A PARALLEL APPROACH TO GRAMMATICAL EVOLUTION IN PYTHON

Submitted by Michael Stallard
for the degree of
Bachelor of Science
2006

A PARALLEL APPROACH TO GRAMMATICAL EVOLUTION IN PYTHON

Submitted by Michael Stallard

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Bachelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Signed.....(Michael Stallard)

This dissertation may be made available for consultation within the University Library and may be photocopied or lent to other libraries for the purposes of consultation.

Signed.....(Michael Stallard)

Abstract

Grammatical Evolution is the creation of computer programs using Evolutionary Computation over the search space of a language grammar. Previous systems have been written in C, C++ or Java, however due to the compiled nature of the languages used in the implementations there is the disadvantage of a clear separation between the Grammatical Evolution system and the generated solutions. Furthermore, these systems lack clarity and readability in their code and design. An implementation of Grammatical Evolution in the Python programming language would be able to take advantage of Python's inherently easy to understand syntax and Python's ability execute to itself, producing a neat integration of program production and interpretation. However, Python has the disadvantage that its execution speed is relatively slow. This document details the specification, design, and implementation for a Python Grammatical framework and also documents the experimental analysis of applying parallel computing techniques in order to attempt to alleviate performance issues as a result of using Python. It concludes that a Python Grammatical Evolution framework is a valuable tool, and that parallel computing techniques bring performance gains in terms of both the execution speed and the success rate of a Grammatical Evolution system.

Acknowledgements
I wish to thank my supervisor, Dr. Alwyn Barry for his help, support, encouragement and excellent supervision throughout the project. Thanks also go to Tom Crick for providing me with access to the Department's cluster

Special thanks go to Elvio Caruana of the Department of Computer Science at the University of Malta who provided me with access to their Linux cluster at very short notice, with tireless support and assistance, enabling me to complete the testing of the parallel GE code.

I also wish to thank my family for helping me to get this far - my parents for their love, support and guidance (not to mention proof reading!) and my brothers for achieving so much as to inspire me to attempt to emulate their successes.

Contents

Chapter 1

Introduction

"...whilst this planet has gone cycling on according to the fixed law of gravity, from so simple a beginning endless forms most beautiful and most wonderful have been, and are being, evolved."

The Origin of Species, Charles Darwin

1.1 Background

Grammatical Evolution (GE) is a Genetic Algorithm that uses a variable length linear genome to govern the mapping of Backus Naur Form (BNF) grammar definition to a program [Ryan et al., 1998]. As the target language program is described by the BNF grammar definition, any programming language may be used. GE can theoretically be used to generate programs of arbitrary complexity.

Genetic Algorithms (GAs) are search algorithms derived from the biological processes of genetics and natural selection. The purpose of a GA is to find the best possible solution to a problem from a search space of many possible valid solutions. Their history can be traced back to the 1960s and work undertaken by John Holland at the University of Michigan.

A GA works by initially randomly creating a population (gene pool) of valid possible solutions to the problem (chromosomes) which are in the form of binary strings. These chromosomes are evaluated in order to identify which of these are the most optimal solutions to the problem, with a measure of fitness - some measure of profit, utility or goodness [Goldberg, 1989]. The

GA will then breed a new generation of solutions, using the existing solutions as parents to the new generation. The higher the fitness of a chromosome, the more likelihood that its genes will be passed down into the next generation. The process of identifying and matching up the parents is called selection and there are several different ways of implementing this, each with its own advantages and disadvantages. The process of selection and creation of new generations is repeated until a satisfactory terminal condition, usually attaining a certain level of fitness of solution, has been reached.

As previously stated, Grammatical Evolution uses a GA to map a BNF grammar to a program (i.e. a solution to the problem). Each gene within a chromosome is mapped to a production rule within the BNF grammar, eventually generating a complete program that is grammatically valid. The programs in a generation are then evaluated for their fitness and if necessary a new generation is created from the current gene pool.

1.2 Project Aims

GE has previously been implemented in a variety of languages, including the original version developed in C++ [Ryan et al., 1998] and a version in Java written by a University of Bath MSc student [Griffith, 2002]. The aim of this project is to design and implement a GE system in Python.

Whilst these previous systems have successfully implemented GE, due to the compiled nature of the languages used in the implementations there is the disadvantage of a clear separation between the GE system and the generated solutions. Python is an interpreted language and can execute itself, thus providing a neat integration of both program production and fitness evaluation of programs, skipping any extraneous program compilation tasks. Indeed, interpreted languages such as Lisp have been preferred in the past for other evolutionary computing systems, such as the initial version of Genetic Programming [Koza, 1992].

Another disadvantage of existing implementations is the lack of clarity in their code. One of the natural features of Python is its tendency to produce readable clear code. A simple to understand GE system would be of great advantage to novices looking for a starting point to understand the internals of GE, as opposed to having to understand obfuscated C++ or Java code.

However, these advantages come at a cost. Pythons execution speed when compared to C++ or even Java is very slow, and given the intensive computational nature of GE, Python would be far from the first choice for an ideal language for writing a robust, efficient GE system. The next part of this project will investigate one possible way of alleviating Pythons performance limitations.

One technique for tackling computationally intensive problems is that of parallel processing - essentially, splitting a large computational task up into smaller tasks which are then executed on a number of processors. The use of parallel techniques with GAs has been widely used previously and has been found to be of great effectiveness when compared to a sequential GA [Cantú-Paz, 1997]. Whilst there has not previously been a parallel implementation of GE, it is a technique that is highly likely to bring performance gains. Hence we shall use the Python GE framework we develop to construct a parallel implementation of GE and then conduct experimental work with this to measure the effects of parallelism.

Therefore, the formal objectives of this project are to:

1. Produce a simple, easy to understand and easy to use GE framework in Python.
2. Use this framework to investigate the effect on performance that a parallel implementation of GE would bring.

This document shall begin with a literature review, providing background on the theory of GAs (and parallel variants) and GE. We shall then present a brief overview of related existing systems, before moving on to a requirements analysis for the framework. After this, we shall describe both the high level design, then the low level design and implementation, giving justification for the various design decisions. We shall then describe the testing used to ensure the system is consistent with the requirements. Then we shall present a description and discussion of the experimental work into a parallel implementation of GE, conducted using the framework we have developed. Finally, we shall present a conclusion to the project, discussing whether the project met the formal objectives listed above, identifying future work related to the project and providing personal reflection on the project as a whole.

Chapter 2

Literature Review

The purpose of this review is to research the background of Grammatical Evolution, illustrating the technology and ideas that underpin it. By doing this, we will be able to attain the understanding that is necessary to design and implement a Grammatical Evolution framework in Python.

2.1 Genetic Algorithms

2.1.1 Introduction to Genetic Algorithms

As briefly described in the Introduction chapter, Genetic Algorithms (GAs) are search algorithms that attempt to find the best possible solution to a problem from a search space of many possibly valid solutions. They are derived from the biological processes of genetics and Darwinian natural selection [Darwin, 1859]. Their history can be traced back to the work of John Holland at the University of Michigan in the 1960s. Holland's goal was to study the phenomenon of adaptation in nature and develop ways in which the mechanisms of natural adaptation could be used in computer systems. The GA was originally presented as being an abstraction of biological evolution - a method for evolving from one population of chromosomes (in a GA, these are binary strings) to a new one, using a derivation of Darwinian natural selection to identify "parents" and models of genetic operators - crossover, mutation and inversion [Mitchell, 1996].

Holland's work was highly innovative for its time, introducing multiple individual populations,

crossover and inversion operators. Previous work on Evolutionary Strategies by Rechenberg [Rechenberg, 1965] was based around the idea of simply having one parent and one child, with the child being a mutated version of the parent, and work by Fogel, Owens and Walsh on Evolutionary Programming [Fogel et al., 1966] only used the mutation operator. Holland also later developed a theoretical basis for GAs in The Fundamental Theorem of Genetic Algorithms [Holland, 1975], a topic that we shall discuss in greater detail later [Mitchell, 1996].

Goldberg identified [Goldberg, 1989] that GAs differ from more traditional search and optimisation methods in four ways:

1. GAs work with a coding of the solutions, not the solutions themselves.
2. GAs search from a population of points, not a single point
3. GAs use fitness information, not derivatives or other auxiliary knowledge
4. GAs use probabilistic transition rules, not deterministic rules

GAs have been used to solve a wide and varied range of problems - from stock market trading systems, and robotic learning; to the recommendation of music to people [Westergren, 2005].

2.1.2 Biological Foundations

At this point, we shall introduce some basic genetics in order to provide some background on the natural processes that underlie GAs.

All biological organisms are constructed from building blocks called "cells". Be it a simple single cell (*unicellular*) bacterium or complicated organism such as a human (*multicellular*), all organisms are made up from one or more of these units. This is called "The Cell Theory", and was originally elaborated by Schwann and Schleiden in the 19th Century [Schwann and Schleiden, 1847] (cited by [Mitchell, 1996]). Each cell contains the same set *chromosomes* - these are strings of DNA which act as a "blueprint" for that organism. There are two possible locations that a cell can store its chromosomes - most commonly inside a nucleus, so called *eukaryotic* organisms. The alternative are organisms that store their chromosomes in a non membrane based region called a *nucleoid*, or in small *plasmids* throughout the cell Wikipedia [2005b]. These are called *prokaryotic* organisms. For the sake of simplicity, we shall only discuss eukaryotic organisms. Furthermore, an organism can be *diploid* where the chromosomes exist in pairs, or *haploid*, in which the chromosomes are unpaired [Mitchell, 1996]. Again for

simplicity, we shall only discuss haploid organisms, as these are the basis for most GAs.

Each chromosome can be conceptually split into *genes*, each having a particular position on a chromosome called its *locus*. A gene is essentially the encoding of the information that is required in order to produce proteins - these govern the growth and functioning of the organism. These proteins lead to *traits* - physical characteristics of the organism, such as eye colour or hand size. A gene can hold various possible "value" - a certain value is called an *allele*, which gives way to variance in the expressed proteins, and hence, variance in traits, and the organism as a whole. The complete set of genetic information for an organism is called its *genome*, with a particular set of genes within a genome is called a *genotype*. A genotype will give rise to a *phenotype* - the resultant physical expression of these genes, such as eye colour, etc. [Mitchell, 1996].

Darwinian natural selection is based around the idea of "fitness" of organisms, which is typically defined as the probability that the organism will live to reproduce or as a function of the number of offspring the organism has. The higher fitness of an organism, the more likely that its genes will be passed down to future generations. In natural populations, fitness is determined by a creature's ability to survive predators, pestilence, and the other obstacles to adulthood and subsequent reproduction [Goldberg, 1989].

As we have now set a foundation for what makes up an organism, and through natural selection how potential parents with "good" genes are more likely to mate and produce offspring, we shall turn our focus to how this genetic information is passed down from generation to generation with sexual reproduction. During sexual reproduction, *recombination* (or *crossover*) occurs: in haploid reproduction, genes are exchanged between the two parents' single-strand chromosomes. Offspring are subject to mutation, in which single nucleotides (elementary bits of DNA) are changed from parent to offspring, the changes often resulting from copying errors. This is how natural selection gives rise to evolution [Mitchell, 1996].

DNA within a chromosome is gives rise to proteins through the processes of transcription, then translation. In the process of transcription, the information stored in DNA is copied into ribonucleic acid (RNA), which has three distinct roles in protein synthesis. Messenger RNA (mRNA) carries the instructions from DNA that specify the correct order of amino acids during protein synthesis. The assembly of amino acids into proteins occurs by translation of mRNA. In this process, the information in mRNA is interpreted by a second type of RNA called transfer RNA (tRNA) with the aid of a third type of RNA, ribosomal RNA (rRNA), and its associated

proteins. As the correct amino acids are brought into sequence by tRNA, they are linked by peptide bonds to make proteins. This is the so-called *Central dogma of molecular biology* [Lodish et al., 1995].

In genetic algorithms, the term chromosome typically refers to a candidate solution to a problem, often encoded as a binary string. Genes are either (more commonly) defined as being single bits, or short blocks of adjoint bits that encode a particular element of a candidate solution. It goes without saying that the number of possible alleles for a gene in a binary string is $\lceil 2^n \rceil$, with n being the number of bits within that gene (for example, a 1-bit gene can only take two values - 1 or 0). Naturally, an extended encoding system (i.e. more advanced alphabet) would give rise to a greater number of alleles per gene Lodish et al. [1995]. The initial population will consist of a predetermined number of randomly generated individuals. Generally, GAs don't draw a distinction between genotype and phenotype, and also vastly simplify the transcription and translation phases. We shall see, however, that GE is different from the vast majority of GAs by adding in a translation layer, and breaking the direct mapping between genotype and phenotype.

