

Evolutionary Fault Repair of Electronics
in Space Applications

by Sverre Vigander

Preface

This dissertation describes a research project undertaken as a part of my degree at the Department of Computer and Information Science at the Faculty of Physics, Informatics and Mathematics, in the Norwegian University of Science and Technology (NTNU) in Trondheim. The project was performed at the Centre for Computational Neuroscience and Robotics (CCNR) at the University of Sussex.

The project involved familiarizing with the field of artificial evolution, in particular its applications in the space industry. A field was found where it would be possible to add to general knowledge by conducting experiments designed by the student, namely the issue of fault tolerance in space applications, with particular emphasis on the use of evolutionary techniques. The experiments involved enhancing a much used method for fault tolerance by adding a genetic algorithm.

Here, I present only my final ideas, experiments and conclusions, and not everything along the winding path to get there. For example, several ideas for experiments on similar topics were developed and discarded before the project finally took its current form.

I would like to thank my supervisors Adrian Thompson and Pauline Haddow for their support in my work.

To contact the author, please see <http://go.to/sverre>.

University of Sussex, 28th of February, 2001

Sverre Vigander

Abstract

This project has explored the use of genetic algorithms (GAs) in the field of fault tolerance in electronics. Special emphasis has been put on fault tolerance in space applications, a field where autonomous systems are important and faults can have great implications. It is the author's view that this field can benefit greatly from an evolutionary approach.

Several experiments were conducted to explore the possibility of using evolution for fault repair. These experiments indicate that repair of digital logic is a hard task for general genetic algorithms.

An enhanced version of a fault tolerance scheme commonly used in space applications (a voting system) was proposed. The experiments show that although genetical fault repair failed to impress, the newly proposed system performs potentially very well. The report also highlights some practical issues that should be investigated before the idea will be applicable in real systems.

Contents

1 Introduction	1
2 Evolution	2
2.1. Genetics	2
2.2. Natural Evolution	2
2.3. Computational Evolution	3
3 Faults in Electronic Circuits	5
3.1. Physical faults	5
4 Reliability	7
4.1. Avoid or Tolerate?	7
4.2. Redundancy	7
4.3. Mean Time To Failure/Repair	7
5 Evolutionary Approaches to Fault Tolerance	9
5.1. Embryonics	9
5.2. Mutational Robustness	9
5.3. Evolution in the Presence of Faults	10
5.4. Populational Fault Tolerance	10
5.5. Anytime Learning	10
5.6. Reactive Navigation System	11
6 Fault Repair in Space Applications	12
6.1. Voting System	12
6.2. Evolutionary Voting System	13
6.3. Problems	13
6.3.1. Possible Faults	14
6.3.2. Radiation Hardness	15
6.3.3. Failure Rate	15
7 Experiment Outline	16
7.1. Experiment Background	16
7.1.1. The Application	16
7.1.2. The FPGA Model	16
7.1.3. Representation of an Individual	18
7.1.4. Fitness Evaluation	18
7.1.5. Selection Method	18
7.1.6. Genetic Operators	18
7.1.7. Elitism	19
7.1.8. Miscellaneous Parameters	20
7.1.9. Seeding the Population	20
7.2. List of Experiments	20
Experiment 1: Repair of Damaged Circuit	21
Experiment 1.1: Extra Cells	21
Experiment 1.2: Long Run	22
Experiment 2: Accepting Imperfect Repair	22
Experiment 2.1: Voting System with Imperfect FPGAs	23
Experiment 3: Repeats of Same Run	23

8 Results	24
Experiment 4: Repair of Damaged Circuit	24
Experiment 4.1: Extra Cells	25
Experiment 4.2: Long Run	25
Experiment 5: Accepting Imperfect Repair	28
Experiment 5.1: Voting System with Imperfect FPGAs	29
Experiment 6: Repeats of Same Run	31
9 Conclusion	32
9.1. General Conclusion	32
9.2. Further work	32
<i>Appendix A Stuck-At Model</i>	<i>A-1</i>
<i>Appendix A.1. Assumptions</i>	<i>A-1</i>
<i>Appendix A.2. An Example</i>	<i>A-1</i>
<i>Appendix A.3. Advantages and Disadvantages</i>	<i>A-2</i>
<i>Appendix B Multiplier Implementation</i>	<i>B-1</i>
<i>Appendix B.1. SIS</i>	<i>B-1</i>
<i>Appendix B.2. Multiplier Truth Table Script</i>	<i>B-1</i>
<i>Appendix B.3. Mapping</i>	<i>B-2</i>
<i>Appendix C Glossary</i>	<i>C-1</i>

1.0 Introduction

Space technology is a particularly costly field, with high risks and where failure is common. The launch of, for example, a satellite can cost millions of pounds, but even when the satellite has been placed in a stable orbit, the risk is not over. Space is a harsh environment, and a satellite with delicate electronics is being continuously bombarded by harmful radiation. Despite physical shielding, space systems are more likely to fail than similar, earth-bound counterparts. What is more, once a system in space fails, any physical repairs can be very costly, if not outright impossible.

This project's main goal is to look at ways of increasing reliability in space applications with the help of evolutionary techniques. The project investigates the use of genetic algorithms (GAs) to repair faults on Field Programmable Gate Arrays (FPGAs), a reconfigurable technology that can implement any electronic circuit.

For the experiments, simulation of a simple, feed-forward FPGA model has been employed to implement a conventionally designed 4-bit multiplier. These experiments involved introducing errors into the simulated FPGAs to represent damage from radiation, and then using a GA to evolve a new circuit that performs well on the damaged chip.

Finally, a new system for fault tolerance and repair is proposed, using a GA to enhance a normal fault tolerance technique: A voting system. It is shown through experimental results that this enhanced voting system will be able to cope with faults in the system.

Section 2.0 is a short introduction to an evolutionary approach, meant as an introduction for the reader unfamiliar with GAs. Section 3.0 and 4.0 discusses faults in electronic systems, and what they do to the reliability of a system. To show what else has been done in this field, Section 5.0 summarizes some related work performed by various other people. The authors ideas are outlined and discussed in Section 6.0, followed by the experiments performed, as described in Section 7.0 and 8.0. Finally, the conclusion along with suggestions for further work can be found in Section 9.0.

2.0 Evolution

This chapter is written as an introduction to natural and artificial evolution. It is meant as a straightforward and very simple introduction to the basic concepts and terminology of artificial (and natural) evolution. For a more thorough description of artificial evolution, see [14].

2.1. Genetics

In 1865, an Augustian monk named Mendel published a theory about how individuals inherit traits from their parents [6]. Unfortunately, his work was overlooked for 35 years. Working with pea plants, Mendel discovered that the plants had several traits of a discreet character (e.g. round or wrinkled peas, yellow or green seeds) that were passed down from parents to offspring. This finding contradicted the accepted theories at the time which held that traits were inherited as an average mix between the traits of your parents. He formulated a theory which stated that inheritance were controlled by "particles", or "*factors*." What he had found evidence for were the discrete inherited units that we now call *genes*.

Today, we know that Mendel was right, and that every living creature on Earth consists of cells with genes in their cores. Each gene is a code for a specific trait¹ (e.g. seed shape in peas) and each of these traits can have several alternatives (e.g. wrinkled or smooth seeds in peas), called *alleles*. During the middle of the 20th century the physical basis of genes were discovered. Genes are coded in an information string and "stored" in a molecule called DNA. A single strand of DNA contains hundreds or thousands of genes, and what is more, an individual can have several strings of DNA, called *chromosomes*. The full set of genes in an individual is called its *genotype*, while the set of traits coded by the genotype is called the individual's *phenotype*. The phenotype, however, is not defined by the genotype alone, but is also affected by the individual's *environment*.

2.2. Natural Evolution

Natural evolution was a theory first formally described by Charles Darwin in his work "The Origin of Species by Means of Natural Selection" in 1859 [2]. He described a mechanism called natural selection. The idea is that in a population of organisms, some individuals will be more *fit* than others (some wilderbeests run faster than others, some birds are better at camouflaging their eggs than others, and so on). The fittest animals will be more likely to survive and reproduce, thereby having a higher chance of passing their genes² on to the next generation. Less fit individuals will have a higher chance of dying before they can reproduce and the (less fit) genes these organisms carry will eventually be *selected* out of the population.

When reproducing (sexually), two individuals with different chromosomes "splice" their DNA to form the genotype for their offspring. This new genotype has a mixture of genes from the organism's parents. For simplicity, assume that some parts of the offspring's chromosomes come from

-
1. This is a simplification. Genes can code for one or several traits, or for activation of other genes, or even for nothing at all. In general, genes really code for the production of proteins, and the proteins may have very different effects.
 2. Note that Darwin didn't know about genes, he had a different suggested mechanism for inheritance of traits.

one parent, and some other parts from the other (sexual reproduction is of course much more complicated than this, but that is beyond this report). This crossing of DNA and mixing of genes can be called a *crossover*.

Natural selection can be seen as a natural algorithm that chooses between alternative alleles from a gene pool. However, for natural selection to power evolution, a continuous source of new variation in the gene pool is needed. Without this, variation would die out, and there would be nothing left to select between. The final, critical component of natural evolution is therefore the creation of new alleles by *mutations*.

A mutation is a random change to an individual's genotype. This results in a change to one or more genes, and this may in turn effect a change in its offspring's phenotype. A caused by a mutation will in the vast majority of cases reduce the individual's fitness, in which case the individual most likely will die before it can reproduce. Very occasionally, the change increases an individual's fitness, and the individual will be more likely to survive and pass its genes on to its own offspring.

2.3. Computational Evolution

Computational or artificial evolution is inspired by natural evolution, and embodies the same basic mechanisms. One type of computational evolution is *Genetic Algorithms* (GAs). Invented and developed by Holland ([5]), a GA searches the solution space; evolving a good solution (high fitness genotype) from a set of initial solutions (gene pool), often randomly created. A *fitness value* is given each individual, according to how well the individual solves the task at hand (e.g. if you want to evolve a robot controller, you can define an individual's fitness value according to how well the robot performs, using that individual as controller).

Be aware of the main difference between natural and (most types of) computational evolution; in natural evolution fitness is *implicit*. That is, the individuals that produce most surviving offspring are per definition most fit. In computational evolution, fitness is (often) made *explicit*, calculated from how well an individual solves a certain task. Natural evolution does not have a specific goal, while computational evolution is a tool for finding a solution.

In computational evolution, the chromosome is often implemented as a bit string. A gene can be composed of a single bit, or in some cases several bits, for example if several bits encode a single trait. Crossover can be implemented by giving a new individual some genes from one parent, and the rest of the genes from the other. Mutation will, for example, be the inversion of a random bit.

Figure 2.1 illustrates the evolution process in a GA. The initialisation step generates a population, often with random gene material, evaluating each individual and thus assigning them fitness values. From this first generation two individuals will be selected (selection) to allowed reproduce (genetic operators, e.g. mutation and crossover). Their offspring are evaluated (fitness calculation) and inserted in the next generation. These steps are repeated until the next generation is full (the set population size has been reached). When generation $n + 1$ is full, the process can start again,

filling generation $n + 2$ and so on. This goes on until an individual providing a good enough solution is found, or until a set number of generations have been made.

