input_nodes is a space seprated list of inputs connected to the gate.

xor *output_node input_nodes*
Defines the XOR-gate. Output_node gives the name of the output of the gate and input_nodes is a space seprated list of inputs connected to the gate.

inv *output_node intput_node*
defines inverter in the circuit. Node names can be those defined with **input** and **output** lines, but can also be new ones.

Example
The following example describes voting circuit from the figure 2.8.

```
input i1
input i2
input i3
output o1
and oa1 i1 i2
xor ox1 i1 i2
and oa2 ox1 i3
or o1 oa1 oa2
```

Truth-Table File
This file contains a list of output values for input values in the normal binary order. For example a full-adder circuit from the figure 5.2. is shown in the table 3.4. The file corresponding to that circuit is shown below:

```
00
10
10
01
10
01
01
11
```

B. Appendix B
Distributed System IDL File (/common/genetic.idl)

```
interface CORBANode{

      typedef sequence<string> StringSeq;
      typedef sequence<octet> OctetSeq;
      typedef sequence<OctetSeq> Chromosomes;
      typedef sequence<long double> Fits;

      enum RadFitnessType{
          Normal,
          Added,
          Log2,
          Proportional,
          Sq
      };

      long shortEvaluate(in Chromosomes question, out Fits answer);
      long radEvaluate(in Chromosomes question, out Fits answer, in
RadFitnessType radType, in double maxFit);
      void setArchitecture(in StringSeq arch);
      void setCircuit(in StringSeq circuit);
      void setResults(in StringSeq results);
      void reinitializeSimulator();
      void isAlive();
};

interface CORBAManager{
      exception CannotRegisterException{
          string msg;
      };

      exception CannotStartPopulationException{
          string msg;
      };

      exception CannotStopPopulationException{
          string msg;
      };

      enum CrosserType {
          OnePoint,
          TwoPoint,
          TwoPointConstLength,
          TwoPointConstLengthMoving,
          Uniform
      };

      union CrosserArg switch(CrosserType){
          case TwoPointConstLength:
          case TwoPointConstLengthMoving:
            short length;
      };

      struct Crosser {
          CrosserType type;
          CrosserArg  argument;
```

```
};

enum StrategyType {
    Overlap,
    OverlapStrong,
    Greedy
};

struct OverlapArg {
    long popSize;
    long overlapSize;
    double crossProb;
    double mutProb;
};

struct OverlapStrongArg{
    long popSize;
    long overlapSize;
    double crossProb;
    double mutProb;
    double strongRatio;
    double strongTreshold;
};

struct GreedyArg {
    long popSize;
    long overlapSize;
    double crossProb;
    long mutExpected;
    long strongReq;
    long maxPopSize;
    long minReqFit;
    long minReqRadFit;
};

union Strategy switch(StrategyType){
    case Overlap:
      OverlapArg OverlapArgs;
    case OverlapStrong:
      OverlapStrongArg OverlapStrongArgs;
    case Greedy:
      GreedyArg GreedyArgs;
};

enum ReferenceType{
    Circuit,
    Results
};

struct Reference{
    ReferenceType type;
    CORBANode::StringSeq refData;
};

enum FinishCondType{
    Number,
    Fitness,
    NumberORFitness
};
```

```
      struct NumberORFitnessArg{
          long maxGenNumber;
          double desiredFitness;
      };

      union FinishCond switch(FinishCondType){
          case Number:
            long maxGenNumber;
          case Fitness:
            double desiredFitness;
          case NumberORFitness:
            NumberORFitnessArg NumberORFitnessArgs;
      };

      enum FitnessScalerType{
          Linear,
          Square,
          Cubic
      };

      void registerNode(in string reference, in string name)
raises(CannotRegisterException);
      void startPopulation(in string name, in CORBANode::StringSeq
architecture, in Reference ref, in Strategy genStrategy, in Crosser
cross, in FitnessScalerType fitScaler, in CORBANode::RadFitnessType
fitType, in FinishCond finish, in short length, in long priority)
raises(CannotStartPopulationException);
      void stopPopulation(in string name) raises
(CannotStopPopulationException);
      CORBANode::StringSeq listPopulations();
      CORBANode::StringSeq listNodes();
};
```

C.Appendix C
The Best Configuration Obtained During Simulations

Length: ***448***
Fitness: ***15.9955***
Wrong outputs produced: ***2***
Sensitive bits: ***2***

```
1000000000011000010110000001010001100100000101000101100000000
0011010100000101000100000000000100101100100000000011011110000
1011111000100000000010110010000000101010100000000000101000000
0010001000010100000001000000101000000010000001010000000000111
1111100010110010100000000000000010101111110000010100000000000
0010100000000010110110111011011101001000000000001001000001010
0001010100000000000100110000000000010110111101101100001111111
0001011111100010111001
```