

**Michael Martin Garvie**

**Gripping Evolvable Hardware**

Part II Computer Science

Gonville & Caius College

2001



# Proforma

Name : **Michael Martin Garvie**  
College : **Gonville & Caius**  
Project Title : **Gripping Evolvable Hardware**  
Examination : **Part II Computer Science, 2001**  
Word Count : **11280**  
Project Originator : M. M. Garvie  
Supervisor : Dr S. W. Moore

## Original Aims of the Project

### 1. Core Section

- (a) Develop an evolutionary system for developing hardware intrinsically and extrinsically.
- (b) Use this to evolve circuits to perform simple tasks such as:
  - i. a D-Latch.
  - ii. an arbitrary Boolean function of many inputs.

### 2. Extensions

- (a) Replicate more complicated experiments done previously by Thompson [11]. Specifically to evolve circuits that:
  - i. generate an output signal of low frequency.
  - ii. differentiate an input frequency of 1Mhz from an 100Hz one.
- (b) Make evolved circuits resistant to changes in temperature.
- (c) Attempt to evolve circuits that perform more ambitious tasks such as:
  - i. simple image recognition.
  - ii. simple voice recognition.

## **Work Completed**

The core and extensions a.i, b, c.i and others.

## **Special Difficulties**

None.

## **Declaration of Originality**

I Michael Garvie of Gonville & Caius College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Preparation</b>	<b>3</b>
2.1	Overview . . . . .	3
2.2	Proposal Refinement . . . . .	3
2.3	Requirements Analysis . . . . .	4
2.4	Directed Research . . . . .	4
2.4.1	Genetic Algorithms . . . . .	5
2.4.2	Programmable Hardware Properties . . . . .	10
2.4.3	Parallel Port Communication . . . . .	11
2.4.4	Other Programming/Scripting . . . . .	13
2.4.5	Miscellaneous . . . . .	13
2.5	Strategies . . . . .	13
<b>3</b>	<b>Implementation</b>	<b>15</b>
3.0	Overview . . . . .	15
3.0.1	Development Cycle . . . . .	15
3.0.2	System Structure . . . . .	18
3.0.3	Glue . . . . .	18
3.1	Genetic Algorithms — package <code>es.evolve</code> . . . . .	18
3.1.1	Aim . . . . .	18
3.1.2	Algorithm Properties . . . . .	19
3.1.3	Product . . . . .	20
3.2	Deployment — package <code>es.deploy</code> . . . . .	20
3.2.1	Aim . . . . .	20
3.2.2	Circuit Encoding . . . . .	20
3.2.3	Extrinsic Deployment — The Logic Simulator . . . . .	22
3.2.4	Intrinsic Deployment — Interacting with the FLEX . . . . .	23
3.2.5	Product . . . . .	25
3.3	Experiment — package <code>es.experiment</code> . . . . .	27

3.3.1	Aim . . . . .	27
3.3.2	Fitness Functions . . . . .	27
3.3.3	The test inputs . . . . .	29
3.3.4	Fitness Functions and Test Inputs used for the Experiments . .	30
3.3.5	Allowing for Time Delays . . . . .	35
3.3.6	Product . . . . .	35
3.4	Control — package <code>es.control</code> . . . . .	36
3.4.1	Aim . . . . .	36
3.4.2	Distributed Coevolution . . . . .	36
3.4.3	Product . . . . .	39
3.5	Testing — package <code>es.testing</code> . . . . .	39
3.6	External — package <code>es.external</code> . . . . .	39
3.6.1	Statistical Functions package . . . . .	39
3.6.2	Billy the Logic Simulator part 1B Project India 2001 . . . . .	40
3.6.3	Class File Server . . . . .	40
3.7	Overall Product . . . . .	40
<b>4</b>	<b>Evaluation</b>	<b>41</b>
4.1	System Testing . . . . .	41
4.1.1	Parallel Port Interface . . . . .	41
4.1.2	Fitness Proportionate Selection . . . . .	42
4.1.3	Simulator & Verilog . . . . .	42
4.1.4	D-Latch Experiment . . . . .	43
4.1.5	Evolver . . . . .	43
4.1.6	Deployment-Experiment Integration . . . . .	43
4.1.7	Overall System Integration . . . . .	44
4.1.8	Distributed coevolution . . . . .	44
4.2	Experiments . . . . .	45
4.2.1	Getting started — AND gates . . . . .	47
4.2.2	Core Objective (i) — D-Latches . . . . .	48
4.2.3	Core Objective (ii) — Complex Boolean Functions . . . . .	49
4.2.4	Extension — Multiplexers . . . . .	50
4.2.5	Extension — Oscillators . . . . .	50
4.2.6	Extension — Simple Vision . . . . .	54
4.3	Overall . . . . .	56
<b>5</b>	<b>Conclusions</b>	<b>59</b>
<b>A</b>	<b>Appendix</b>	<b>63</b>

# Glossary of Terms

**AI** Artificial Intelligence.

**CPLD** a Complex Programmable Logic Device is different from an FPGA in that its interconnections aren't fully programmable.

**ECAD** Electronic Computer Aided Design.

**ECP** Extended Capabilities Port very much like EPP but can use DMA and a FIFO buffer.

**EH** Evolvable Hardware.

**EPLD** Erasable Programmable Logic Device.

**EPP** Enhanced Parallel Port is a standard signaling method for bi-directional parallel communication between a computer and peripheral devices.

**FPGA** Field Programmable Gate Array.

**GA** Genetic Algorithms.

**Gene** Unit of information in the genotype, may be composed of many symbols.

**Genotype** Encoding of an individual in a sequence of symbols.

**IDE** Integrated Development Environment.

**Jam STAPL** The Jam<sup>TM</sup> STAPL is an interpreted language optimized for programming PLDs via the JTAG interface.

**Javacomm** "The Java<sup>TM</sup> Communications API can be used to write platform-independent communications applications for technologies such as voice mail, fax, and smartcards." <http://java.sun.com/products/javacomm/index.html>

**JNI** Java Native Interface. “The JNI allows Java code that runs within a Java Virtual Machine (VM) to operate with applications and libraries written in other languages, such as C, C++, and assembly. In addition, the Invocation API allows you to embed the Java Virtual Machine into your native applications.”

**JTAG** Joint Test Action Group, or IEEE Standard 1149.1 is a standard specifying how to control and monitor the pins of compliant devices on a printed circuit board.

**Phenotype** Incarnation of a genotype into a deployed individual.

**RMI** Remote Method Invocation.

**SAGA** Species Adaptation Genetic Algorithm [5].

**SPP** Standard Parallel Port or Centronics is an early standard for transferring data from a host to the printer.

**UART** Universal Asynchronous Receiver/Transmitter.

**UML** “Unified Modeling Language is a modeling language for specifying, visualizing, constructing, and documenting the artifacts of a system-intensive process.”

# Chapter 1

## Introduction

I was first exposed to the idea of Evolvable Hardware (EH) five years ago when reading a New Scientist magazine article [9] describing Adrian Thompson's work in which an array of a 100x100 logic gates was configured to perform simple speech recognition. This inspired me to believe in the power of the 'hidden' analog properties of chips as the silicon counterpart of biological systems based on analogue processing. If these could be exploited fully then perhaps we could evolve systems with similar efficiency and robustness to those found in nature. Also, using more resources of the chip would mean each chip would be more powerful, which coupled with biological-like properties made Artificial Intelligence (AI) — albeit comparable to that of a snail — seem a not so far fetched goal. After reading more papers on the subject by researchers around the world it was clear that the field was emerging, promising vast fertile unsearched space. It was also clear there would be many technical difficulties and possibly no results. However I thought this was the most promising method of possibly achieving AI, having read about other methods like the massive Neural Network project called CAM-Brain, hard-coded rule systems, and self-replicating viruses. Even then, before setting myself out into evolving AI circuits it would be necessary to get to grips with GAs and their application to hardware.

Genetic Algorithms (GA) have been used for solving problems where conventional methods have failed because they are NP complete, of overwhelming complexity, dynamic or ill-defined. GAs provide a unified framework which is increasingly accepted as an extra tool-case in fields such as optimizing functions, coding AI routines, designing real-time control systems, generating timetables. Other examples of dynamic and ill-defined problems are image recognition, system robustness under component failure, robotics, economic modelling, etc...

Hardware fits into this group for two reasons. On one hand if we wish to make use of analog properties the problem is ill-defined because we don't know the exact properties of chips, and even if we did, then making use of them when designing large-scale

circuits would be extremely complex using existing tools. On the other hand, there is great interest [8] in the use of self-reconfiguring hardware in real-time controllers to make use of their unparalleled reaction time of around ten nanoseconds. This is a dynamic problem, in which the controller would have to adapt to hardware failures and new conditions. Such as a Mars Explorer controller having to adapt to moving with one damaged wheel by cutting its power and increasing that of its neighbours, or adapting to a different gravity, ground friction, or air viscosity than expected and learning new ways of operating effectively under them. Moreover, in applications like rocket booster controllers, nanoseconds can make a large difference, which is one of the reasons NASA is funding research in evolvable hardware.

Evolvable hardware is currently in an exploration stage as a field. Many papers have been published which approach the subject from very different angles and there isn't yet an established set of best recognized methodologies. The purpose of this project is to be introduced to the subject by exploring some of these methods, gain a better understanding of the problems involved and possibly attain a vision for future progress in the field.

# Chapter 2

## Preparation

### 2.1 Overview

As was mentioned earlier, this project was meant as an introductory exploration of the different approaches, methodologies and applications involved in EH. This exploration progressed through many areas of Computer Science including ECAD, hardware soldering, digital electronics, evolutionary programming, software engineering, Java and C programming, distributed computing and function analysis. This chapter will describe why these topics were covered and the strategies employed while developing the system that applied them.

### 2.2 Proposal Refinement

After hearing my high flying vague ideas, my supervisor Simon Moore thought I should attempt a third year project that could be completed. This is how the project title was transformed from “Towards AI” to “Gripping EH”. So the proposal was for me to get to grips with evolving hardware intrinsically (on the hardware itself) and extrinsically (on a simulator). The hardware used would be the Altera FLEX Field Programmable Gate Array (FPGA) on the Computer Lab Teaching Boards used for the ECAD practicals. The simulator and related software would be written in Java. GAs would be used for evolving the genotypes representing the circuits (phenotypes). The proposed goal was to develop a framework in which hardware could be evolved and then use this to evolve simple circuits — stateless functions of many inputs, D-Latches — as the core of the project and then move on to more complicated ones — oscillators, image recognizers, neural net neurons as the extensions.

## 2.3 Requirements Analysis

The evolving system framework needed to have the following properties:

- a flexible Evolver module to configure the evolutionary process with different combinations of genetic operators and their properties, in order to study them and use the most efficient ones.
- a flexible Deployment module to allow circuits to be deployed intrinsically and extrinsically.
- a flexible Experiment module so that the system could be used to evolve circuits with different functionalities.
- a Control module that would manage the information flow between the previous three.

An analogy I found useful was to imagine the individuals in the Evolver as sports players, the Deployment as a sports pitch, and the Experiment as the rules of a game. So any player can be evaluated on any pitch using the rules of any game. An example being a table-tennis player on a basketball court playing water-polo which may not work very well, but a badminton player on a table tennis table playing tennis may kind of work. In the same way a genotype which represents an XOR circuit when deployed on some simulator would be relatively good as a 2 bit adder as well.

It was one of the aims of this project to make the evolutionary system framework developed flexible enough to evolve individuals (players) for any purpose (game) to be deployed anywhere (pitch). In this way the system could be extended to evolve anything from timetables implemented on a scheduling software to AI routines for a Pac-Man player implemented on hardware chips as well as the simulated and hardware deployed circuits of this project.

## 2.4 Directed Research

Various issues had to be researched before starting the design and specification of the system. Others needed only be researched before the implementation of the module they belonged to. This is why not all subjects listed here were researched strictly before coding. For example the nature of the programming of the FLEX chip wasn't necessary for developing the GA, experiments and simulator modules; only the interfaces needed be defined.

### 2.4.1 Genetic Algorithms

M. Tomassini (“Evolutionary Algorithms” [8]) describes a GA as “an iterative procedure that consists of a constant-size population of individuals, each one represented by a finite string of symbols encoding a possible solution in some problem space. This space, also known as the *search space*, comprises all possible solutions to the problem at hand.” He then proceeds to give a schema summarizing the standard GA:

```
produce an initial population of individuals
evaluate the fitness of all individuals

while termination condition not met do

    select fitter individuals for reproduction
    recombine individuals
    mutate some individuals
    evaluate the fitness of the new individuals
    generate a new population by inserting some new good individuals
    and by discarding some old bad individuals

end while
```

The termination condition mentioned above is usually met when a satisfactory solution to the problem at hand is found or the maximum number of generations has been reached. If we generalize mutation and recombination to instances of genetic operators (of which there could be other yet unknown ones) the schema can be simplified to:

```
produce an initial population of individuals

while termination condition not met do

    evaluate the fitness of new individuals
    select fitter individuals for reproduction
    apply genetic operators to individuals adding the results to
    the new population

end while
```

This schema is visualized in Figure 2.1 using a binary symbol set.

The GA<sup>1</sup> was discovered by John Holland in 1975 with the main purpose of function optimization problems such as finding the  $\max(f(x))$  where  $f(x)$  needed not

---

<sup>1</sup>Most information on GAs was acquired from [8], [7], [11] and the comprehensive [2].

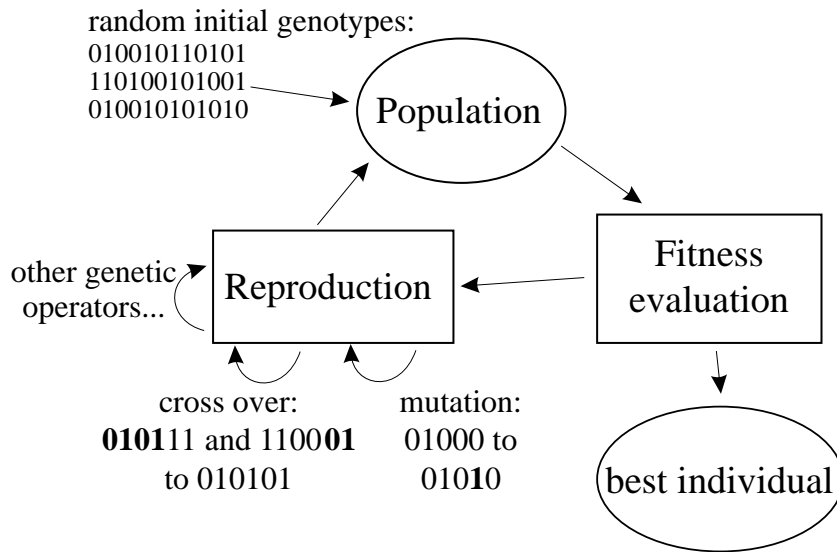


Figure 2.1: The Genetic Algorithm

be continuous or differentiable or even explicitly closed-form. Later on other people applied the idea to searching a wide range of problem spaces in fields like game strategy, economics, engineering optimization, battlefield communications and chemical processing.

The GA explores a *fitness landscape*, which is the n-dimensional search space with an associated height at each point (individual) corresponding to its fitness, in a process called *genetic drift*. The maxima of this landscape is the desired solution to our problem. However, there usually are local maxima and ridges. Holland's *Schema theorem* [6] shows that during the execution of the GA, the best and average fitnesses of the population improve in a better than random way. Genetic drift sometimes occurs along ridges in the landscape where the genotype changes but the fitness doesn't. This process is called *neutral drift* and is essential for exploring the landscape from a local maxima to a higher maxima.

There are many variants of the standard GA, one example being being Species Adaptation Genetic Algorithms (SAGA) [5], which could be applied for evolving circuits of variable size. Note that the GA is inherently parallelizable since the fitness evaluation of individuals is independent from each other. Hence different portions of a population could be evaluated simultaneously on different processing units. Island based coevolution is a model in which this is achieved by allowing whole populations to evolve semi-independently of each other on "islands", with individuals migrating between them sporadically.

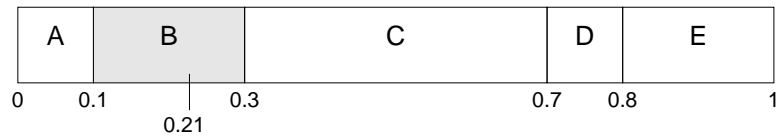


Figure 2.2: Example of fitness proportionate selection showing a cumulative fitness graph of individuals  $A, B, C, D, E$  with fitnesses 1,2,4,1,2 respectively. The random number generated is 0.21, selecting individual  $B$ .

Another variant of the standard GA is GP. GAs were also chosen above GP for the following reasons. First, the GA binary string genotype could be used to program the hardware directly. Secondly, GAs are more efficient using simpler more compact data structures than GP. Thirdly, there is no need for GP when the problem space is constrained, as in the circuits evolved for this project. Fourthly there always exists a mapping from a variable length bit-string to a tree. Last but not least, the simple way of using GPs to evolve circuits — tree with  $Nodes = \{\text{NAND}\}$ ,  $Terminals = \{0, i_0 \dots i_n\}$  — doesn't allow for circuits with feedback.

### Selection

This is a policy for choosing individuals from a population — usually the fittest ones for reproduction when creating the next generation. Various methods exist:

- *Fitness proportionate*: The probability of an individual to be chosen is directly proportional to its fitness as a fraction of total population fitness. To implement this a cumulative fitness table is created and adjusted so that the total fitness — equal to the last value in the table — is 1. Then a number is picked from a continuous  $Uniform(0, 1)$  probability distribution and the first position in the table whose value exceeds this number is the individual selected. See Figure 2.2 for an example. This method is effective but favours outstanding individuals too little when all have similar fitnesses and too much when the outstanding one is much fitter than the rest. The latter circumstance could annihilate genotype variety and make the population converge too fast on a local maxima, while the former circumstance would inhibit final convergence towards the top of a gentle slope.
- *Rank based*: The probability of an individual been selected depends on its position in the fitness ladder. The probabilities of each position are predefined. Note this method can also be used to discard individuals, like  $A$  and  $B$  in Table 2.4.1. It doesn't matter how much better  $C$  is, it will still have the same reproductive advantage, solving both problems of fitness proportionate selection. However it ignores the potentially useful relative fitness information.

Rank	Probability	Individual
1	0.66	<i>C</i>
2	0.23	<i>B</i>
3	0.1	<i>E</i>
4	0	<i>A</i>
5	0	<i>D</i>

Table 2.1: Predefined rank based selection probabilities and individuals they'd apply to from population of Figure 2.2

- *Tournament*: A number  $n$  (tournament size) of individuals are picked randomly with uniform distribution from the population and the one with highest fitness is the selected one. This is useful when we can't compute absolute fitness, but can compute relative fitness.
- *Elitism*: this can be used in conjunction with any of the selection methods mentioned above. It consists of copying through to the next generation some number of the best individuals and killing off some number of the worst ones. This can be done to make sure the evolution process never goes backwards by losing a very good individual.

### Genetic Operators

Genetic Operators take a number of individuals and create a new one from them. There are two main methods of applying GO's: one after another as in the first schema in §2.4.1 or assigning each GO to a different proportion of the population. The former allows the possibility of a GO acting twice on a genotype between generations while the latter allows for more control over how many individuals are affected and how. There are two main known types of GOs:

- *Recombination*: This is the main operator used to converge a population to some maxima. It operates on two or more genotypes analogous to DNA strands and sexual reproduction. The main variants are:
  - N-point cross over: Two genotypes are chopped at  $n$  random positions  $p_1 \dots p_n$ . The offspring is generated by alternating the pieces of each parents genotype as in Figure 2.3. If we define  $p_0 = 0$  and  $p_{n+1} = \text{genotypelength}$  then we have symbols  $p_k$  to  $p_{k+1}$  of the offspring coming from the first parent if  $k$  even and from the second if  $k$  odd. A variant of this operator uses  $2 < \text{parents} \leq n + 1$ .

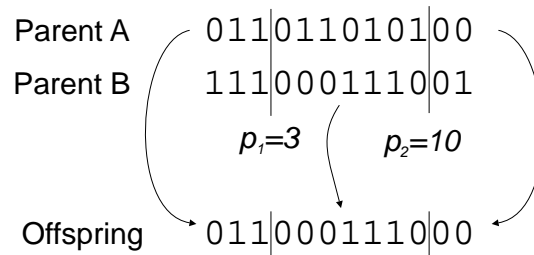


Figure 2.3: Example of n-point cross over with  $n = 2$ .