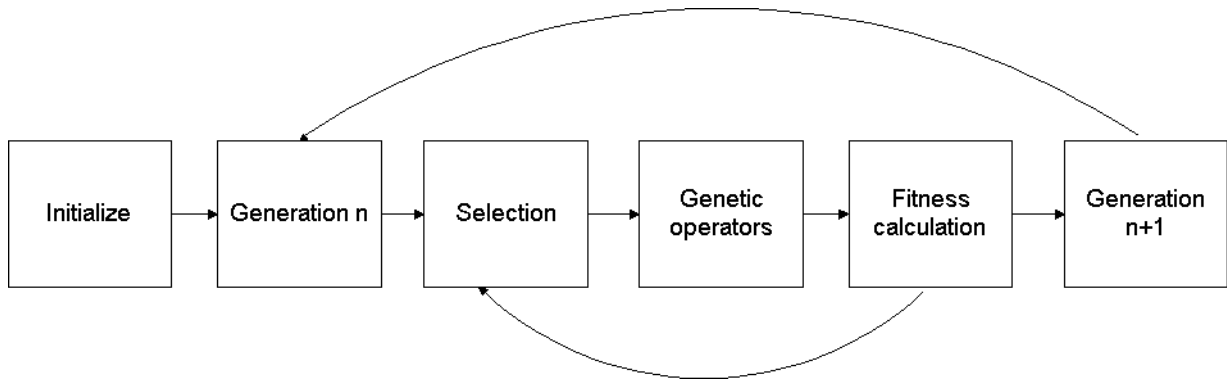


Figure 2.1: The evolution process

3.0 Faults in Electronic Circuits

In today's complex computer systems, every now and then something goes wrong. Faults can range in magnitude from small, imperceptible calculation errors to major disasters that can even take human life. To catch all potential faults requires rigorous and costly testing, something not always economically or physically possible. This chapter is a short introduction to the possible faults to look out for, with emphasis on the type that is important for this project.

The literature on fault tolerance (see H. Castros thesis, [1]) distinguishes between three different groups of faults in hardware: Design, interaction and physical faults.

Design faults. As the name indicates, these faults are inherent system faults made during the design of the system.

Interaction faults. What will normally be referred to as "human error." Usage of system in another way than what was originally intended by designers.

Physical faults. Hardware faults due to impurities in the manufacturing process, aging or external events, for example radiation.

This paper is primarily concerned with using evolution to design systems that are tolerant to the effects of physical faults. This is especially important in space applications, in which the harsh environment of space threatens to disrupt the physical attributes of electronic circuits.

3.1. Physical faults

Physical faults have been further classified by the following attributes:

Duration. What is the time aspect of the fault? Is it a transient or a permanent fault?

A transient fault is a fault that disappears after a certain amount of time. Such a fault can be caused by noise (e.g. sun spot activity), and does not have to be repaired. The important thing is being able to detect that a fault has occurred. Dealing with a transient fault may be as simple as re-evaluating the circuit after a short time period.

Permanent faults include for example a broken connection. A permanent fault will not disappear on its own accord, but needs to be repaired. Repair of permanent, physical faults traditionally means manually replacing a part. This is particularly costly, and often impossible, in space applications.

Value. What is the resulting value caused by the fault? Typically, the faults in electronic circuitry cause either a stuck value ('1' or '0'), or a seemingly randomly alternating value.

Extent. How much of the circuit is affected by the error? This can be a single variable (a local fault), or a whole subsystem (a distributed fault). Since single, local faults are much less complicated than distributed ones, many fault tolerance strategies focus only on local faults. An example of this is the "stuck-at" model, which basically models faults as short-circuits to ground ('0') or

power ('1'). Due to its simplicity and versatility, this is the model which is used in this work. See Appendix A for more details on the stuck-at fault model.

4.0 Reliability

This chapter briefly discusses reliability in an electronic circuit. That is, how to make sure a circuit does not break down, even when it's prone to errors. This discussion is in general taken from [1]. Section 4.3. defines the key term *availability*, a measure of how reliable a system is.

4.1. Avoid or Tolerate?

There are two main ways of increasing reliability in a system; fault avoidance (or prevention) and fault tolerance.

Fault avoidance. With fault avoidance the system is designed so that foreseen, potential faults are prevented from happening. This can include radiation shielding or careful testing and debugging. With good fault avoidance faults are much less likely to occur, but if a fault actually happens, the system may break down completely, necessitating a manual repair. Clearly, this is not optimal in a system where it is difficult and costly to prevent faults (using e.g. heavy radiation shielding) and expensive to fix faults manually (needing in worst case a space shuttle mission, or similar).

Fault tolerance. Fault tolerance is a different approach; allowing that faults will happen but designing the system to continue working as well as possible while faults are occurring. This can be done for example with redundant logic, in other words, doing the calculations several times and comparing. Note that most fault tolerance schemes will be disadvantageous the system in terms of performance, size and cost. A sub-group of fault tolerance is automatic fault repair; for example using evolutionary techniques to fix a damaged circuit (e.g. bypassing the fault).

Obviously, a truly robust system may well incorporate both fault avoidance and -tolerance techniques in its design. This way, it will be able to avoid most faults, but still be able to tolerate the faults that "slip through."

4.2. Redundancy

Redundancy is the use of extra (or backup) hardware, and it can be either static or dynamic.

Static. Static redundancy is when the redundant logic automatically finds the best answer, without reconfiguration of any kind. An example is a voting system, where several identical units "vote" for the correct solution. The identical units work in parallel, and the majority answer is chosen as the system's answer.

Dynamic. Dynamic redundancy is where a detected fault makes the system reconfigure itself. E.g. a system can have "reserve" logic, unused before needed. When a faulty unit is detected, it can be replaced with a reserve unit by reconfiguring the routing between them.

4.3. Mean Time To Failure/Repair

The average time it takes between starting up a system and when it fails, is usually referred to as *Mean Time To Failure*, or MTTF. The time it takes to fix a system once it has failed, is called *Mean Time To Repair*, or MTTR. The system *availability* (average part of time the system is working properly) is defined as

$$\text{availability (A)} = \text{Time system is available} / \text{Total time} \quad (4.1)$$

or

$$A = \text{MTTF} / (\text{MTTF} + \text{MTTR}) \quad (4.2)$$

Equation (4.2) gives the following conclusions:

$$\text{If } \text{MTTF} \gg \text{MTTR}, A = 1 \quad (4.3)$$

$$\text{If } \text{MTTR} \gg \text{MTTF}, A = 0 \quad (4.4)$$

In words, for the system to be available for as much time as possible, the time to repair the system should be much less than the time between failures.

ling a stuck-at fault (see Appendix A). In his encoding, the fault has the same as the effect of a mutation, giving rise to an automatic fault tolerance. Successful individuals will be insensitive to mutations, otherwise the majority of their offspring will be adversely affected by mutations. Therefore, insensitivity to faults are inherent in such applications of genetic algorithms.

5.3. Evolution in the Presence of Faults

The argument in Section 5.2. only holds when the genetic encoding makes sure that faults have the same phenotypic effect as a mutation. Thompson evolved other fault tolerant circuits by deliberately subject the system to faults during evolution [20].

The problem with this approach is that you will have to anticipate what kind of faults your system will be exposed to. Using *co-evolution* to generate faults have been suggested as a way of finding the "right" faults to expose the system to.

5.4. Populational Fault Tolerance

Arguing that the inherent qualities of evolved circuits are different than those of hand designed ones, Paul Layzell has found evidence for what he calls Populational Fault Tolerance (PFT) [11]. PFT is the potential for a population of evolved circuits to contain an individual that performs well in the presence of a fault that would render the currently best individual useless. This is due to the incremental nature of the evolutionary process, allowing ancestors that would be unaffected by this particular fault to exist relatively undisturbed in the final genotypes.

5.5. Anytime Learning

Seeing that applying evolution directly on a real system can be very costly, in terms of testing out bad individuals, the Anytime Learning scheme [3] separates the evolution from the real world. By dividing the system into a learning system and an execution system, robot controllers can be

evolved in a simulation of the real world. The simulation is updated by an environment monitor. Figure 5.2 shows Anytime Learning schematically.

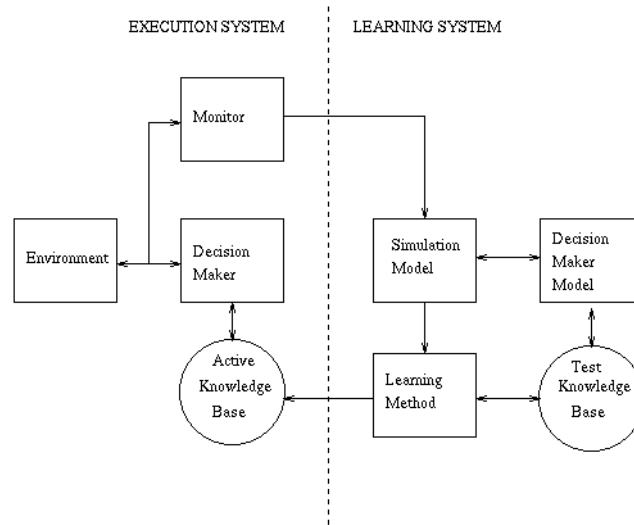


Figure 5.2: Anytime Learning. By separating the learning system from the execution system, evolution can be run on a simulation, without adversely affecting the real system. Figure from [3].

5.6. Reactive Navigation System

[9] describes a system for robot navigation, using evolutionary hardware. They use what they call a *model-based* evolution method. With this method, they continuously download a controller to the robot, while evolution is running on-line on a model of the real world. This has much in common with the Anytime Learning scheme (Section 5.5.).

A central issue for many of these methods is that they can be used in applications where fault avoidance won't be possible. It is for example impossible to foresee all obstacles a Mars rover will encounter, and problems not thought of during the design may easily appear. These evolutionary systems are ways of tolerating and adapting to new problems as they arise.

6.0 Fault Repair in Space Applications

The aerospace industry is spending a lot of resources trying to make space systems more reliable. A failure of a space system very costly and often impossible to repair, and space is a harsh environment with radiation and temperature fluctuations making electronic equipment perform worse than optimal. Making fault tolerant systems is therefore a high priority in organizations like NASA.

This chapter reflects fault tolerance in space applications and introduces some ideas from the author on how to apply evolution to existing fault tolerance schemes.

6.1. Voting System

A widely used way of coping with errors in a space system is the voting system scheme. The system will consist of several (for example five) identical¹ circuits, and if no faults exist, they will agree on the correct solution. The degree of redundancy (number of circuits) should be set according to the importance of error-free performance and the probability that a fault will occur (three being the minimum number needed). For no performance loss, the total number of faulty circuits should never be more than half of the available ones. If this happens, there is a chance that the faulty circuits agree on some wrong answer. Figure 6.1 shows a typical voting system with five identical circuits.

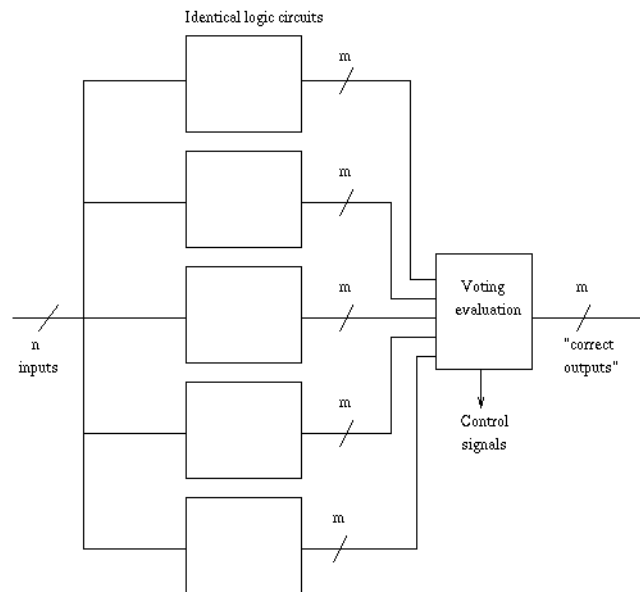


Figure 6.1: The voting system. The control signals from the “Voting evaluation” block can shut down a module if it disagrees with the majority.

1. It is possible to make voting system with non-identical circuits. For example, a system including several different versions of the same circuit may be able to vote away design faults that appear in each of them.

A fault is detected when the outputs of the originally identical circuits are in disagreement. The voting element will select as system output the value that a majority of the circuits agree on. The system is fault tolerant because of the voting - a fault in one (or more, up to half of the available) circuit does not change the behaviour of the system. A circuit in disagreement with the majority should be assumed to have faults, and can be switched off.

As long as the redundancy is high enough that the majority of circuits are unharmed, the voting system makes sure that physical faults won't affect the system's behaviour. But if more than half of the circuits contain faults, the system will behave more erratically. Without error detection circuitry on each of the units, there is no way of telling what circuit to trust.

The voting system will detect and tolerate transient faults, as well as permanent faults. When a fault is detected, the faulty chip can be re-evaluated to check for transience, before evolution is applied. This should be done after a small time span, to allow a possible transient fault to go away.