The selection process in GAs is still based on the concept of fitness, which is given a more GA-centric definition by Goldberg [Goldberg, 1989], in that we can think of fitness as being some measure of profit, utility or goodness that we want to maximise. In a GA, this objective measure of fitness is the determinant for whether a particular individual will live or die. There are numerous ways to implement selection within a GA, each with their own advantages and disadvantages, and are typically very task-specific.

In general, reproduction in GAs is much like haploid reproduction in nature; crossover consists of exchanging genetic information between two parents' binary strings - there are numerous methods for doing this, each with their own peculiarities and features, which we shall look at in greater detail later. Mutation is achieved by simple bit flipping of a gene at a randomly chosen locus (or, for larger alphabets, replacing a the symbol at a randomly chosen locus with a randomly chosen new symbol) Mitchell [1996]. This operation is typically limited to only occurring with a pre-determined probability, otherwise the GA would essentially degenerate into being a random search. In contrast to nature, where mutation is often stated as being the driving force behind evolution, mutation plays a decidedly secondary role in the operation of GAs. However, it still plays a vital and important role. It can perhaps be thought of as being an "insurance" operator - whilst selection and crossover effectively search and recombine existing individuals, they may occasionally be overzealous in their operation and lose potentially use-

ful genetic information. With a sufficiently small probability of application and combination with selection and crossover, mutation protects against irrecoverable loss of important genetic information [Goldberg, 1989]. Although we can see how GAs are analogous to nature, it is clear that they are a massive simplification of the immensely complex biochemical processes involved in nature.

Other genetic operators have been invented with various genetic algorithms, typically to adapt a GA to a particular task - indeed, GE has two of its own genetic operators, which are covered in more detail in the GE section of this review.

2.1.3 The Fundamental Theorem of Genetic Algorithms

We shall now look at some of the theory that underpins GAs. The traditional theory behind GAs stems from work done by Holland in the 1970s [Holland, 1992]. This theory is known as the *fundamental theorem of genetic algorithms*, or the *schema theory*. The theory, taken at a high level of description, states that GAs work by discovering, emphasising, and recombining good "building blocks" of solutions in a highly parallel fashion - this is known as the "Building Block Hypothesis". Better solutions are made up of more "good" building blocks; these are combinations or patterns of bit values that lead to higher fitness in individuals in which they are present [Mitchell, 1996].

A building block is a highly fit, short-defining-length schema. A schema in terms of a GA is a similarity template, which describes a subset of the population with similarities at certain loci along the genotype. Assuming we are using binary strings, we can create a pattern matching template by extending the alphabet with the $*$ character - this is the *wild-card* or *don't care* symbol. The template can be then used as follows - a 1 will match a 1 on a genotype, similarly a 0 will match a 0. The $*$ symbol can match either a 1 or a 0 on the genotype. A string that matches a schema is called an *instance* of that schema. The *order* of a schema is how many non-wild-card characters it has defined, and the *defining length* of a schema is the distance between its outermost non-wild-card characters [Goldberg, 1989]. For example, consider the following schemata:

$$H = \{1 * 1 * *1\}$$

We can see that it is of the order 3, and its defining length is 5.

According to Goldberg, the schema theorem states that "short, low order schemata are given exponentially increasing or decreasing numbers of samples depending on a schema's average fitness" [Goldberg, 1989]. The schema theorem is defined as being the following equation:

$$m(H, t+1) \geq m(H, t) \cdot \frac{f(H)}{\bar{f}} \left[1 - p_c \frac{\delta(H)}{l-1} - o(H) p_m\right]$$

Where:

- m is a function that gives us the number of examples of a schema at a particular time (e.g. H at time $t+1$).
- $f(H)$ gives us the average fitness of strings representing schema H
- $\bar{f}$ is the average fitness of the entire population
- $\delta(H)$ gives us the defining length of schema H
- l is the total length of a string
- $o(H)$ is the order of schema H
- p_c and p_m are the probabilities of the crossover and mutation operators being applied, respectively

This equation describes the exponential growth of "good" schema from one generation to the next. There are several implications for this theorem, the most important of which for our case is the building block hypothesis as stated previously. The schema theorem has been criticised, with arguments being given that it is not sufficiently powerful to make predictions concerning the direction of genetic search [Vose, 1993], however this is beyond the scope of this literature review and we shall not elaborate on this further.

2.1.4 Selection Methods

There are many various selection methods for GAs that have been developed over the years, each with their own particular advantages and disadvantages, and as with seemingly any competing technologies, endless claims and counterclaims have been made regarding their effectiveness. As there are too many to investigate and list in this literature review, we shall cover the details of some of the most common selection methods, which have been analysed in work by

Goldberg [Goldberg and Deb, 1991], Hancock [Hancock, 1994] and Mitchell [Mitchell, 1996].

The purpose of selection is, of course, to emphasise the fitter individuals in the population in the hope that their offspring will in turn have even higher fitness. Selection has to be balanced with variation from crossover and mutation, which is termed the "exploitation/exploration balance" [Mitchell, 1996], or "selection pressure". Too much pressure, and the search will terminate prematurely as a result of lack of diversity in the gene pool from an excessive amount of sub-optimal individuals taking over the population. Too little pressure, and evolutionary progress will be slower than necessary [Hancock, 1994].

Roulette Wheel Selection

This is also known as fitness-proportionate selection. This is the selection method used in Holland's original, and is simply based upon giving a probability of selection directly in proportion to its fitness. One can visualise this as being a biased roulette wheel, where each current string in the population has a roulette wheel slot sized in proportion to its fitness. To calculate an individual's slot size, simply divide its fitness, f, by the summation of fitness values in the population, $\sum_{i=0}^{n} f_i$ This virtual roulette wheel is then "spun" N times, where N is the number of individuals you wish to produce for your new population. On each spin, the slot that the wheel's marker lands on selects the individual to be in the pool of parents for the next generation [Mitchell, 1996].

One way of implementing this is to use the *expected values* of the individuals. This is the number of times that we expect an individual to be selected for reproduction, and is defined as being $\frac{f}{\bar{f}}$, where f is the fitness of the individual and $\bar{f}$ is the average fitness of the entire population. Let T be the summation of expected values in the population. Now, for N, choose a random integer between 0 and T. Loop through the individuals that are in the current population, and sum each their expected values until the summation is equal or greater than r. The individual selected for reproduction will be the individual who causes the summation to be equal to or greater than r.

For the relatively small populations typically used in GAs the actual number of offspring allocated to an individual is often far from its expected value; in fact, one can see that although unlikely, it is not inconceivable for a series of "spins" to result in the allocation of all the offspring for a new population to just one individual [Mitchell, 1996]. This has been termed

sampling bias.

There have been attempts to circumvent the problems with fitness-proportionate selection, such as *stochastic universal sampling* or *SUS*, developed by James Baker [Baker, 1987] (cited by [Mitchell, 1996]). This tries to combat the stochastic effects such as sampling bias associated with fitness-proportionate selection. However, the biggest problem with fitness-proportionate selection, is premature convergence. Typically, early on in the search the fitness variance of the population is high and a small number of individuals are of substantially higher fitness compared to the others. Under fitness-proportionate selection, these high fitness individuals and their descendants will multiply quickly in the population, thus preventing the GA from doing any further exploration of the other individuals. With a view to the exploitation/exploration balance, this selection method often puts too much emphasis on "exploitation" of highly fit strings at the expense of exploration of other regions of the search space early on. Later in the search, when all individuals in the population are very similar (i.e. the fitness variance is low) there are no real fitness differences for selection to exploit and evolution slows down massively. Thus, it can be seen that the rate of evolution or selection pressure depends on how varied the fitness is within the population [Mitchell, 1996].

Scaling Selection

Various attempts have been made to combat the premature convergence problems of fitness-proportionate selection by using "scaling" methods. These methods scale the expected values of individuals so that they fall within an acceptable range - if the fitness values of a population are too close to one another, the scaling will make them more distinct, and vice versa.

Sigma scaling attempts to exploit the fact that selection pressure is related to the variance of fitness within the population [Hancock, 1994]. Sigma selection aims to keep the selection pressure relatively constant over the course of the run rather than depending on the fitness variances in the population. Under sigma scaling, an individual's expected value is a function of its fitness, the population mean and the population standard deviation. An example of sigma scaling would be:

$$ExpVal(i,t) = \begin{cases} 1 + \frac{f(i) - \bar{f}(t)}{2\delta(t)} & \text{if } \delta(t) \neq 0 \\ 1.0 & \text{if } \delta(t) = 0 \end{cases}$$

Where $ExpVal(i,t)$ is the expected value of the individual i at time t, $f(i)$ is the fitness of i, $\bar{f}(t)$ is the mean fitness of the population at time t, and δt is the standard deviation of population fitness at time t. This function gives an individual with fitness one standard deviation above the mean 1.5 expected offspring. If ExpVal(i,t) was less than 0, Tanese arbitrarily reset it to 0.1, so that individuals with very low fitness still have some chance of reproducing. This expected value can then be "plugged in" to the fitness-proportionate selection method as shown above.

At the beginning of a run, when the standard deviation of finesses is typically high, the individuals of higher fitness will not be many standard deviations above the mean, and hence they will not be allocated a disproportionately large number of offspring. Likewise, later in the run, when the population is typically more converged and the standard deviation is consequently lower, the fitter individuals will stand out more, allowing evolution to continue.

Whilst this may help prevent search stagnation, it could well exacerbate the problem of premature convergence to an extremely fit individual because this method increases its advantage relative to the more average scoring individuals in the population. The sigma scaling method exhibits a slight improvement over fitness-proportionate selection, as good individuals will increase the standard deviation, thereby reducing their selective advantage somewhat, however there is still plenty of room for improvement [Hancock, 1994].

Another scaling method is that of Linear Scaling, as described in [Goldberg, 1989]. Rather than working with the expected values, this method scales the fitness values directly. The scaled fitness f' is defined as:

$$f' = af + b$$

Where f is the raw fitness, and the coefficients a and b are constants to be chosen. These can be chosen in a number of ways, but the goal is to arrange them so that the average scaled fitness f'_avg is equal to the average scaled fitness f_avg, and to cause the maximum scaled fitness to be a specified multiple of the average finesses. This is usually set to be 2. By setting these conditions, we ensure that the average population members receive one offspring copy on average, whilst the best receive the specified multiple number of copies [Goldberg, 1989].

The disadvantage of this technique is that it is possible for negative fitness calculations to arise, preventing its use. This most commonly occurs in the latter stages of a GA run, when the majority of population members are highly fit, but there are still a few members with very low

values [Goldberg, 1989]. Discarding the raw fitness information allows us to move away from the convergence problems we saw before, however it can place us at a disadvantage should we ever need to know if one individual is significantly fitter than it's nearest competitor.

Both scaling methods are rather computationally inefficient - they require two passes through the population at each generation: one pass to compute the mean fitness (and, for sigma scaling, the standard deviation) and one pass to compute the expected value of each individual [Mitchell, 1996].

Rank Selection

This, along with scaling, is another method that prevents premature convergence. Rather than creating an expected value directly from fitness, it depends on a ranking of the individuals, based on their fitness.

Rank selection prevents giving a much greater share of offspring to a small group of highly fit individuals, hence reducing the selection pressure in a population with a diverse range of individual fitness. It also keeps up selection pressure whilst there is a variance of fitness: the ratio of expected values of individuals ranked i and $i + 1$ will be the same whether their absolute fitness differences are high or low [Mitchell, 1996].

A linear ranking method was proposed by [Baker, 1985] (cited by [Mitchell, 1996]), which is as follows. Every individual in the population is ranked in increasing order of fitness, from 1 to N. The user chooses the expected value Max of the individual with rank N. Min is defined as being the expected value of the individual assigned with rank 1. Hence, the expected value of each individual i in the population at a specific time t is given by:

$$ExpVal(i,t) = Min + (Max - Min)\frac{rank(i,t)-1}{N-1}$$

One possible disadvantage of rank selection is that slowing down selection pressure can cause the GA in some cases to be slower in finding highly fit individuals. However, in many cases the increased preservation of diversity that results from ranking leads to more successful search than the premature convergence that can result from standard fitness proportionate selection. It is also computationally inefficient, as it requires sorting the entire population by rank, potentially a time consuming procedure [Mitchell, 1996], which is shown in [Goldberg and Deb, 1991] to be $O(n \log n)$.