- Single point cross over: A special case of n-point cross over with  $n = 1$ . Most commonly used since simpler and more effective at keeping large contiguous pieces of genotypes — which are responsible for phenotypic features — together.
- Uniform cross over: Another special case of n-point cross over, this time with  $n = \text{genotypelength} - 1$  so each symbol of the offsprings' genotype can be independently copied from any of the parents.
- Mutation: Operates on one genotype randomly altering some of its symbols. This is analogous to strands of DNA being modified by ultra violet light or being miscopied during cell reproduction and then being transmitted to the offspring. With recombination alone, genetic drift would often get stuck at local maxima in the fitness landscape, but mutation remedies this by pulling individuals away from convergence points. However too much of it would not allow the population to converge at all which would result in random search. The suggested mutation rate per symbol value for keeping quasi-species together [4] is the reciprocal of the genotype length. There are two main types of mutation:
  - Per symbol: each symbol of a genotype is altered with a probability defined by the bit mutation rate.
  - Per genotype: the number of symbols mutated per genotype is specified. This can be the exact amount or the mean of some probability distribution. The position of the symbols flipped is chosen from a uniform distribution. Note that a rate of one per genotype is roughly equivalent to the optimal mutation rate mentioned above.
  - Adaptive Mutation: this can be applied to any of the previous schemes. It works by changing the mutation rate based on the population diversity calculated by taking the average Hamming distance between all individuals. If this distance is low then the population may have converged on a local

maxima and mutation is increased. When the diversity gets too high, it may not be converging properly and mutation is dropped.

### Examples of parameters which have been used during evolution

Parameters controlling the evolutionary process can be optimized for different experiments. Those used by different people for experiments similar to this project's was researched:

Purpose	Symbols	Pop Size	Selection	Cross Over	Mutation
Oscillators and robot controllers intrinsically [11]	Binary	30	Rank based with elitism	Single Point, 70% of pop.	Rate: $6.0 \times 10^{-4}$ per bit.
Optimizing a function [8]	Binary	50	Fitness prop.	Single Point, 60% of pop.	Rate: 0.01 per bit
Oscillators intrinsically [12]	Binary	3	Only fittest	none	One per genotype
Various experiments using GP [7]	Program nodes and terminals	500	Fitness proportionate	Branch cross over, 60% of pop.	none

### 2.4.2 Programmable Hardware Properties

The Altera FLEX (EPF10K20RC240-4) available on the Computer Laboratory teaching boards was a good choice of chip because of its large size having  $\approx 20000$  gates compared to  $\approx 1024$  for the Xilinx XC6216 used by Thompson [10], rich connectivity to other elements on the teaching board — useful for I/O and debugging — and a complete set of tools for configuring it.

A major drawback is the fact that it is difficult to configure directly without compilation [10] since Altera keeps the mapping from programming file to circuit structure private. Also, an SRAM LUT based CPLD like the FLEX shown in Figure 2.5 may exhibit different analog properties from those of the Xilinx FPGA shown in Figure 2.4.

I/O from the PC to the FLEX could be performed by going through the ARM on the board, using PC controlled custom hardware [13] like tone generators and analogue integrators to generate inputs and process outputs, using a Verilog implementation of a UART with a voltage adapter, or a direct parallel port connection to FLEX user I/O pins.

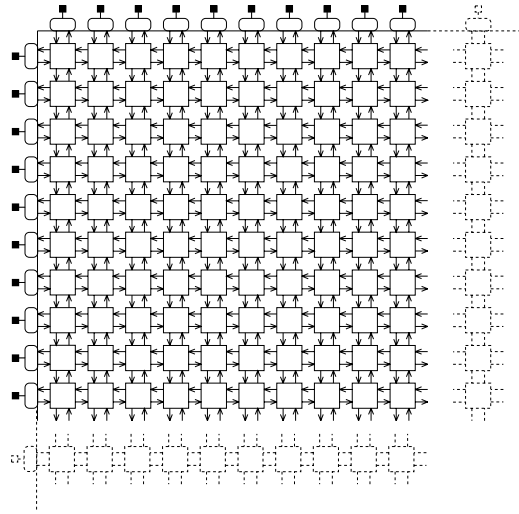


Figure 2.4: Simplified view of a corner of the Xilinx XC6216 FPGA composed of an array of cells.

### 2.4.3 Parallel Port Communication

A direct parallel port connection was chosen to communicate with the FLEX because it was the simplest most direct method having an extra port available. This connection involved building a hardware interface between the FLEX and the parallel port, and then a software interface between the parallel port and Java, as in Figure 2.6.

#### Hardware interface

The EPP and ECP protocols were researched but Javacomm libraries didn't implement them or any kind of parallel port input. To make both interfaces simpler, SPP mode was chosen restricting FLEX → bus PC width to five bits. Soldering skills were learnt before building the cable.

#### Software interface

On the software side there would have to be a two way communication path between Java and the parallel port registers. This was implemented in native code and linked into Java through JNI. GNU gcc and Microsoft Visual Studio were used in conjunction with the Microsoft Drivers Kit to compile the .DLL.

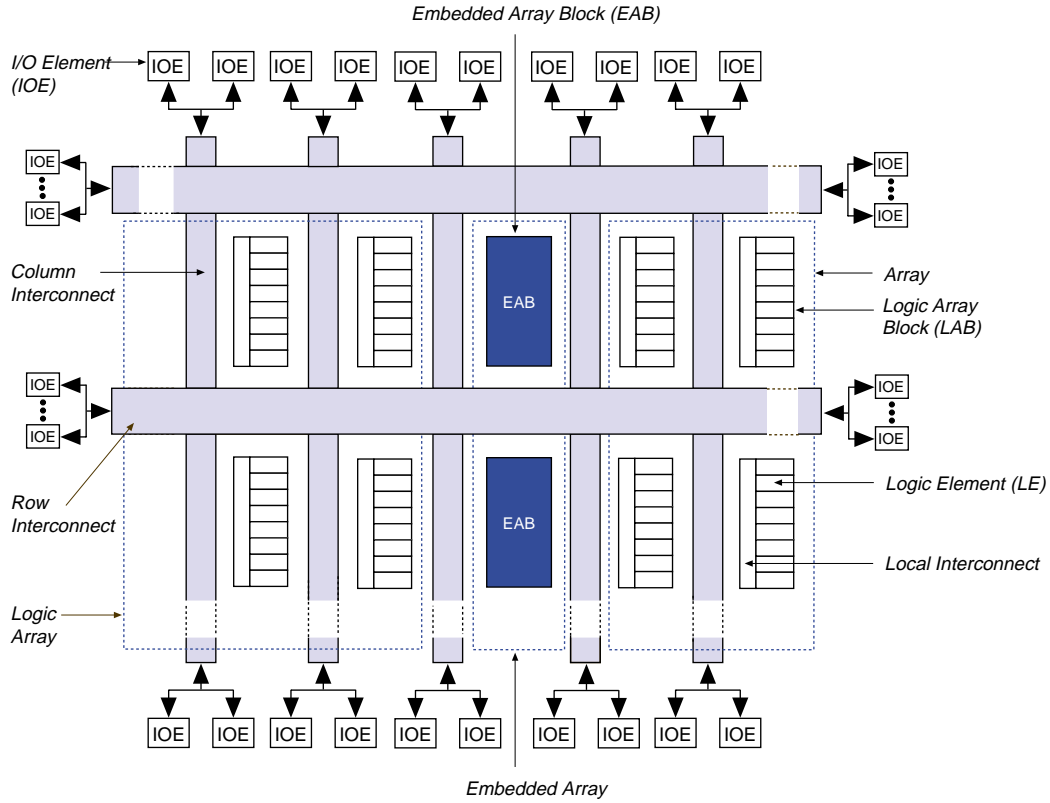


Figure 2.5: The FLEX EPF10K device block diagram made up of logic elements and interconnects. Diagram extracted from [3].

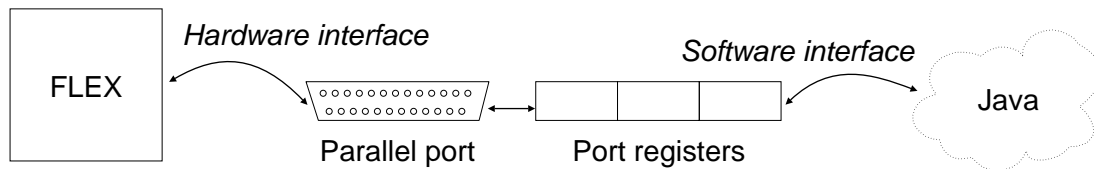


Figure 2.6: Schematic of parallel port communication between Java and the FLEX.

### 2.4.4 Other Programming/Scripting

- ARM programming: how to write to the UART in assembler and C in order to communicate with the PC.
- RMI programming: how to use remote objects to perform distributed computing.
- C programming: basic skills to program a library.
- Java data representation: The `java.util.BitSet` class (using a long internally) was found to use only 0.2MB of memory for allocating  $2^{20}$  bits while a `boolean` array used 2.1MB. The former also has a mutable length.
- a Verilog implementation of a UART was found as an alternative method for PC  $\leftrightarrow$  FLEX I/O.
- Macro Recorder: a shareware macro editor tool was used for which a script was written to manipulated MAX+plus II to invoke the Programmer.
- Billy The Digital Logic Simulator Part 1b Group Project India: This group kindly allowed me their source code to connect it to my project to visualize how evolved circuits functioned. Their Javadoc and comments in source code was read to bridge the circuit representations.

### 2.4.5 Miscellaneous

- MAX+plus II tools: how to control from Java through command line execution to compile Verilog and generate Jam byte code for it.
- Jam STAPL Byte-Code Player: how to control to configure the FLEX.
- CVS: how to build a repository and access it using WinCVS.
- Backup: how to automate a daily backup to the Pelican server.
- Books related to the field [14] [1]. What other people have achieved in Artificial Life through totally different methods, where they have failed and succeeded, how to link these methods with EH.

## 2.5 Strategies

Throughout the development of the system a set of design strategies was employed:

- only using constants and hard-coding where necessary by making the system fully configurable through parameters and constructors.
- making clean interfaces and module implementations independent of each other.
- making the system extensible by trying to keep in mind the most general case of what's been implemented instead of the particular case currently needed.
- keeping it simple.
- keeping in mind integration issues between components and their testing harnesses, and between components themselves.
- commenting the code as its written aimed not only at myself but at anyone else who may want to make use of the system.

Most of these strategies were aimed at making the system framework reusable by anyone researching or applying GAs, so that they could extend it to for example evolve timetables inside schedules or AI monsters inside different games. The independence of functionality between modules also makes them reusable and possible to code in stages. The decomposition of the system also makes it friendlier to testing and maintenance.

Having said this there are places in which following these strategies strictly would have been impossible or too time consuming. Examples being the *WireSwapper* (§A) genetic operator needing to know how a genotype will map onto a circuit when deployed and the whole system being constrained to fixed-length genotypes.

All software was written in Java using the Forte Community Edition IDE, except where stated. JavaDoc comments were used to provide an automatically generated documentation in HTML. A CVS repository was set up on Thor and WinCVS was used to synchronize with it.

# Chapter 3

## Implementation

### 3.0 Overview

In this chapter the evolutionary system framework proposed in §2.3 is developed and extended to evolve hardware. Each module will be treated in separate sections, and their intersection covered in §3.0.3. Feedback from the evaluation of certain evolutionary runs lead to improved understanding of the principles to use when designing and implementing `Experiments`, which is covered in §3.3.2.

#### 3.0.1 Development Cycle

The core section of the system — essential for performing any evolutionary experiment — was developed first using the waterfall model as depicted in Figure 3.1. This part of the system included the `GA` module, the common data structures, the `Deployment` and `Experiment` interfaces, and their respective testing harnesses. First the requirements of the system were set out in plain English, diagrams and pseudo-code. These described the purposes of the components of the system, how information would flow as they interact and in some cases a rough schematic of the structure or algorithm used internally. Then, the `Specification` was written which incarnates all components into classes and lists their variables, constructors and methods in Java form, sometimes including sections of code for them. The design strategies followed are those described in §2.5. From these sheets the implementation was coded revisiting the previous stages (`Validation` and `Verification`) when design issues were encountered. A testing harness was coded for each class to supervise its sound functioning and system integration testing was performed once different combinations of classes were ready. System maintenance and documentation was performed during part of the next cycle shown in Figure 3.2.

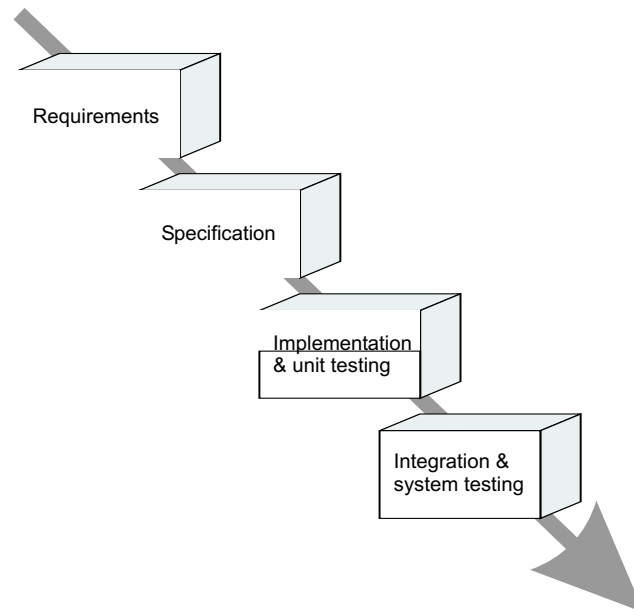


Figure 3.1: The Waterfall Development Cycle Model.

After this, the nature of the project allowed for an evolutionary model to be used, evolving some circuits while code was developed for evolving others. The stages used can be seen in Figure 3.2 in which the richness of the system (how many configurations could be used to evolve circuits) is represented by the thickness of the spiral. The “System & Experiment results Documentation” stage includes Validation, Verification, Documentation of code and of evolutionary experiment runs.

### Chronology of major mile stones

**15/11/00** Completed implementation of core section of system.

**23/12/00** D-Latch evolved extrinsically.

**28/12/00** D-Latch evolved intrinsically.

**2/1/01** Oscillator evolved extrinsically.

**4/2/01** Complex boolean function evolved extrinsically. Core Completed!

**13/4/01** Oscillator evolved intrinsically. First extension Completed!

**23/4/01** Multiplexer and simple vision evolved through distributed coevolution.

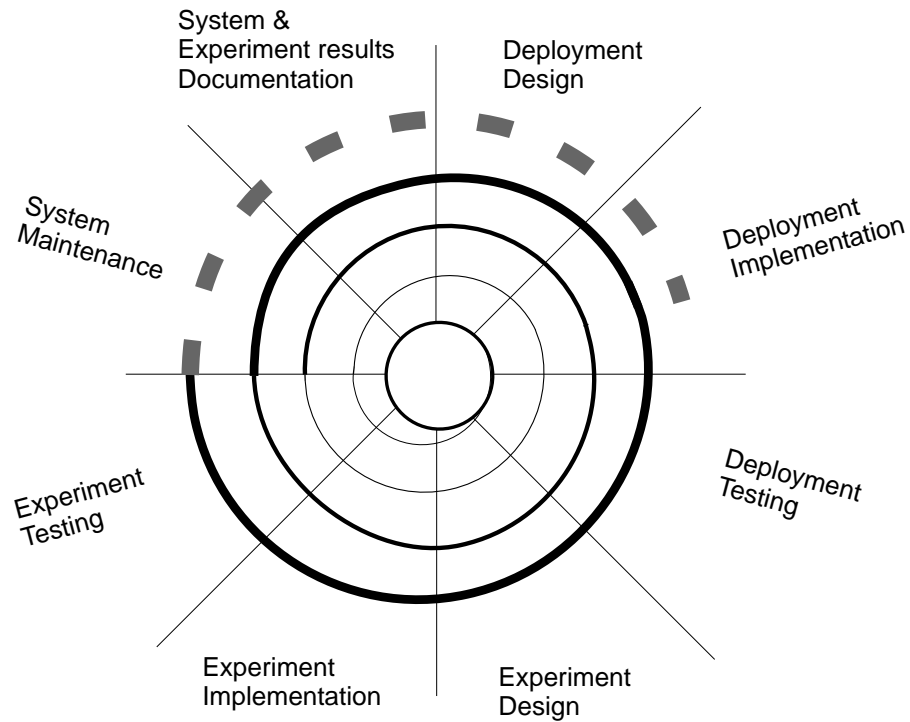


Figure 3.2: The spiral model used for the second half of the project. The thickness of the line denotes how many configurations of Deployment/Experiment were ready to be run, for example by the first round the SimulatorDeployment and DLatchExperiment were ready and were being executed as the FLEXDeployment and boolean function experiments were being developed. The central core is where the waterfall model (Figure 3.1) was used.

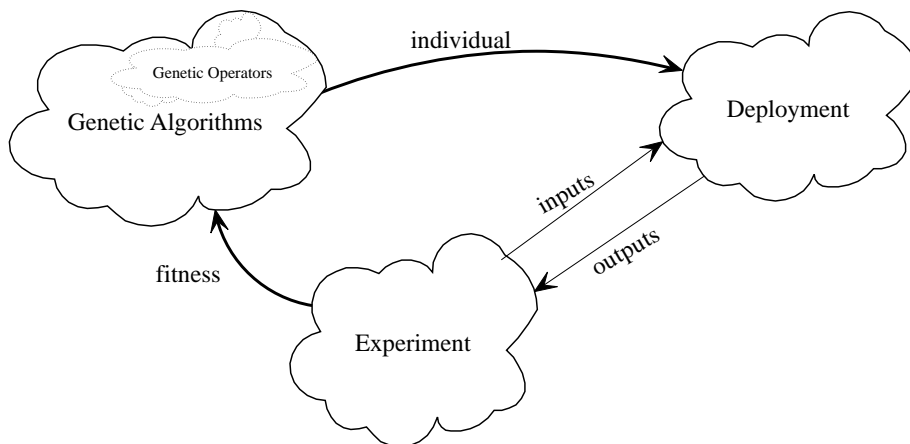


Figure 3.3: Module Structure. In the context of Figure 2.1 the Deployment and Experiment can be seen as the fitness evaluation mechanism.

### 3.0.2 System Structure

Following the decisions taken in §2.3 the evolutionary system is divided into the following main modules:

1. **Genetic Algorithms:** In charge of evolving the genotypes.
2. **Deployment:** Where individuals are tested to see how they perform.
3. **Experiment:** Defines what we want to evolve by providing inputs and a fitness function.
4. **Control:** In charge of managing the flow of information between the previous modules.
5. **Testing:** Collection of test harnesses.
6. **External:** Collection of components incorporated from other systems.

The first three are represented as modules in Figure 3.3 and the fourth as its arrows. The first three main modules can be treated as independent systems. Class hierarchies and object overviews can be found in the Appendix.

### 3.0.3 Glue

One on hand the common representation of individuals' genotypes is the glue between the GA and Deployment modules; on the other data samples, between the Experiment and Deployment as shown as the input and output arrows in Figure 3.3.

Genotypes are represented by a bit string and a fitness value in adjusted fitness form <sup>1</sup> which is reset only when any bit is modified so individuals aren't reevaluated when they survive intact into the next generations.

Data samples are represented by again a string of bits (the sequence of sampled values) and a value specifying the sampling rate. For example when using the simulator we may wish to sample the output faster than the input so we can wait for the output value to settle as in Figure 3.4. Not applicable to the FLEX because its gate delay times are minimal compared to the I/O the parallel port latency.

## 3.1 Genetic Algorithms — package `es.evolve`

### 3.1.1 Aim

What was needed was a module that could use genetic algorithms to evolve individuals.

---

<sup>1</sup>value ranging from 0, as worst fitness, to 1 as perfect fitness.

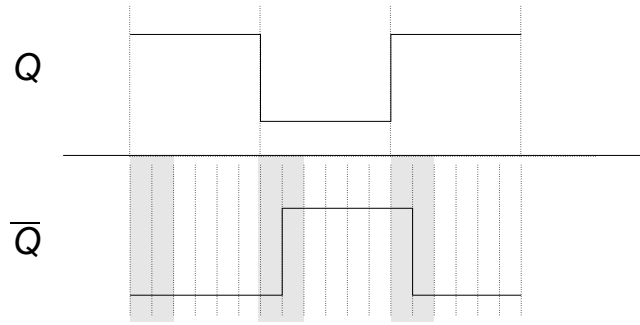


Figure 3.4: Outputs being sampled in this case at six times the frequency of the inputs. The greyed out areas of the outputs are ignored when evaluating fitness, allowing for the circuit to settle.

### 3.1.2 Algorithm Properties

The version of the genetic algorithm chosen is the simplified schema in §2.4.1 because it generalizes the type of genetic operators resulting in a more configurable system. The operators chosen to be implemented first were the per bit mutation and single point cross over ones because they were the simplest and most widely used basic set. Later on, a wire swapping <sup>2</sup> and per genotype mutation were implemented because of their convenient properties. Adaptive mutation was also implemented. To improve efficiency, an approximate of the Hamming distance of the population was calculated as:

$$H = \frac{1}{n-1} \times \sum_{i=1}^{n-1} h(g_i, g_{i+1}) \quad (3.1)$$

where  $g_i$  is the  $i^{th}$  genotype of the population,  $n$  is the population size and  $h(a, b)$  is the Hamming distance between genotypes  $a$  and  $b$ . This approximation is adequate for unsorted populations.

The genotype length was kept fixed because the search space of the experiments to be carried out in this project is constrained, and there is no need to make evolution open ended. Moreover, Holland's Schema theorem only applies to genotypes of fixed length <sup>3</sup>.