Any voting redundancy scheme like this has a single weak point, where a fault will be devastating. In this case, the voting logic is imperative for good performance. Care has to be taken so as to not make the system perform worse than each individual unit. In practice though, the voting evaluation block is so small that the chance of a fault there is very small.

The voting system has a speed advantage over other fault tolerance schemes in that the longest path through the circuit is only lengthened by the voting block, even though the area is increased n times (n being the level of redundancy). To compare this with another scheme, embryonics (Section 5.1.) adds to the longest path in every cell, thus making the system much slower.

6.2. Evolutionary Voting System

To enhance the voting system with evolution, a GA can be connected to the circuits. If a unit fails (disagrees), it will evolve instead of being shut off. This allows the unit to improve, and eventually reach full functionality. Even if evolution cannot bring the circuit back to full functionality, an improvement may be sufficient. Several, nearly perfect circuits have a high probability of failing on different computations, thus the majority may still always come up with the right answer.

The difference between an evolutionary and a conventional voting system (as described in Section 6.1.) may not be apparent at first. As long as less than half of the circuits involved have errors, this system performs exactly like a normal voting system.

If more than half the circuits in a conventional voting system have errors, a serious dilemma emerges. Since it is not known which circuits are faulty (all that is known that they disagree), which one will you choose? This project focuses on a solution that can be used instead of area- and power-expensive error-detection mechanisms. The proposed solution should be able to repair enough damage that the voting system works well, even if more than half of the components are damaged.

6.3. Problems

This section discusses some of the problems the suggested voting system will face.

6.3.1. Possible Faults

A big problem for using evolution on a space-craft (or any other “harsh” environment) is that the evolutionary algorithm itself takes up silicon space that can turn faulty as well. There will, for example, be a major problem if the logic involving fitness evaluation fails. To overcome this, it could be possible to have communication with a ground station. The ground station can send configuration and input to the system, the signals will be processed and then sent back to Earth for evaluation of the results. (See Figure 6.2)

A satellite in geostationary orbit (more about this can be read in [8]) is roughly 35,786 kilometers above the surface. The propagation delay between the satellite and the ground station, assuming

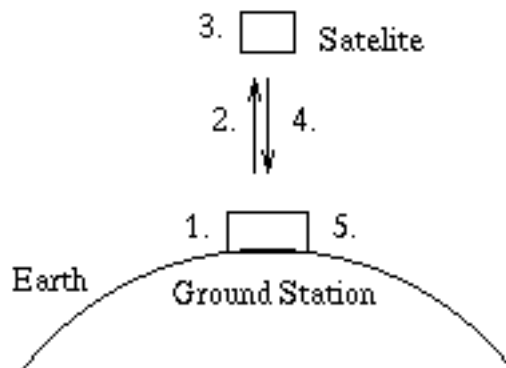


Figure 6.2: Communication between a satellite and its ground station. The ground station (1) sends out hardware configuration and inputs to the satellite (2), which then produces output signals as normal (3). These are transferred back to Earth (4), where they can be evaluated in a genetic algorithm (5).

the ground station is placed right beneath the satellite, will thus be

$$t = 2 \cdot \frac{h}{c} = 2 \cdot \frac{35,800 \text{ km}}{300,000 \frac{\text{km}}{\text{s}}} = 0.239 \text{ s}, \text{ where } h = \text{height and } c = \text{speed of light} \quad (6.1)$$

This delay is not a major issue, since the fitness evaluations can be pipelined (the steps involved are performed at the same time, but for different individuals in the GA). Nonetheless, the method would work best with satellites, as the delays to a space probe might be too substantial. The distance from Earth to Mars, for example is approximately 0.5 astronomical units (75,000,000 km, see [15]), giving a total delay of

$$t = 2 \cdot \frac{h}{c} = 2 \cdot \frac{75,000,000 \text{ km}}{300,000 \frac{\text{km}}{\text{s}}} = 500 \text{ s} \quad (6.2)$$

6.3.2. Radiation Hardness

Another possible problem that has not been looked further into in this project, is the effect reconfigurable chips have on radiation resistance. Since reconfigurability is needed for evolution, different technology (than conventional) must be used, and the reconfigurable technologies are in general less radiation-hard. That is, the basic chance of faults is increasing, but so is the ability to cope with faults. The balance between adding faults and coping with faults is a delicate one, and this should be looked into. More on radiation hardness in programmable devices can be found in [7].

6.3.3. Failure Rate

An important assumption for this scheme, is that it will be possible to let $MTTF \gg MTTR$ (see Section 4.3.). Evolution must be happening fast enough compared to the rate of errors, otherwise the system will never be able to repair itself before the next fault appears. If this is not the case, other methods, like shielding and better radiation hardness, must be applied as well.

7.0 Experiment Outline

This section goes through the different experiments performed in order to examine the subject. Section 7.1. describes the genetic algorithm used in the experiments. Section 7.2. presents short descriptions of each experiment conducted, followed by a brief description of the results. The full details of the results of each experiment can be found in Section 8.0.

7.1. Experiment Background

All experiments were performed with a Field Programmable Gate Array (FPGA) simulator written specifically for this project, described in detail in Section 7.1.2. The simulator and genetic algorithm was programmed in C++, using GCC 2.95.1 compiler. It should be noted that the simulation used is a starting point for doing experiments. It has different constraints than a physical FPGA, due to the simplicity wanted for these experiments. Thus, some of the results may not apply to real FPGAs.

7.1.1. The Application

The chosen application was a 4-bit multiplier, in other words a combinatorial circuit that takes two 4-bit numbers and outputs the product of them (an 8-bit number). This was chosen primarily because it can represent the sort of digital equipment needed on a spacecraft to do simple calculations. It is also a relatively simple, combinatorial circuit, but it is still complicated enough to be very difficult to evolve from scratch (this was in fact quickly tested in preliminary work). Indeed, to the authors knowledge, the biggest successfully evolved multiplier to date is a 3-bit multiplier.

The multiplier was hand designed by generating the truth table (using a Perl-script) and optimizing this with a tool called SIS [21]. SIS allows a design to be mapped down to a customized model, such as the FPGA model presented in Section 7.1.2. This method gave a compact design, with little extra space or redundant logic.

7.1.2. The FPGA Model

The FPGA model is a simple, feed-forward network of cells, where each cell can perform a small number of logical functions. Due to simplicity of implementation, the array is restricted to a regular rectangle with x by y cells and exactly y inputs and outputs. Each cell has one output and three

inputs, each of which can be connected to any earlier output in the array (meaning “above and to the left” in the following figure). See Figure 7.1 for an example of a legal FPGA, where $x = y = 3$.

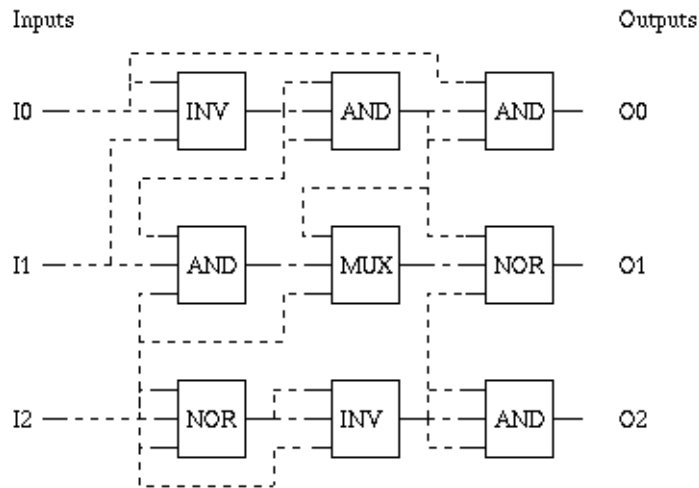


Figure 7.1: Example of a legal FPGA

Table 7.1 shows the possible cell functions allowed. The notation uses a , b and c as the three cell inputs.

Table 7.1: The possible cell functions

Name	Function
AND	2-input logical AND [$a * b$]
OR	2-input logical OR [$a + b$]
NOT	1-input logical inversion [$!a$]
MUX	Multiplexer [$a!c + bc$]
GND	Logical 0
VCC	Logical 1
NAND	2-input logical NAND [$!(a * b)$]
NOR	2-input logical NOR [$!(a + b)$]
AND3	3-input logical AND [$a * b * c$]
OR3	3-input logical OR [$a + b + c$]
NAND3	3-input logical NAND [$!(a * b * c)$]

Table 7.1: The possible cell functions

Name	Function
NOR3	3-input logical NOR $!(a + b + c)$

The main difference between the simulation model used and a real FPGA is the restrictions on routing. Artificial restrictions stronger than necessary have been set on the model to ensure a feed-forward network, with focus on implementation ease. In other words, there are strict feed-forward networks that would not be allowed in this model. On the other hand, a real FPGA may have more restrictions on how cells can be routed, depending on where they are physically located in relation to each other.

7.1.3. Representation of an Individual

Each individual in the genetic algorithm represented a circuit in the FPGA model described in Appendix A. The individual was described using two arrays, one defining the functions of all cells in the FPGA, and one defining the inputs of all the cells.

7.1.4. Fitness Evaluation

The fitness of an individual is evaluated by testing all input combinations ($2^8 = 256$), and checking whether the result is correct. A fitness value is assigned as the number of correct *minterms*, or specific input combinations. (Minterms are often numbered in binary order, so in a circuit with inputs A, B and C, the combination (A=1, B=0, C=1) is referred to as minterm 5).

Note that all figures in Section 8.0 use absolute fitness. 100% fitness means an absolute fitness of 256, a success in all test cases.

7.1.5. Selection Method

The GA uses a Ranking Selection, where the chance of being selected out of n individuals is

$$P = \frac{Rank}{\sum_n Rank} \quad (7.1)$$

where *Rank* is descending (highest number is highest rank). The advantage of this over simple, fitness proportional selection methods like the one used originally used by Holland [5], is that the scaling of fitness does not affect the selection. For more information on selection methods in general, see Mathis Landsverk's dissertation [10].

7.1.6. Genetic Operators

Several types of genetic operators were used. This is a short list of them.

Crossover. The crossover operator was implemented on an application level (as opposed to on a genotype level). That is, instead of crossing two strings at random points, a way of crossing two modelled FPGAs was made. As can be seen in Figure 7.2, a random rectangle was "drawn".

Within the rectangle, cells were inherited from one parent. The cells on the outside of the rectangle were inherited from another. The the intention is that this crossover method will be better able to distribute good building blocks, with the assumption that closely related cells will also be closely located in the FPGA. No formal experiment was conducted to see whether this assumption holds, or if the crossover method indeed performed better than a "normal" two-point crossover.

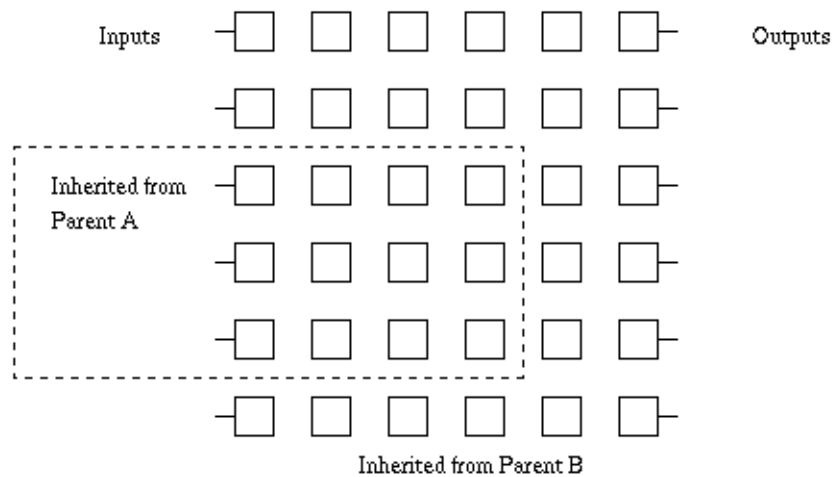


Figure 7.2: The crossover operator

Mutation. The normal mutation operator went through the chromosome, and had a certain chance of mutating each locus. This is a quite normal way of implementing mutation. See Section 7.1.8 for a short discussion on the mutation rate.