Tournament Selection

Tournament selection is a remarkably efficient method of selection. Quite simply, n individuals are randomly chosen from the population, and then the fittest of the individuals is selected as a parent. The chosen individuals are then returned to the original population to be made available for re-selection [Mitchell, 1996]. Typically, the tournament size (n) is set to 4 or 5.

Tournament selection is very similar to ranking in terms of selection pressure and expectation except it is very efficient with a complexity of $O(n)$, compared to ranking's time complexity of $O(n \log n)$ [Goldberg and Deb, 1991].

2.1.5 Crossover Methods

Crossover or recombination is the genetic operator that exchanges genetic information from two parents to create a child. It is typically based around the use of a crossover probability; that is, the likelihood that crossover will be used to create a child (otherwise the offspring is created using verbatim copies of the parents). There are a wide range of crossover operators, but we shall only describe the ones that have researched with regard to GE, details of which can be found in [O'Neill et al., 2001] and [O'Neill and Ryan, 2000].

One Point Crossover

This is perhaps the simplest method of crossover, and was the one originally used by Holland. Once two parents have been chosen, the resultant pair of strings crosses over as follows: an integer position k is chosen at random between 1 and the string length minus one $[1, l-1]$, Two new strings are created by swapping all the characters between positions $k+1$ and l inclusively [Goldberg, 1989].

Consider the following strings:

$$S_1 = 111011$$
$$S_2 = \textit{0 1 1 1 0 1}$$

If we obtained a k of 4, we would swap the last two bits of the string, resulting in the offspring:

$$S'_1 = 1110\textit{0 1}$$
$$S'_2 = \textit{1 1 1 0}11$$

The advantage of this method is that it is clearly very simple to implement. However, it has shortcomings. Firstly, it cannot combine all possible schemas - if we have instances of 1*1****1 and ****11**, it cannot combine these to produce 1*1*11*1. Secondly, schemata with long defining lengths are prone to destruction by one-point crossover [Mitchell, 1996].

Two Point Crossover

Two-point crossover is unsurprisingly similar to one-point, except it uses two positions on the parents to swap information. For example:

$$S_1 = 111011$$
$$S_2 = \textit{0 1 1 1 0 1}$$

If we obtained a k of 2 and 4, we would swap the last bits of the strings as follows

$$S_1' = 11\textit{1 0}01$$
$$S_2' = \textit{1 1}10\textit{1 1}$$

Two-point crossover has historically been the most popular solution for the combating the schemata problems of one-point crossover. Two-point crossover is less likely to disrupt schemas with large defining lengths and can combine more schemas than one-point crossover. In addition, the segments that are exchanged do not necessarily contain the endpoints of the strings. Whilst there are still schemata that two-point crossover disrupts, it is a marked improvement over one-point crossover [Mitchell, 1996].

Homologous Crossover

Homologous crossover (HX) is a crossover operator that was originally developed for use in Genetic Programming (GP), to counter claims that crossover in GP performs no better than mutation and is destructive. Rather than swapping genes without regard to the function that the genes perform in the code generated, HX is based on the idea of "sticky crossover" Put simply, the HX operator only permits instructions at the same position in the genome to be exchanged with other instruction in the same position. Put more precisely, "sticky crossover" chooses a *sequence* of code randomly from one parent, and this is then swapped with the sequence of code in the same position from the second parent. It was found to have a significant and beneficial effect on the fitness of the best individual on the validation data and on code

bloat when applied to GP [Francone et al., 1999].

When tested with GE, it was found that the extra computational complexity of HX didn't yield significant benefits against one-point crossover. In fact, it was found that one and two point crossover led to homologous type crossover, due to the linear nature of individuals in GE.

2.1.6 Mutation

As stated in section 2.1.2, the action of mutation is very similar to mutation in nature in that a random gene has its value altered. However, unlike nature, it is not the primary driver for evolution. It is needed to guard against overzealous operation of crossover and the resultant loss of potentially useful genetic information. It is implemented as a simple bit flip operation of a randomly chosen bit of a binary string[1]. It is typically applied with a very low probability in order to prevent the GA becoming a random search as a direct result of the application of the mutation operator.

2.1.7 Population Update Models

There are different methods for creating a new generation population of individuals. The first, and more traditional approach is that of "generational replacement". This method simply creates an entirely new population for each generation (although this may contain some verbatim copies of parents, depending on the crossover probability).

An alternative approach is to only replace a few individuals in each new generation, so called *steady state* replacement. The idea behind this is that this may be a better model of the natural process. A feature of species that have short life-spans such as some insects, is that the parents lay eggs and then die before their offspring hatch. But in longer-lived species, including mammals, offspring and parents are alive concurrently. In nature this allows the parents to raise their offspring, but of use to us with GAs is that it also gives rise to competition between them [Beasley et al., 1993]. The number of individuals replaced in each generation is defined as a proportion called the *generation gap*. Whilst generational replacement has a generation gap of 1, steady state replacement often only replaces as few individuals as two, thus giving a higher

[1]Note that if we were not using a binary string for our genotype, mutation would simply be a replacement with a random value from the set possible of values; i.e. if we were using numbers from the set $\{x \text{ in } \mathbb{R} : [0, 1)\}$, one would simply choose a random real value between 0 and 1.

generation gap.

A third method is that of "elitism". Elitism forces the GA to retain some number of the best individuals at each generation. These "elite" individuals can be lost if they are not selected to reproduce or if they are destroyed by crossover or mutation. Many researchers have found that elitism significantly improves the GA's performance [Mitchell, 1996].

As we shall see later, GE uses steady state population update. However it is worth bearing these other techniques in mind, as insights into them could prove useful in the implementation stage (for example, we may need to use an alternate population update method in order to debug some part of the framework during development).

2.1.8 Parallel Genetic Algorithms

As we are aiming to implement GE in a parallel system, we shall now look at the different ways there are of implementing GAs to execute in a parallel way. Parallel GAs are in fact regarded as being easy to implement and can provide substantial gains in performance - leading to a vast amount of research being done on this particular area of GAs. This entire section is based heavily upon the information found in [Cantú-Paz, 1997], which is regarded as an authoritative survey of different parallel GAs.

There are deemed to be three main categories of parallel GAs:

1. Global single-population master-slave GAs
2. Multiple-population coarse-grained GAs
3. Single-population fine-grained GAs

We shall look at each of these in turn.

First though, we shall introduce some information on the basic principles of parallelisation. Parallel architectures are specialised computer systems that break away from the traditional Von-Neumann Single Instruction Stream Single Data Stream (SISD). The two most common parallel architectures are Multiple Instruction Stream Multiple Data Stream (MIMD) and Single Instruction Stream Multiple Data Stream (SIMD). A MIMD system is one in which multiple functional units (i.e. processors) perform different operations on different pieces of data;

simple examples of this are multiprocessor SMP machines, or a Beowulf cluster of computers. SIMD systems are those in which a single sequential processing element (the *control unit*) is connected to multiple simple processing elements, which in turn are each connected to memory, which may or may not be shared between the processing elements. The *control unit* fetches an instruction which is then broadcast to all processing elements, and causes a data element to be fetched from memory for each processing element. Each processing element will then execute the instruction on the data that had been fetched. Once all the processing elements have finished, the results are all written back to memory [Barry, 1996].

MIMD machines can be further subcategorised into *loosely coupled* and *tightly coupled*. A tightly coupled MIMD architecture system has globally shared memory available which is used by the processing units, whilst a loosely coupled MIMD architecture system has no shared memory, but relies on bus, network, or dedicated processor to processor connections to communicate data between processors [Barry, 1996].

A SIMD architecture consists of a sequential processing element controlling multiple simple processing elements that work on data from the memory. They are also known as *vector processors* or *array processors*, for obvious reasons.

The time taken for a parallel computation T_{par} will depend on three primary factors:

- T_{comp} - the time taken for the actual computational work to be completed
- T_{comms} - the time taken for the communication between processing elements
- K - a constant for things that cannot be parallelised, such as start up of the algorithm, etc.

The time taken for a parallel computation can be expressed simply as:

$$T_{par} = T_{comp} + T_{comms} + k$$

Both T_{comp} and T_{comms} are functions of n, the number of processors in a system:

$$T_{comp} = \frac{T_{seq}(1-A)}{n}$$
$$T_{comms} = T_{comm} \times n$$

Where T_{seq} is the time taken for the computation to be done sequentially, A is the fraction of work that cannot be parallelised (the *Amdahl portion*) and T_{comm} is a rough approximation of

the communication time between two processing elements.

If we define speed-up, S_n as:

$$S_n = \frac{T_{seq}}{T_{par}}$$

And define the ratio of computation time to communication time as:

$$R = \frac{T_{seq}}{T_{comm}}$$

Then speed-up can be expressed as:

$$S_n = \frac{T_{seq}}{T_{par}} = \frac{1}{A + \frac{(1-A)}{n} + \frac{n}{R}}$$

This shows that as the size of the computation increases, more worker processes can be added to increase speed up, but that only holds true as long as the communication time does not increase. Conversely, if the communication time can be reduced, more workers can also be added effectively.

It is often the case that as the size of the problem increases, the Amdahl portion A of sequential operations will decrease, and hence the problem will be more suitable for parallelisation - this is known as the *Amdahl Effect*.

The fact that communication of data is inherently slower by orders of magnitude than the processing speed of any processor means that the amount of communication required, and hence T_{comms}, by a parallel algorithm will be the limiting factor for the speed-up gained from paralellisation [Barry, 1996].

The maximum speed up S_n, known as Ahmdal's law [Amdahl, 1967] (cited by [Barry, 1996]) is :

$$S_n =< \frac{1}{A + \frac{(1-A)}{n}}$$

Global single-population master-slave GAs

This is perhaps the simplest method of parallelisation for a GA. The algorithm uses a single population, with, more commonly, the task of fitness evaluation of the individuals, and/or the

application of genetic operations done in parallel. As with the canonical GA, each individual may compete and mate with any other (hence, selection and mating are global). Global parallel GAs are commonly implemented in a master-slave topology - the master node stores the population, whilst the slave nodes carry out fitness evaluation and/or genetic operations.

We shall discount the parallelisation of genetic operations, as these operators are so simple that they are unlikely to bring significant gains in performance in comparison with parallelising the fitness evaluation, and that the added communications overhead in parallelising them would offset any benefit that it would bring.

The fitness evaluation of individuals in a population is parallelised by partitioning the population, and assigning a certain set of individuals to each processor. Communication between processors only occurs in two situations: when each slave receives its allocated subset of individuals to evaluate from the master, and then when the slaves return the fitness values to the master. If a GA stops and waits for fitness values to be returned from all slaves before proceeding into the next generation, it is termed a *synchronous* GA, otherwise, it is an *asynchronous* GA.

Communication overhead can become a significant factor as you add more processors to the system - you can reach a point where it becomes damaging to performance of the GA to add more processors, as the amount of communication performed is directly proportional to the number of processors in a system. Assuming a synchronous GA, a formula for obtaining the optimal number of slaves in a system is described in [Cantú-Paz, 1998]:

$$S^* = \sqrt{\frac{nT_{comp(f)}}{T_{comm}}} - 1$$

Where n is the size of the population, $T_{comp(f)}$ is the average time taken to evaluate one individual, and T_{comm} is the time delay of communication between a slave and the master.

The conclusion drawn by [Cantú-Paz, 1997] is that global single-population master-slave GAs are simple to implement, and are seen to be a very efficient parallelisation method when the evaluation stage requires considerable computation time. The method also has the benefit of not altering the search behaviour of a GA, and as such we can simply "plug" the parallelisation in as a fitness evaluation function.

Whilst this method is suitable for tightly coupled MIMD machines, it suffers from the lack

of parallelisation of crossover - there will be a large Amdahl portion caused by the gathering and sending of results still being sequential. Once one fitness evaluation process is complete, it must wait until all the other processes have completed, then for the master to produce the next generation and then send individuals out again for evaluation. Similarly, if all the slave processors finish their fitness evaluation at the same time, the master processor can suffer from contention. It is for these reasons that master-slave GAs have not been so popular, and tend to only be used in extreme cases where the fitness evaluation is of a very high complexity.