The genotype symbols were chosen to be binary because a mapping can always be found from binary strings to other data structures, making them general enough to encode an individual for any problem.

<sup>2</sup>swapping wire connections over is analogous to the subtree swapping GP cross over operator.

<sup>3</sup>Although it is proven in [5] that it can be extended to apply to variable length genotypes if their length is not allowed to change in big jumps.

The selection method chosen to be implemented was fitness proportionate over rank selection because of two reasons: on one hand it doesn't ignore the relative fitnesses of individuals. On the other rank selection's preparation time is bounded by sorting the population which is  $O(n \log n)$ ; while fitness proportionate's, by creating the cumulative fitnesses table which is only  $O(n)$ . Once prepared, they both need to find the position of a random number in their sorted table, which is  $O(\log n)$ . However the effectiveness of each at improving the search hasn't been proved either way.

### 3.1.3 Product

What is implemented is a GA module with pluggable genetic operators and selection policies. Single point cross over, per bit mutation, per genotype mutation, adaptive mutation and fitness proportionate selection were implemented and could be used for general purpose evolution. The WireSwapper GP operator was implemented and is restricted to evolving circuits.

## 3.2 Deployment — package `es.deploy`

### 3.2.1 Aim

A deployment must be able to program an individual and run it. Following the analogy used in §2.3 programming an individual would be putting the football player's atoms into the rugby pitch, and running it would be equivalent to sending stimulus to the player and recording how he reacts to it. This framework was to be extended to perform intrinsic and extrinsic deployment of circuits.

### 3.2.2 Circuit Encoding

To deploy an individual as a circuit, a mapping from a binary genotype to a circuit structure (phenotype) is needed. The circuit structure must allow feedback because it is essential to building D-Latches and oscillators. It also gives more freedom for evolution to explore and generate novel solutions to all problems.

To make this mapping simple yet powerful, NAND gates, inverters and C-Muller majority flip-flops were chosen as the building blocks of the circuits. NAND gates were chosen because they are powerful enough on their own to express any other logic operation. Inverters allow for a more powerful compact encoding of circuits allowing evolution explore the search space faster.<sup>4</sup> The majority gates are included for stabi-

---

<sup>4</sup>This can be thought of in terms of epistasis which is a measure of the amount of interdependencies between genes in a problem representation. High epistatic representations aren't suitable for being used

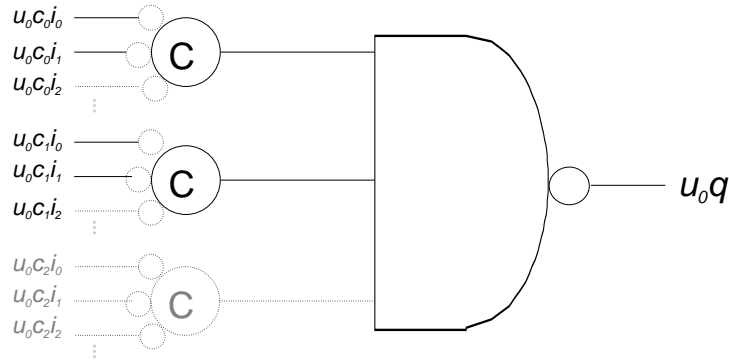


Figure 3.5: Basic unit of circuit structure with configurable components showed as dotted lines. C-Muller latches stabilize the inputs to the NAND gates. The former's inputs are optionally inverted.

lizing the circuit in the event that temperature dependant analog effects are occurring and want to be suppressed. The circuit is built up from configurable units as in Figure 3.5.

Each gene in the genotype will encode the sources of the  $u_xc_yi_z$  in Figure 3.5 in terms of the  $u_xq$  and the circuit inputs. The number used to encode each  $u_xq$  is the position this unit is defined in the genotype and highest numbers are reserved for circuit inputs, see Table 3.1 for an example. Substituting  $u_2q$  in  $Q$  we get  $Q = \overline{\overline{A} \wedge \overline{B}} = A \wedge B$  and thus an AND circuit is encoded. An extra bit can be added to each gene indicating inversion of this source.

Verilog code is generated by defining a wire for each  $u_xq$  and each C latch output, and then generating the appropriate code. Using the example of Table 3.1 we would have:

```
assign U0C0 = ( U0C0 & ( U2Q ) ) | ( U2Q );
```

and so on.. and then

```
assign U0Q= ~(U0C0 & U0C1);
```

and if we were using two inputs to C gates we could have:

```
assign U0C0 = ( U0C0 & ( U3Q | U7Q ) ) | ( U3Q & U7Q );
```

Other possible mappings are discussed in §3.2.4.

---

with GAs [5]. To evolve an inversion using only NAND gates ( $\overline{Q} = \overline{Q\overline{Q}}$  or  $\overline{Q} = \overline{\overline{Q\overline{A\overline{A}}}}$ ) both inputs to it would have to be equal so there would be an interdependency between the genes defining those inputs. However if a single bit is attached to each gene saying if this input is inverted the interdependency is avoided and epistasis is reduced. The representation is also made more compact by avoiding the need for adjacent duplicate genes.

wire defined	gene	encoded source	logic
$u_0c_0i_0$	010	$u_2q$	$u_0q = Q = \overline{u_2q \wedge u_2q}$
$u_0c_1i_0$	010	$u_2q$	
$u_1c_0i_0$	110	$A$	$u_1q = \overline{A \wedge u_1q}$
$u_1c_1i_0$	001	$u_1q$	
$u_2c_0i_0$	110	$A$	$u_2q = \overline{A \wedge B}$
$u_2c_1i_0$	111	$B$	
$u_3c_0i_0$	010	$u_4q$	$u_3q = \overline{u_4q \wedge u_3q}$
$u_3c_1i_0$	011	$u_3q$	
$u_4c_0i_0$	101	$u_5q$	$u_4q = \overline{u_5q \wedge u_3q}$
$u_4c_1i_0$	011	$u_3q$	
$u_5c_0i_0$	110	$A$	$u_5q = \overline{A \wedge u_1q}$
$u_5c_1i_0$	001	$u_1q$	

Table 3.1: Three bits are used to encode each unit while the two highest encodings are reserved for inputs leaving a total of six available encodable units. Output  $Q$  is defined to be  $u_0q$ .  $C$  latches have been set to have only one input which is passed on directly to the NAND gate input.

### 3.2.3 Extrinsic Deployment — The Logic Simulator

It is usually the case that simulations run slower than the hardware itself making the intrinsic evolutionary cycle faster. This is why simulations are rarely used when evolving hardware, specially when trying to make use of analog particularities which may not be included in the simulator. However in the case of this project the intrinsic evolutionary cycle was slowed down heavily by the need for compilation and the extrinsic cycle accelerated by keeping the simulator simple and by the natural increase in computer processing power. The simulator was also simpler to implement because a custom genotype→circuit mapping was used instead of one defined by an FPGA manufacturer for configuring the chip, which can be very complex in the case of multi-featured chips like the FLEX. Another advantage of simulation is that evolution can be readily distributed over a network without each client needing a FLEX connected to it.

The only analog effect chosen to simulate is a configurable gate delay because it is the most significant analog effect present in circuits and would probably be critical for evolving oscillators. The implementation defines the notion of a `SimulatorLogicElement` whose output settles to the right value a configurable time after its inputs changed. `SimulatorNANDLE` and `SimulatorCLE` extend this notion by defining what the output should settle to.

When the simulator is run it takes input samples and generates output samples. If the sampling rate of the inputs has been set up to be  $n$  times slower than the default value, then the simulator will reevaluate the whole circuit and sample its output  $n$  times

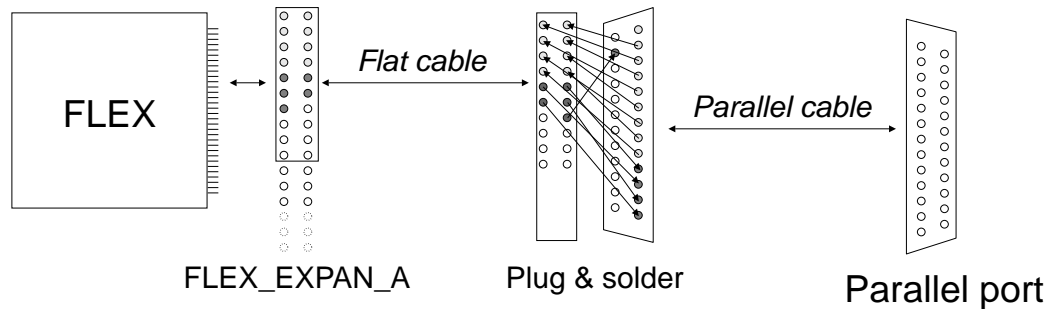


Figure 3.6: Schematic of hardware interface for parallel port communication.

for every input sample.

### 3.2.4 Intrinsic Deployment — Interacting with the FLEX

Two operations need to be performed on the FLEX: program an individual and perform I/O to this individual through the parallel port §2.4.3.

#### The parallel port hardware interface

The connection between the parallel port and the pins on the board could be direct because both the FLEX and the parallel port operate at five volts sharing the same ground when the FLEX is powered by the PC. The hardware interface built is a direct pin to pin connection between the FLEX user I/O pins mapped onto the FLEX\_EXPAN\_A patch on the board and the pins of a DB25 parallel cable board mounting plug, as in Figures 3.6 and 3.7. The eight parallel port data pins are used to send inputs to the FLEX, and the five status pins are used to read its outputs back.

#### The parallel port software interface

On the software side there would have to be a two way communication path between Java and the parallel port registers. The code to write to the data register at the parallel port base address and read from the status register at offset +1 was to be written in native code and then linked into Java through JNI. A JNI aware DLL was written in C to read and write bytes to any port address. To access ports under Windows 2000 and NT the “MS Driver Development Kit” libraries including the generic portIO driver were used. The final software interface was comprised of the JNI callable C DLL calling the portIO driver, as in Figure 3.8.

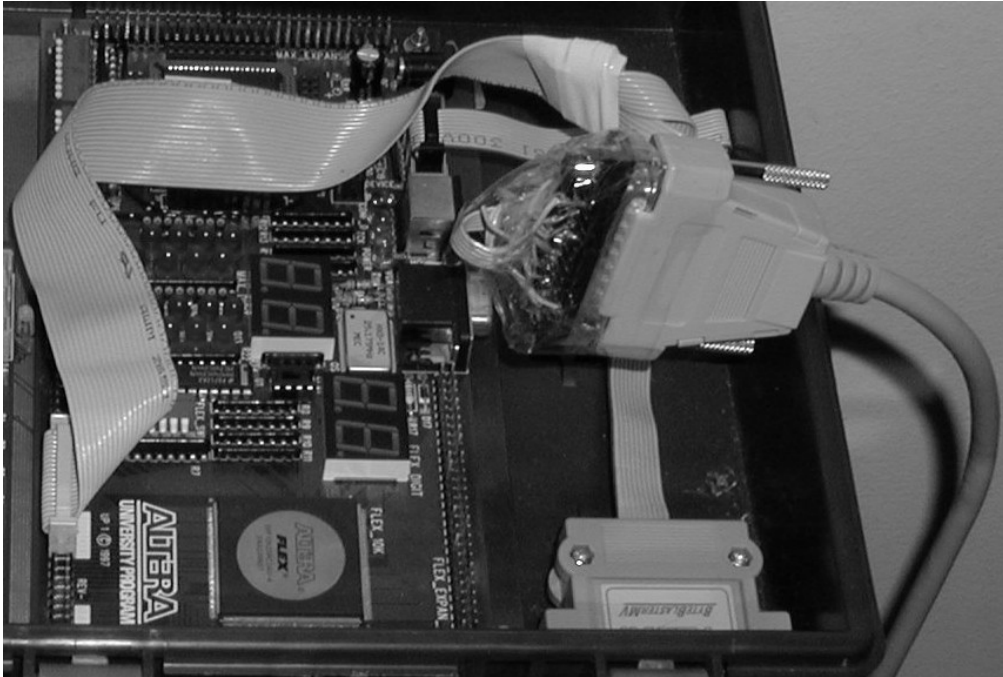


Figure 3.7: The parallel port hardware interface.

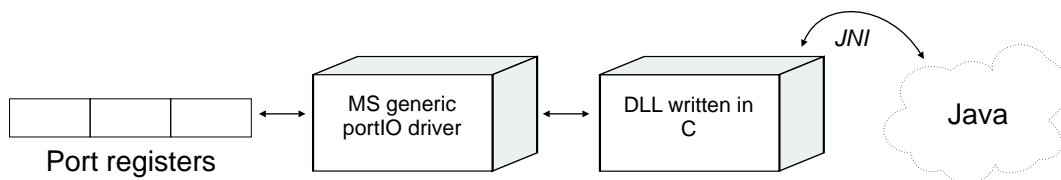


Figure 3.8: Schematic of software interface for parallel port communication.

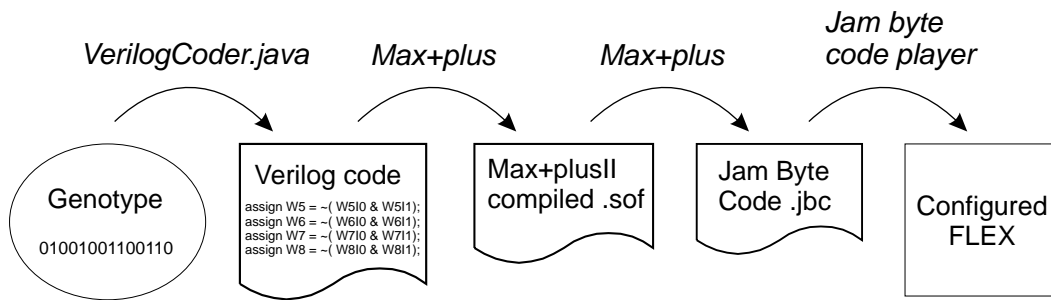


Figure 3.9: Programming an individual onto the FLEX: from genotype in Java to a configured circuit. Note all intermediate formats are different representations for the same individual. All arrows are functions.

### Programming the FLEX

Before a circuit can be tested through I/O it has to be programmed into the FLEX. This involves

1. Generating Verilog code from the genotype (see §3.2.2).
2. Compiling it.
3. Sending the compiled programming file to the FLEX.

The first step was performed by adding the generated code onto an appropriate header with the appropriate chip and pin definitions file already created.

The second step was performed by invoking Max+plus II from Java through the `System.exec()` invoking call.

The last step was complicated because Max+plus II tools don't allow the programmer to be invoked through command line parameters. As a quick initial implementation a shareware macro-recorder script was written to manipulate Max+plus II to execute the programmer. Later on, a Jam STAPL byte-code player was used to configure the FLEX from the command line as in Figure 3.9. Other possible methods include using Altera's own bit string  $\rightarrow$  circuit as in Figure 3.10 however running the danger of short-circuiting the FLEX and rendering the simulator incompatible, or adding an extra step to map a genotype into a bit string as in Figure 3.11 by performing a primitive compiling step and then program this which would require knowing Altera's mapping.

### 3.2.5 Product

A deployment framework was developed and extended to achieve a common interface for testing circuits intrinsically on the FLEX and extrinsically in a logic simulator.

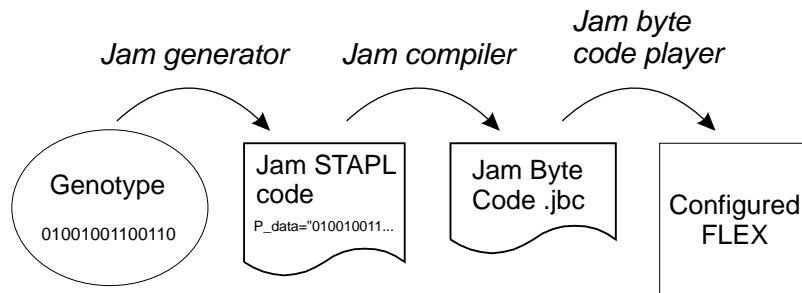


Figure 3.10: Programming the FLEX without compilation using Altera's mapping from a binary string to a circuit structure. No knowledge of their mapping required.

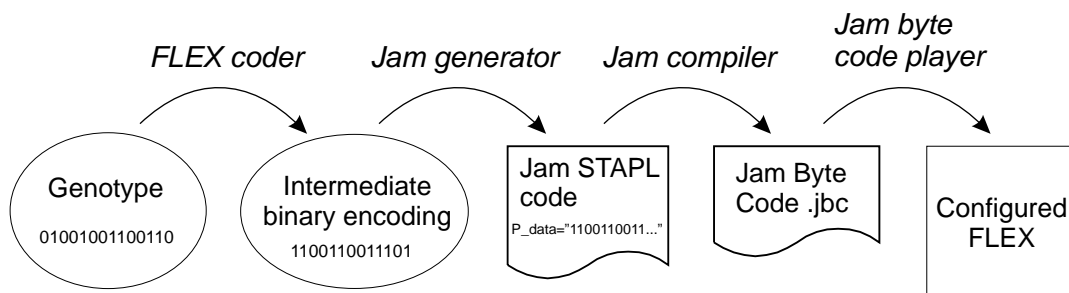


Figure 3.11: Programming the FLEX without compilation using custom mapping by adding converter from binary string to a FLEX binary file that will convert into the desired structure under Altera's mapping. Knowledge of their mapping required.

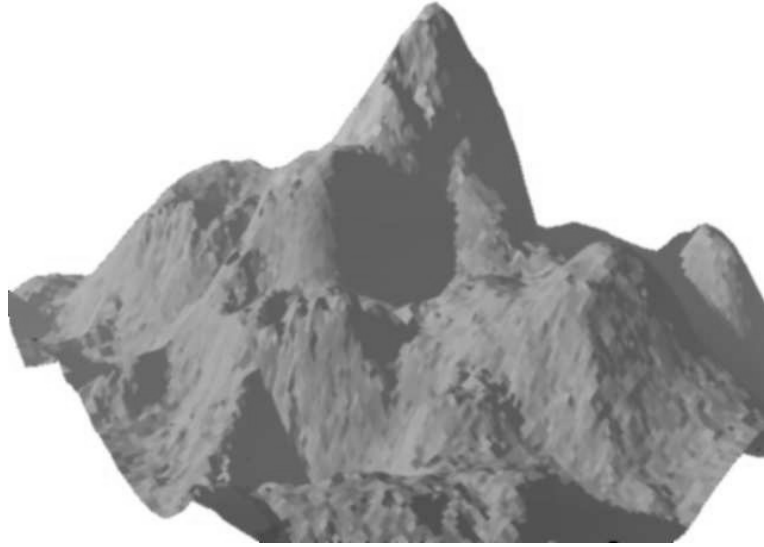


Figure 3.12: A good fitness landscape with a correlation between climbing up and moving towards the maxima.

### 3.3 Experiment — package es .experiment

#### 3.3.1 Aim

Continuing with the analogy in §2.3 an Experiment must on one hand define the stimulus to be given to the football player on the rugby pitch and combined with its reactions say how good a bowler she was. The aim was to begin by creating experiments for evolving D-Latches and some arbitrary Boolean functions and later on move to more demanding experiments.

#### 3.3.2 Fitness Functions

An appropriate fitness function is essential for the functioning of GAs since it defines entirely the shape of the fitness landscape. This function must be able to guide the GA towards its maxima by providing a good correlation between climbing up in the landscape and moving towards the maxima as in Figure 3.12. Hence it is bad for a fitness function to have too many local maxima which are unrelated (there is no ridge between) to the maxima as in Figure 3.13, because the GA will get stuck at the top of the wrong hills.

This will be illustrated with an example from this project. The fitness of D-Latches and Boolean functions were initially calculated as [7]:

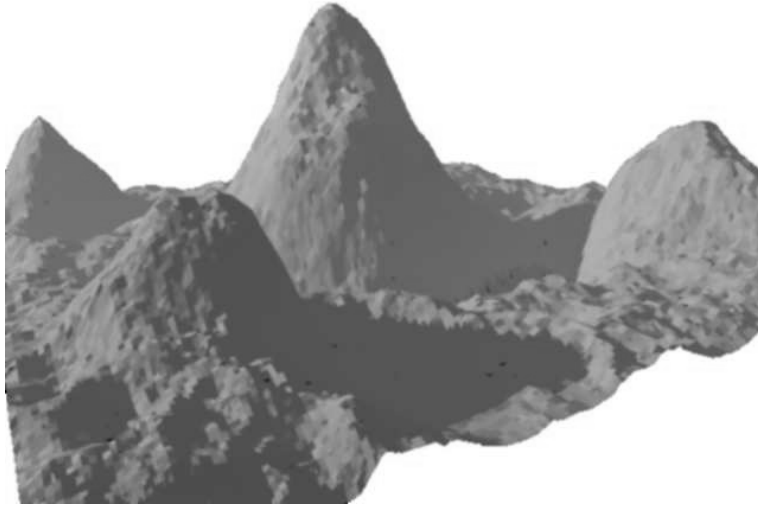


Figure 3.13: A bad fitness landscape with too many misleading local maxima.

$$fitness(Q, Q') = \frac{1}{n} \times \sum_{i=1}^n \delta_{Q_i Q'_i} \quad (3.2)$$

where  $Q$  and  $Q'$  are the actual and desired output of the circuit respectively — for example  $Q' = A \wedge B$  in the case we're trying to evolve an AND gate — and  $n$  is the number of output samples<sup>5</sup>.