Cell swapping. In addition to crossover and mutation, a special form of mutation operator was designed. This operator simply swapped two cells in the array, so the cells exchange both cell function and inputs. The thought was that for repairing a damaged cell, the GA would want to simply swap that cell with an unused one, thus repairing the damage in a simple way. There is one major difficulty with making this work properly, namely the restriction that we want the array to be purely feed forward (see Section A.1.). This means that one cell often will be illegally placed after a cell swap. The solution to this was to pick new, random inputs whenever the rules for the FPGA model were broken. No experiments have been conducted to validate these choices.

7.1.7. Elitism

Elitism was used in all the experiments. In other words, the fittest individual was always cloned. If several individuals had equally high fitness values, they would have an equal probability of being chosen as "the elite". This was done to allow as much neutral drift in the fitness landscape as possible, to lessen the chance of being caught in local optima (see [4] for a discussion on neutrality).

7.1.8. Miscellaneous Parameters

The genetic algorithm used a population size of 50, crossover rate of 70% (30% chance that an individual has only one parent). The hand-designed FPGA used had a size of $34 * 8 = 272$ cells, increased to $34 * 12 = 408$ cells in Experiment 1.1. Each cell had four different possible mutation points, the three inputs and the cell function. The mutation rate was set to $p = 1/1000$ per locus of the genome, giving an expected number of mutations

$$E = n * p = 272 * 4 * 1/1000 = 1.088 \quad (7.2)$$

($E = 1.632$ when using extra cells in Experiment 1.1). The rate of cell swapping was set to 10%, in other words 1 out of 10 individuals would have two cells swapped (in addition to the normal mutation).

7.1.9. Seeding the Population

The population was initially seeded with identical individuals - all equal to the original design. This means the population was *converged*. According to popular myths about GAs, too early convergence is not a good thing, making the algorithm unable to explore the fitness landscape enough. According to [4], though, convergence happens in GAs after very few generations, and the population as a whole must explore the landscape in the vicinity of the best individual, looking for a way to reach higher fitness areas in the landscape. With this view, seeding the population with identical individuals should not be a problem. Indeed, in the experiments performed, no apparent loss has been seen by using this method.

7.2. List of Experiments

The following is a list of the experiments that were made in order to investigate the use of a genetic algorithm to improve reliability of a space system. Figure 7.3 summarizes the different experiments, and shows the relations between them. In the first branch of experiments, attempts were made to "repair" faults in an FPGA, that is redesign the circuit so it can function properly even though physically damaged. The second branch is using evolution to enhance a voting sys-

tem. The last experiment investigates whether restarting the genetic algorithm is a feasible way of improving its performance.

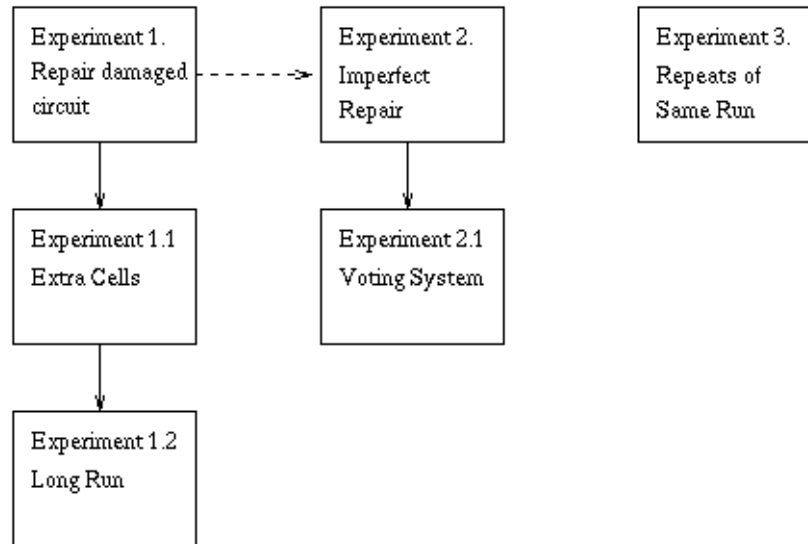


Figure 7.3: The different experiments made, and how they relate to each other.

Experiment 1: Repair of Damaged Circuit

How well can the GA repair a damaged circuit?

To see whether a genetic algorithm would be able to completely repair faults in FPGAs, this experiment invoked random stuck-at faults in the simulated FPGA. The circuit described in Section 7.1.1 was implemented, and a single, random cell in the FPGA were then forced to either '1' or '0'. The hand design was usually, but not always, adversely affected by this fault. A genetic algorithm was started, filling the initial population with clones of the hand design.

Result. As it turned out, complete repair of a damaged circuit was quite hard with the genetic algorithm used (see Section 8.0). It seemed to be very difficult to take the last step to 100% fitness. In a more complicated circuit, this would probably be even more difficult. This sets the stage for the next experiments.

Experiment 1.1: Extra Cells

In an attempt to improve the genetic algorithm, this experiment added extra (“reserve”) cells to the FPGA, increasing the search space and allowing the design more room to avoid the failed cell.

Figure 7.4 shows an FPGA with extra cells added. With this setup, a particularly difficult fault was chosen, and used in all runs, comparing extra cell space with no extra cell space.

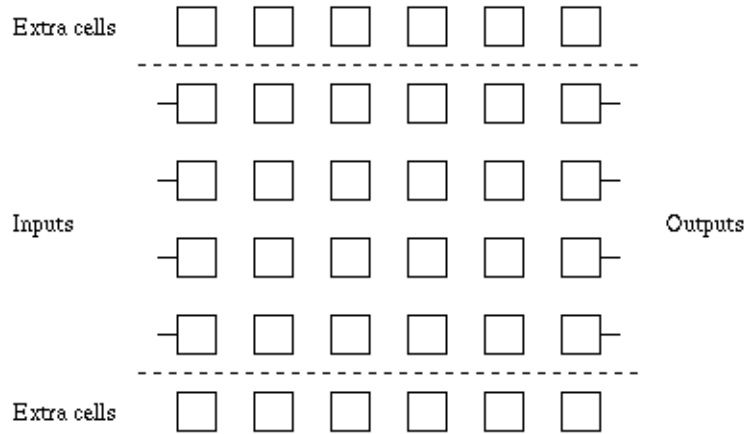


Figure 7.4: An FPGA with extra cells

Result. This experiment showed no improvement by having extra cells, as indicated by Figure 8.2. See Section 8.0 for more details.

Experiment 1.2: Long Run

One extra long run was made to see how the GA performs with more time available. The main question to be answered in this experiment was whether the population stabilizes completely, or whether it is doing neutral moves¹ over a longer time aspect, and eventually finds a way to climb to higher fitness values. This experiment was done with extra cells, using the same, difficult fault as in Experiment 1.1. The run lasted for 21000 generations (approximately two weeks computing time).

Result. Although no high fitness was reached in this experiment, the run does show improvement late in the run (see Section 8.0 for more details). Since only one run was made of this length, it is hard to tell whether this is a general trend.

Experiment 2: Accepting Imperfect Repair

Is imperfect repair of a damaged circuit just as bad as no repair? It seems very difficult to reach 100% fitness, but is this something absolutely necessary for a system to repair itself? The aim of this experiment, was to see whether any “problem minterms” exist that always fail. In other words, is there a set of inputs that faulty circuits tend to get wrong more often than other inputs? If no such set of input exist, different faulty circuits will not make errors with the same inputs. That would imply that a set of faulty circuits could vote and always get the right answer together.

1. A neutral move is a mutation not affecting the fitness.

Whether such minterms exist is, of course, dependent on the application used.

Each run in this experiment is done with a different, random stuck-at fault. The setup was identical to the one in Experiment 1, but in this experiment the way in which circuits fail is studied more closely.

Result. It turns out that the different, semi-repaired circuits generally fail on different minterms. This finding suggests that it is possible to make voting system work, even with damaged circuits. See Section 8.0 for the full results.

Experiment 2.1: Voting System with Imperfect FPGAs

In this experiment, a voting system was assembled with faulty FPGAs, slightly repaired with evolution. The object is to see that the assumption from last experiment works in practice (i.e. non-overlapping faults in a voting system should make no difference). Three FPGAs were used, with no extra space. A fault was introduced in all three FPGAs at the same time, simulating a worst-case scenario like a quick burst of strong radiation.

Result. The results from this experiment were quite encouraging, usually reaching a good solution after just a few generations, even when the individual circuits didn't behave perfectly alone. See Section 8.0 for more details.

Experiment 3: Repeats of Same Run

Due to the randomness involved in genetic algorithms, sometimes running the algorithm several times yield different results. This depends on the application and the details of the GA used. This experiment investigates whether rerunning the GA is a useful approach with this particular setup, by testing whether the same minterms are left unrepaired each time the algorithm is run.

The same fault as in Experiment 1.1 was chosen. Extra cells were used in this experiment to increase the search space.

Result. The results of this experiment indicate that the same minterms do indeed fail every time the algorithm is run. The full results are presented in Section 8.0.

8.0 Results

This section will go through the results from the experiments in more detail.

Experiment 1: Repair of Damaged Circuit

In this experiment, 57 runs were made with the GA, picking a random fault in every run. The experiment showed that solving this task in full is quite hard for the general genetic algorithm used. No perfect repair was made, although a lot of runs came close. It seems very hard to lift the population up from high fitness values all the way up to 100% fitness (getting all 256 test cases right). This may be related to the discrete nature of digital logic. When the GA has no information on where to find the maximum, it may deteriorate to a random search. Some faults may in fact be completely unrepairable, for example if an output pin has been stuck to a particular value.

Figure 8.1 shows three selected runs, to illustrate typical runs. One thing that can be seen in the figure is what effect random faults have. Many faults only harm the circuit in a small way, while a few almost cripple the circuit (thus the different starting points). It is interesting to see that the runs that start out with very low fitness values (<150) usually find higher fitness areas very quickly (~ 200). Some times, though, no improvement is made on the fitness value during the first 1000 generations (this happened with only two of the ten runs that started with fitness values <200).

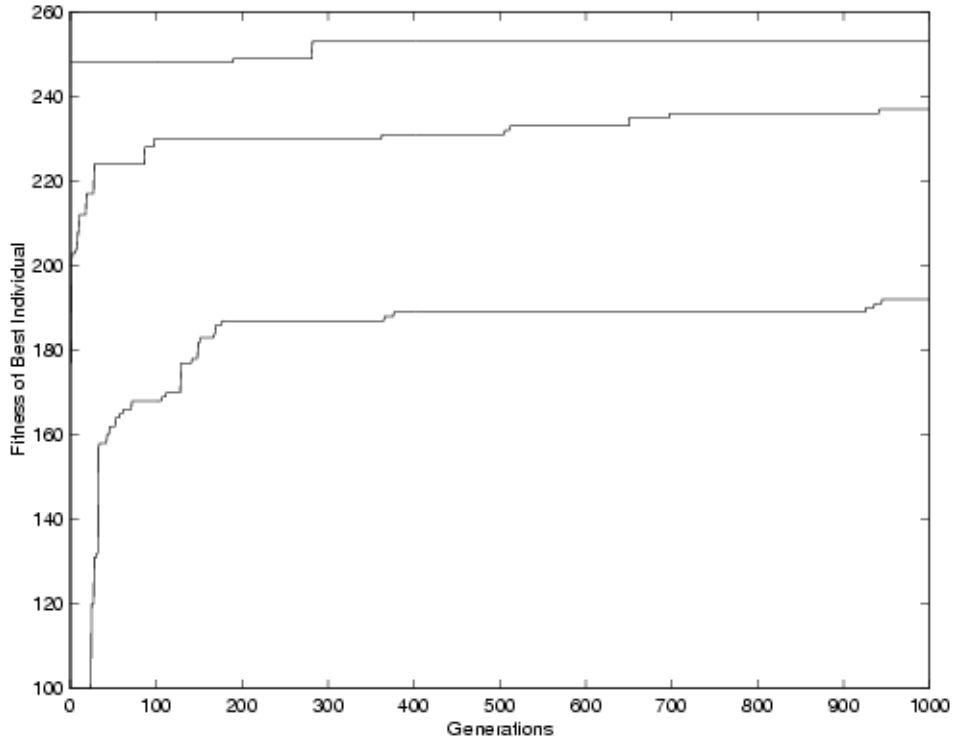


Figure 8.1: Three typical runs of Experiment 1

Experiment 1.1: Extra Cells

This experiment was made to see whether extra, unused space in the FPGA could be exploited by the genetic algorithm. The algorithm was applied to FPGAs on which a single, permanent stuck-at fault¹ was inflicted, giving the hand-designed circuit a fitness value of 120. Runs were made on FPGAs with and without extra cells. Figure 8.2 shows the best individual's fitness values, averaged over all runs (the dashed line represents extra cells, the solid line no extra cells).