Multiple-population coarse-grained GAs

Multiple-population (or *multiple-deme*) coarse-grained GAs are currently the most popular method of parallelising GAs. They are characterised by the use of a few relatively large sub-populations, and migration of individuals between populations. This migration is usually synchronous, occurring at predetermined constant intervals, although it can occur asynchronously.

Research has uncovered a number of facts about the performance of different configurations of multiple-deme GAs. Firstly, early work by Grosso [Grosso, 1985] (cited by [Cantú-Paz, 1997]) identified that using smaller demes resulted in a faster improvement rate than when compared to a single large population. However, he identified that the quality of the final solution found was dependent on the migration rate between the populations, and that there is a critical migration rate below which the performance of the algorithm is hampered by the isolation of the demes, and above which a multi-deme system will find solutions of the same quality as a single population system.

It is asserted by [Cantú-Paz, 1997] that the "tuning" of multiple-deme GAs is not yet understood - there are three questions that stem from these observations that are still unanswered today:

1. How to identify the level of migration required to make a multiple-deme parallel GA behave like a single population GA
2. What is the cost of the communication involved in migration
3. Is the communication cost small enough to make this a viable alternative for the design of parallel GAs?

The effect of migration on the rate of evolutionary change has been compared to the theory of punctuated equilibria [Cohoon et al., 1987] (cited by [Cantú-Paz, 1997]). This theory states

that for the majority of time, a population is in equilibrium; that there are no significant changes in the genetic composition. However, changes on the environment can trigger a rapid evolutionary change. One thing that can change the environment is the immigration of individuals from other populations. Cohoon's study found that whilst there was not significant change between migrations, new solutions were found shortly after individuals were exchanged between populations.

The conclusion drawn by [Cantú-Paz, 1997] is that multiple-deme GAs are a simple extension of the canonical serial GA, and it requires little effort to convert a serial GA into a multiple-demed version.

Multi-deme GAs are suitable for implementation on loosely coupled MIMD architecture systems,. This method brings much greater parallelism than the master-slave method, avoiding the problems of the wait between fitness evaluation and crossover operations, and the problem of possible contention of a master processor. If migration is asynchronous, then k is practically negligible. However, is important to consider the size of the populations; as a population grows larger, the start-up time for the computation will grow, and hence the Amdahl portion will grow, reducing the benefits of parallelisation.

Single-population Fine-grained Genetic Algorithms

Single-population fine-grained GAs are GAs that are designed to run on massively parallel computers, dividing the population into a spatial structure, limiting interactions between individuals so that they can only compete and mate with neighbours. The overlapping of neighbourhoods results in the dissemination of good solutions across the entire population.

As we shall not be have access to massively parallel SIMD computers, then we shall discount their possible use.

Hierarchical Parallel Genetic Algorithms

The final type of parallel GA we shall look at is the hierarchical type. This is when different methods of parallelisation are combined to form a hierarchy. One example taken from [Cantú-Paz, 1997] is shown in figure 2.1, which at the upper level uses a multi-deme parallel GA, and uses a master-slave GA for fitness evaluation in each individual population.

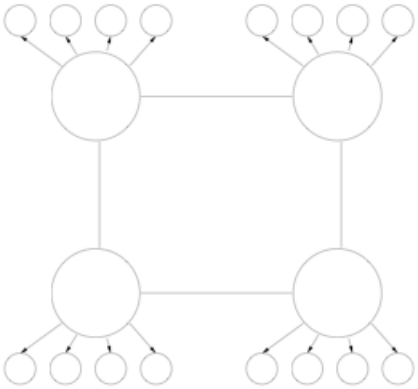

Figure 2.1: A hybrid parallel genetic algorithm.

There are many methods of parallelising GAs at multiple levels, which can introduce both theoretical and computational complexity. Due to the complex nature of hybrid parallel GAs, we shall discount them at this stage.

2.2 Grammatical Evolution

2.2.1 Introduction to GE

Grammatical Evolution is a grammar based, variable length, linear genome system for generating valid program code from a BNF definition of an arbitrary language [O'Neill, 1999]. GE originated at the University of Limerick in 1998, and the first implementation was developed by Conor Ryan, John Collins and Michael O'Neill. GE has been applied to problems such as Symbolic Regression, Symbolic Integration, finding Trigonometric Identities, the Santa Fe Trail, and Grammar Induction.

GE works by taking a BNF grammar specification of the target language as an input, and then searches over this grammar using a GA to find the optimal solution[2]. The genotype is a variable length binary string, consisting of 8-bit genes[3]. These genes are transcribed into

[2]Note that [O'Neill, 1999] asserts that the system is not limited to just using GAs as a search technique - conceivably any technique that can search variable length binary strings could be used.

[3]Whilst no solid explanation as to the choice of an 8-bit length for genes has been given in publications on GE, it can be seen from [O'Neill and Ryan, 1999b] that having a range of possible values greater than the maximum number of expansions in a production rule for a given grammar, causes *degeneracy*, which is seen to be advantageous. A possible hypothesis for using 8-bit genes is that the 256 possible values from an 8-bit gene is far in excess of the typical number of expansions for a grammar, thus giving sufficient degeneracy. It is also a relatively small

codons as integer values, the use of which shall be described later (note that this transcription correlates to the process of transcribing DNA to RNA in natural biology). Before we look at GE in any further depth, we should define what a BNF grammar is.

2.2.2 Backus-Naur form

The Backus-Naur form (BNF) is a metasyntax used to express context-free grammars: that is, a formal way to describe formal languages [Wikipedia, 2005a]. A BNF grammar consists of a quad N, T, P, S, where:

- T - set of terminals of language, which are symbols that can appear (e.g. +,-,int,return,etc.)
- N - set of non-terminals of language, which can be expanded into terminals or further non-terminals
- P - set of production rules which governs how the non-terminals will be expanded
- S - the start symbol (n.b. $S \in N$)

An example BNF grammar is given below, which can be used to solve the Trigonometric Identity problem [Ryan et al., 1998]

$$N = \{expr, op, pre\_op\}$$
$$T = \{Sin, Cos, Tan, Log, +, -, /, *, X(,)\}$$
$$S =< expr >$$

Aside from these three elements of the quad, we need to define P, the set of production rules. These have their own syntax, where $< non - term >$ represents a non-terminal, ::= means "is defined by" and | means "OR". With this notational definition out of the way, we can now list our production rules:

1. $< expr >::=$ $< expr >< op >< expr >$ (A)
   $| (< expr >< op >< expr >)$ (B)
   $|< pre\_op > (< expr >)$ (C)
   $|< var >$ (D)

gene size, thus providing computational efficiency.

2. $< op >::=$ $+$ (A)
$\mid -$ (B)
$\mid /$ (C)
$\mid *$ (D)

3. $< pre\_op >::=$ Sin (A)
$\mid Cos$ (B)
$\mid Tan$ (C)
$\mid Log$ (D)

4. $< var >::= X$

There are two important observations to take from the above grammar. Firstly, one can see that the grammar is recursive - i.e., a non-terminal may expand to itself (visible in A,B and C of rule 1). Secondly, any non-terminal can expand to a multiple of choices.

2.2.3 Expansion selection

With GE, the codons (integer values transcribed from the 8-bit genes) act as the basis of the mechanism of producing the phenotype for an individual; each codon will map to a certain expansion of a production rule in the BNF grammar.

For example, consider production rule 1. There are four possible expansions to consider; GE determines which expansion to use by calculating $c \, mod \, n$, where c is the current codon integer value, and n is number of expansion choices for current production rule. For example, if the current codon's value was 241, then the expansion to be selected for the above rule would be:

$$241 \; mod \; 4 = 1$$

Therefore the expansion $(< expr >< op >< expr >)$ would be used. Note that we are using a *linear degenerate genetic code* - there is essentially a many-to-one mapping of codons to production rules. That is, through the mod operation, codons of different values can map to the same production rule[4]. It has been found that this degeneracy is responsible for the preservation of the functionality of the phenotype, whilst still allowing for an unrestricted search over the genotypic search space. From the work by O'Neill as described in [O'Neill and Ryan,

[4]Note that a genotype is deterministic - it will always produce the same phenotype (i.e. resultant program), because the same choices will always be made when translating codons to terminals.

1999b], it has been hypothesised that the basis for this lies in Kimura's neutral theory of molecular evolution [Kimura, 1983] (cited by [O'Neill, 1999]). This theory suggests that molecular evolution is a result largely of mutations that have no effect on the phenotype (i.e. functionality). Another result from O'Neill's work on degeneracy and a corollary of Kimura's neutral theory of molecular evolution is that degeneracy has a clear and definite beneficial effect on maintaining genetic diversity within the population [O'Neill, 1999].

It is important to note at this point that this layer of translation from codon to physical attribute differs somewhat from the traditional direct genotype to phenotype mapping of most GAs. In fact, this brings the system closer still to natural biology, where, as identified in section 2.1.2, there is no direct mapping from gene to physical expression. This translation to terminals is thought by O'Neill to be very much like the translation from RNA to protein in natural genetics, albeit in a very simplified way. When genes are expressed, they generate proteins, which, either independently or, more commonly in conjunction with other proteins created by other genes, affect physical traits. We treat each transition as a protein; on their own, each transition simply cannot generate a physical trait (unless we had an unrealistically simplified BNF grammar). However, when other proteins are present, physical traits can be generated through their combinatory nature. Moreover, while a particular gene always generates the same proteins, the actual physical results depend entirely on the other proteins that are present immediately before and after its locus [Ryan et al., 1998] [5] Also, as in nature, there are no genetic operations applied to the phenotype, which means that it can be as complex as necessary, and problems such as over- and under-specification, and infeasible crossovers simply do not occur [Ryan and O'Neill, 2002].
Figure 2.2 is a useful diagram illustrating the similarities between GE and nature, from [O'Neill and Ryan, 1999a]

[5] It is interesting to note that the original GE paper [Ryan et al., 1998] contains an error - it states that a particular gene always produces the same *protein*, which is not necessarily true, given that a protein is defined in GE as being a *terminal* - the author seems to have confused the fact that a gene will always produce the same *codon*, but the protein generated will depend entirely on the codon's context in the genotype (i.e. what other codons surround it). This one-to-many protein mapping incidentally runs parallel to the recent discovery of how genes act in nature, dispelling the classical genetic hypothesis of "one-gene-one-protein". However, nature achieves this through the greatly different process of *alternative splicing* [Lopez, 1998]. This genetic process describes how the transcripts of many eukaryotic genes are shortened by RNA splicing, in which the intron sequences are removed from the mRNA precursor. The primary transcript can be spliced in different ways and thereby make different polypeptide chains from the same gene - hence, leading to different proteins being expressed [Alberts et al., 2002]. Whilst somewhat similar to GE's prune operator due to its intron removal, it is different in that it does not alter the genome whatsoever - it is carried out post-transcription. This genetic process has had little to no research as to its uses within genetic algorithms, and could prove to be of great benefit, allowing for higher efficiency with more information being stored in the genotype with greater economy.

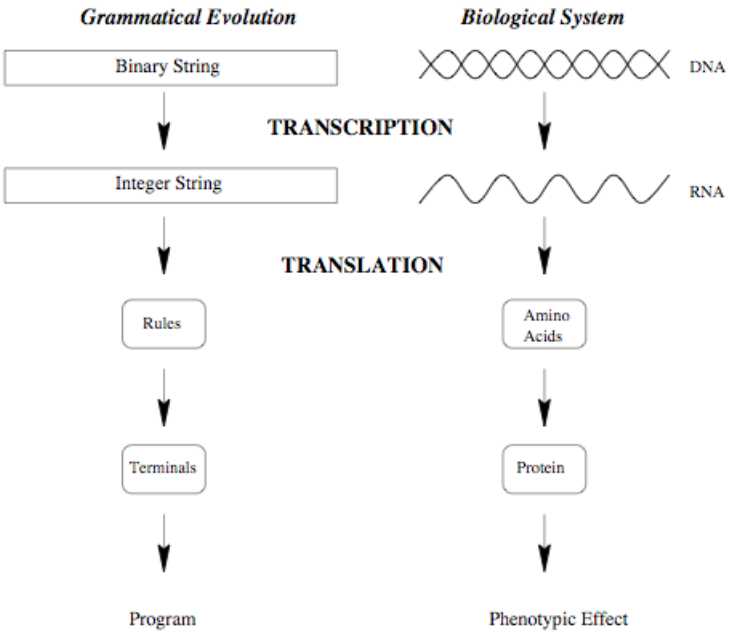

Figure 2.2: A Comparison Between GE and nature.