The problem with equation 3.2 is that for an average Boolean function in which the output is high half of the time, the fitness of a “dead” circuit which never changes its output will be a half. This is very misleading for evolution since it will drift the population towards a local maxima of “dead” circuits which is totally unrelated with the solution, like a peak in Figure 3.13. This gets worse with the case of AND gates since our “dead” circuit will be 75% times correct if it always outputs low. In order to fix this we could modify our test inputs so that the output should be high half times, but this is the best we can do.

A naive solution to this problem is to punish fitness if the output does the wrong thing. So now

$$fitness(Q, Q') = \frac{1}{n} \times \left( \sum_{i=1}^n \delta_{Q_i Q'_i} - p \times \sum_{i=1}^n \delta_{Q_i \overline{Q'_i}} \right) \quad (3.3)$$

where  $p$  is a punishing factor. However now we face more problems. Not only can the fitness be negative, but the  $Q$  with lowest fitness will be the exact inversion of  $Q'$

<sup>5</sup>we divide by  $n$  to have adjusted fitness §3.0.3

which, being only a step (or inverter) away from the solution, should have very high if not perfect fitness. Again, this is very misleading for evolution forcing it to make more use of random search than GAs to find the solution. Taking the modulus of the right hand side of equation 3.3 is better but still misleads evolution by giving scores to “dead” circuits when the number of 1s and 0s in  $Q'$  aren't equal; and very low scores to individuals which are correct half of the time and inverted the other half, which again could be one inversion away (one bit under our circuit representation) from the solution.

Hence a fitness function is needed that gives a better measure of how the  $Q$  and  $Q'$  data sets behaves relative to one another. The statistical correlation between  $Q$  and  $Q'$  gives this and provides a smoother landscape which is more efficient at guiding evolution towards the real maxima.

$$fitness(Q, Q') = \begin{cases} correlation(Q, Q') & correlation(Q, Q') > 0, \\ correlation(Q, Q') \times -p & \text{otherwise.} \end{cases} \quad (3.4)$$

where  $p$  is this time punishing factor for inverted outputs, 0.97 being used for many experiments in this project. Equation 3.4 gives zero fitness to any “dead” circuits and a good score to any circuit behaving in a similar pattern to the desired one, creating a landscape more like the one in Figure 3.12. This fitness function improved the performance of the GA by about 4000%.

### 3.3.3 The test inputs

These are also important for an efficient GA. For most non-trivial problems in this project it's impossible to generate a complete test-set because in allowing feedback circuits can become sequential and not all sequences of arbitrary length can be tested. So sometimes circuits learn to use test sequences instead of current inputs to behave. However random test sets are counterproductive because randomness on both ends can find an unhelpful perfect match of random circuit and test inputs <sup>6</sup>. This applies even when only a portion of the inputs are random. This is why all individuals must be tested with the same inputs for the competition to be fair and results meaningful.

A test set has to be complete enough to discourage individuals using wrong approaches and focused enough on the key issues of the circuit to distinguish correct solutions above others. For example, the test pattern in Table 3.2 allows  $Q = D + Q\overline{C}$  to respond to it exactly as a D-Latch, yet Table 3.3 includes line four to focus on the fact that  $Q$  should never change when  $C$  is low.

---

<sup>6</sup>for example a “dead” circuit evaluated as an AND never tested for  $A = B = 1$

$C$	$D$	$(Q)$
0	0	N/A
1	0	0
0	0	0
1	1	1
0	1	1

Table 3.2: Unhelpful D-Latch test inputs.

$C$	$D$	$(Q)$
0	0	N/A
1	0	0
0	0	0
0	1	0
1	1	1
0	1	1

Table 3.3: D-Latch test inputs

### 3.3.4 Fitness Functions and Test Inputs used for the Experiments

#### D-Latches and Boolean functions

Equation 3.4 was used as the fitness function for all these experiments. Table 3.3 was made focused on the latching behaviour of the D-Latch, and Tables 3.4 and 3.5 provide a good balance between high and low values. These tests were repeated a few times and sometimes complemented with random inputs. For the more complex functions the full set of inputs was used, with repeated and inverted sections.

$A$	$B$	$(Q)$
0	1	0
1	1	1
1	0	0
0	0	0
1	1	1

Table 3.4: And function test inputs

$A$	$B$	$(Q)$
1	0	1
0	0	0
0	1	1
1	1	0

Table 3.5: Exclusive OR function test inputs

### Multiplexers

These are used as a standard benchmark when evolving circuits [7] [2]. The full set of inputs was used to test them and equation 3.4 used as a fitness function. The number of data and address lines was allowed to be configurable.

### Oscillators

Since the behaviour of these circuits lies on a smaller time scale than the parallel port latency, a different approach was used for recording oscillations on the FLEX and the simulator.

#### Simulator

The fitness function used was [11]:

$$\bar{\lambda} = \frac{1}{k} \times \sum_{n=1}^k (t_n - t_{n-1}) \quad (3.5a)$$

$$fitness = \frac{1}{|\bar{\lambda} - \lambda| + 1} \quad (3.5b)$$

where  $k$  is the number of rising edges recorded and  $t_n$  are the times they were recorded at (initial edge  $t_0 = 0$ ) and  $\lambda$  is the desired separation between edges (wavelength). See Figure 3.14.

#### FLEX

Since the parallel port — and the ARM<sup>7</sup> — were too slow to measure the frequency of oscillation of some circuits another method had to be found. A frequency counter was implemented in Verilog that Counted rising edges from the evolved circuit's output during half a cycle of a Divided clock signal as in Figure 3.15 and stored them in a

<sup>7</sup>some circuits oscillated at 30Mhz, sampling at the ARM's 24Mhz aliasing effects would give erroneous frequency measures.

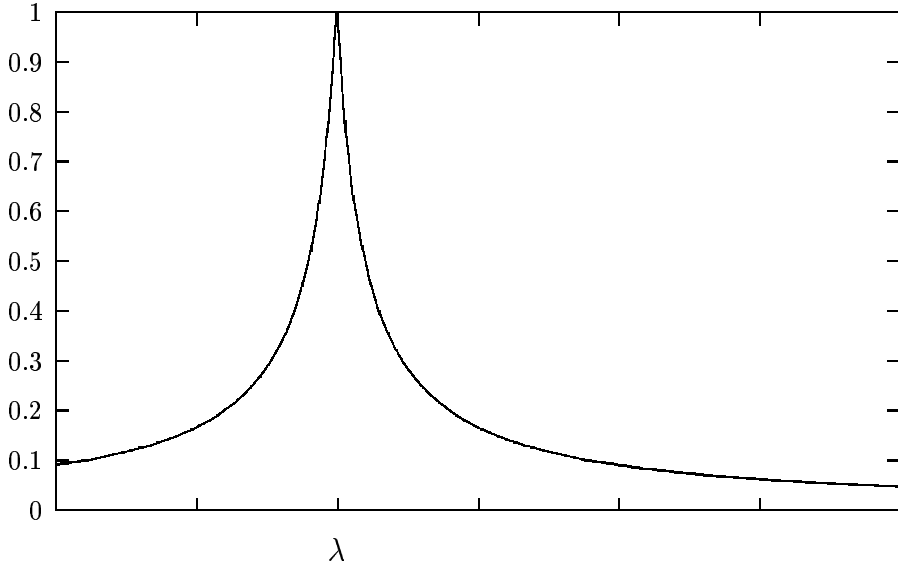


Figure 3.14: A plot of *fitness* vs.  $\bar{\lambda}$  from Equation 3.5b and  $\lambda \approx 20$ .

40bit Frozen register made accessible from the PC through a MUX as in Figure 3.16. This value was sampled a few times to punish unstable circuits and get a more accurate value for good ones.

Now this sampled average frequency in Hertz  $\bar{\mu}$  could take values ranging from thousands to millions for which equation 3.5b would be inadequate. Hence a scalable function was needed with shape invariance under  $\mu$ . Equation 3.6 plotted in Figure 3.17 has this property:

$$fitness = \begin{cases} 1 - \left(\frac{\mu - \bar{\mu}}{\mu}\right)^2 & x < \mu, \\ \left(\frac{|\mu - \bar{\mu}|}{\mu} + 1\right)^{-1} & \text{otherwise.} \end{cases} \quad (3.6)$$

### Tone Differentiation

Another extension of this project was to evolve circuits that could differentiate input frequencies, using similar skills to those used in voice recognition for example. These were only evolved on the FLEX by implementing a module that could generate a different tone depending on an input from the PC. This tone was then fed into the evolved circuit which reported its output back to the PC as in Figure 3.18. The fitness is calculated as the correlation between the *Tone code* fed in and the value of  $Q$  received. This model can be adapted to differentiate between more than two tones by simply making these two lines more than 1 bit wide.

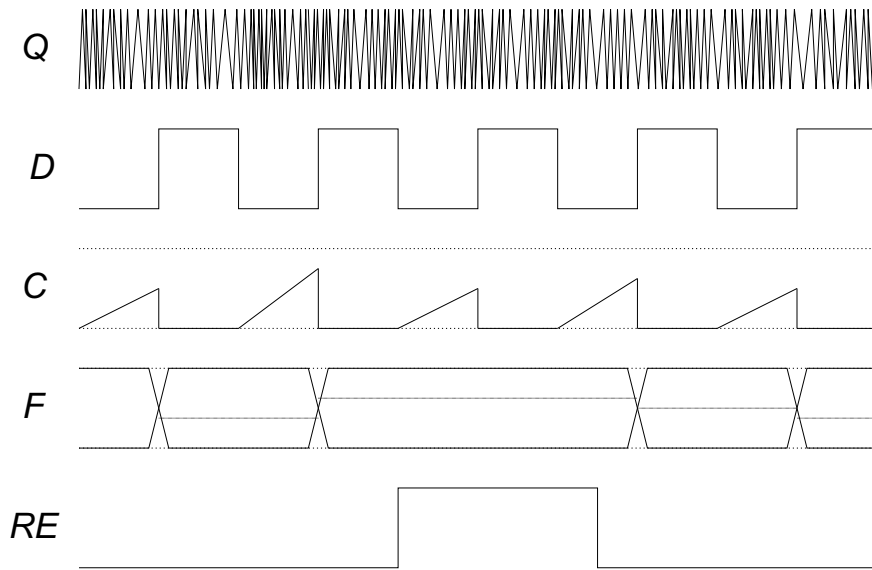


Figure 3.15: Timing diagram for frequency counter shown in Figure 3.16. *C* and dotted line in *F* represent register value. Note the latter is constant when *RE* is high.

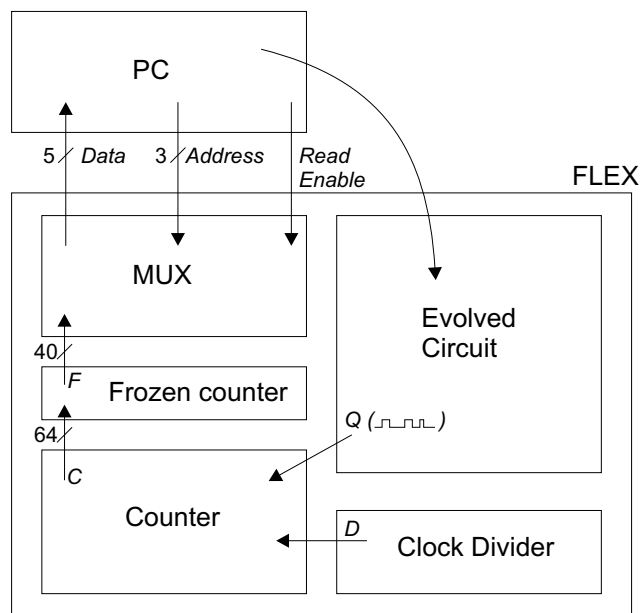


Figure 3.16: The frequency counter implemented on the FLEX to evolve oscillators.

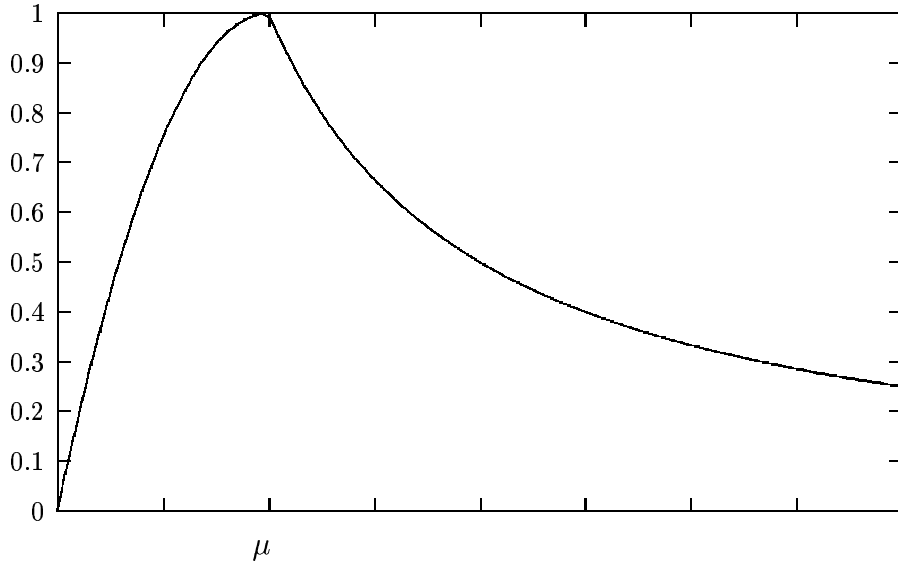


Figure 3.17: A plot of *fitness* vs.  $\bar{\mu}$  from fitness function in Equation 3.6.

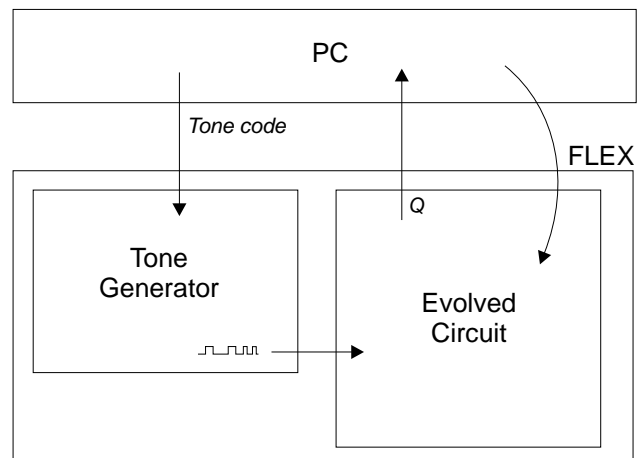


Figure 3.18: The setup used to evolve tone differentiators intrinsically.

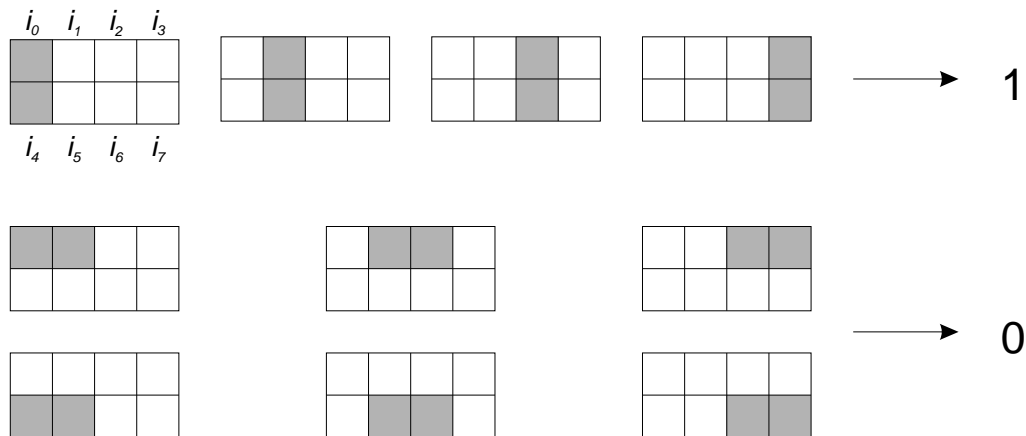


Figure 3.19: Test cases used for simple image recognition using eight stimuli. Vertical lines (above) must generate high output. Horizontal lines (below) must generate low output.

### Simple Vision

A crude case of the image recognition project extension was implemented. All that the circuit had to do was to guarantee a high output when a vertical bar was inputted and a low one if a horizontal bar was inputted, as in Figure 3.19. All other (don't care) cases weren't tested. Clearly this is just a Boolean function with minterms  $i_0i_4 + i_1i_5 + i_2i_6 + i_3i_7 + i_1i_2 + i_2i_3 + \dots + i_6i_7$  plus optional (don't care) terms that may simplify it. However it is still a valid small step into the realm of image recognition.

### 3.3.5 Allowing for Time Delays

Simulated gate delay (§3.2.3) results in circuit delay. Hence a finer grained sampling is used for the outputs than for the inputs so we can discard a configurable initial proportion  $t_{delay}$  of output samples per input sample, see Figure 3.4 for an example.

### 3.3.6 Product

The experiments framework was developed and extended for evolving D-Latches, various Boolean functions, oscillators, multiplexers, tone differentiators and simple image recognition.

## 3.4 Control — package `es.control`

### 3.4.1 Aim

This module should manage the flow of information between the previous three and also perform distributed island based coevolution.

### 3.4.2 Distributed Coevolution

This extension of the project exploits the parallel nature of GAs using spare computing time on processors connected through TCP/IP. A more general case was implemented: a model for processor intensive low communication distributed computing. The advantage being that the projects code could be modified on the server side and the client wouldn't need altering, nor restarting. This was implemented using RMI and could not be performed by using a remote code base (as an applet) because clients would be more sensitive to server crash and the browser closing. Also, a Java application can be ran automatically as a background service so people can always offer their computer's spare time to any task that's being ran.

#### Distributed Computing Model

The model implemented provides interfaces for a `Task`, an `InteractiveTask` and an `InteractiveTaskServer`. The first is a task which only takes a parameter when started and returns a value when it finishes <sup>8</sup>. The second is a task which can perform I/O during its operation to report and modify its internal state. The last is a server which will provide clients with their ID and task and will interact with them.

An `InteractiveTaskClient` was implemented which connects to a generic `InteractiveTaskServer`, downloads a generic `InteractiveTask`, runs it and then facilitates the interaction between the server and the task. The client is only a 5Kb JAR file and powerful enough to run any processor intensive low communication task.

This client is entirely in control of initiating communication between the task and the server and of handling failure recovery by continuing processing when server connection fails and by not restarting a task when the server gives it the same one, as in 3.20. The advantage of this centralized control is that the implementations of the task and the server can be failure transparent. The state can be distributed throughout the whole system and nothing is lost if any single (either server or client) host crashes.

This model can be used to run various tasks on heterogenous platforms, for example physics/network simulations, numerical computations, code breaking, information processing.

---

<sup>8</sup>multiple parameters and return values can be packed into one `Vector` object for example.

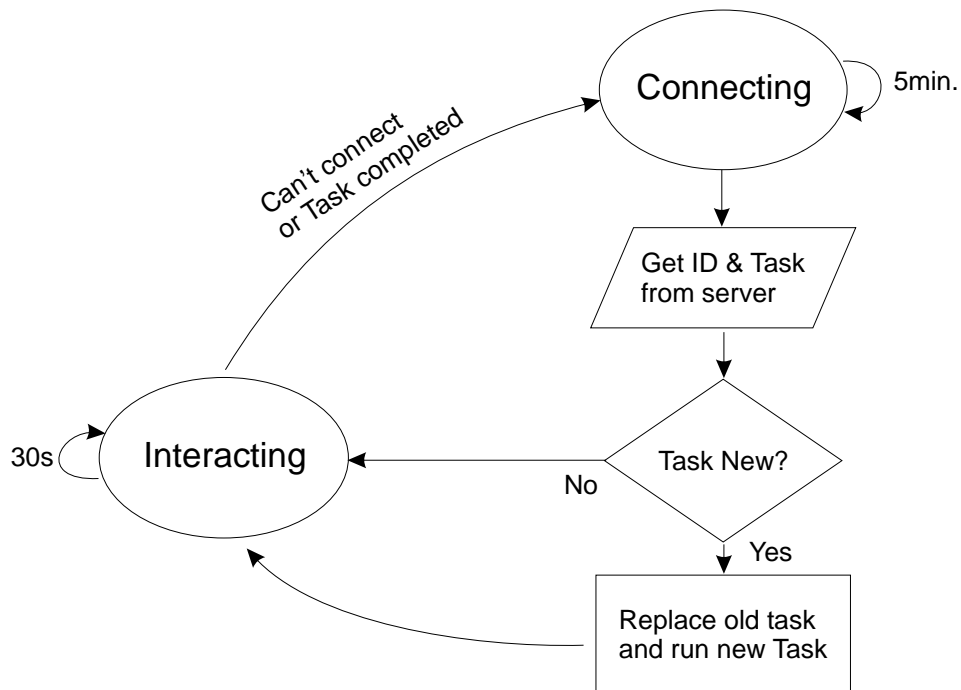


Figure 3.20: The `InteractiveTaskClient`'s control thread activity diagram, allowing for a server and client crash proof system.