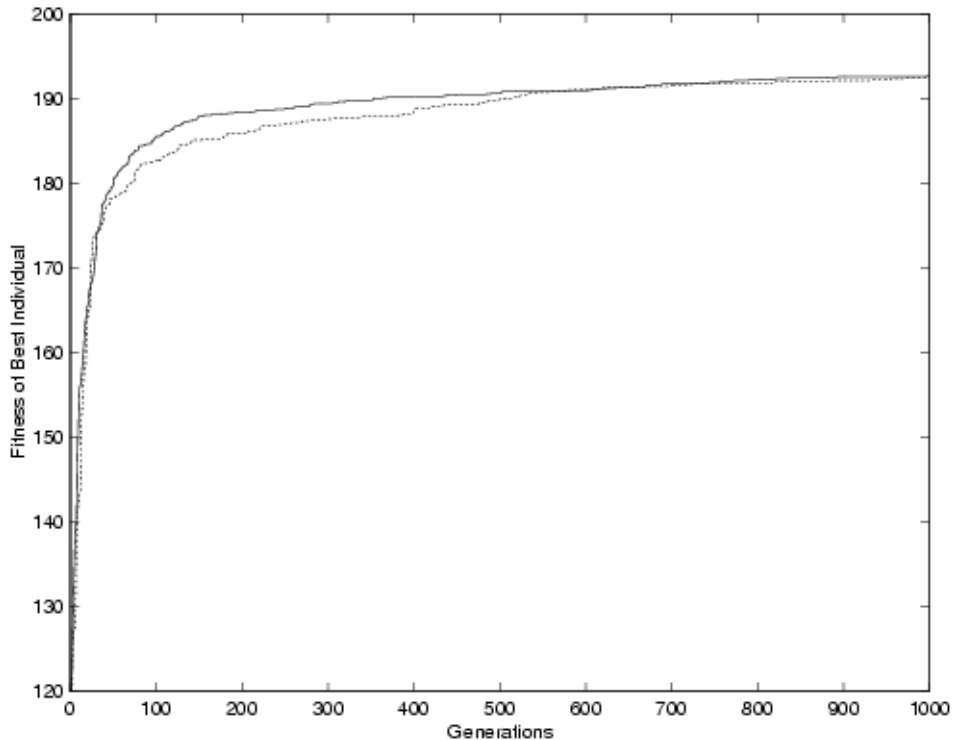


Figure 8.2: The fitness values averaged over all runs. The dashed line represents the runs with extra space, the solid line represents the "normal" ones.

Figure 8.2 shows no big difference between use of extra cells and not. No statistical analysis has been conducted to see whether this difference is significant or not. One possible explanation for the small loss in performance seen in the figure, is the higher number of dimensions. By introducing extra cells (50% cells not used by the original design were added), the search space was also greatly increased, and this may have adversely affected the GA.

Experiment 1.2: Long Run

This run was made to see whether the GA can increase fitness over a longer period of time, when it seems to have reached a stable position, as in Figure 8.3 (from Experiment 1). Most experiments were run for 1000 generations, while this experiment lasted for 21000 generations. Only

1. The fault introduced was cell no. 207 SA0. See Appendix B for implementation details of the multiplier.

one run was made in this experiment, because of the time needed to run such an experiment (approximately two weeks).

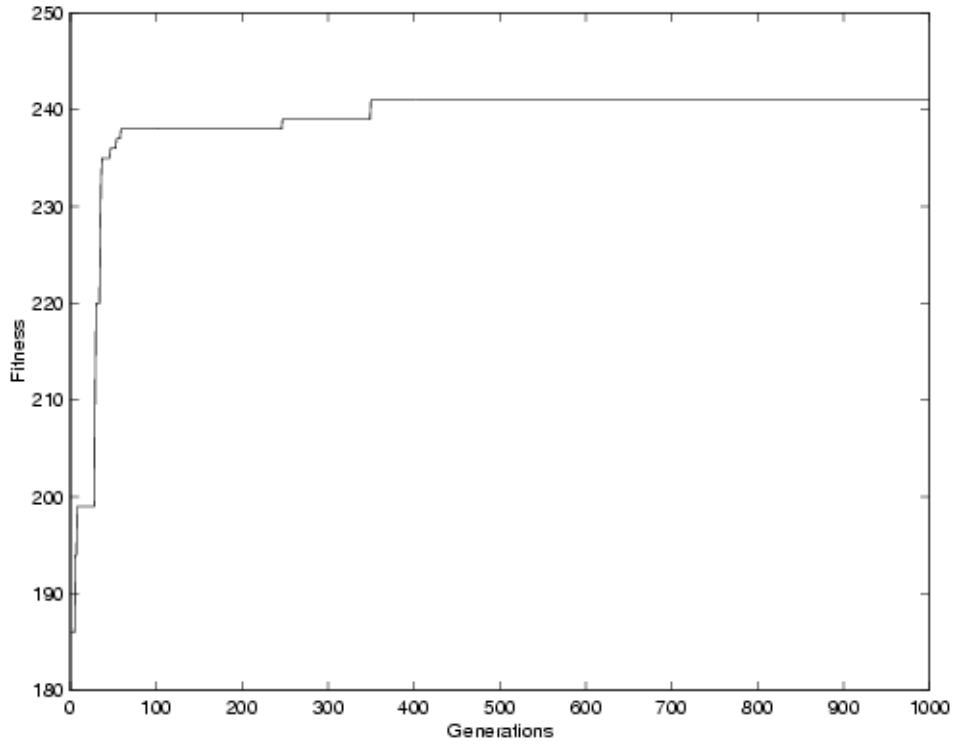


Figure 8.3: Example of stagnant run, taken from Experiment 1.

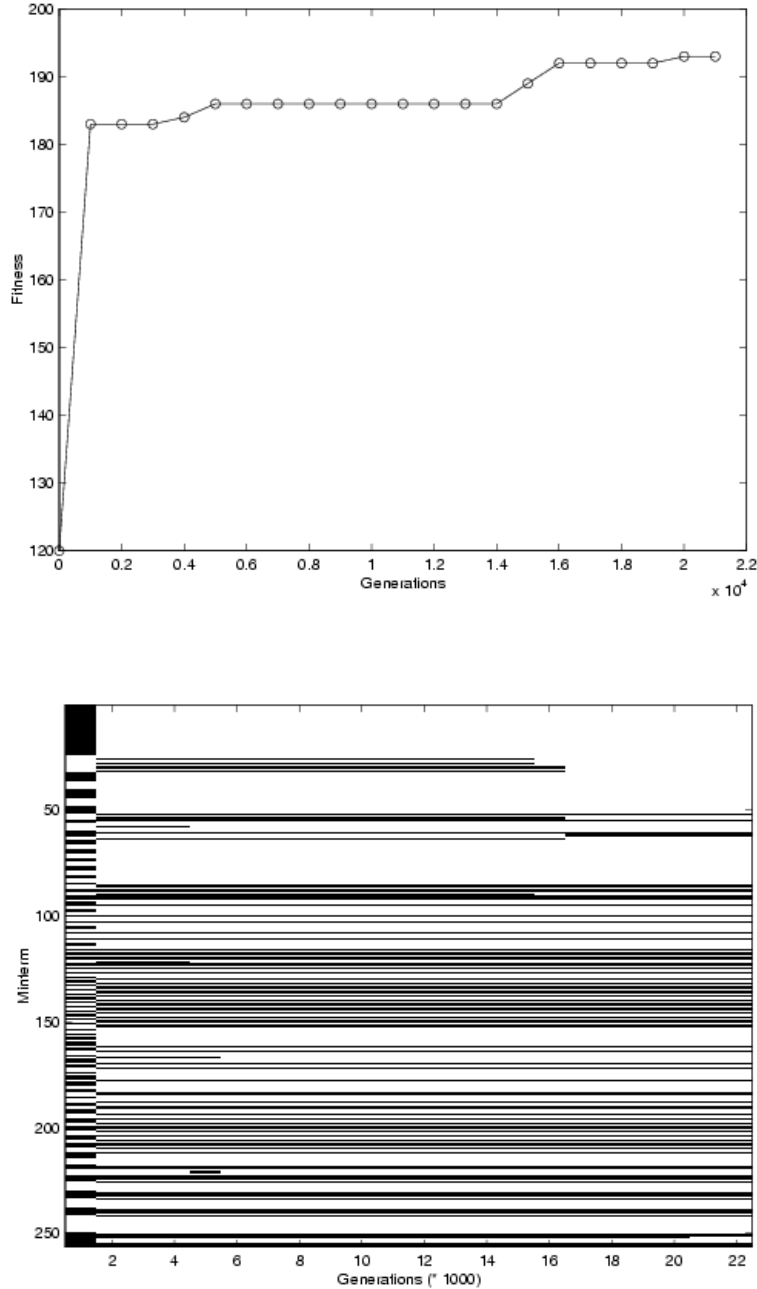


Figure 8.4: Results of long run. Top: Fitness curve, Bottom: Failed Minterms (in black). Both graphs are made by sampling the population every 1000 generations.

Figure 8.4 shows the result of this run, presented as a fitness curve, and as a plot of the failed minterms. It is apparent that improvements are still happening late in the run, even though the GA seemed to have stagnated after about 6000 generations (see top graph). This means that if the time gap between faults (MTTF) is large enough, seemingly difficult faults may yet be repaired.

The reason for the seemingly radical change in minterms after 1000 generations is the high sampling rate, every 1000 generations. The small, gradual changes happening over the first 100 generations or so seems like a big jump in this figure.

Experiment 2: Accepting Imperfect Repair

For a voting system to work well with damaged circuits, not completely repaired with evolution, the damaged circuits need to overlap each other's faults. In other words, it is important that the circuits don't fail in the same way. This experiment monitored what minterms failed on a simulation of a single FPGA, each run with a randomly picked fault.

Figure 8.5 shows the result of the 57 runs conducted. If a minterm in a certain generations is marked as completely white, no runs failed on that minterm. At the other end of the spectrum, completely black means all runs failed that minterm.

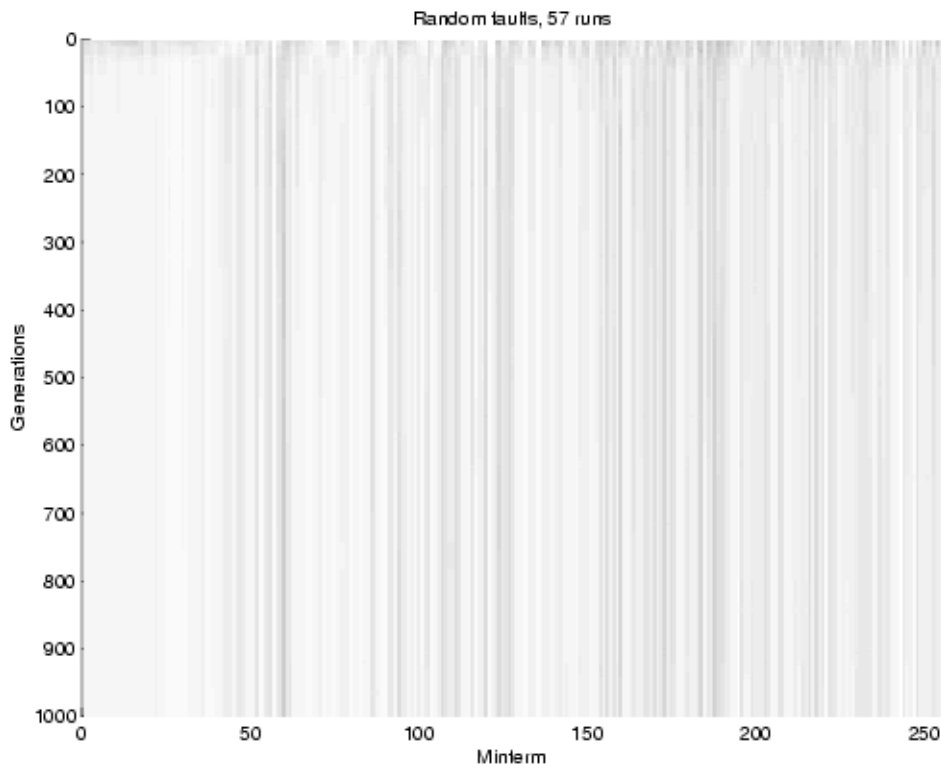


Figure 8.5: The failing minterms of several runs, shown in grayscale.