2.2.4 Gene wrapping

Should a particular individual run of out genes during the genotype to phenotype mapping process, there are two possible strategies that can be employed. A genotype with insufficient genes can be assumed to be an invalid individual, and assigned a harsh fitness value in order to prevent it from passing into the next generation. The alternative method is the one that is used in GE, which is a somewhat unusual approach for GAs. Rather than discarding the individual, simply "wrap" the genotype by starting reading genes from the start of the genotype once you have exhausted genes [O'Neill and Ryan, 1999c]. The wrapping technique, whilst not as commonly applied in nature, has its origins in the natural process of gene overlapping [Gerald D. Elseth, 1995]. Whilst this wrapping mechanism is beneficial, it cannot guarantee that complete individuals will be generated. This results from having a recursive grammar - consider the following production rule:

$$
\begin{aligned}
< expr > ::= \quad & < expr >< op >< expr > && (A)\\
& | \ (< expr >< op >< expr >) && (B)\\
& |< pre\_op > (< expr >) && (C)\\
& |< var > && (D)
\end{aligned}
$$

For the purposes of an example, imagine an individual with three codons. If the first codon specified expansion A from above, and then so did the second and third codon, then the translation process would never terminate, infinitely looping through expansion A. Whilst it is possible to guard against this through having a sufficiently long genotype, it is no guarantee that an infinite loop will not take place. A heuristic approach can be employed, with a counter incrementing each time the genotype is wrapped. If the counter reaches a predefined limit, then the individual will be marked as being incomplete and assigned an appropriately harsh fitness value. Wrapping has beneficial effects for genetic diversity - [O'Neill and Ryan, 1999c] identifies that when wrapping is switched off, the variety within a population is much less. Wrapping also has a large beneficial effect on increasing the probability of success within a specified number of generations.

2.2.5 Grammatical Evolution's Genetic Operators

Duplication

Gene duplication is the concept of making a copy or copies of a gene or genes, and has been used many times in evolutionary computing, it was originally proposed by Holland [Holland, 1975] (cited by [Goldberg, 1989]). It is stated in [Ryan et al., 1998] that gene duplication can:

1. Reduce the probability of a harmful mutation by provision of "backup" copies of genes within the genotype
2. Permit the possible production of new functionality from mutation of one of the copies of the gene, whilst maintaining the functionality of the original gene
3. Increasing the presence of a protein within a cell

It is stated in [Goldberg, 1989] that the use of duplication in a variable length genotype allows for the solving of problems by duplicating short defining-length building blocks to build up longer strings that increasingly cover more features of a problem. With GE, duplication is based on copying multiple genes at one time, the number of which is random. The duplicated gene is placed into the position of the last gene on the genotype. As with other genetic operators, whether the duplication operation is applied is determined by a probability.

Pruning

The stated aim of pruning is to remove *introns* from the end of the genotype[6] from a genotype - these are redundant genes that, whilst they do not contribute any feature to the phenotype, they serve to protect crucial genes from being destroyed during crossover. This is counter to what is ideal with GE and reduces the number of potentially beneficial crossovers. Therefore, a prune operator was introduced for GE, which works by simply removing any genes that are not used in the genotype to phenotype mapping. As with other genetic operators, pruning is applied probabilistically.

It was stated in [O'Neill and Ryan, 1999a] that the prune operator can actually prove to be disruptive if it is applied too often - as we use gene wrapping, the prune operator could remove an excessive amount of introns at the end of a gene, resulting in the loss of useful genetic information. It has also been found elsewhere [Levenick, 1991] (cited by [O'Neill and Ryan, 1999a]) that introns have been useful in other GAs. Therefore, it is recommended to use a low probability for the application of the prune operator, typically 0.01.

2.3 Conclusion

As seen in this literature review, there is a wide range of possible approaches implementing GAs, each with their own unique benefits and drawbacks. However, in reviewing each area of research, it seemed that there was always an ideal solution that was particularly suited to GE. We shall now look at the conclusions drawn from this literature review.

2.3.1 Choice of Selection Method

From reviewing the various methods listed in section 2.1.4, whilst we have seen that the most efficient method computationally is that of Tournament selection, the method that has been used in existing GE solutions has been Roulette Wheel selection. Therefore, whilst Tournament selection appears to offer advantages over Roulette Wheel selection, we shall implement Roulette Wheel selection in our GE system in order to maintain consistency.

[6] However, this statement may be an error on the part of the authors of GE - introns are located within the coding region of mRNA, whereas in our case, the information that is removed is after what could be considered to be GE's coding region. This is analogous to the three prime untranslated region, or 3' UTR, which is a section of mRNA defined as being anything after a stop codon, which as such, is not translated. For the sake of consistency with GE, however, we shall continue to use their notion of pruning introns [Lodish et al., 1995].

We shall also implement steady state population replacement. This is based upon findings in [Ryan and O'Neill, 1998], which show that steady state replacement is of great benefit to GE. They highlight that it helps reduce the number of invalid individuals in a population by preventing them from being passed on to the next generation as a result of generational replacement. It also had the knock-on effect that it is more likely for individuals of a higher fitness to be preserved from one generation to the next - in essence, the population as a whole is working towards finding a solution.

2.3.2 Choice of Crossover Method

From the examination of the research done by O'Neill and Ryan into crossover methods in GE ([O'Neill and Ryan, 2000] and [O'Neill et al., 2001], one-point crossover remains the current ideal operator for GE that has had research behind it. It is simple to implement and code, and due to the linear nature of individuals in GE, implicitly has homologous crossover effects.

Further analysis of the success of one-point crossover was specifically done in [O'Neill et al., 2001], identifying a *ripple* property of its effect on parse trees which could explain its success with GE.

Hence, from the current evidence that one-point crossover is the best known crossover operator for GE at the current time, we shall implement our GE system using this operator.

2.3.3 Choice of Parallel GA Method

As previously stated in section 2.1.8, we do not have access to the hardware required to use a fine grained parallel GA, which leaves our choice of parallel GA to be between either multiple-population coarse-grained or global single-population master-slave. Whilst they both have their advantages in that they are simple to implement and can bring significant performance benefits, it seems that the multiple-population coarse-grained model would bring the most benefit to us. As stated in section 2.1.8, it doesn't suffer from the inherent problems with the master-slave model, and provides the greatest parallelism of the simple (i.e. non-hybrid) methods.

Chapter 3

Discussion of Existing Systems

In this section, we shall briefly discuss the main features of existing systems relating to GE and Python in order provide a background to comparable products already available. By doing this, we shall be able to identify the main strengths and weaknesses of them, and then use this as a basis for the requirements and design process of the project.

3.1 libGE

libGE is a C++ library that implements the GE mapping process and allows for the use of any kind of search algorithm (not just a GA). It is an object oriented library, with a highly modular design approach. For example,the data structures provided with libGE are highly configurable, and can be easily used for different kinds of problems. A set of mappers is provided, which can be extended to comply with different demands (e.g. different grammar formats, different mapping processes, etc.).

Whilst this can greatly empower a researcher through configuration options, it results in a highly obfuscated and complex system that has an extremely steep learning curve. In order to highlight the complicated nature of its design, a class hierarchy diagram is included in appendix A (this has been taken from [O'Neill and Ryan, 2005]. It is felt that this serves well to illustrate the problems of trying to understand the existing code that many face when trying to learn about GE, and should we should bear this in mind throughout the design of the Python GE framework.

3.2 SciPy

SciPy is a module of scientific tools for the Python programming language. One of the included tools is the sub-module `ga`, which is a genetic algorithm library. It provides researchers with a vast range of options, from a variety of different crossover methods and different gene types, to the inclusion of a simple island model parallel GA.

The module has a highly Object Oriented (OO) approach to design, with classes being of a very fine level of granularity. For example, even the genes that make up a genotype are defined as being classes. As with libGE, this detailed and complex nature of the design results in a system that is very powerful, yet extremely difficult to get to grips with. Hence we should also bear this system in mind throughout the design stage of this project, so as to ensure that we are able to produce a framework that is simple and easy to understand.

3.3 PyGP

PyGP is a full tree-architecture genetic programming algorithm written in Python. Whilst this software is still in the alpha stage of development, it was felt worthwhile to examine it due to how GE is related to GP. It is a much simpler module that either libGE or SciPy, and takes a different approach to how it has been designed.

Rather than trying to make a "one size fits all" piece of software like libGE or SciPy, the approach with PyGP has been to build a system that does one thing, and does it well with simplicity. It has an interesting mix of OO and procedural design, with the bulk of the system consisting of procedural code. OO design techniques have been used in the design of the parse tree, with nodes consisting of user customised instantiations of abstract classes provided with the system.

Whist it is arguable that an inconsistent approach to the design of a framework such as this is not recommended, it has resulted in a system that is much easier to understand and use that either libGE and SciPy. The OO nature of the parse tree provides a researcher with the ability to extend and customise the system to their needs, whilst the simple procedural core of the system allows for a researcher to quickly grasp how the system works. This approach clearly has its advantages, and the notion of a simple core combined with OO extensibility is something that could well serve to be a useful influence in the design stage.

3.4 PyGA

PyGA is GA written in Python that has a very simple, lightweight approach to its design. It is entirely OO, but unlike libGE or SciPY, takes a much coarser level of granularity to its design, consisting of only two classes.

Whilst it does lack the vast range of features contained in the SciPy GA module, this lack of clutter leads to an implementation that is very easy to get to grips with and understand. Furthermore, most of the functionality it is missing when compared to SciPy can simply be added in by the researcher as and when they need it - the OO design of PyGA means that it is easily extended through inheritance.

3.5 Conclusion

Through the brief examination of other systems, it is clear that whilst it is useful to have tools such as libGE and SciPy that have a wide range of features and functionality, this can get in the way of providing a tool that is easy to understand and quick to learn. The example by PyGA (and to some degree PyGP) of a concise system that can easily be extended should be something that is used as a starting point for the design of our Python GE framework.

Chapter 4

Requirements

In this chapter, we shall examine the requirements elicitation and analysis techniques used throughout the project and explain the reasoning behind their choice. We shall then see the constraints within which the project and hence the requirements elicitation and analysis must be conducted within. From this basis, we shall then present a discussion of the critical findings of the requirements process, rather than listing the requirements in their entirety (note that the formal software requirements specification document is located in the Appendix). Finally, the chapter will close by examining the effectiveness of the requirements elicitation techniques used and suggest any changes to be made.

4.1 Requirements Elicitation and Analysis Techniques

This project's primary aim is to produce a system that will allow researchers to conduct experimentation with GE in a simple to use, easy to understand Python based framework. The secondary aim is to investigate the benefits that employing parallel processing techniques to GE can bring. Therefore, the remit of the project is neither a clear cut "Application" project or "Experimental" project - it contains elements of both. This in itself impacts upon the choice of requirements elicitation methods in terms of a stake-holder perspective.

There are essentially two stake-holders for the GE framework - the end-users, who will be using the framework to conduct experiments, and the actual developer of the framework itself. We are somewhat fortunate in that, due to the secondary aim of the project, we are in the position of being able to use ourselves as a resource for investigating the requirements. In order

to avoid presenting a potentially biased perspective of requirements it would clearly have been advantageous to be able to incorporate the views of other end users. However, the number of GE researchers is relatively tiny - in fact the only other real end-user resource available was the supervisor of the project. One further resource available are the existing systems we have described in the previous chapter. In particular, the code of libGE and PyGA - by examining this we were able to identify requirements that the developers of this had been able to find previously, and also to identify any potential weaknesses or improvements that could be made in the development of the Python framework.

As the supervisor's availability for the project was limited, it was decided that than rather rely upon him as a central requirements resource, he should be used to provide consultancy and feedback regarding the requirements elicited through other methods. Hence, from an end-user perspective, we provided the primary resource ourselves.