### Island based Coevolution implementation of model

The `InteractiveTaskServer` was implemented to perform island based coevolution. This assigns each client an island on a two dimensional grid (Figure 3.21) which is used as the client ID.

The task given out to clients is on one hand the whole `Monica` evolution task including the `Evolver`, `Experiment` and `Deployment` and on the other a GUI which displays the client's progress and that of its neighbours as in Figure 3.22. An extra button allows for a special function to be performed on the best individual which in this case loads `Billy the Logic Simulator` (§3.6.2) to view it and simulate it. Both tasks were packed into one task by implementing a `MultiInteractiveTask` which would run all its contained tasks simultaneously and pack and unpack I/O from the server to them in `Vectors`. Various concurrency issues were attended because `Monica`'s inner state was now accessible to an external thread.

During interaction, the evolution task sends the server its best individual, a migratory individual and some statistics. The server then performs migration between neighbouring clients and returns the incoming migrator to the evolution task and neighbours statistics to the GUI, as in Figure 3.23.

This subsystem is general enough to be adapted to other genetic evolution projects



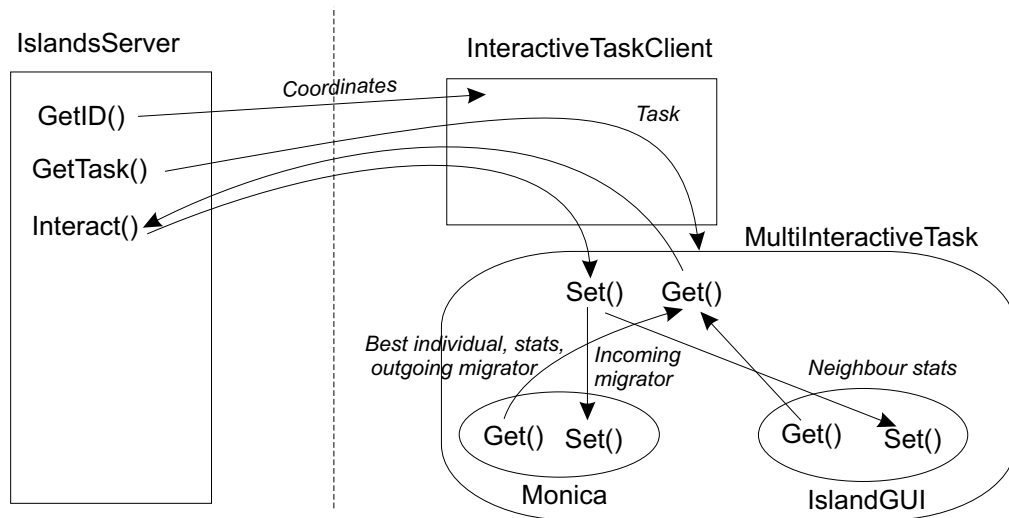


Figure 3.23: The `IslandsServer` interacting with `Monica` and the `IslandsGUI` through the `InteractiveTaskClient`.

by only implementing a couple of interfaces.

### 3.4.3 Product

A control module is implemented which manages the information flow between the previous three modules to perform evolutionary runs. Also, a framework for performing distributed computing was developed and extended to provide a framework for performing distributed island based coevolution.

## 3.5 Testing — package `es.testing`

A limited amount of debug code was included in each object, most of it being in external test harnesses making the objects lighter at runtime. The `DebugLib` library can be instructed to toggle objects into and out of debug mode making it output or ignore their debugging reports. Sample testing results are in §4.1.

## 3.6 External — package `es.external`

### 3.6.1 Statistical Functions package

Thanks to Sundar Dorai-Raj for these classes, downloaded free from <http://www.stat.vt.edu/~sundar/java/> used for calculating statistical correla-

tion (equation 3.4).

### 3.6.2 Billy the Logic Simulator part 1B Project India 2001

A bridge was implemented so that the individuals evolved in this project could be displayed and simulated here as in Figure 4.9. Thanks to the neatness of their code this did not take long. When bridging circuits into Billy redundant gates — whose outputs are disconnected — are automatically discarded using the algorithm:

```
while changed( Gates )  
   $\forall g_i \in \text{Gates}$   
    if output disconnected remove  $g_i$ 
```

### 3.6.3 Class File Server

Sun's lightweight http server was used to serve classes while using RMI, which can be found at: <ftp://ftp.javasoft.com/pub/jdk1.1/rmi/class-server.zip>.

## 3.7 Overall Product

An evolutionary system framework was developed with configurable GA properties, pluggable deployment of individuals, and pluggable evaluation procedure describing what they're to evolve towards.

The system was extended to evolve hardware circuits intrinsically on a FLEX10K20 CPLD and extrinsically on a simple logic simulator, through the use of various genetic operators. More extensions were implemented to evolve D-Latches, various Boolean functions, multiplexers, oscillators, tone differentiators and simple image recognition. A further extension allowed extrinsic evolution to be performed distributedly over the internet.

# Chapter 4

## Evaluation

This chapter will cover the testing of the evolution system framework, summaries of some of the evolutionary runs carried out, and an appreciation of what was and what wasn't completed.

### 4.1 System Testing

All modules were tested with their own test harnesses as they were developed. Later on, integration harnesses were made to test the consistency and sound interaction between modules. This section will go through some examples of the many (  $\approx 20$  ) test harnesses.

#### 4.1.1 Parallel Port Interface

Once the hardware interface had been tested using a freeware parallel port debugging tool, the whole interface was tested using the GUI harness shown in Figure 4.1, which allows bits or bytes to be written and read from Java to the FLEX. This was mainly used in a feedback setup with the FLEX mapping its lowest five inputs to its outputs. The harness was also used later to interact with evolved circuits.

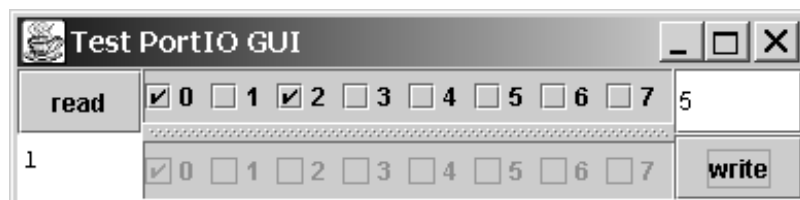


Figure 4.1: The GUI to test the parallel port interface.

### 4.1.2 Fitness Proportionate Selection

This test harness selected ten thousand individuals from a known population checking how many times each was selected. Harness output:

```
Now will test select() method used to select different individuals according
to their fitness, we'll set up a population of individuals with fitnesses:
```

```
ind#0 => 0.25
ind#1 => 0.0078125
ind#2 => 0.015625
ind#3 => 0.125
ind#4 => 0.03125
ind#5 => 0.5
ind#6 => 0.0
```

```
Test results in selection hits per individual / total hits:
```

```
ind#0 => 0.2697
ind#1 => 0.0104
ind#2 => 0.017
ind#3 => 0.136
ind#4 => 0.0354
ind#5 => 0.5315
ind#6 => 0.0
```

### 4.1.3 Simulator & Verilog

This harness tests if the Simulator and Verilog coder implement the same genotype → circuit mapping. It was also used as a regression test when either of them was modified. The sample output below shows that the simulator reports (top) the same structure as the Verilog code (bottom).

```
The same simple individual will be programmed into a simulator and then coded in Verilog.
```

```
Simulator's Structure:
```

```
**CGate connections
C0 = N1 N1
C1 = N1 N1
C2 = N2 N2
C3 = N3 N3
**NAND connections
N0 = C0 C1
N1 = C2 C3
```

```
and Verilog is:
```

```
wire U0Q;
wire U0C0;
wire U0C1;
wire U1Q;
wire U1C0;
wire U1C1;
wire U2Q;
wire U2C0;
wire U2C1;
```

```

wire U3Q;
wire U3C0;
wire U3C1;
assign outputs[0] = U0Q;
assign U3Q = inputs[0];
assign U2Q = inputs[1];
assign U0C0 = U0C0 & ( U1Q | U1Q ) | ( U1Q & U1Q );
assign U0C1 = U0C1 & ( U1Q | U1Q ) | ( U1Q & U1Q );
assign U1C0 = U1C0 & ( U2Q | U2Q ) | ( U2Q & U2Q );
assign U1C1 = U1C1 & ( U3Q | U3Q ) | ( U3Q & U3Q );
assign U0Q = ~( U0C0 & U0C1);
assign U1Q = ~( U1C0 & U1C1);

```

#### 4.1.4 D-Latch Experiment

This harness generated random, perfect and delayed D-Latch output samples and then called the `DLatchExperiment` to evaluate their fitness in debug mode.

#### 4.1.5 Evolver

Apart from testing harnesses for each genetic operator, another was written to inspect if the population size remained the same from generation to generation, the right number of individuals were reproduced through each operator, etc...

#### 4.1.6 Deployment-Experiment Integration

This harness deploys a given individual on the tested simulator, runs it and evaluates its fitness using a given tested experiment. This tested the simulation, the inputs-outputs interaction with the experiment and the fitness evaluation. Later on this was used to inspect the functioning of evolved circuits.

In the edited sample output we're testing a circuit which achieved high fitness in the simple vision experiment (§3.3.4). The first columns show the input data sample (*ids*) number, then the value of the inputs, and the last two columns show the desired and actual outputs *desQ*, *actQ*. The data shows *actQ* oscillates at *ids* = 0 where *i<sub>0</sub>*, *i<sub>4</sub>* and *desQ* are high. Then at *ids* = 1 *actQ* is a perfect inversion of *desQ*. At the start of *ids* = 2 it exhibits a small race condition as many inputs change. At *ids* = 3 it exhibits a delay at changing but this will be ignored while calculating fitness (§3.3.5). At *idl* > 3 (edited out) *actQ* keeps being a reasonable inversion of *desQ*. Fitness is lowered by the badly handled case when *i<sub>0</sub>* = *i<sub>4</sub>* = 1. Note that the simulated circuit displays sound realistic behaviour.

```

ids < inputs >      desQ actQ
0 true false false false true false false false true true
0 true false false false true false false false true false

```

```

0 true false false false true false false false true false
0 true false false false true false false false true false
0 true false false false true false false false true true
0 true false false false true false false false true false
0 true false false false true false false false true true
0 true false false false true false false false true false
[...]
1 false false false false false false true true false true
1 false false false false false false true true false true
1 false false false false false false true true false true
1 false false false false false false true true false true
1 false false false false false false true true false true
1 false false false false false false true true false true
1 false false false false false false true true false true
1 false true false false false false false false false true
2 false true true false false false false false false false
2 false true true false false false false false false true
2 false true true false false false false false false true
2 false true true false false false false false false true
2 false true true false false false false false false true
2 false true true false false false false false false true
2 false true true false false false false false false true
3 false true false false false true false false true true
3 false true false false false true false false true true
3 false true false false false true false false true false
3 false true false false false true false false true false
3 false true false false false true false false true false
3 false true false false false true false false true false
[...]
Fitness: 0.8588594491702484

```

### 4.1.7 Overall System Integration

This module ran the evolutionary process with a tested deployment and experiment, optionally configuring objects into debug mode and outputting the status at each generation and the best individual at the end.

### 4.1.8 Distributed coevolution

Code was written to test the client placement algorithm (§3.4.2) and the migration procedure. The server running on a spare 486 DX25 with 32MBs of RAM handled 15 clients on computers in Cambridge and home totalling  $\approx 7$ Ghz of spare computing power.

## 4.2 Experiments

This section will go through some of the many (> 60) evolutionary experiments carried out. The most successful of the many GA and system parameters used were:

- Population size: 500
- Inputs to NAND gates: 2
- Inputs to C latches: 1 (ignores them)
- Single point cross over proportion: 0.6
- Adaptive mutation: 1 bit per genotype for high diversity (0.5 average Hamming distance per bit), 30% of bits mutated for no diversity.
- $t_{delay}$  proportion of cycle for outputs to settle: 0.5
- Selection policy: fitness proportionate.
- Simulator gate delay: 1.

Some interesting properties were observed. On one hand the number of individuals evaluated per evolutionary run is distributed around a mean value  $n$  such that:

$$n = k \times popSize \times generations \quad (4.1)$$

where  $k$  depends on the the fitness landscape. On the other, the GP operator `WireSwapper` had a similar effectiveness over many runs to the GA operator `SinglePointCrossOver`. This is not revolutionary since GP has repeatedly proved as successful as GAs even if Schema Theorem doesn't apply to it.

Another experiment was carried out early on to evolve a D-Latch using only a mutation rate of 0.5 per bit, so all individuals were randomly generated at each generation, equivalent to random search. This experiment ran till its maximum of 50000 generations without finding a solution, compared to under 20 generations using the normal, showing that solutions weren't been found by brute force.

Other experiments were carried out to measure the usefulness of Adaptive Mutation. Figure 4.2 shows how the evolutionary process got stuck at top fitness 0.8 because the population converged on local maxima having very low diversity (an average Hamming distance of only 0.05), while in Figure 4.3 adaptive mutation is used and population diversity is always maintained avoiding premature convergence.

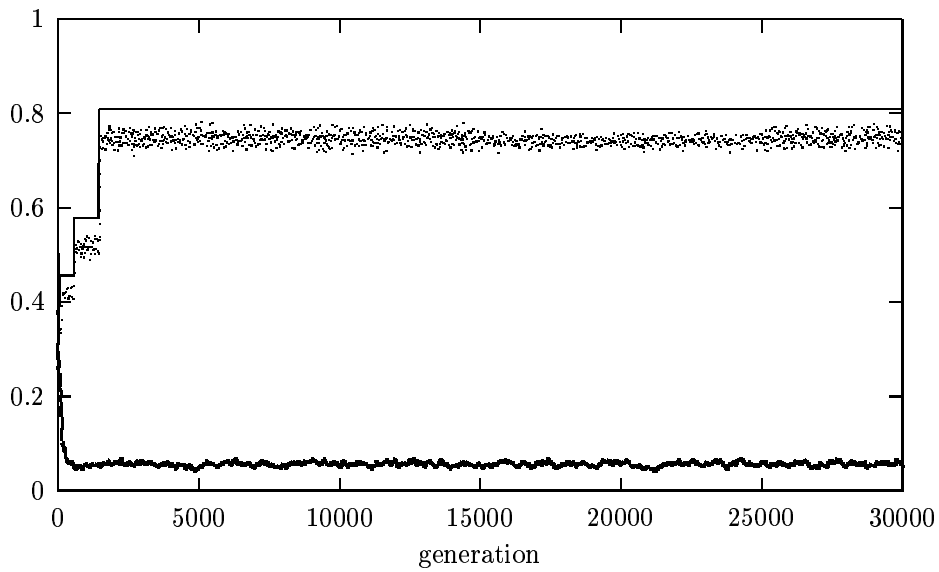


Figure 4.2: Top fitness, average fitness, average Hamming distance vs. generation with no adaptive mutation.

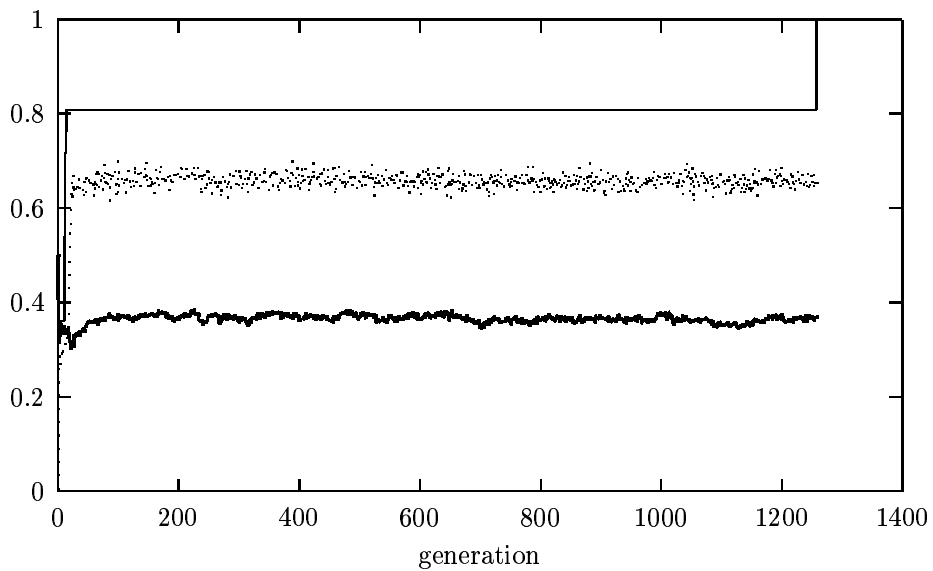


Figure 4.3: Top fitness, average fitness, average Hamming distance vs. generation with adaptive mutation.

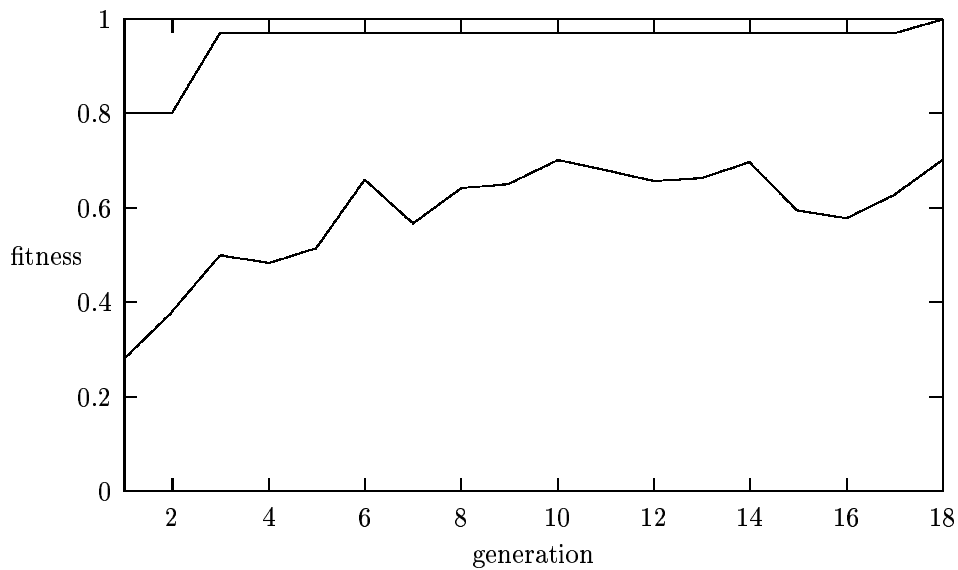


Figure 4.4: Top and average fitnesses during evolutionary run 1. The average fitness rises steadily and top fitness in jumps as is usual for GAs.

### 4.2.1 Getting started — AND gates

Ironically, I later found out that D-Latches were much easier to evolve, because of the high epistasis required to invert the output §3.2.2.

#### Evolutionary Run 1

Inverted (trivial) solution found in third generation achieving 97% fitness (see Figure 4.4), took 15 more generations to evolve the inverter which can be seen in Figure 4.5.

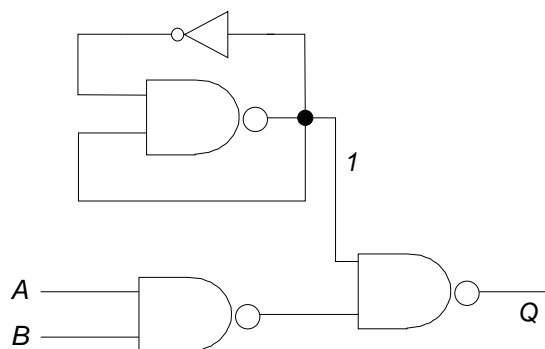


Figure 4.5: Circuit diagram of first AND circuit evolved. Feedback is used to tie a NAND gate high which is fed into another NAND gate making it an inverter.

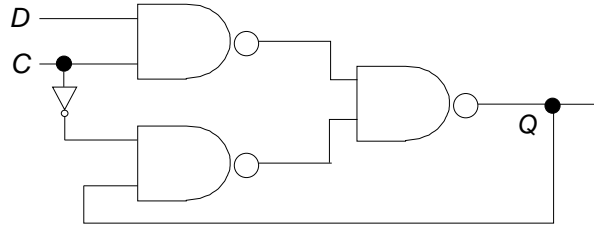


Figure 4.6: Simplest (transparent) D-Latch using NANDs and inverters.

### Evolutionary run 2

A circuit was evolved with a larger number of gates between  $B$  and the output making it react faster to changes in  $A$  than to  $B$ .

## 4.2.2 Core Objective (i) — D-Latches

Most of these were evolved quickly, usually in less than a minute. Most times the simplest possible D-Latch was evolved shown in Figure 4.6. All D-Latches were transparent.

### Evolutionary run 3

The simplest possible (not analog based) solution was evolved intrinsically over night.