Some interesting things can be seen in Figure 8.5. Firstly, although some vague lines can be seen, the whole figure has a very light gray tone. This can easily be seen if compared to Figure 8.8. Secondly, two distinct phases can be seen. During the first 30 generations or so, a rather radical change can be seen. After that, the system stabilizes, and only now and then turns ever so slightly whiter.

The figure certainly shows that there may be hope for an evolutionary voting system, even after the majority of FPGAs are damaged.

Experiment 2.1: Voting System with Imperfect FPGAs

In this experiment, a voting system was implemented and tested. Three FPGAs were used in the system, and a different fault were introduced to each of them. This differs slightly from how the system would be used in "reality" - there would be a certain amount of time between each fault, and evolution could have started on the faulty FPGAs as they were marked faulty. This, however, allows us to see how the voting system performs in extreme cases.

The experiment was run 44 times, and most of the runs manage to find maximum fitness after about only 10 generations. Figure 8.6 shows a typical run of this experiment. The dashed lines are the individual FPGAs' performances, the solid line is the voting system's. This particular run achieved 100% fitness (256) after almost 350 generations. Note that an improvement of an individual FPGA does not always mean an improvement of the system. If the FPGA "traded" some minterms for others, the newly failing minterms may arbitrarily be minterms failing in another FPGA as well, thus decreasing the performance of the system.

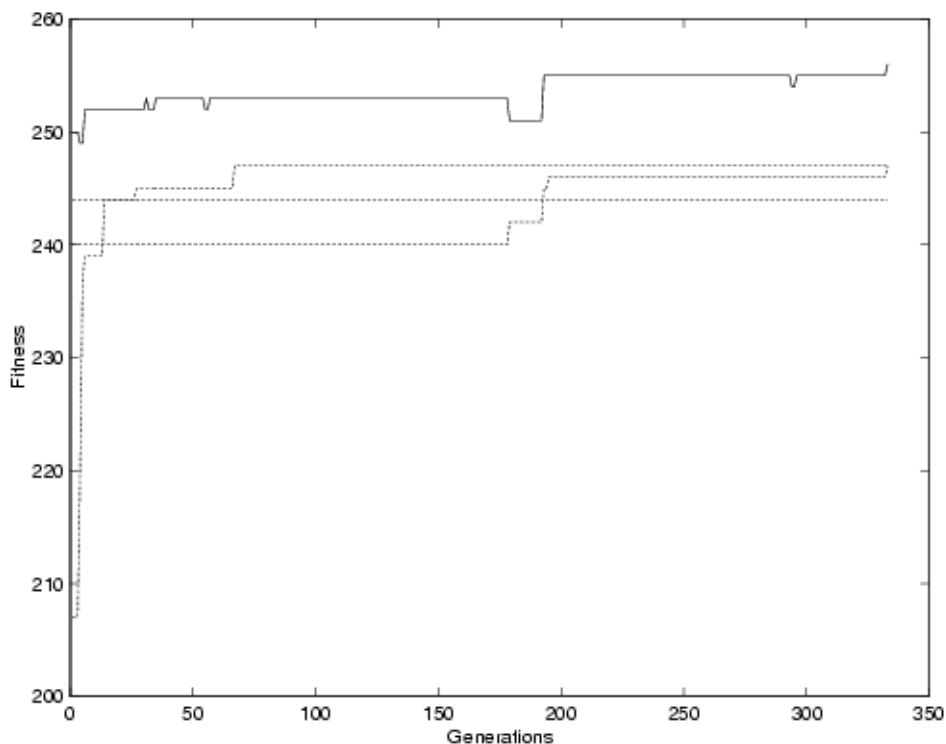


Figure 8.6: A run of the voting system. System fitness shown with solid line, individual FPGAs' fitness shown with dashed lines.

Another thing to note is that it is not guaranteed that the voting system will perform better than each of the FPGAs. Figure 8.7 shows this in a run where 100% fitness was not reached in the first

1000 generations. Nonetheless, in a more realistic use, the time gaps between individual faults in the FPGAs should be much larger, and so this effect may be much smaller.

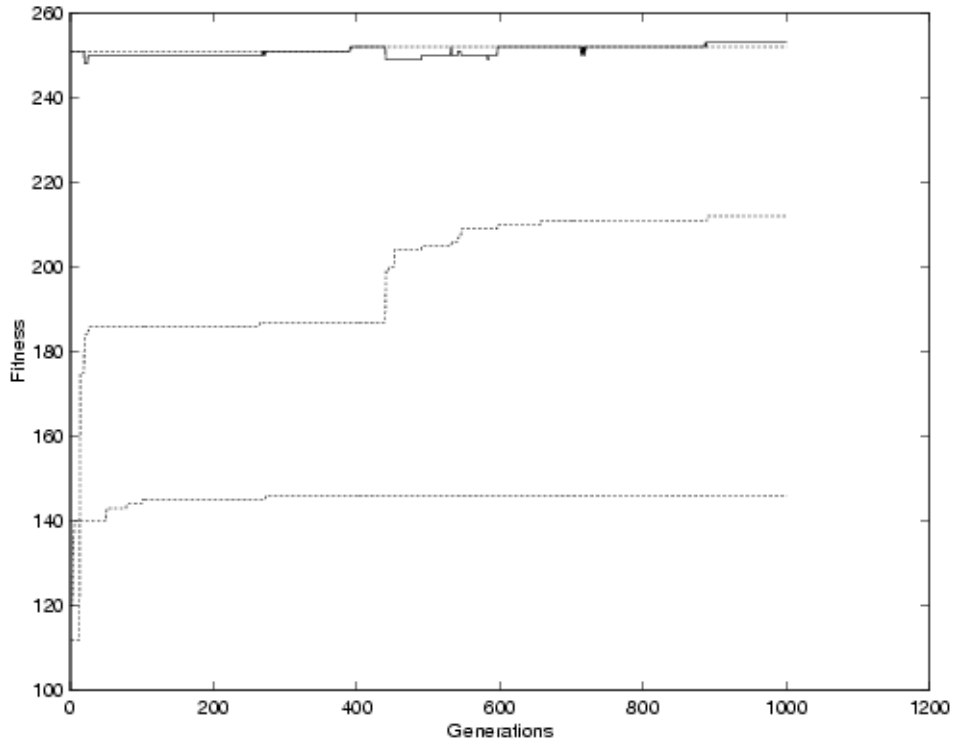


Figure 8.7: Another voting system run. The system (solid line) doesn't always perform better than its components (dashed lines).

This experiment has shown that it is indeed possible to improve a voting system with evolution.

Experiment 3: Repeats of Same Run

Trying to answer whether re-runs of evolution can be advantageous. This experiment looks at how many minterms that fail in the same way every time. Figure 8.8 is a grayscale figure, ranging from white where all runs managed a minterm, and black where none did.

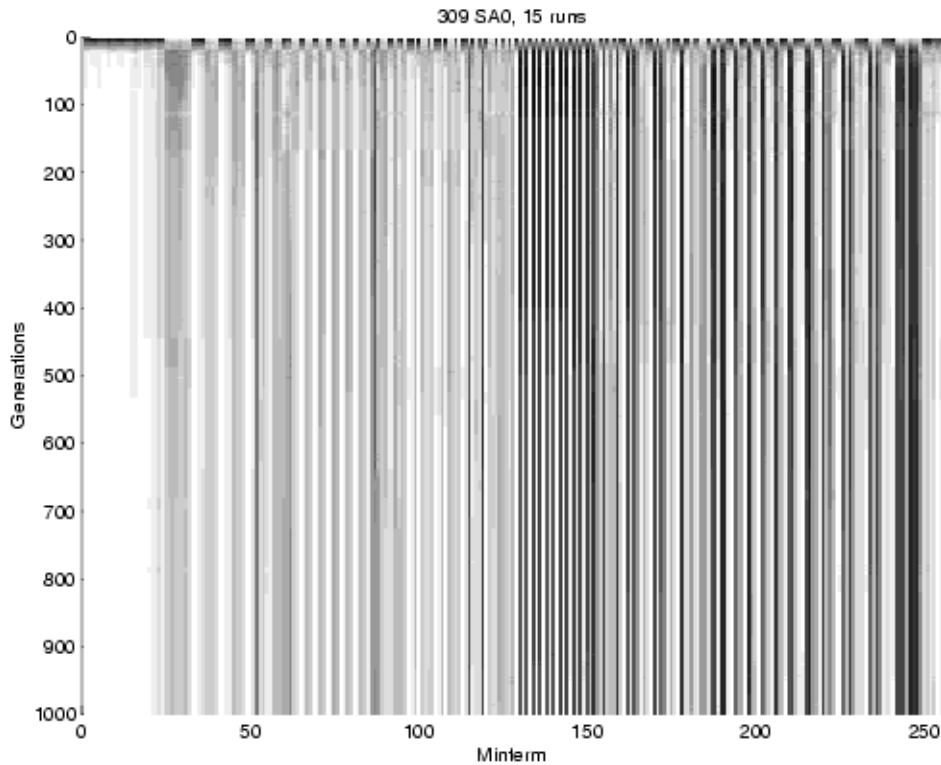


Figure 8.8: Grayscales showing failing minterms when all runs start with the same fault

The figure clearly shows that some minterms are more likely to fail than others. The result is of course dependent on this particular application. For this application, re-running the GA from scratch would not seem to achieve anything, since it is highly likely that it will fail in the same way again.

9.0 Conclusion

This chapter goes through the general things that have been learned from this project, and then goes on to discuss what can be done on the subject in the future.

9.1. General Conclusion

The field of fault tolerance in space applications was explored, with emphasis on the use of genetic algorithms. Several experiments were conducted around this issue, and a system for fault tolerance, enhancing an existing redundancy method (a voting system), was suggested.

The experiments show that evolutionary repair in itself is disappointing somewhat in finding perfect solutions. Nonetheless, the new proposed enhancement of a voting system, using evolutionary repair, have great potential for making a system more reliable. Some issues still have to be worked out, like exactly how to prevent the genetic algorithm itself from failing.

Also, an experiment indicates that for this particular application, re-running the genetic algorithm is not a useful option, since a similar solution appears each time evolution is run.

It should be remembered that all the experiments are performed with a simple simulation model of an actual FPGA, and some differences may occur if the same experiments are performed on a physical one. Nonetheless, it is the author's opinion that these experiments show that genetic algorithms can be beneficially applied to fault tolerance in space applications.

9.2. Further work

This section suggests what could be done in the future, to further investigate the use of genetic algorithms in voting systems.

- The GA used in this project has been a standard GA, with few alterations to suit the application. The effects of the special crossover and the cell swapping operators have not been properly investigated. In general, little "tweaking" of the parameters in the GA has been done to increase its performance. It is likely that adding application-specific knowledge to the implementation of the GA will be beneficial for the system as a whole.

- According to Experiment 1, 100% fitness was difficult to reach with the GA used. It may be interesting to see whether other search methods, like hill climbing, could be used in a hybrid algorithm to take the last step.

- The problems mentioned in Section 6.3. should be investigated more.