As it was clearly important to understand the workings of GE and PyGA in order to usefully conduct end-user requirements elicitation, it was decided that the first stage of requirements gathering should be a detailed analysis of the existing software, incorporating and extending from the evaluation already performed in the Literature Review. An event scenario technique as discussed in [Sommerville, 2001], was used for documenting the functionality of the relevant parts of the existing systems, as this provides a high level view of a system, whilst still retaining a suitable level of detail. Whilst performing this before brainstorming is sometimes frowned upon in requirements elicitation as being somewhat of a double-edged sword (in that it can potentially stifle creativity by leading to preconceptions about how the system should be designed), it is felt that the background knowledge and understanding that doing this would provide would be essential to effectively elicit end-user requirements. This process is termed *domain understanding* by [Sommerville, 2001] - in that analysts must develop their understanding of the application domain in order to work effectively.

Once this first phase was complete, we moved onto the next stage of our requirements strategy. This was a process of brainstorming, to identify as intended users of the system, what kind of features would be desirable to have in a system intended for performing experiments. These requirements were generated and captured using spider diagrams, allowing for easy analysis and identification of relationships and patterns. Whilst this method was simple and quick to put into practice, the drawback was the limited availability of end-users, which meant that the process had the potential to suffer from a lack of variety of thoughts and opinions, and also a potential bias as to the needs of the selected users.

The final stage of requirements elicitation then followed. Having examined the requirements gleaned from the first and second stages, it was felt that there was room for improvement in the documentation of the requirements, and also the possibility for new requirements to be identified by using an alternative method. Therefore, sequence diagrams [Pooley and Stevens, 1999] were employed. These are a fundamental feature of UML notation for describing object-oriented systems. These, along with collaboration diagrams are collectively known as interaction diagrams, which describe how objects interact to perform some piece of behaviour. Collaboration diagrams are better at showing the links between the objects in a system, whereas sequence diagrams are better at describing the sequence of message passing between objects. Given that the main problem from the analysis of the existing systems was a clear description of the actions and behaviours rather than the links between objects, it was felt that sequence diagrams were the most appropriate approach in our case. The main drawback to employing this technique was its time consuming nature - it was not feasible to fully document the existing system using sequence diagrams, so a limited number of diagrams had to be produced representing a selection of the key features and entities of the existing systems.

4.2 Constraints

It was important that we identified constraints upon the project before we embarked upon the actual elicitation process, as constraints have a critical impact upon the determination of validity and usefulness of, and the ability to filter, the requirements gathered. In order to determine what these constraints were, and the extent to which they would affect the project, a number of resources were used. These included the specification document for the final year project module, the literature review, experience of previous projects, discussion with the project supervisor and the project proposal.
The main constraints were:-

1. Time - perhaps the greatest constraint on the project, the project must be handed in by 12pm on Monday the 8^{th} of May 2006. This means that the functionality and extent of the project will have a finite limit.

2. Finance - there shall be no financial support for the project, and as such, any tools to be used should be freely available.

3. Cluster computing facilities - the project will have the Department's 20 node Linux cluster at its disposal for the testing and experimentation of the parallel implementation

of GE, therefore the configuration and design of the parallel code should take this into consideration.

4. Use of standard Python - the code should use a standard version of the Python interpreter, compatible with version 2.3 of Python in order to allow for use on the Department's cluster. This means that any special features added in version 2.4 of Python must be omitted from implementation.

5. Personnel - only one developer is available for the entire project, and hence, as per the Time constraint, this limits the scope and extent of the project.

6. Readability of code - one of the key aims of the project is to produce a GE implementation that is written in clear, simple to understand code. Hence, caution should be applied when selecting features from the requirements elicitation process so that they don't result in extraneous and overcomplicated code.

7. Cross-platform compatibility - The system shall be developed using Mac OS X 10.4, however it must be able to work on Linux in order for the parallel experimentation to be carried out on the Department's cluster.

4.3 Requirements Specification

Having described the methods used for requirements elicitation and analysis, and the constraints that impact upon the project and hence requirements, we shall now highlight the key findings of the requirements gathering process.

4.3.1 Grammatical Evolution Framework

4.3.2 General Requirements

Given that one of the aims of the project is to produce an implementation of GE that is easy to read and understand, one of the very first requirements that was found was that the code should adhere to a relevant standard. More precisely, the code should be written to the de facto standard that is illustrated in the official Python style guide [van Rossum and Warsaw, 2001]. This standard should be reflected within any existing code in PyGA, thus the code should be re-factored if necessary. It was decided from a combination of the findings of the examination of the original GE code and the brainstorming phase that code readability should be given

precedence over performance, albeit within reason.

There is no need for a graphical user interface to be included with the framework, although the code should not prevent any extension of the framework to allow this. There are no specific security requirements with regard to the framework. No formal framework documentation is required, as it is assumed the researcher will be aware of how GE works and the code should be concise enough to be easily understood by a competent programmer. However, some limited documentation within the code should be provided in the form of pydoc [van Rossum and Drake, 2006b] compatible comments.

Further from the constraint that the framework should work with a standard Python interpreter, the framework should aim to minimise the burden of installation for the researcher - it should avoid the use of non-standard libraries where possible, and utilise freely available modules that are written in pure Python (and hence can be included in the framework without the need for special installation on a test system) should the need for external modules be unavoidable.

Finally, the framework should support two standard GE tests, the Symbolic Regression and Symbolic Integration problems (as described in [Ryan and O'Neill, 1998]), "out of the box", in order to provide researchers with an example problem with which to learn and understand the framework.

Genetic Algorithm

From the evaluation of GE and PyGA, a number of requirements were obtained relating to the genetic algorithm that would be used. Naturally, the first requirements that were obtained were the basic and obvious ones regarding GAs. For instance, the default behaviour of the GA should be that it will operate over a search space of individuals with binary string genotypes, mapping each individual's genotype to its respective program using the specified grammar for the GE problem. It should run this program, and evaluate it, assigning it with a fitness value (i.e., with a maximisation problem, a "better" individual should receive a higher score). Any individuals that are deemed to be invalid should be assigned a suitably harsh fitness value (e.g. zero for a maximisation problem) to see that they do not pass into the next generation. A selection of parents should be made, using a specified method, and then in line with the specified population update method, these parents should be crossed over using the appropriate method. Any offspring created should have any applicable genetic operators applied before being placed into the new population. This process should iterate until the defined end conditions are met -

by default, either the optimal individual has been found, or the specified maximum number of generations has been met.

In line with the GA that is used with GE, it must support the use of a variable length genotype for individuals, with the ability to specify a default initial starting length when the individual is first created. The GA must be modular, in the sense that it should be straightforward for a researcher to customise it by altering the choice of selection method, crossover method, population update method, etc. At the very least, the GA should support the one-point crossover, roulette wheel selection and steady state population update methods highlighted in the literature review as being the suitable choices for the implementation. In addition to mutation, the GE specific genetic operators duplication and pruning should be made available.

Further to the provision of ease of customisation of the GA to use different selection methods, etc., through the brainstorming session, a requirement that a simple way to configure many of the commonly altered settings in an experiment (e.g. population size, mutation rate) should be included. This should be some form of configuration file external to the Python GE code, to save the researcher hunting through code to alter parameters.

The brainstorming also elicited a performance related requirement of the framework. The Python implementation of GE should theoretically match the original implementation of GE in terms of average number of generations to find the optimal solution. However, due to the lack of specific information on parameterisation in the papers published on GE, the performance requirement was deemed to be more of a loose fit in that it should approach or match the performance of GE.

Code Generation

As stated above, the GA must convert the genotype into a program in accordance with the grammar specification associated with the GE problem. Through a combination of examination of the original GE code and the brainstorming session, it was decided that it should be required that the framework should read in the grammar in standard BNF format from a plain text file (rather than some sort of meta-format defined in XML) in order to preserve consistency.

When generating the code, the framework should support both modes of operation of GE with regard to gene wrapping - it should either wrap the genotype if the mapping process runs out of genes before it terminates, or assign the individual a suitably harsh fitness value. This should

be configurable from an external configuration file. Also in accordance with the original GE code, there must be a heuristic in place in order to prevent infinite wrapping of the genotype should a grammar that allows this be used.

Fitness Evaluation

In order to evaluate the fitness of an individual its generated program code should be run, with zero or more possible variables, and potentially be compared with one or more target values (e.g. with a symbolic regression problem). Hence, it is required that the program should have a mechanism for reading in any variables and/or target values from a file.

Given the vast potential for different methods of running generated code (e.g. the difference between the symbolic regression problem and the Santa Fe ant trail problem), fitness evaluation should be modular and allow for easy adaptation and customisation by a researcher.

Reporting of Results

It was determined in the brainstorming phase that it is essential for a researcher to not only log statistics about a GE run, but that the framework should provide feedback throughout the duration of the GE run to allow the researcher to see how the run is progressing. In line with the general theme of ease of customisation, the reporting facility should be simple to adapt and change to the researcher's own needs.

4.3.3 Parallel Implementation

The requirements for the parallel implementation of GE were a result of the brainstorming session, rather than the analysis of GE and PyGA. The first clear requirement obtained was that the parallel implementation should be a simple extension of the Python GE framework, in order to maintain the simplicity of code and also to test the usability of the framework as a research tool. The implementation shall use a multiple-population coarse-grained model of parallel genetic algorithm, in line with the findings of the literature review. The implementation should allow variables such as migration rate, number of nodes in the topology and the means of communication within the topology (i.e. 2-D Toroidal Mesh vs. Ring, etc.) to be altered, although it is not critical that these should be contained within an external configuration file.

4.4 Conclusion

Overall, the strategy of employing multiple techniques worked well, providing a full and useful set of system requirements. Through the combination of techniques, a greater range of requirements was found, and also it aided the prioritisation of requirements based upon the frequency of occurrence of a given requirement.

The examination of the system with event scenarios worked particularly well, allowing for a much clearer view of the convoluted original GE code. This was however, a time consuming process to undertake, although it's more likely that this is a side effect of the complexity of the GE code rather than any inefficiency in the method used.

The brainstorming process was quick and efficient, providing the bulk of the requirements pertaining to the need to allow ease of customisation of the system for effective use by a researcher. The main drawback associated with this process was that, due to the limited range of needs of the participant of the brainstorming session, it could have resulted in an unavoidable bias in the requirements generated. This is something that can only really be seen with time when other researchers use the Python GE framework. The other problem with the brainstorming session was that the requirements were found to be not immediately clear from the way that they were documented, which resulted in the extra phase of sequence diagrams.

The sequence diagrams were successful in their application of trying to help formalise and make clearer certain aspects of requirements found, although they were slow to create. They also did not add substantially to the requirements already known, however it was felt that the benefit that they brought in organising the findings of the previous phases outweighed this drawback.

Chapter 5

Design

In this chapter, we shall describe the high-level design process for both the Python Grammatical Evolution framework and the parallel implementation of GE. As they are essentially two separate entities we shall split this chapter into two key sections discussing both. We shall begin by providing some background as to the key pre-design stage decision of basing the framework on the existing Python Genetic Algorithm, PyGA. Following on from this is a discussion of the various design life cycles that were considered, and the choice of life cycle for the project. With this in mind, we will then discuss the actual high level design problems and design choices made for the GE framework. Finally, the chapter will build on this to illustrate the high level design of the Parallel GE implementation.

5.1 Grammatical Evolution Framework

5.1.1 Background

The earliest, and perhaps most critical design decision to be made was whether or not to create a GE framework completely from scratch or to base it upon existing publicly available code. There were strong arguments for and against both approaches. By creating a completely new framework we could tailor it precisely towards GE, which could avoid any potential for cluttering the implementation with unneeded or unsuitable legacy code. However, doing so would result in a much larger implementation stage of the project, resulting in less time available for testing and experimentation, which has been cited by [Griffith, 2002] as being a factor in the difficulty of being able to reproduce the results of the original GE system. Conversely, in bas-

ing the GE framework upon an existing system, time must be taken to understand that system's code, and adapt it to our needs. Furthermore, the code would have to match the requirements for readability and conciseness stated in section 4.3.2. However, the clear advantages in this approach are the reduced implementation time and, to a certain extent, a reduced design time, as the foundations of the system will already be in place.

As mentioned previously (see section), PyGA is a clean, simple and concise implementation of a Genetic Algorithm. Whilst it lacks much of the key functionality needed for a GE system as detailed in our requirements, it fulfils the need for simplicity, readability and ease of use. It is designed around the Object-Oriented (OO) paradigm. Despite the performance hit of using OO with Python, it provides excellent modularity of code and ease of modification and extension. Customisation of PyGA is provided through inheritance rather than object aggregation. Whilst in many cases, hiding the basis of the implementation through aggregation is of benefit (again in line with the criterion of clarity of implementation), an inheritance approach to customisation is more desirable.