### Evolutionary Run 4

The solution found was very complex. Analyzing the circuit's interconnections by tracing  $Q = u_0q$  backwards:

$$\begin{aligned} Q &= \overline{u_{10}q\overline{u_5q}} & u_9q &= \overline{\overline{u_1q}D} & u_5q &= \overline{u_1q\overline{u_6q}} \\ u_{10}q &= \overline{D\overline{u_{13}q}} & u_1q &= \overline{Q\overline{C}} & u_6q &= \overline{u_2q\overline{u_6q}} \\ u_{13}q &= \overline{u_9q\overline{u_8q}} & u_8q &= \overline{\overline{Q}\overline{C}} & u_2q &= \overline{u_6q\overline{u_2q}} \end{aligned}$$

Now note that  $u_6q = \overline{u_6q\overline{u_2q}\overline{u_6q}} =_{DM} \overline{u_2q\overline{u_6q}} + u_6q$  which expands recursively to  $u_6q = \dots \overline{u_6q\overline{u_6q}\overline{u_6q}} + u_6q = 1$ . Now:

$$\begin{aligned} Q &= \overline{u_{10}q\overline{u_5q}} = \overline{D\overline{u_{13}q}\overline{u_1q\overline{u_6q}}} = \overline{D\overline{u_9q\overline{u_8q}}\overline{Q\overline{C}}\overline{1}} = \overline{D\overline{u_1q}D\overline{Q}\overline{C}\overline{Q\overline{C}}} = \\ &= \overline{D\overline{Q\overline{C}D}\overline{Q}\overline{C}\overline{Q\overline{C}}} =_{DM} \overline{D\overline{Q\overline{C}D}\overline{Q}\overline{C}} + \overline{Q\overline{C}} =_{DM} \overline{D(\overline{Q} + C + \overline{D})(Q + C)} + \overline{Q\overline{C}} = \\ &= \overline{D\overline{Q}Q} + \overline{DCQ} + \overline{D\overline{D}Q} + \overline{D\overline{Q}C} + \overline{DCC} + \overline{DDC} + \overline{Q\overline{C}} \\ &= \overline{DCQ} + \overline{D\overline{Q}C} + \overline{DC} + \overline{Q\overline{C}} \end{aligned}$$

which is a correct solution with two redundant terms which are useful to avoid race conditions. Another interesting property of this circuit is that it is quicker at dropping its output than at raising it, as in Figure 4.7.

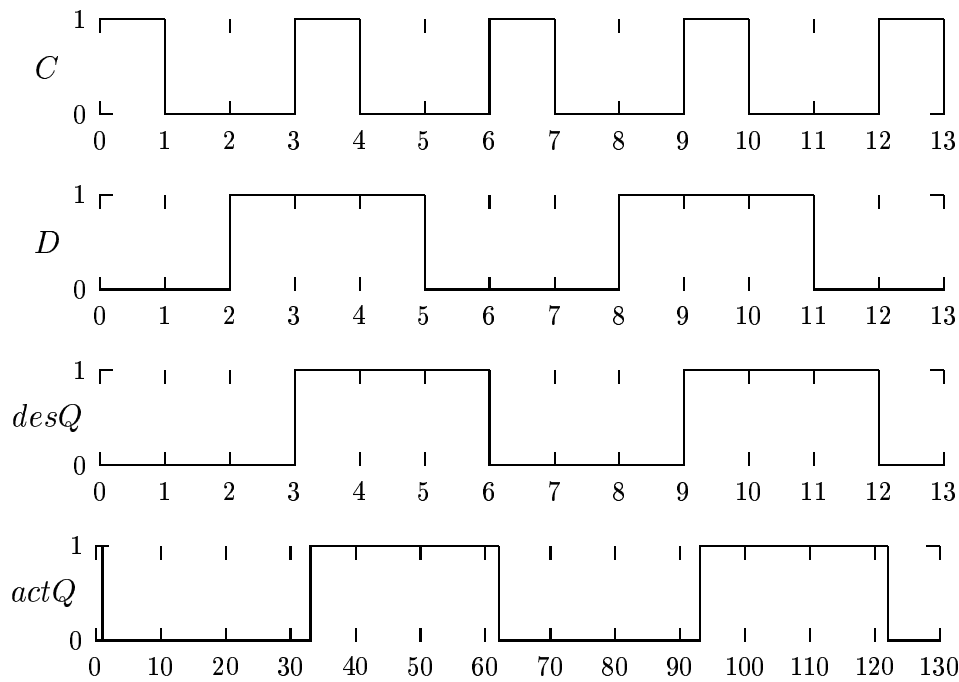


Figure 4.7: *actQ* exhibits a greater delay w.r.t. *desQ* when rising than falling.

### 4.2.3 Core Objective (ii) — Complex Boolean Functions

Functions were evolved:

1.  $Q = A \oplus B$ .
2.  $Q = (A \vee B) \oplus (C \wedge D)$ .
3.  $Q = (A \vee B) \wedge (C \vee D)$ .
4.  $Q = \overline{\overline{A \oplus B \oplus C} \wedge D}$ .

Due to the increased complexity of these functions more gates were given to evolution to play with, opening exploration to many complex structures with a lot of feedback and complex behaviour. This enabled them to sometimes get past the fitness evaluation tests without really being the circuit sought.

#### Evolutionary Run 5

A solution to Boolean function 4 above was found in 10373 generations, using 23 NANDs and 14 inverters interconnected with many complex feedback paths. This solution is incorrect, it even ignores one of the inputs. However it got past the fitness test

by oscillating its output at the right point at the same frequency as the disconnected input changed. It is a perfect solution for the task given, in which the inputs are provided in a fixed repeating sequence (which was lengthened to guard from been 'learnt') and the desired output oscillates in parts of this sequence. This is an example of genetic algorithms finding a creative solution to a problem. When this circuit was implemented on the FLEX with the right inputs, the output oscillated at 18Mhz, halfway to a low frequency oscillator.

### **Evolutionary Run 6**

Boolean function 2. above is evolved. An individual scored perfect fitness by making use of the test data pattern (which was again lengthened) instead of implementing the correct combinatorial logic.

### **Evolutionary Run 7**

This attempted to evolve boolean function 3 above. The solution found was correct though had varying delays and race conditions as can be seen in Figure 4.8.

### **Evolutionary Run 8**

This found the correct solution to Boolean function 4 above with no feedback and several redundant terms.

## **4.2.4 Extension — Multiplexers**

### **Evolutionary Run 9**

A multiplexer with two address lines and four data lines was evolved, which is not the simplest solution even including a feedback path, as in Figure 4.9. However it behaves correctly under all circumstances.

This evolutionary run was performed distributedly. In Figure 4.10 the average fitness shadows the top fitness as usual, but at generation 1223 a new client connects into the system reporting its fresh low average fitness population, which quickly improves as it receives fitter migrators.

## **4.2.5 Extension — Oscillators**

Surprisingly, most extrinsically evolved circuits were also good oscillators when later deployed on the FLEX, sometimes with unplanned interesting properties. Most oscil-

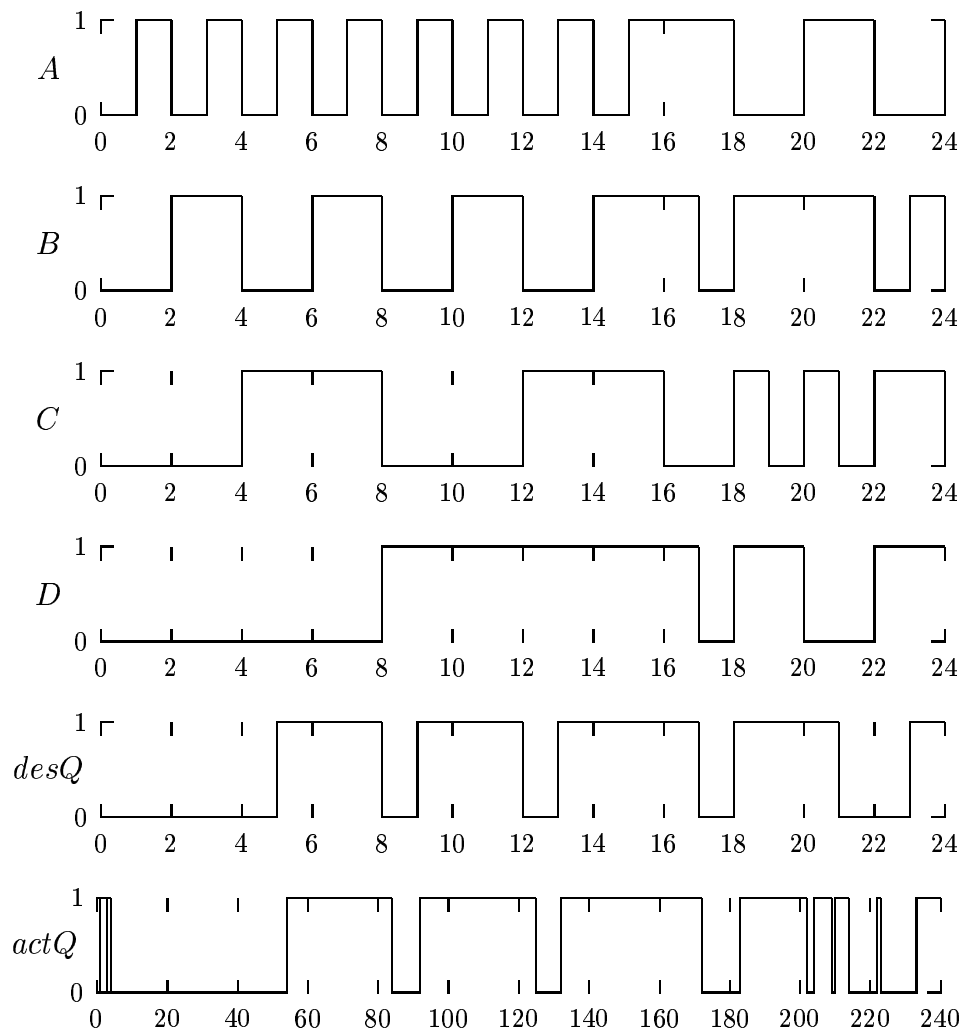


Figure 4.8: *actQ* from evolutionary run 7 is delayed and shows signs of race conditions.

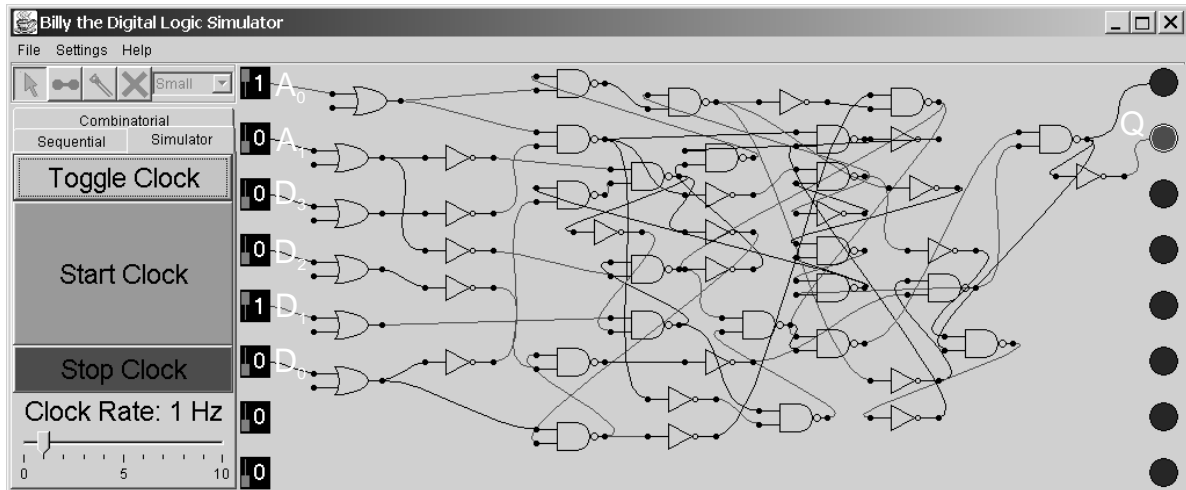


Figure 4.9: Multiplexer evolved in evolutionary run 9 loaded into Billy.

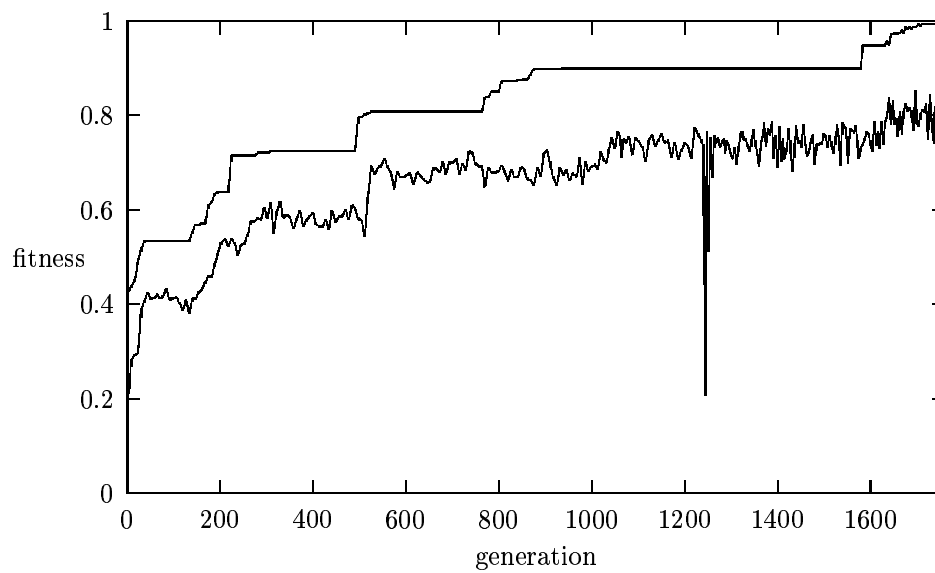


Figure 4.10: Top and average fitness during distributed evolutionary run 9.

lated in the order of 20Mhz with the fastest at 48Mhz and the slowest being arbitrarily low. All oscillator circuits contained rich feedback paths.

Some circuits exhibited odd behaviour when deployed on the FLEX, not working after every compilation or programming. This may be because of routing, placing and startup conditions varying. Another problem arose when measuring the frequency of some oscillators by incrementing a register at every evolved circuits output rising edge and displaying the high bits of the register on the LEDs. With some circuits the counter sporadically decremented as well as steadily incrementing, which could be because these circuits were oscillating at a frequency which didn't allow the carry to propagate correctly.

### **Evolutionary Run 10**

This oscillator was evolved on the simulator. When deployed on the FLEX it has some interesting properties. On one hand it is temperature dependant. At normal operating temperature it oscillates at 156Khz, but when a hot lamp is shone on it, oscillations go up to around 800Khz. However if a cold heat sink is placed on the FLEX oscillations decelerate gradually to a complete halt. This is a clear example of evolved hardware using analog properties being affected by changes in temperature. The fact that this oscillator also worked on the simulator would suggest that gate delay is responsible for its behaviour and that gate delay is reduced with increase in temperature, but the opposite is in fact the case. However crosstalk and other effects would increase with temperature and by chance may also be responsible for the oscillations. This circuit only uses 60 gates and can be reduced to 30 and may be useful as an onboard thermometer, perhaps controlling a fan directly so it spins faster as it warms up. Thorough testing hasn't been performed, but this circuit may be accurate enough to measure temperature. The circuit exhibited the same behaviour on other FLEX chips when deployed with the same programming file, but didn't work when recompiled, meaning it uses properties of some part of the FLEX but which are present in all of them.

Another surprise of evolution was that the oscillations stopped when the circuit input went high, and they resumed when it dropped again.

When this circuit was loaded into the Billy (Figure 4.11) simulator all connections oscillated in unison at a very high frequency. This would lead to massive race conditions on the hardware, so most oscillations may be ignored altogether only detecting them with a probability proportional to the temperature of the wires. Another explanation could be that on hardware some of these loops in the circuit oscillate at slightly different frequency and their signals are getting coupled through crosstalk onto a wire whose signal will exhibit a beating pattern.

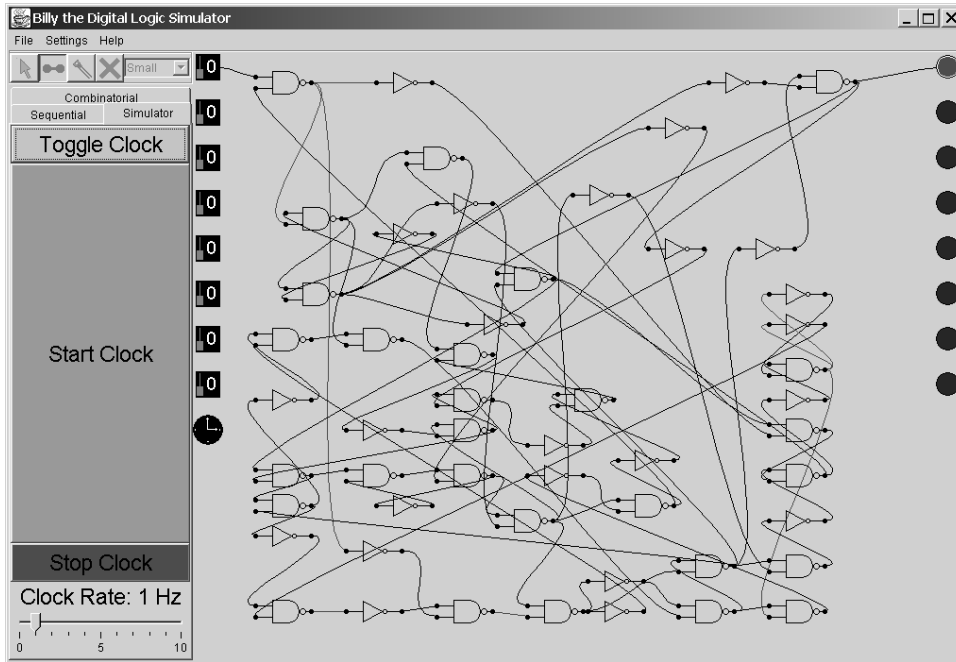


Figure 4.11: Oscillator from evolutionary run 10 loaded into Billy. Various intersecting loops can be observed.

### Evolutionary Run 11

This was an oscillator evolved intrinsically which kept its 14.6Mhz frequency stable regardless of changes in temperature.

## 4.2.6 Extension — Simple Vision

### Evolutionary Run 12

The output of the server and one of the clients of this distributed run was captured and placed side by side below to illustrate the robustness of the system to server failure. In the output below it can be seen how the server fails and Client AvT! continues processing after loosing contact and then when the server is back online, realizes its task is the same as before and just continues interacting. This run is also an example of why coevolution is effective. By allowing many sometimes unavoidable local maxima to be explored at each client evolution is less likely to reach dead-ends. For example AvT! seemed to be stuck at a local maxima of fitness 0.58 even up to generation 44, while Deity had arrived at fitness 0.69 already by generation 11 by exploring another more promising maxima. Without the help of a migrator from Deity, AvT! could've stayed for a long time at its top fitness of 0.58.

Sample output:

## 4.2. EXPERIMENTS

55

SERVER

Islands Evolution Server  
launched Fri May 04 14:45:53 GMT+01:00 2000

Evolution Task:  
Experiment: ArbitraryFunctionExperiment with:  
tSetup = 0.45  
Function: VerticalOrHorizontal Function with 2x4 1bit grid

Deployment: SimulatorDeployment with  
bitsPerVariable = 5  
stabilizers = 1  
delay = 1

Evolver: Standard Evolve with  
popSize = 500  
genotypeLength = 288  
numOfElites = 1  
Genetic Operators: [proportion - operator]  
0.3 - ExactGenotypeMutator with  
mutationsPerGenotype = 1  
0.7 - SinglePointXOver  
Selector: FitnessProportionateSelector

AvT! has registered as client ( java.awt.Point[x=0,y=0] )  
AvT! (0, 0) says Gen: 3 Top: 0.5039526306789684 Avg: 0.17152281834984892  
Deity has registered as client ( java.awt.Point[x=1,y=0] )  
AvT! (0, 0) says Gen: 7 Top: 0.5654527761937885 Avg: 0.30126643802834896  
Deity (1, 0) says Gen: 1 Top: 0.5039526306789689 Avg: 0.08534258526411576  
AvT! (0, 0) says Gen: 9 Top: 0.5654527761937885 Avg: 0.3640563796966795  
Dhruv has registered as client ( java.awt.Point[x=0,y=1] )  
Deity (1, 0) says Gen: 3 Top: 0.5654527761937885 Avg: 0.1731756189652586  
AvT! (0, 0) says Gen: 11 Top: 0.5654527761937885 Avg: 0.37417822423021907  
Dhruv (0, 1) says Gen: 1 Top: 0.478426124519397 Avg: 0.0829488978524553  
Deity (1, 0) says Gen: 5 Top: 0.5654527761937885 Avg: 0.25578140659532467

<Server Crashes>

Islands Evolution Server  
launched Fri May 04 14:49:27 GMT+01:00 2001  
Evolution Task:

Experiment: ArbitraryFunctionExperiment with:  
tSetup = 0.45  
Function: VerticalOrHorizontal Function with 2x4 1bit grid

Deployment: SimulatorDeployment with  
bitsPerVariable = 5  
stabilizers = 1  
delay = 1

Evolver: Standard Evolve with  
popSize = 500  
genotypeLength = 288  
numOfElites = 1  
Genetic Operators: [proportion - operator]  
0.3 - ExactGenotypeMutator with  
mutationsPerGenotype = 1  
0.7 - SinglePointXOver  
Selector: FitnessProportionateSelector

AvT! has registered as client ( java.awt.Point[x=0,y=0] )  
Dhruv has registered as client ( java.awt.Point[x=1,y=0] )  
Deity has registered as client ( java.awt.Point[x=0,y=1] )  
AvT! (0, 0) says Gen: 49 Top: 0.589255650988791 Avg: 0.4273713820002709  
Dhruv (1, 0) says Gen: 5 Top: 0.6765296059244602 Avg: 0.25173121239458723  
Deity (0, 1) says Gen: 9 Top: 0.6943296507508822 Avg: 0.3615984510575078  
AvT! (0, 0) says Gen: 53 Top: 0.589255650988791 Avg: 0.42791349691824687  
AvT! (0, 0) says Gen: 57 Top: 0.589255650988791 Avg: 0.42858358791948803

CLIENT1 (AvT!)