- Naturally, an interesting way of proceeding with this work, would be to perform experiments with a real FPGA. This would remove possible bad assumptions in designing the FPGA simulator.

Appendix A: Stuck-At Model

This appendix describes the fault model used, the single stuck-at fault model.

A.1. Assumptions

The single stuck-at model has a number of assumptions. Firstly, as the name suggests, it assumes only one fault occurs at a time. In other words, the time gap between faults occur should be big. Secondly, the fault is permanent, and affects the circuit as if a signal is tied directly to ground ("Stuck-At 0") or power ("Stuck-At 1"). Lastly, the logic gates in the circuit are unaffected by the fault (in a direct manner).

In spite of these seemingly tight assumptions, the stuck-at model is widely used. This is mainly because of its practicalities, it is "good enough" for most needs.

A.2. An Example

As mentioned, the model basically sees any fault as a short-circuit to ground or power (VCC). Figure A.1 shows a faulty cell, in which the signal 'A' is damaged, and as a result, is short-circuited to VCC. We say that A is stuck-at 1 (SA1).

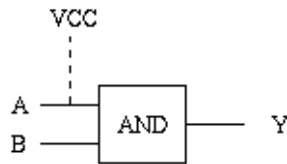


Figure A.1: Pin A is stuck-at 1 (A SA1)

Table A.1 shows the truth table for the example in Figure A.1. Note that the fault only affects the circuit if A is 0, and can only be seen on Y if, in addition, B is 1. This means that it may be difficult to observe a fault, and indeed, some times a fault may be unobservable.

Table A.1: The truth table for the circuit in Figure A.1

A	B	Correct Y	Y with A SA1
0	0	0	0
0	1	0	1
1	0	0	0
1	1	1	1

A.3. Advantages and Disadvantages

There are several advantages to this fault model that makes it very popular. These advantages include

Simplicity. Since a fault is defined as a stuck signal, only $2n$ possible faults exist, where n is the number of signals. This is a low enough number for it to be possible to look for all faults.

Abstractability. The model can be applied either on a logic level, or on a modular level.

Automatic testing. Algorithms are well developed for automatic pattern generation (ATPG) and fault simulation on this fault model.

Coverage. The model covers about 90% of all possible physical faults that can occur in a CMOS circuit. (This includes source-drain short circuits, oxide pinholes, missing features, diffusion contaminants, metallization short circuits and metallic drift.)

Usability. Other, more detailed models, like stuck-open and bridging faults, can be mapped down to sequences of stuck-at faults.

In spite of the good coverage, the biggest disadvantage of the stuck-at model, is that there are some defects that it cannot model well enough.

Appendix B: Multiplier Implementation

This appendix shows the exact details of the implemented multiplier.

B.1. SIS

SIS [21] is a tool for electronic design with features for optimizing digital logic. This project used SIS to optimize a truth table of the 4-bit multiplier and map it to the self-defined FPGA model. [18] explains how to use the different features in SIS.

The following sections goes through some of the steps in the process of implementing the multiplier.

B.2. Multiplier Truth Table Script

The truth table was generated with the following Perl script. The script is run once for each output bit, a total of 8 times. Each time, the output bit number should be given as argument.

```
#!/local/bin/perl

my $output = shift;

sub dec2bin {
    my $str = unpack("B32", pack ("N", shift));
    my $bits = 32-shift;
    $str =~ s/^[01]{ $bits }([01]*)/$1/;
    return $str;
}

print "# 4x4 bit multiplier truthtable, espresso format\n";
print "# Made by mult.pl (Sverre Vigander)\n";
print "# Output no $output, where 0 is LSB\n";
print ".i 8\n";
print ".o 1\n";

for ($m1 = 0;$m1<2**4;$m1++) {
    for ($m2 = 0;$m2<2**4;$m2++) {
        my $b1 = dec2bin($m1,4);
        my $b2 = dec2bin($m2,4);
        my $product = $m1*$m2;

        if (($product & 2**$output) == 2**$output) {
            print "$b1$b21\n";
        } else {
            print "$b1$b20\n";
        }
    }
}
# print "$b1$b2$b3\n";
}
```

B.3. Mapping

SIS uses a library format called Genlib, that lets the user define the target system. Using this format, it was possible to implement the multiplier using the FPGA model in this project. SIS produced the following output. Each line represents a cell in the FPGA, starting in the "upper left" corner, going down, then right (see Figure 7.1). The right side of each equation defines both the inputs to that node and the logical function, the first node being a 3-pin OR gate, for example.

```
[1641] = a0 + a1 + b0
[2110] = [1641]' + b1'
[1023] = a1'
[1026] = a0' + b0'
[2112] = [1023] + [1026]
[1635] = [2110]' + [2112]'
[2106] = [1635]' + a2'
[1639] = a0 + b0 + b1
[339] = [1639] a1
[1021] = b1'
[340] = [1021]' [1026]'
[1637] = [339] + [340] + a2
[2108] = [1637]' + b2'
[1633] = [2106]' + [2108]'
{p7} = [1633] a3 b3
[1030] = a3'
[617] = [1030]' a0'
[1022] = b0'
[1027] = b2'
[2080] = [1027]' + a2' + b0'
[1020] = a2'
[2082] = [1020]' + a0' + b2'
[1693] = [2080]' + [2082]'
[2076] = [1693]' + a1' + b1'
[2084] = [1023]' + [1026]'
[1800] = [1023] b1 + a1 b1'
[2102] = a1' + b0'
[2104] = a0' + b1'
[1802] = [2102]' + [2104]'
[7] = a0 b0
{p1} = [1800] [7] + [1802] [7]'
[1024] = a0'
[2086] = [1024] + {p1}
[135] = [2084] [2086]
[1037] = a2' + b2'
[2078] = [1037] + [135]
[126] = [2076] [2078]
[2000] = [126] + a3 + b3
[2092] = a3 + b3
[139] = [2092] a1
[2088] = [139] + b1
[142] = a0' b0'
[2090] = [142] + {p1}
[1689] = [2088]' + [2090]'
[1710] = [1037]'
```

[2002] = [1026]' + [1689]' + [1710]'
 [773] = [2000] [2002]
 [2032] = [1026]' + [1027]'
 [2028] = {p1} + a0 + a2
 [2030] = [1030]' + a0' + a2'
 [140] = [2028] [2030] [2032]
 [2008] = [1023] + [140] + b1
 [2016] = [1021] + [1037]
 [2018] = [1020]' + a1'
 [128] = [2016] [2018]
 [2012] = [128] + a0
 [2020] = [1023]' + {p1}' + a2'
 [2024] = [1020]' + [1026]'
 [2026] = {p1} + b0
 [1724] = [2024]' + [2026]' + b1'
 [2022] = [1724]' + a1'
 [133] = [2020] [2022]
 [2014] = [133] + b2
 [124] = [2012] [2014]
 [2010] = [124] + a3
 [1677] = [2008]' + [2010]'
 [2004] = [1677]' + b3'
 [2060] = [1023] + b0
 [1031] = b3'
 [2062] = [1031] + a0
 [1699] = [2060]' + [2062]'
 [2056] = [1699]' + a2'
 [2064] = [1026]' + b1' + b3'
 [2066] = a2 + b1 + b3
 [1705] = [2064]' + [2066]'
 [2058] = [1705]' + b2'
 [1691] = [2056]' + [2058]'
 [2038] = [1691]' + {p1}'
 {p0} = [1026]'
 [802] = [1037] {p0}
 [2068] = [1023]' + [802]' + b3'
 [796] = a0 b3
 [2072] = [1020]' + [796]' + a1'
 [2074] = [1021] + a1 + b0
 [136] = [2072] [2074]
 [2070] = [136] + b2
 [132] = [2068] [2070]
 [2040] = [132] + {p1}
 [778] = [2038] [2040]
 [1048] = [1037] + b0
 [2034] = [1023] + [1048] + b3
 [2042] = [1020]' + [1023]' + [1026]'
 [2050] = b0 + b2
 [2046] = [1023]' + b0' + b2'
 [2052] = [1026]' + [1027]'
 [2054] = {p1} + b0
 [144] = [2052] [2054] a1
 [2048] = [144] + a2
 [141] = [2046] [2048] [2050]

[2044] = [141] + b3
 [1685] = [2042]' + [2044]'
 [2036] = [1685]' + b1'
 [1681] = [2034]' + [2036]' + [778]'
 [2006] = [1681]' + a3'
 {p4} = [2004]' + [2006]' + [773]'
 [1942] = [1022]' + {p4}' + [617]'
 [1944] = a1 + a2
 [1651] = [1942]' + [1944]'
 [1063] = a3' + b1'
 [1974] = [1026] + [1063] + {p1}
 [1982] = a2' + b1'
 [1663] = [1982]' + a3'
 [1976] = [1023]' + [1663]' + {p4}'
 [649] = [1974] [1976]
 [1998] = a1 + a3
 [1675] = [1026]' + {p1}'
 [1994] = [1675]' + {p4}'
 [1996] = {p0} + b1
 [121] = [1994] [1996] [1998]
 [1978] = [1020] + [121] + b2
 [1990] = [1030] + a1 + b0
 [1992] = [1030]' + a0' + a1'
 [1669] = [1990]' + [1992]'
 [1988] = [1669]' + b2'
 [1071] = a2' + a3'
 [1984] = [1071] + {p1} + b1
 [1986] = [1023] + a2 + b2
 [118] = [1984] [1986] [1988]
 [1980] = [118] + {p4}
 [1657] = [1978]' + [1980]' + [649]'
 [1950] = [1657]' + b3'
 [1956] = a1' + b2'
 [1659] = [1956]' + b3'
 [1952] = [1021]' + [1659]' + {p4}'
 [1958] = [1031]' + a2' + b0'
 [1960] = a2 + b2
 [117] = [1958] [1960]
 [1954] = [1021] + [117] + {p4}
 [1653] = [1952]' + [1954]'
 [1946] = [1653]' + a3'
 [1972] = {p0} + b3
 [1671] = [1972]' + a1'
 [1968] = [1020]' + [1671]' + a3'
 [1970] = {p0}' + a2' + b1'
 [115] = [1968] [1970]
 [1966] = [115] + {p1}
 [1081] = [1020] + a3
 [1962] = [1081] + {p4} + b3
 [1964] = [1020]' + [1023]' + {p4}'
 [1655] = [1962]' + [1964]' + [1966]'
 [1948] = [1655]' + b2'
 {p5} = [1946]' + [1948]' + [1950]'
 [1924] = [1651]' + {p5}' + b3'

[1928] = [1031]' + a3' + b2'
 [1930] = [1030]' + a2' + b3'
 [106] = [1928] [1930]
 [1926] = [106] + {p5}
 [588] = [1924] [1926]
 [1932] = a1' + a3' + b1'
 [1936] = a0' + a1' + b3'
 [109] = [1936] {p5}
 [1934] = [109] + {p4}
 [1645] = [1932]' + [1934]'
 [1920] = [1645]' + a2' + b2'
 [1938] = [1021]' + {p5}'
 [1940] = [1020]' + b3'
 [1647] = [1938]' + [1940]'
 [1922] = [1027]' + [1647]' + a3'
 {p6} = [1920]' + [1922]' + [588]'
 [1096] = a2' + b1'
 [533] = {p0}' {p1}'
 [1848] = [1096]' + [533]'
 [1852] = b1 + b2
 [1854] = a1 + a2
 [158] = [1852] [1854]
 [1850] = [158] + b3
 [1736] = [1848]' + [1850]'
 [1846] = [1736]' + b0'
 [1856] = {p1}' + a2' + b2'
 [1858] = [1023] + [1096]
 [1744] = [1856]' + [1858]'
 [1842] = [1031]' + [1744]' + a0'
 [1860] = [1022] + [1031] + [1096]
 [1758] = [1860]' + a0'
 [1844] = [1027]' + [1758]' + {p1}'
 [1728] = [1842]' + [1844]' + [1846]'
 [1836] = [1728]' + a3'
 [585] = [1020]' b3'
 [1832] = [1022]' + {p1}' + [585]'
 [1916] = [1027]' + a1' + b1'
 [1106] = a1' + b2'
 [1773] = [1106]'
 [1918] = [1773]' + {p1}'
 [1750] = [1916]' + [1918]'
 [1912] = [1750]' + a3'
 [1914] = {p0} + a1
 [148] = [1912] [1914]
 [1107] = a0' + b3'
 [1834] = [1107] + [148] + a2
 [405] = [1832] [1834] [1836]
 [1878] = [1022]' + [1106]'
 [1880] = a1 + a3
 [157] = [1878] [1880]
 [1866] = [1096] + [157]
 [511] = a2 a3
 [1868] = [1023]' + [511]' + b2'
 [1870] = [1026]' + [1106]'