Given the excellent foundation that PyGA provides us, it was decided that the optimal design choice was to build upon and extend it rather than design and implement a system from scratch, given the time constraints of the project.

At this stage of the project, it was deemed appropriate to give the framework a name. After some deliberation, the name GEPy was chosen. Its etymology can be traced to two key factors: firstly, "Py" is used as an abbreviation for Python (this is somewhat a tradition for modules and programs written in Python), with the "GE" abbreviating "Grammatical Evolution". Secondly, the term GEPy is a slight play on words, being a phonetically similar word to Guppy, a fish whose most famous characteristic is its propensity for breeding [Wikipedia, 2006].

5.1.2 Design Life Cycle

Before any further design work was to be carried out, it was important to decide upon a software design process, or life cycle, for the project. A formal process of development is desirable, as it will provide a stronger structure to the project by enabling more precise time planning and putting in place more well defined project milestones. As there is no "perfect" life cycle for a software project, the choice of life cycle should made by judging the advantages and disadvantages of various models, and selecting the most appropriate for our situation. Be-

low, we shall briefly discuss the decision making process behind our choice of software life cycle.

As the project already has a solid set of requirements, a Rapid Application Development approach, whilst desirable for its characteristics of speed of development and quality of code, is inappropriate. For similar reasons, the Spiral life cycle can be discounted - we have definite goals in mind, as opposed to a rough idea of them, so the spiral approach will not benefit us. Given that this is an individual project as opposed to being team based, this opens up the question of Waterfall derived life cycles. Furthermore, given the rigid nature of the requirements, this negates one of the main criticisms of the Waterfall family of life cycles - that it is inflexible to change.

It was decided to use a modified version of a particular derivative of the Waterfall life cycle known as the V-Model [Cotterell and Hughes, 2002]. Like the Waterfall model, the V-Model is a sequential series of processes, with each phase having to reach completion before the next may begin.

The choice of testing procedures within a software project has a large influence upon the choice of life cycle model to be used. The testing techniques that the V-Model focusses on, and the tightly integrated way of validating requirements (with testing procedures being developed early in the life cycle before any of the implementation work is done) when compared to the standard Waterfall model coincide with the choices we made regarding testing (see chapter 7), and as such lead to it being highly suitable for our needs in this respect. We shall not cover the details of the test planning in this section, however we shall see it in chapter 7.

A graphical representation of our variant of the V-Model is shown in Figure 5.1. The key modification we made was to form an iterative cycle through Low Level Design and Implementation, taking inspiration from iterative design processes. This was due to the relative lack of experience in Python development, which would make it impractical to establish a complete low level design not subject to change and improvement as the implementation of the framework progressed.

The other modification that was made was with regard to testing. Whilst the V-Model has been criticised for lacking a clear strategy for dealing with the problems found in the testing stages, we are fortunate in the fact that this is a relatively small project. Hence we can alter the model slightly and allow ourselves to backtrack to the Implementation stage if problems

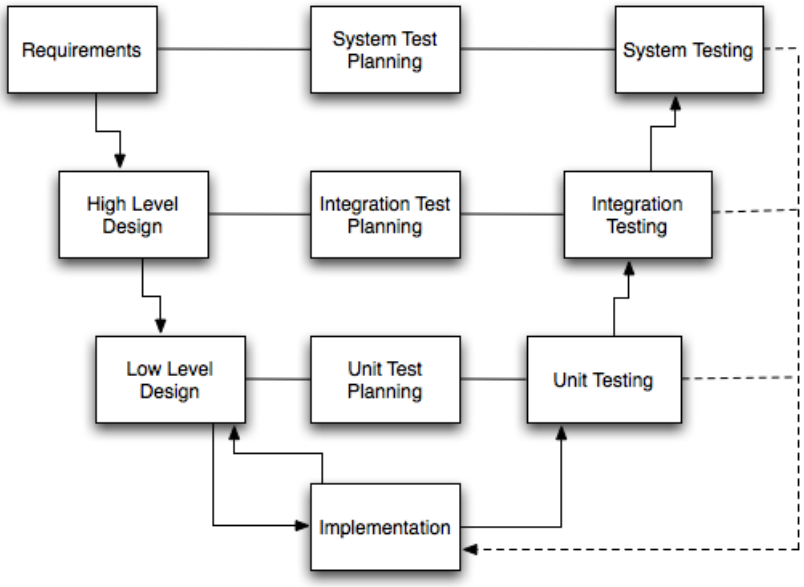

Figure 5.1: The modified V-Model life cycle.

are encountered. Then for example, if a problem was found during Integration Testing, we should backtrack to Implementation in order to construct a solution. Once this solution has been found, we can then progress back through Unit Testing, and continue validating our code through Integration Testing, safe in the knowledge that it is still valid from a Unit Testing point of view. This backtracking path is shown with a dotted line in the diagram.

5.1.3 System Architecture

The first task that must be completed in the design of the framework is to establish a high level architectural design. By doing so, we can identify the classes within the system, and how they relate to one another, thus providing suitable groundwork for developing the high level design of the classes themselves.

In order to provide sufficient background information of the code we are basing our framework upon, we shall first give a high level description of PyGA. PyGA is based around two tightly coupled yet highly cohesive classes called Individual and Environment. Predictably, Individual encapsulates all of the functionality and data of an individual, such as its chromosome (and hence genotype), functions for genotype manipulation, score evaluation, etc. The Envi-

ronment class maintains a population of Individuals, and is responsible for all generation-wide level operations, such as parent selection, determining if genetic operators should be applied to individuals, and deciding whether the goal has been reached or not.

As can be seen in Figure 5.2, these two classes form the basis for our system - any extensions that need to be made to these classes in order to add GE functionality (e.g. genotype to phenotype mapping with a grammar) should be implemented in classes inherited from them, namely GEIndividual and GEEnvironment.

From examining the requirements and the how GE works, it was determined that another further class was required in the system, namely Grammar. Rather than making the parsing and processing of a grammar definition the responsibility of GEIndividual or GEEnvironment, this should be treated as a separate entity in its own right which is tightly coupled to GEIndividual, providing it with access to the grammar used for a given problem. This decision was made on the basis that the complex action of parsing the grammar, which although arguably could be performed by GEEnvironment, would add excessive code to its class and reduce the simplicity and clarity of code that comes from the PyGA implementation. Note that GEEnvironment is still dependent on Grammar, due to implementation issues that we shall explain later.

The final class that is visible in the diagram is ConfigHandler. Whilst it was felt that the handling of configuration files to potentially be a piece of functionality of GEEnvironment, as with the Grammar class, in order to prevent code obfuscation within GEEnvironment, it was elected to make this a class in its own right. Also, by handling configuration in a separate class, it allows for easy extension and customisation of the configuration system separate to the GE itself. Note that neither GEEnvironment or GEIndividual actually depend upon ConfigHandler, allowing a researcher to treat the use of a configuration file as optional.

To recap: the system architecture consists of four main classes, with two of the classes having ancestry in PyGA's Individual and Environment classes. The system architecture is diagrammatically represented using UML in Figure 5.2. Whilst tight coupling is often frowned upon in OO design (due to the "ripple effect" of propagation of errors and bugs), we have elected to allow it in this case on the basis that it is a relatively simple system, and by separating the functionality needed to parse the configuration and grammar files, we maintain the cohesion that exists in GEEnvironment and GEIndividual. By doing this, we have remained aligned with the requirement that the development of the system should be geared towards simple, readable code.

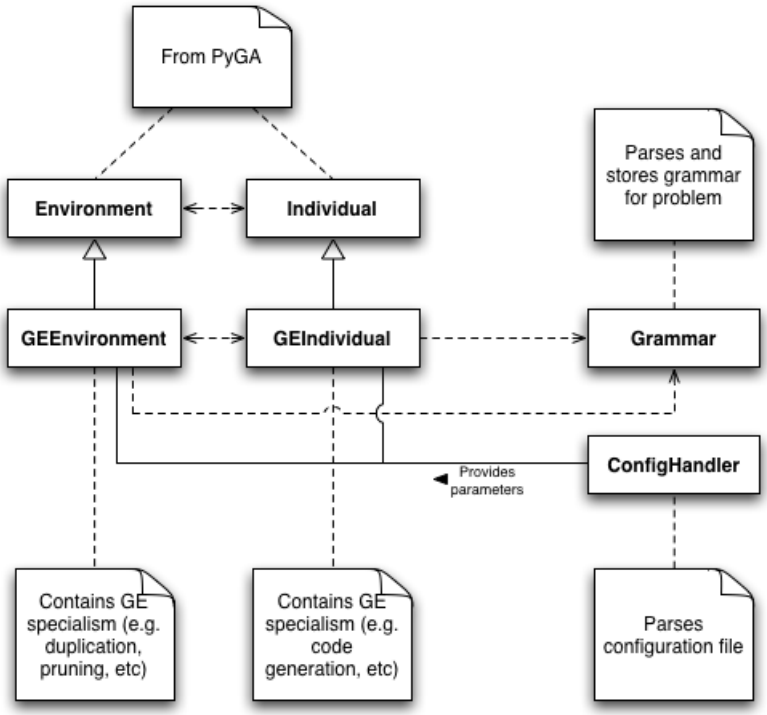

Figure 5.2: UML diagram of system architecture.

5.1.4 Class Design

We shall now provide an overview of the high level design decisions for all of the classes used in the framework. providing background information on the PyGA classes that they are derived from if necessary.

GEEnvironment

As GEEnvironment is based heavily upon the Environment class from PyGA, we shall first discuss the high level design of that class before moving on to highlight the changes that need to be made to make it suitable for use in a GE framework.

As stated previously, Environment is responsible for maintaining a population of Individual objects that are created when `__init__()` (Python's approximation of a constructor) is called. The GA is actually started by calling a function of the Environment class that executes a loop, which iterates until either the goal condition has been met or the maximum allowable number of generations (i.e. iterations of this loop) have been executed. From this loop, a function is called that in turn contains the function calls used in the initialisation of a new generation (pertaining to selection, crossover and mutation). The final task of the loop is to report the current status of the GA.

The extensions that need to be made to Environment in GEEnvironment are not too drastic in their measure. The fundamental structure and mode of operation of the class remains the same - a population is created by `__init__()`, and there is a loop that "runs" the GA.

Firstly, Environment is lacking functions to decide whether or not duplication and/or pruning should be applied to an individual, which can be simply added to GEEnvironment. Secondly, the Environment class is also lacking the steady state population update method that was identified as being required in section 4.3.2. This can simply be added as a new function in GEEnvironment - we choose to do this rather than overriding the existing update method, as it allows us to effectively gain the functionality of elitist population update for "free" (albeit with modification to introduce the use of duplication and pruning).

The final key changes to be made in GEEnvironment relate to setting up variables that will be used by GEIndividuals. Firstly, it will instantiate an object of type Grammar to pass to each GEIndividual to govern the genotype to phenotype mapping process. Secondly, it shall read in

and set up any variables that will be used in the execution of generated code.

GEIndividual

As with GEEnvinroment, we shall begin with a discussion of the high level design of the Individual from PyGA, then move on to describe the required changes that will needed to be made in the inherited class, GEIndividual.

Typically instantiated by an object of type Environment, the Individual class is primarily concerned with the manipulation and use of its chromosome instance variable. Almost all the functions in the Individual class use this in some way - from a function to perform the mutation operation on it, to producing offspring from it via crossover. The chromosome is a randomly generated binary string of a predefined length. Whilst PyGA provides some basic functions, such as a two point crossover function and a mutation function, the evaluation function is deliberately left as an empty function - the task of implementing this is left up to the researcher using PyGA to implement it to suit their needs.

In order to make GEIndividual suitable for use in a GE framework, several critical functions have to be added, pertaining to the genotype to phenotype mapping and the evaluation of the resultant code. When the constructor-like function `__init__()` is called, the calling GEEnvironment shall pass it a reference of a Grammar object (this results in the dependency highlighted in section 5.1.3) that will be used in the genotype to phenotype mapping process, and also a variable containing any variables to be used in the GEIndividual's generated code. The evaluation function that has been overridden from the Individual class simply attempts to generate a code string from the GEIndividual's chromosome and the contents of the passed Grammar. The GEIndividual then actually evaluates this code, assigning the GEIndividual a suitable score. In addition to the code generation and evaluation extensions, GEIndividual had to include functions to apply the GE duplication and pruning operators to the genotype. In line with the requirements regarding crossover in GE, a one point crossover function will be added to GEIndividual, although this shall not override the implementation of two point crossover that is already present.