J:\release>java -Djava.security.policy=java.policy -jar  
ITClient.jar 131.111.13.97 MonicaServer AvT!  
Trying to get task from server 131.111.139.97  
Got it, task is  
Task 0:

Experiment: ArbitraryFunctionExperiment with:  
tSetup = 0.45  
Function: VerticalOrHorizontal Function with 2x4 1bit grid

Deployment: SimulatorDeployment with  
bitsPerVariable = 5  
stabilizers = 1  
delay = 1

Evolver: Standard Evolve with  
popSize = 500  
genotypeLength = 288  
numOfElites = 1  
Genetic Operators: [proportion - operator]  
0.3 - ExactGenotypeMutator with  
mutationsPerGenotype = 1  
0.7 - SinglePointXOver  
Selector: FitnessProportionateSelector

Task 1:  
IslandsEvolutionGUI

Gen: 0, Top: 0.4790701375804062, Avg: 0.08083022513353445  
Gen: 1, Top: 0.48605555238059145, Avg: 0.13503352459494264  
Gen: 2, Top: 0.5039526306789684, Avg: 0.17152281834984892  
Gen: 3, Top: 0.5415100024069428, Avg: 0.20149052138506648  
Gen: 4, Top: 0.5415100024069428, Avg: 0.23127636066727963  
Gen: 5, Top: 0.5415100024069428, Avg: 0.2720821619274588  
Gen: 6, Top: 0.5498128745591895, Avg: 0.30126643802834896  
Gen: 7, Top: 0.5654527761937885, Avg: 0.34381808834999017  
Gen: 8, Top: 0.5654527761937885, Avg: 0.3640563796966795  
[.continues processing.]  
Gen: 27, Top: 0.589255650988791, Avg: 0.4091188651100315  
Gen: 28, Top: 0.589255650988791, Avg: 0.4125675044164598  
Gen: 29, Top: 0.589255650988791, Avg: 0.4163024156264183  
Gen: 30, Top: 0.589255650988791, Avg: 0.41396428009802205  
Gen: 31, Top: 0.589255650988791, Avg: 0.4132658441464468  
Gen: 32, Top: 0.589255650988791, Avg: 0.4228282971063106  
Gen: 33, Top: 0.589255650988791, Avg: 0.41776761865474904  
Gen: 34, Top: 0.589255650988791, Avg: 0.41822407707328824  
Gen: 35, Top: 0.589255650988791, Avg: 0.4202616344567011  
Gen: 36, Top: 0.589255650988791, Avg: 0.4217585325801925  
Gen: 37, Top: 0.589255650988791, Avg: 0.4246113751766949  
Gen: 38, Top: 0.589255650988791, Avg: 0.429373807494162  
Gen: 39, Top: 0.589255650988791, Avg: 0.4269936531671747  
Gen: 40, Top: 0.589255650988791, Avg: 0.4227141155516522  
Gen: 41, Top: 0.589255650988791, Avg: 0.42650048386378536  
Gen: 42, Top: 0.589255650988791, Avg: 0.42389252606348073  
Gen: 43, Top: 0.589255650988791, Avg: 0.4272431236262502  
Gen: 44, Top: 0.589255650988791, Avg: 0.42230936796402163  
Trying to get task from server 131.111.139.97  
Got it, task is  
Task 0:

Experiment: ArbitraryFunctionExperiment with:  
tSetup = 0.45  
Function: VerticalOrHorizontal Function with 2x4 1bit grid

Deployment: SimulatorDeployment with  
bitsPerVariable = 5  
stabilizers = 1  
delay = 1

Evolver: Standard Evolve with  
popSize = 500  
genotypeLength = 288  
numOfElites = 1

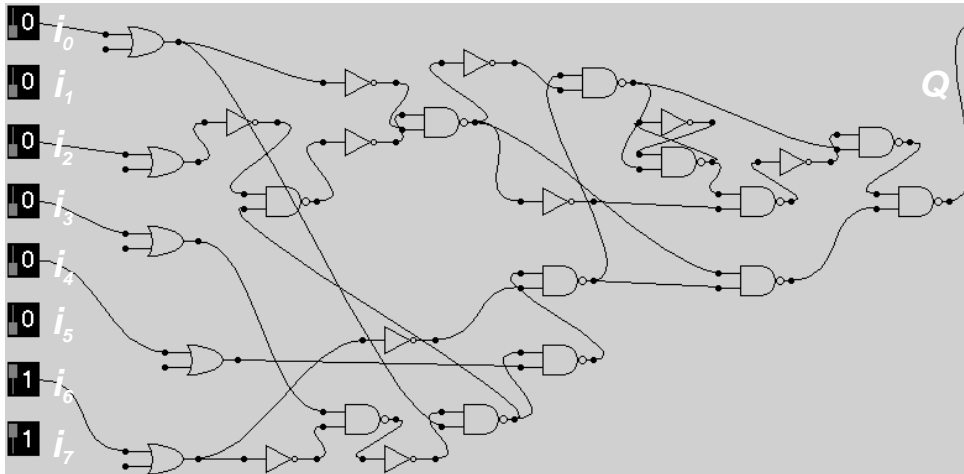


Figure 4.12: Individual evolved in evolutionary run 12.

```
Dhruv (1, 0) says Gen: 7 Top: 0.6765296059244602 Avg: 0.31417635131348415
Deity (0, 1) says Gen: 11 Top: 0.6943296507508833 Avg: 0.39175697760012423
[..etc..]
```

```
Genetic Operators: [proportion - operator]
0.3 - ExactGenotypeMutator with
      mutationsPerGenotype = 1
0.7 - SinglePointXOver
Selector: FitnessProportionateSelector
```

```
Task 1:
IslandsEvolutionGUI
```

```
Same Task! Continue interacting...
```

```
Gen: 45, Top: 0.589255650988791, Avg: 0.42905921375232686
Gen: 46, Top: 0.589255650988791, Avg: 0.42470518196314627
Gen: 47, Top: 0.589255650988791, Avg: 0.4265376885038754
Gen: 48, Top: 0.589255650988791, Avg: 0.4273713820002709
Gen: 49, Top: 0.589255650988791, Avg: 0.4260382634785035
```

```
[..etc..]
```

The circuit diagram for the solution found is shown in Figure 4.12. Only one feedback path exists and it is being used to make an inverter out of a NAND so the solution is entirely combinational. Also it can be seen that two inputs are ignored. However, these aren't necessary for distinguishing vertical from horizontal lines as defined in §3.3.4 due to the large number of don't cares, and the solution is correct. Here evolution has found a simplified solution to the problem without using any method like the Tabular Quine-McCLuskey.

### 4.3 Overall

The system completed is described in §3.7. This was used to evolve various transparent D-Latches, complex Boolean functions, multiplexers and oscillators, completing the core and some extensions of the project. Examples that stand out are “cheeky” creative solutions to boolean functions, an oscillator whose low frequency varies with

temperature, and a simplified simple image recognition circuit evolved on a flexible crash proof distributed evolution system.

However the speech recognition and non-trivial image recognition experiments weren't attempted, mainly due to lack of time. Attempts to evolve a tone differentiator intrinsically didn't succeed either, due to the high intrinsic cycle and difficulty of the task.



# Chapter 5

## Conclusions

This project's main aim was to acquire basic theoretical and working knowledge of evolvable hardware. This was achieved through background research and practical implementation. Knowledge acquired building working systems gave me experience in: designing fitness functions and test sets for problems which make the GA search more effective, analyzing the effectiveness of different genetic operators, understanding hardware particularities, interacting with hardware, simulating hardware, distributing evolution, and looking out for evolution's creative surprises.

In particular, highlights of this project include:

- Analyzing and implementing efficient fitness functions outperforming Koza's [7].
- Displaying the utility of adaptive mutation.
- Developing a configurable extensible evolution system framework.
- Extending it to evolve hardware intrinsically on a FLEX chip and extrinsically on a custom logic simulator using various genetic operators.
- Observing evolution discover new ways of solving problems.
- Evolving a temperature dependant oscillator using analog properties of hardware which also works on different FLEX chips (hadn't been achieved in [11]).
- Evolving MUXes and simplified functions performing crude image recognition.
- Developing a model for processor intensive low bandwidth distributed computing.
- Extending it to perform island based coevolution.

This completes many extensions as well as the core goals of the project, which were achieved on schedule.

This project has allowed me to have one foot in the field ready to begin further exploration if the chance arises. It was a satisfying challenge to apply knowledge gained from Computer Science Tripos lectures to this project, at the same time complementing and extending it with the new knowledge acquired.

## Future Work

In hindsight, more effort should've been put into reducing time of the intrinsic evolution cycle. This could've been done by simplifying or eliminating the compilation step as in §3.2.4. Also having a better knowledge earlier on of how effective GOs were and how to distribute evolution would've meant more circuits would have been evolved and the harder extensions tackled.

Possible extensions of this project include:

- Use the FLEX mounted on an ISA to accelerate programming time and facilitate I/O.
- Try using the genotype to encode a development process for the circuit instead of the circuit structure itself, known as grammar encoding [8].
- Attempt voice recognition by converting .wav files into a series of bytes representing the log of the frequency of each sample or feeding the analog signal in.
- Attempt to evolve fault tolerant circuits. An easy way of doing this with the existing framework is to disable elitism and apply a high mutation operator to all individuals simulating damage of components.
- Attempt an A-Life project trying to evolve primitive organisms that impulse, reproduce themselves and compete for resources; imitating TERRA [14] but on hardware. This would need reprogramming the FLEX from within itself.
- Attempt evolving primitive “Natural Computation” AI by connecting FLEXs mounted on robots with rich interaction to the environment in interactive playgrounds.
- Combine evolvable hardware with hardware specification and verification to fully automate design. The specification of a hardware component could automatically generate a fitness function by comparing the desired behaviour to

the actual behaviour of evolved circuits. And then the best solution could be fed into the theorem prover to be proved correct.

- Small efficient components could be evolved that make use of analog properties and then incorporated into large scale designs as modules. For example a variant of the oscillator described in §4.2.5 could be added as a thermometer module to the Max+Plus II toolkit..

My ventures into EH showed me it has too many paths to explore but not enough time to explore them in.



# Chapter A

## Appendix

### Source Code

Monica control module source code:

```
package es.control.server;

import es.*;
import es.evolve.*;
import es.deploy.*;
import es.experiment.*;
import es.control.*;

import java.util.Vector;

/** Used to control and manage flow of information between Genotypes
 * produced by the Evolve class, the inputs & fitnesses produced by the
 * Experiment and the outputs produced by the Deployment.
 *
 * @author Michael Garvie
 * @version
 */
public class Monica implements InteractiveTask
{
    Deployment deployment;
    Evolver evolver;
    Experiment experiment;

    // Initialize these, they may be read externally:
    private Genotype bestGenotype = new Genotype();
    private int currentGeneration = -1;
    private double currentAvgFitness = 0;

    private final double MAX_FITNESS = 0.9999999; // Using adjusted fitness.
    private final int MAX_GENERATIONS = 50000; // Overridable default value.

    /** Creates new Monica */
    public Monica( Evolver evolver, Deployment deployment, Experiment experiment )
    {
        this.evolver = evolver;
        this.deployment = deployment;
    }
}
```

```

    this.experiment = experiment;
}

/** runs evolution for this generation */
private synchronized void evolveGeneration()
{
    int inputSampleSeparation = deployment.getRecommendedInputSampleSeparation();
    double totalFitness = 0;

    for( int il = 0; il < evolver.getPopulationSize(); il++ )
    {
        Genotype genotype = evolver.getGenotype( il );
        DebugLib.debug( this, genotype ); // Prints out every genotype if we're in debug mode.
        double fitness = genotype.getFitness();
        if( fitness < 0 ) // Needs evaluating.
        {
            deployment.program( genotype );
            SampleData[] input = experiment.generateInput( inputSampleSeparation );
            SampleData[] output = deployment.run( input );
            fitness = experiment.getFitness( input, output );
            evolver.setFitness( il, fitness );
        }
        if( fitness > bestGenotype.getFitness() )
        {
            bestGenotype = ( Genotype )( genotype.clone() );
        }
        totalFitness += fitness;
    }
    evolver.evolve();
    currentAvgFitness = totalFitness / evolver.getPopulationSize();

    String narrator = "Gen: " + currentGeneration;
    narrator += ", Top: " + bestGenotype.getFitness();
    narrator += ", Avg: " + currentAvgFitness;
    DebugLib.println( narrator );
}

/** Used to get output from the task
 *
 * In this case Creates a report of what's going on inside.
 * Made generic so could implement some standard interface if created.
 * In this case it returns:
 * 0: best genotype
 * 1: Integer of current generation
 * 2: Double of average fitness
 * 3: outbound migrating individual
 * @param params can be null if this task doesn't need to know WHAT it has to output
 */
public Object get(Object params)
{
    Vector v = new Vector();
    v.addElement( bestGenotype );
    v.addElement( new Integer( currentGeneration ) );
    v.addElement( new Double( currentAvgFitness ) );
    v.addElement( evolver.pickGenotype() );

    return v;
}

```

```

/** Used to send input to the task
 * @param paramsAndWhat this could be anything, either a single object if the task
 * knows what to do with it, or a packet with variable->value pairs..
 *
 * In this case Receives inbound' migrator.
 */
public synchronized void set(Object paramsAndWhat)
{
    if( paramsAndWhat != null )
    {
        evolver.addGenotype( ( Genotype ) paramsAndWhat );
    }
}

/** Runs whatever experiment on whatever deployment for whatever generations
 *
 * Note the Evolver must already have the first generation of individuals.
 *
 * @return the best genotype
 * @param Integer with generations how many generations of evolutions to run for
 */
public Object run(Object params) throws InterruptedException
{
    int maxGenerations = MAX_GENERATIONS;
    if( params instanceof Integer )
    {
        maxGenerations = ( ( Integer ) params ).intValue();
    }
    Thread.currentThread().setPriority( Thread.MIN_PRIORITY );
    evolver.poolOfMud();
    for( currentGeneration = 0;
        currentGeneration < maxGenerations && bestGenotype.getFitness() < MAX_FITNESS;
        currentGeneration++ )
    {
        if( Thread.currentThread().isInterrupted() )
        {
            DebugLib.println("Monica Interrupted!");
            throw new InterruptedException( "When evolving in Monica." );
        }
        evolveGeneration();
    }
    return bestGenotype;
}

public String toString()
{
    String narrator = "";
    narrator += "\nExperiment: " + experiment;
    narrator += "\n\nDeployment: " + deployment;
    narrator += "\n\nEvolver: " + evolver;
    narrator += "\n";

    return narrator;
}
}

```

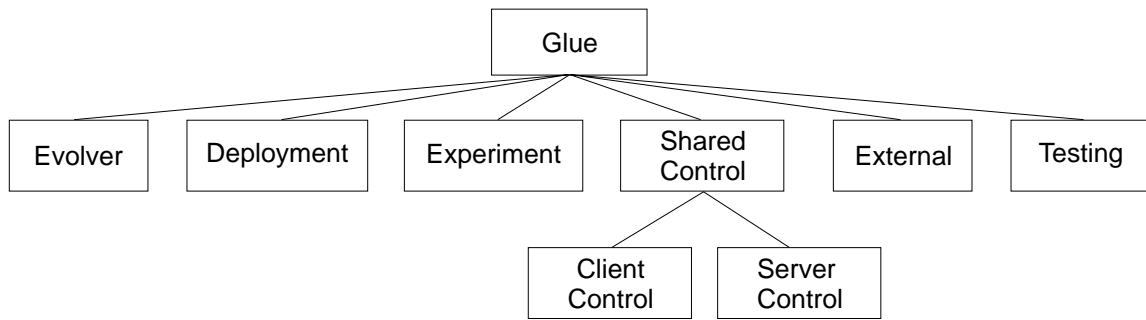


Figure A.1: Modules Hierarchy of Evolutionary System Framework.

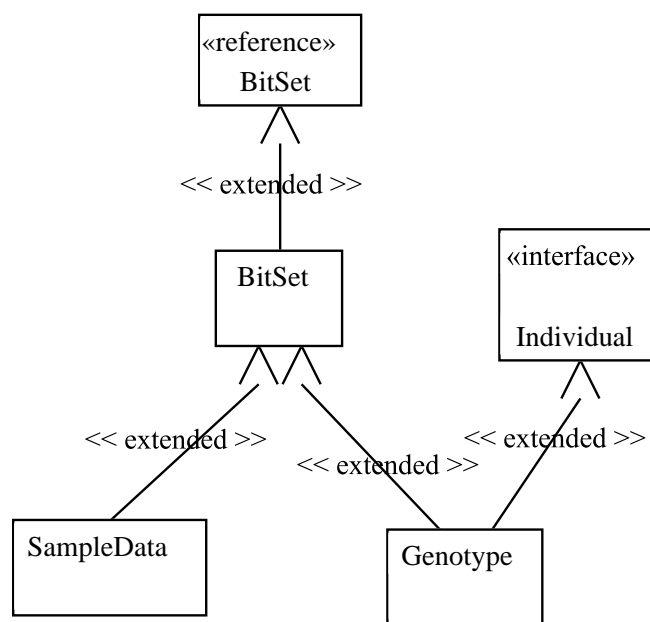


Figure A.2: Glue Module, package es.

## Class Hierarchies

This section includes UML class hierarchy diagrams <sup>1</sup> and summaries of the classes of the implemented system.

### Evolver Objects

- Population: Collection of genotypes.
- Selector: Interface for defining selection policies.

<sup>1</sup>generated using Fujaba ([http://www.uni-paderborn.de/fachbereich/AG/schaefer/ag\\_dt/PG/Fujaba/](http://www.uni-paderborn.de/fachbereich/AG/schaefer/ag_dt/PG/Fujaba/))

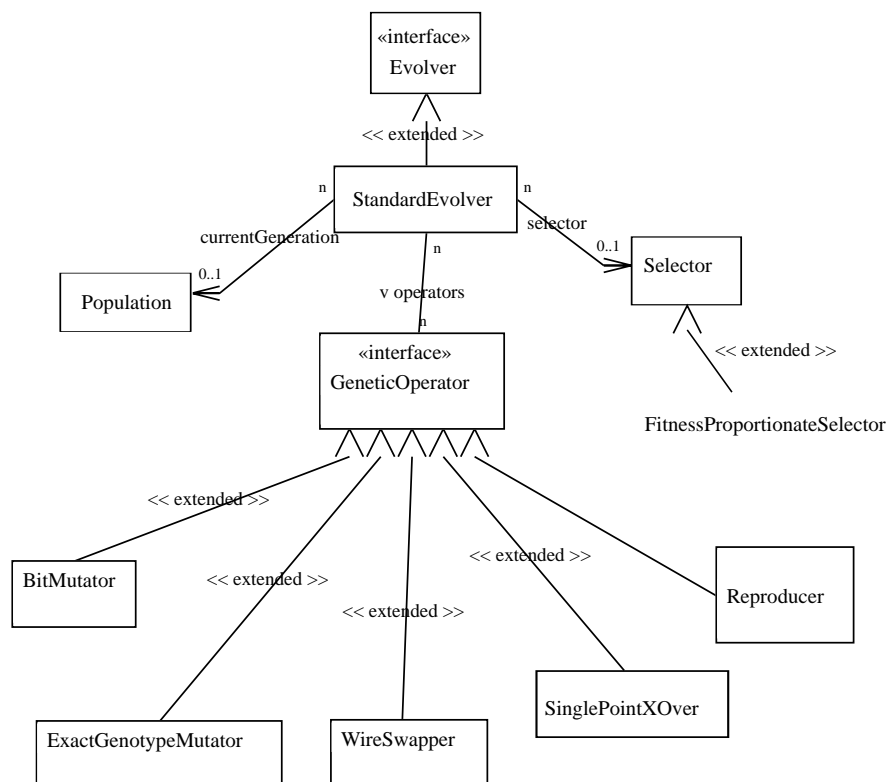


Figure A.3: Evolver Module, package `es.evolve`.