$[152] = [1868] [1870]$
 $[1862] = [1107] + [152]$
 $[1876] = [1031]' + a3'$
 $[165] = [1876] a0$
 $[1872] = [165] + a2$
 $[1874] = [1021]' + [1107]'$
 $[155] = [1872] [1874]$
 $[1864] = [1106] + [155]$
 $[146] = [1862] [1864] [1866]$
 $[1838] = [146] + \{p1\}$
 $[1910] = a2' + b0' + b3'$
 $[1756] = [1910]' + a0'$
 $[1882] = [1756]' + \{p1\}' + b2'$
 $[1890] = [1096] + a1 + b2$
 $[1892] = [1106] + a2 + b1$
 $[150] = [1890] [1892]$
 $[1884] = [150] + b3$
 $[413] = [1882] [1884]$
 $[1894] = a1' + a2' + b3'$
 $[447] = a2' b3'$
 $[1898] = [1027]' + [447]' + b1'$
 $[441] = [1020]' a1'$
 $[1779] = [1020] b3 + [441] b3'$
 $[1900] = [1779]' + b2'$
 $[161] = [1898] [1900]$
 $[1896] = [161] + \{p1\}$
 $[1740] = [1894]' + [1896]'$
 $[1886] = \{p0\}' + [1740]'$
 $[1906] = [1020]' + \{p1\}' + b1'$
 $[1908] = b1 + b2$
 $[154] = [1906] [1908]$
 $[1888] = [1107] + [154]$
 $[147] = [1886] [1888] [413]$
 $[1840] = [147] + a3$
 $\{p3\} = [1838]' + [1840]' + [405]'$
 $[171] = [1020] b1' + a2 b1$
 $[1826] = [171] + a0$
 $[1822] = [1027] + \{p0\} + a1$
 $[1824] = [1023] + b0 + b2$
 $[1794] = [1822]' + [1824]' + [1826]'$
 $[1808] = [1794]' + \{p1\}'$
 $[1810] = [1022]' + a1' + b1'$
 $[1814] = a2' + b0'$
 $[1816] = a0' + b2'$
 $[170] = [1814] [1816]$
 $[1812] = \{p0\} + [170]$
 $[167] = [1810] [1812]$
 $[1804] = [167] + \{p1\}$
 $[1792] = [1020] b2 + a2 b2'$
 $[1806] = \{p0\}' + [1792]'$
 $\{p2\} = [1804]' + [1806]' + [1808]'$

Appendix C: Glossary

This glossary explains some terms and words the way they are used in this report. Most terms are defined from an evolutionary hardware point of view. Words in italics have their own definitions in this section.

- Application:** The task a *genetic algorithm* is set to solve.
- Availability:** The percentage of time a system is working like it should. Defined as $MTTF/(MTTF+MTTR)$.
- Chromosome:** String of information defining an *individual*. Often, an individual's *genome* consists of several chromosomes (especially in natural genetics); but in artificial evolution, usually only one chromosome is used.
- Combinatorial:** A combinatorial circuit is an electronic circuit with no memory components.
- Crossover:** A *genetic operator* splicing two *individuals* together to form offspring. There are various ways of doing this; the simplest being to take 50% of one parent's genes and 50% of the other's.
- Environment:** The surroundings of an *individual*, sometimes defined by the *application*. The environment is the physical restrictions that are placed upon the *GA*, including things like noise or obstacles. If an *individual* is evaluated without an environment, only the *genotype* can be evaluated. If the *individual* is evaluated in an environment, the *phenotype* will be evaluated.
- Evolutionary Hardware (EHW):** The use of artificial evolution to develop electronic circuits (in hardware), using reconfigurable circuits (e.g. *FPGAs*).
- Field-Programmable Gate Array (FPGA):** A programmable logic device which is more versatile than traditional programmable devices such as PALs and PLAs, but less versatile than an application-specific integrated circuit. FPGAs are normally reprogrammable, which makes them especially interesting with respect to evolving circuits.
- Fitness Value:** Measurement of how good an *individual* is. In this project, fitness is always explicit - it is calculated and represented by a number. Can also be implicit, in which it is directly given by whether the *individual* lives or dies. Natural evolution uses implicit fitness, artificial evolution usually (but not always) uses explicit fitness through a *fitness function*.
- Fitness Function:** Explicit way of measuring fitness. The fitness function gives a direct relationship between the *genome* and the *fitness value*, evaluating how well an

individual performs in the *application*. It may score the *individual* for e.g. how fast it solves the problem.

Fitness Landscape: This term was first used by the biologist Sewell Wright in 1932. The fitness landscape is a visualisation of fitness values as a landscape. For example, high fitness values give mountainous peaks or hills in the landscape. Hence the common metaphor "hill climbing", meaning to find the nearest high fitness *individual* in the fitness landscape.

Generation: One class of *GAs* are generational. This means that their *population* is evaluated from generation to generation. In each generation, new *individuals* are made as offspring from the best ones in the last generation.

Genetic Algorithm (GA): A genetic algorithm is a search algorithm, often used at problems that can't be solved or will be difficult to solve using conventional methods. *GAs* were first created at the University of Michigan by John Holland [5], and the approach uses computers to simulate evolution through natural selection. In essence, a correct (or "good enough") answer to the problem gets "evolved."

Genetic Operator: Operators applied to *genomes*, usually as a new offspring is "born". Examples of genetic operators are *mutation* and *crossover*.

Genome: All the genetic material in an *individual*. Can consist of one or several *chromosomes*.

Genotype: The composition of an *individual's* genes. Influences *phenotype*.

Individual: A suggested solution, defined by its *genome*.

Initial Population: A *population* in *generation 0*, before the *GA* actually start. The initial population is often generated at random.

Locus: A position in the *chromosome*.

Mean Time To Failure (MTTF): The average time it takes for a system to break down.

Mean Time To Repair (MTTR): The average time it takes to repair a broken system.

Minterm: The logical AND of the variables associated with an input combination to a logical function. (Example: A function has two inputs, A and B. The minterms are AB, A'B, AB' and A'B'). The number of minterms are generally 2^n , where n is the number of inputs. The minterms can be expressed as binary numbers, ranging from $I_{n-1}'I_{n-2}'\dots I_1'I_0'$ to $I_{n-1}I_{n-2}\dots I_1I_0$, which can be translated into decimal numbers from 0 to 2^n-1 .

- Mutation:** Random change in a gene, often just a "bit-flip", changing a '1' to a '0' or vice versa.
- Mutation Rate:** The rate at which mutations occur. Often, the mutation rate is set to a certain probability per *locus* per *individual*.
- Phenotype:** The observable properties of an *individual*. Influenced by *genotype* and *environment*.
- Population:** A group of *individuals* in a *genetic algorithm*.
- Rank Selection:** A *selection method* where all *individuals* in a *generation* are sorted according to their *fitness values*. The chance of reproducing is then directly dependant on the *individual's* rank, as opposed to its *fitness value*. An advantage of this is that the *fitness values* can be scaled up and down without affecting the selection.
- Selection Method:** Method for deciding which *individuals* should be allowed offspring, based upon their *fitness values*. This project has used a *rank selection* method.

References

- [1] H. Castro, *Fault tolerance through reconfigurability: applications in space instrumentation*, Sussex thesis M0294051US
- [2] C. Darwin, *The Origin of Species by Means of Natural Selection*, Random House Inc., New York, 1859
- [3] J.J. Grefenstette, C.L. Ramsey, *An approach to Anytime Learning*. In Proceedings of the Ninth International Conference on Machine Learning, D. Sleeman and P. Edwards (eds.), pp 189-195, Morgan Kaufmann, 1992
- [4] I. Harvey, A. Thompson, *Through the Labyrinth, Evolution Finds a Way: A Silicon Ridge*, In: Proceedings of The First International Conference on Evolvable Systems: from Biology to Hardware (ICES96), LNCS 1259, Higuchi, T. and Iwata, M. (eds.), 406-422, Springer Verlag, 1997.
- [5] J. H. Holland, *Adaption in Natural and Artificial Systems*, University of Michigan Press, 1975
- [6] S. Jones, *The Language of the genes*, Anchor Books, ISBN: 0385474288, 1994
- [7] R. Katz, J.J. Wang, R.Koga, et al., *Current Radiation Issues for Programmable Elements and Devices*
- [8] Dr. T.S. Kelso, *Basics of the Geostationary Orbit*, Satellite Times, <http://celestrak.com/columns/v04n07/>
- [9] D. Keymeulen, M. Iwata, K. Konaka, et al., *Off-line Model-free and On-line Model-based Evolution for Tracking Navigation using Evolvable Hardware*, in Proceedings of the First European Workshop on Evolutionary Robotics, P. Husbands and J. Meyer (eds.), Springer Verlag: Paris, 1998
- [10] M. Landsverk, *Understanding the Evolution Process: A Case Study in Selection Evaluation*, dissertation at Computer Architecture and Design Group, Department of Computer and Information Science, NTNU, 1999
- [11] P. Layzell, A. Thompson, *Understanding Inherent Qualities of Evolved Circuits: Evolutionary History as a Predictor of Fault Tolerance*, in Evolvable Systems: From Biology to Hardware (ICES2000), J. Miller, A. Thompson, P. Thomson, T.C. Fogarty (eds.), Springer Verlag, 2000
- [12] D. Mange, M. Goeke, D. Madon, et al., *Embryonics: a new family of coarse-grained FPGA with self-repair and self-production properties*, in Towards Evolvable Hardware: The Evolutionary Engineering Approach, LNCS 1062, E. Sanches and M. Tomassini (eds.), Springer-Verlag: Berlin, pp. 197-220, 1996

- [13] P. Marchal, P. Nussbaum, C. Piguet, et al., *Embryonics: The birth of synthetic life*, in *Towards Evolvable Hardware: The Evolutionary Engineering Approach*, LNCS 1062, E. Sanches and M. Tomassini (eds.), Springer-Verlag: Berlin, pp. 166-196, 1996
- [14] M. Mitchell, *An Introduction to Genetic Algorithms*, Massachusetts Institute of Technology, ISBN 0-262-13316-4, 1996
- [15] NASA, *Basics of Space Flight*, <http://www.jpl.nasa.gov/basics/bsf-toc.htm>
- [16] C. Ortega-Sanchez, D. Mange, S. Smith, A. Tyrell, *Embryonics: A Bio-Inspired Cellular Architecture with Fault-Tolerant Properties*, in *Genetic Programming and Evolvable Machines*, ISSN 1389-2576, Wolfgang Banzhaf (ed.), Kluwer Academic Publishers, Vol. 1, pp. 187-215, 2000
- [17] C. Ortega-Sánchez, A. Tyrrell, *Fault-Tolerant Systems: The Way Biology Does it!*, in *Proceedings Euromicro 97*, Budapest, IEEE CS Press, pp.146-151, September 1997
- [18] E.M. Sentovich et al., *SIS: A System for Sequential Circuit Synthesis*, Memo no. UCB/ERL M92/41, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA, 1992
- [19] A. Thompson, *Evolutionary Techniques for Fault Tolerance*, in *Proceedings UKACC International Conference on Control 1996 (CONTROL'96)*, IEE Conference Publication No. 427, pp. 693-698, 1996
- [20] A. Thompson, *Evolving Fault Tolerant Systems*, in *Proceedings of the 1st IEE/IEEE International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications (GALESIA'95)*, IEE Conference Publication No. 414, pp. 524-429, 1995
- [21] University of California, Berkley, *Sis*, Design Technology Warehouse, <http://www-cad.eecs.berkeley.edu/Software/software.html>