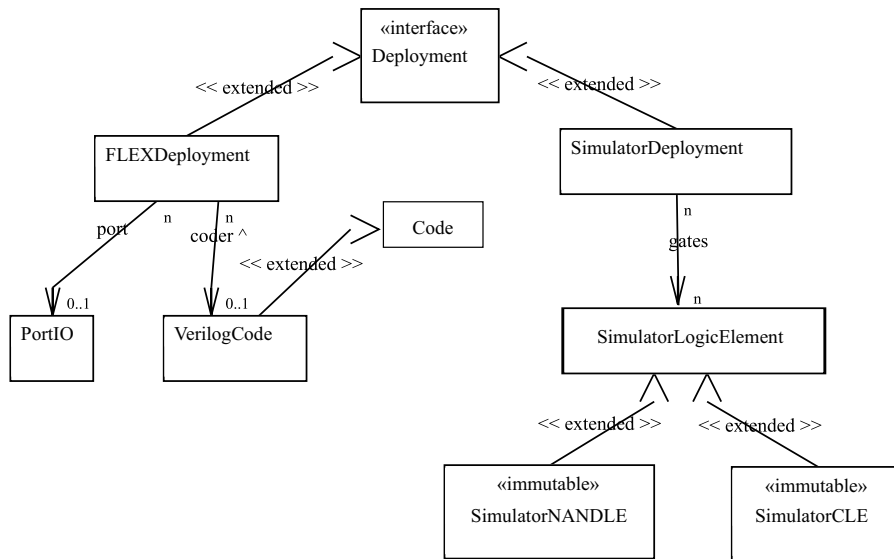


Figure A.4: Deployment Module, package `es.deploy`.

- `FitnessProportionateSelector`<sup>2</sup>
- `GeneticOperator`: Interface for defining genetic operators.
  - `Reproducer`: Copies individuals into the new population.
  - `BitMutator`: Copies individuals with their bits mutated at a per bit probability.<sup>2</sup>
  - `ExactGenotypeMutator`: Copies individuals with their bits mutated at an exact configurable number of random places.
  - `SinglePointXOver`<sup>2</sup>
  - `WireSwapper`: Swaps wire connections round. Must be instantiated with enough information to know how to map a genotype to a circuit.
- `Evolver`: Interface for a module capable of creating new populations (optionally including seed genotypes), interfacing with other modules and creating the next generation of a population.
- `StandardEvolver`: Implementation of `Evolver` by holding the current population and genetic operators and implementing the chosen GA.

<sup>2</sup>See §2.4.1 for a more detailed description of these operators and selection mechanisms.

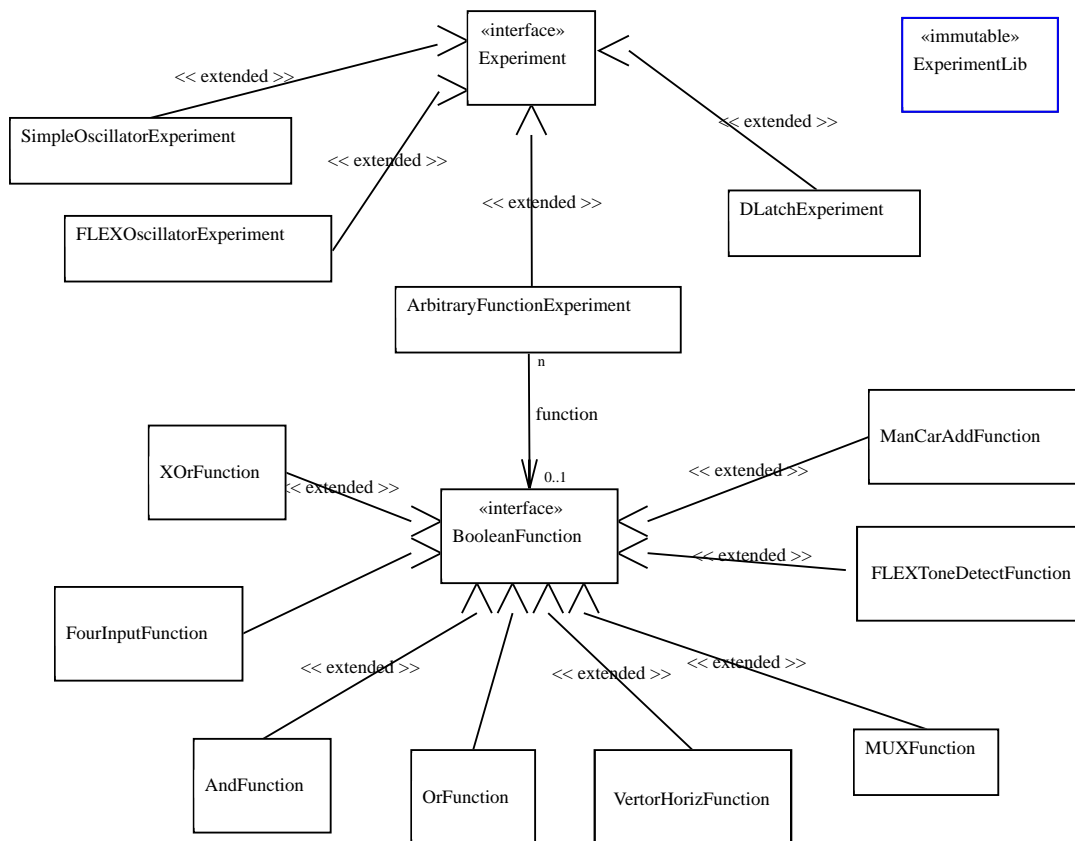


Figure A.5: Experiment Module, package `es.experiment`.

## Deployment Objects

- **Deployment**: interface for creating deployments of genotypes. A deployment must be able to program a genotype into it, and given a sequence of inputs return the sequence of outputs generated. Implementations:
  - **FLEXDeployment**: Used for intrinsic evolution, this programs a genotype onto an Altera FLEX 10K20 chip, and then performs I/O to it.
  - **SimulatorDeployment**: For extrinsic evolution, instantiates virtual circuits into a logic simulator and then runs it. It uses:
    - \* **SimulatorLogicElement**: an abstract basic virtual logic element.
    - \* **SimulatorNANDLE**: extension of the **SimulatorLogicElement** to act as a NAND gate with  $Q = \overline{i_0 i_1 i_2 \dots}$ .
    - \* **SimulatorCLE**: extension of the **SimulatorLogicElement** to act as a C-Muller majority flip-flop with  $Q = Q(i_0 i_1 i_2 \dots) + i_0 i_1 i_2 \dots$ .

## Experiment Objects

**Experiment:** Interface that must provide test inputs and a fitness function. Implementations:

- **DLatchExperiment:** Used for evolving D-Latches. A D-Latch has two inputs,  $C$  and  $D$ , and its output  $Q = CD + \overline{C}Q$ .
- **ArbitraryFunctionExperiment:** Used for evolving circuits that implement combinatorial Boolean functions.
  - **BooleanFunction:** Interface for specifying Boolean functions. Implementations:
    - \* **AndFunction:**  $Q = AB$ .
    - \* **XORFunction:**  $Q = A \oplus B$ .
    - \* **FourInputFunction:**  $Q = (A + B) \oplus (CD)$ .
    - \* **ManCarrAddFunction:**  $Q = \overline{A \oplus B} \oplus CD$ . A Manchester carry for an adder with enable.
    - \* **MUXFunction:** A multiplexer function with configurable data and address lines.
    - \* **VertorHorizFunction:** Detect a Vertical or Horizontal line.
    - \* **FLEXToneDetectFunction:** Used for evolving tone differentiators on the FLEX.
- **SimpleOscillatorExperiment:** Used for evolving oscillators extrinsically.
- **FLEXOscillatorExperiment:** Used for evolving oscillators on the FLEX.

## Control Objects

Shared Objects:

- **Task:** interface defining a runnable task.
- **InteractiveTask:** interface defining a Task which can interact during its operation.
- **InteractiveTaskServer:** interface defining an RMI remote object which is handles distributed processing of InteractiveTasks.
- **DebugLib:** library used for debugging other objects.

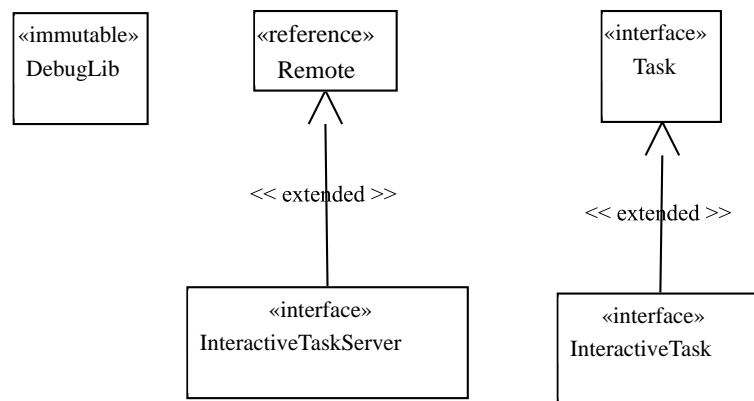


Figure A.6: Shared Control Module, package `es.control`.

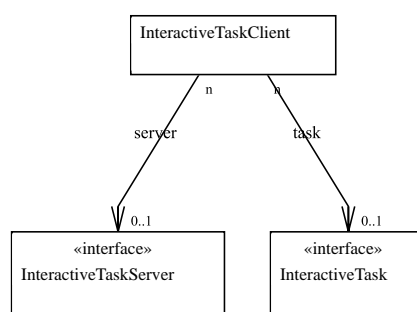
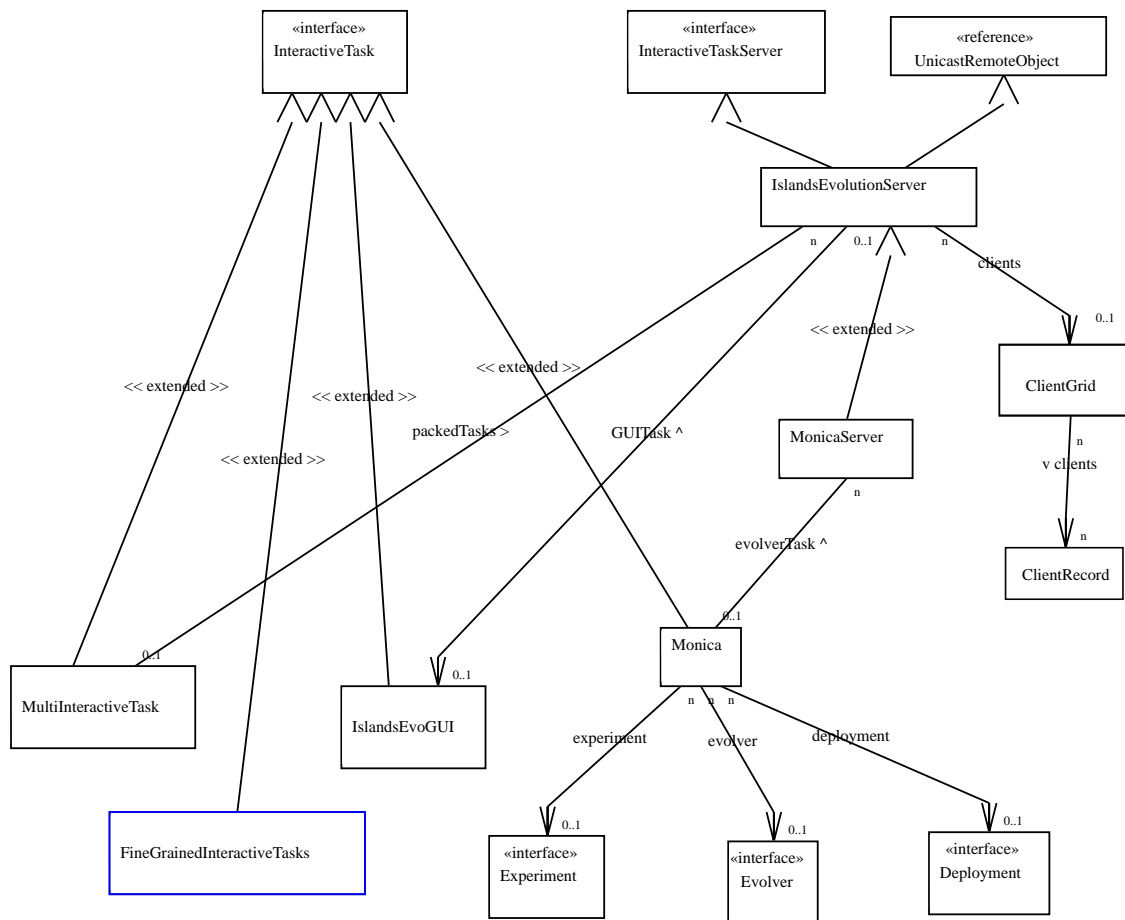


Figure A.7: Client Control Module, package `es.control.client`.

Figure A.8: Server Control Module, package `es.control.server`.

#### Client Objects:

- `InteractiveTaskClient`: connects to any `InteractiveTaskServer` and downloads its task, runs it and then interacts with the server at set intervals.

#### Server Objects:

- `MultiInteractiveTask`: bundles many `InteractiveTasks` into one.
- `FineGrainedInteractiveTasks`: allows for total server control at every interaction over which `InteractiveTasks` are being simultaneously executed on the client.
- `IslandsEvolutionServer`: implementation of `InteractiveTaskServer` to perform distributed island based coevolution.
- `ClientRecord`: holds information about clients and also knows how to paint itself as a `JComponent`.
- `ClientGrid`: stores and retrieves clients from a two dimensional grid.
- `MonicaServer`: extension of `IslandsEvolutionServer` to perform coevolution using the evolutionary system implemented in this project.
- `Monica`: implementation of `InteractiveTask`. This class is instantiated with an `Evolve`, a `Deployment`, an `Experiment`. It can then run the evolutionary process looking for a solution, with a maximum number of generations to run for. See the Appendix for source code of the main evolution loop.
- `IslandsEvoGUI`: implementation of `InteractiveTask`. Provides a GUI for the clients to check their progress and that of their neighbours.

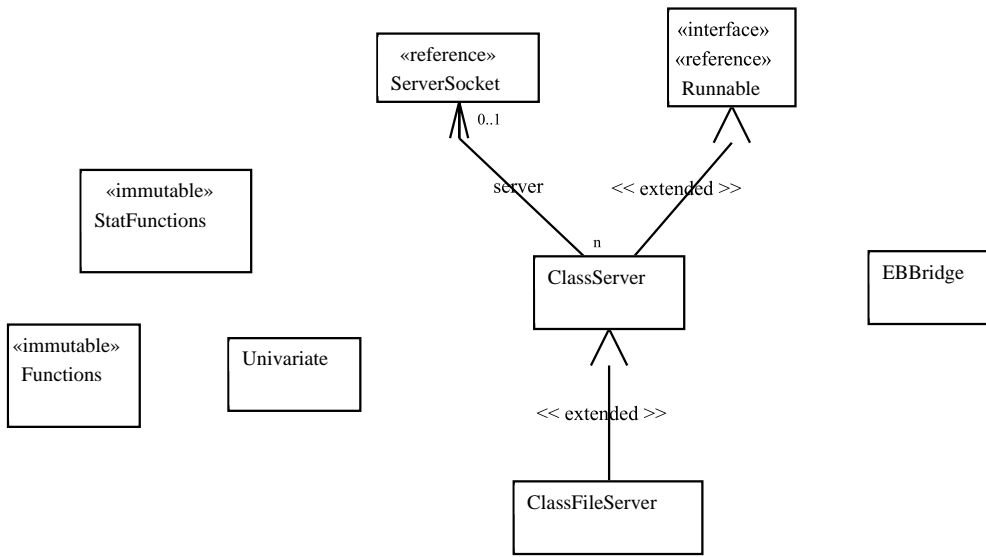


Figure A.9: External Module, package `es.external`. On the left the Statistics classes, in the middle Sun’s class file server, on the right the Bridge between the Evolutionary System Framework and Billy The Digital Simulator.

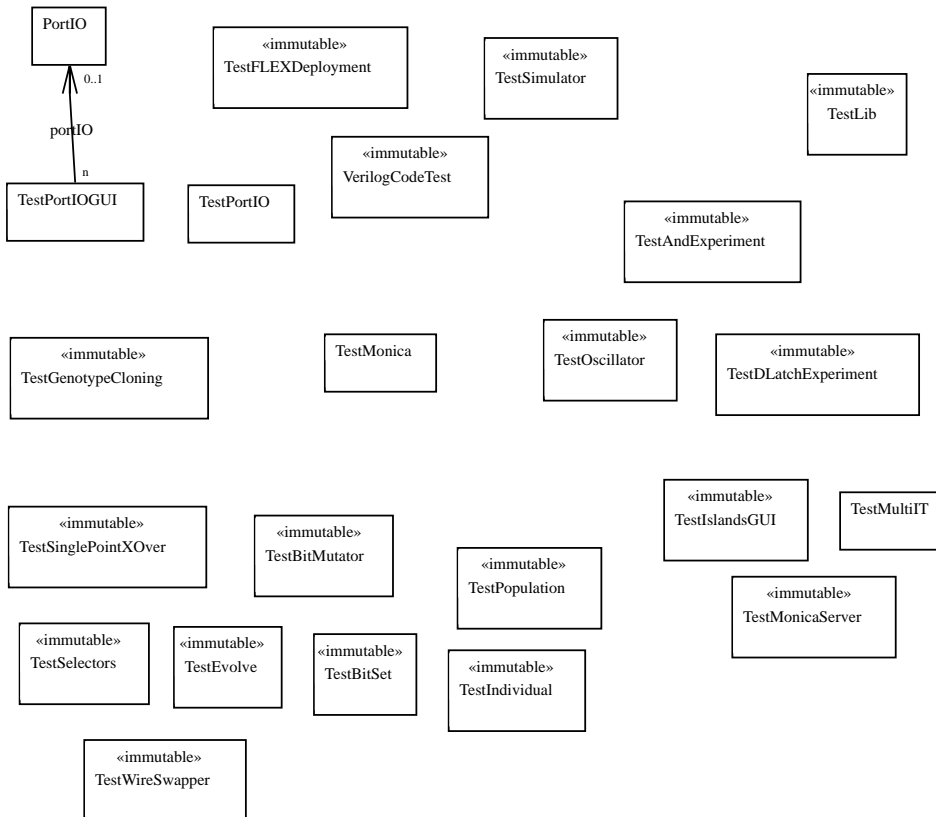


Figure A.10: Testing Module, package `es.testing`.

# Bibliography

- [1] Igor Aleksander. *How to build a mind*. Weidenfeld & Nicolson, 2000.
- [2] Alan C. Schultz at the US Naval Research Laboratory. The genetic algorithms archive. <http://www.aic.nrl.navy.mil/galist/>.
- [3] Altera Corporation. Flex 10k embedded programmable logic family data sheet, 1999. version 4.02.
- [4] M. Eigen and P. Schuster. *The hypercycle: A principle of natural self organization*. Springer-Verlag, 1979.
- [5] Inman Harvey. *Species adaptation genetic algorithms: A basis for a continuing saga*. MIT Press, 1992.
- [6] John H. Holland. *Adaptation in natural artificial systems*. University of Michigan Press, Ann Arbor, 1975.
- [7] J. Koza. *Genetic Programming*. The MIT Press, 1992.
- [8] E. Sanchez and M. Tomassini. *Towards Evolvable Hardware*. Springer, 1995.
- [9] New Scientist. Creatures from primordial silicon. *New Scientist*, November 1997.
- [10] A. Thompson. An evolved circuit, intrinsic in silicon, entwined with physics. In Tetsuya Higuchi, Masaya Iwata, and L. Weixin, editors, *Proc. 1st Int. Conf. on Evolvable Systems (ICES'96)*, volume 1259 of *LNCS*, pages 390–405. Springer-Verlag, 1997.
- [11] A. Thompson, I. Harvey, and P. Husbands. Unconstrained evolution and hard consequences. In E. Sanchez and M. Tomassini, editors, *Towards Evolvable Hardware: The evolutionary engineering approach*, volume 1062 of *LNCS*, pages 136–165. Springer-Verlag, 1996.

- [12] A. Thompson and P. Layzell. Evolution of robustness in an electronics design. In J. Miller, A. Thompson, P. Thomson, and T. Fogarty, editors, *Proc. 3rd Int. Conf. on Evolvable Systems (ICES2000): From biology to hardware*, volume 1801 of *LNCS*, pages 218–228. Springer-Verlag, 2000.
- [13] A. Thompson, P. Layzell, and R. S. Zebulum. Explorations in design space: Unconventional electronics design through artificial evolution. *IEEE Trans. Evol. Comp.*, 3(3):167–196, 1999.
- [14] Mark Ward. *Virtual Organisms*. Macmillan, 1999.