Grammar

The Grammar class is to be very simple in its operation from a high level - all it should do is read and parse the contents of a file containing a grammar in BNF format (in line with the section 4.3.2). The class should make the start symbol of the grammar available in an instance variable, and provide some mechanism for returning the expansion of a given non-terminal.

ConfigHandler

As with the Grammar class, the high level design of ConfigHandler is not complex - all it shall do is parse a configuration file containing various parameters for a GE run, and make the values of these available through instance variables in the actual setting up of the GE framework.

5.2 Parallel Implementation

The parallel implementation will essentially just subclass GEIndividual and GEEnvironment, adding and modifying functionality in order to create an asynchronous coarse-grained, multiple-population version of GE. The bulk of the changes required will reside in the subclass of GEEnvironment; in fact, GEIndividual need only to be sub-classed in order to specify selection and crossover method, etc.

A Python program that is separate to the subclasses of GEIndividual and GEEnvironment will be required to act as a controller or master node. It will be responsible for control of the parallel system - if a population reaches the optimum, then it will inform the master node of this before closing down. The master node will then inform all other nodes in the parallel system that they should halt execution and close down.

The two key functions that will be added to the subclass will allow for the emigration and immigration of individuals to and from other populations. In order to provide this migration, each population will need to be aware of its neighbours, and store this in some way. Alongside this, each population should have a specified *migration interval* - how often (in terms of generations) should it emigrate individuals to its neighbours. Once a population finds the optimal solution to the GE problem, it should inform the master node in order to halt execution in the parallel system.

Chapter 6

Low-Level Design and Implementation

In this chapter, we shall expand upon the high level design, detailing the problems encountered and the decisions made during the low-level design and implementation of the project. Given the closely interrelated nature of low-level design and implementation in the project life-cycle we have adopted (as described in section 5.1.2), it was felt that presenting the low-level design and implementation details together would more clearly document the decision making process throughout that stage rather than presenting them separately.

6.1 Grammatical Evolution Framework

6.1.1 General Concerns

Development Environment

Whilst it was planned at the project proposal stage to develop the framework using the Eclipse IDE and the PyDev plug-in, it soon became apparent that this was not an ideal solution for the project given the hardware constraints (the development platform was a 1.33GHz G4 PowerPC processor with 768MB of RAM, which was not sufficient for Eclipse to be usable). Therefore, the decision was made to use the vim editor, which offers full syntax highlighting for Python. Pychecker, a lint-like debugger for Python was also integrated into the vim environment, to allow for simple testing for common errors as the development work progressed.

Variable Storage Problem

Before we describe the low level design and implementation of GEEnvironment and GEIndividual, we should explain a problem that cross-cut both classes, and subsequently influenced the design of both. According to the high level design of the GEIndividual class (see section 5.1.4), each instance should evaluate the code generated from its chromosome. This code may contain zero or more variables, so not only may there be the need to store multiple variables, a range of values for each may be required.

This posed the problem of how the values for these variables should be stored and loaded into the GE system.

A variety of solutions were considered. For example, a simple comma separated values (CSV) file, containing pairs of variable names and values. Whilst simple for a researcher to use for trivial problems, it would not be practical to allow support for Python specific data structures, such as lists, etc. Furthermore, by using CSV, we would have to specially create a parser for the variables file, which adds unnecessary work to the low level design and implementation stage. The solution that was eventually chosen takes inspiration from how Python itself provides access to variables - the concept of namespaces. A namespace is essentially a Python dictionary where the keys are names of variables in the current scope, and the dictionary values are the values of those variables. For example, each function has its own namespace, called the local namespace, which keeps track of the functions variables, including function arguments and locally defined variables [Pilgrim, 2004]. Quite simply, the variables for a problem are specified in Python as a list of dictionaries, where the keys are the variable names used in the evaluation of each individual's code. This is stored in a plain text file external to the system, and will be read in during a GE run. For example, here is an extract from a variables file for a Symbolic Regression problem:

```
1 [
2 { 'X' : 0.0, 'ans' : 0.0},
3 { 'X' : 0.314159265, 'ans' : 0.72187230281887804},
4 { 'X' : 0.628318531, 'ans' : 1.6108879599186472},
5 { 'X' : 0.942477796, 'ans' : 2.6397591862827423},
6 ...
7 ]
```

Note that `X` is an input value that will be used in code string execution, and `ans` is the value that the generated code's result will be compared to in that specific case, and hence evaluate the fitness of each individual.

Whilst this is quite an elegant solution, it is not without its problems. It can be ungainly for a researcher to create and edit such a file, as opposed to say, a comma separated value file. However, this solution was far simpler to implement than other solutions (including the process of opening and reading the file, the variables file is 'parsed' into a usable state in three lines of code). Perhaps the key problem with this solution is that it can be somewhat insecure - there is nothing to prevent a researcher from giving a variable the same name as another used in GE, thus potentially introducing unexpected results. This problem can be avoided, however, as long as the researcher is aware of this potential pitfall and they use sensible variable names (they could also simply check potential variable names against the GEPy source using a standard tool such as `grep`). Another possible solution to this problem would be to implement a mechanism for identifying the variables within the BNF grammar, and then have GEPy automatically obfuscate the variable name in order to prevent any possible clashes with variables used within GEPy itself (somewhat akin to how Python simulates private variables).

Harsh Fitnesses

A harsh fitness value is one which will ensure that an individual that is considered to be invalid has a minimal chance of its genetic material passing into the next generation. A harsh fitness is essentially one of two values, depending on the nature of the problem being investigated with a GA:

1. For a maximisation problem, the harsh value should be zero
2. For a minimisation problem, the harsh value should be a large, positive number

Initially, it was assumed that using simply a very large number (e.g. 99999999999999) for minimisation problems would be sufficient. However, no matter how large you make the number, there is still a technically conceivable situation in which an invalid individual could stand more chance of progressing than valid individuals in the population[1]. Ideally, we would like to use infinity as our minimisation punishment value, as this would prevent invalid individuals

[1]Note that long integers in Python have unlimited precision, and as such there is no "maximum" integer that can be represented [van Rossum and Drake, 2006a]

being fitter than valid individuals. Whilst Python does not prevent native support for infinite values, a commonly used module called `fpconst` that provides IEEE 754 double-precision special values (positive infinity, negative infinity and not-a-number (NaN)) is available for use. Whilst we wish to minimise the burden of installation (as per the requirements detailed in section 4.3.2, it is written in pure Python, and as such, has no special installation requirements and can simply be packaged with the GEPy framework distribution.

The harsh fitness value shall be stored in a global variable, with the appropriate value set in GEEnvironment's constructor depending upon the problem type. Whilst using global variables is not a generally desired design choice with an OOP paradigm, it was decided that it would be preferable to follow PyGA's use of global variables for constant values that cross cut both classes in our system, as it was noted during the requirements analysis of PyGA that this improved the readability of the code.

6.1.2 Grammar

From the requirements analysis, it was determined that the grammar definition should be in the standard BNF format. Whilst during the requirements gathering process other options such as using an XML definition of the grammar, which would ease the implementation were considered, it was felt that it was preferable to maintain consistency with the original GE system.

As there is not a Python module for the parsing of BNF grammars, it was necessary to develop a bespoke parser to suit our needs. Three possible methods of implementation were identified:

1. Using the SPARK compiler/parser Python framework
2. Using the pyparsing, a simple recursive descent parser module
3. Writing our own parser from scratch using regular expressions, etc.

It became immediately apparent that it would not be a valuable use of time to develop our own parser from scratch when there are modules available that have already implemented the bulk of the work involved. Hence, we were lead to decide between SPARK or pyparsing. Whilst SPARK provides a feature-rich environment for developing a full compiler in Python, it is complex to implement something simple as our BNF parser when compared to pyparsing. Furthermore, pyparsing integrates the language definition with the parser code - it takes a

"building-block" approach, in which you assemble the grammar using expression objects such as Literal, Word, OneOrMore. There is no separate lex/yacc syntax, leading to the provision of a neat, self contained solution. Hence, we elected to implement our Grammar class using the pyparsing module. We should note that as with the fpconst module, pyparsing is written in pure Python, and as such, can be simply packaged with the GEPy code, rather than the researcher having to specially install it.

Before we describe the Grammar class, we shall first briefly explain the format of our BNF grammar. In line with the standard BNF format, non-terminals are differentiated from terminals by angled brackets, i.e. $< nonterminal >$. Our implementation required one slight modification of the BNF syntax, by including the definition of the start symbol S within the grammar. For example, if the start symbol was $< expr >$, this would be defined in our grammar as:

$$S ::=< expr >$$

In our high level design it was determined that the Grammar should provide the ability to return the expansions for a given non-terminal in the grammar. Hence we had to decide upon a suitable data structure to store this information. The most natural solution was to store the expansions for a given non-terminal in a list and then provide access to this list in a hash table with the non-terminal's name as the key (this is a variable of type `dict` in Python). The hash table provides good performance (constant-time $O(1)$ lookup on average), and the fact that this is a built in data structure type in Python removes some of the burden of implementation through other methods.

Now that we have a target data structure defined, we shall describe the code used in the parser.

We define our grammar using pyparsing constructs such as OneOrMore, Literal, etc. For example, we define terminals and non-terminals as follows:

```
1 self.terminal = Word(alphas,alphanums+"-"+".") | oneOf("( ) + - * /")
2 self.nonterminal = Combine(Literal("<") + self.terminal + Literal(">"))
```

Note that in our implementation we support a limited range of alphanumeric characters, but it is clearly easy to add more should a researcher find the need. These definitions are then used to define rules. For example, we define the rules for handling the start symbol :

```
1 self.startSymbolRule = Literal("S").suppress()\
2          + Literal("::=").suppress() + self.nonterminal
3 self.startSymbolRule.setParseAction(self.__handleStartSymbol)
```

The use of the pyparsing objects builds up a structure of pattern matching that result in a token string being generated (note the use of `suppress()`, which states that the associated object should be matched but not included in the token string). For example, the above code would convert $S ::= < expr >$ into the token string `<expr>`. In line 3 of the above code listing, we see that we call the function `setParseAction()` of the start symbol rule. This associates an instance function that is called whenever the particular rule is matched in text, and we shall see the purpose for this shortly.

The parser essentially consists of a loop inside a `try...except` block, which catches a ParseException (this would be thrown by a syntax error in the BNF grammar file). The loop iterates over the contents of the grammar file, calling the `grammar_rule.parseString()` function on each line, returning a ParseResults object, stored in `_current_terms`. This call will apply the parse rules which will result in either a startSymbolRule or productionRule being matched. As stated above, parse actions are associated with these rules - helper functions that set variables external to the grammar definition. If a start symbol is matched, then a helper function is called, setting a local variable representing the current non-terminal to `S`, and the variable `_start_symbol` is assigned with the actual start symbol. For the production rules, another helper function is called, simply resulting in the local variable storing the current non-terminal to be assigned whatever is on the left hand side of the ::=. Once the call to `parseString()` is complete, the value of the current non-terminal is checked - if it is `S`, then nothing has to be done (as we have already assigned the `_start_symbol` variable the appropriate value), otherwise, the contents of `_current_terms` will be the expansions for the current non-terminal. The contents of `_current_terms` are converted to a list, and then stored in the `_term_dict` dictionary, using the current non-terminal name as a key[2].

We should note the above use of Python's pseudo-private variables, which prevent access to these variables external to the class. This is in order to ensure that the variables cannot be accidentally altered by GE. All read access to the start symbol and term dictionary are provided through the accessor functions `get_startsymbol()` and `get_expansions()`, respec-

[2]Through discussion of the BNF parser code used in this project with the author of pyparsing Paul McGuire, it was identified that the Grammar class could be re-entrant (or thread-safe), as long as any threads do not share Grammar objects.

tively.

It is important to note that the use of the `pyparsing` does not break our adherence to Python code style standards, despite what may appear to be contrary. For example, the `OneOrMore` rule may appear to be a function in its application, but it is actually an object and hence is named accordingly.

From the experience of using pyparsing to implement the BNF grammar parser, it became clear that this is an incredibly easy to use and straightforward tool to use, that gives results very quickly. The nature of implementing the parser and the grammar purely within the same Python file (as opposed to using a separate lexx/yacc syntax) lead to very readable code. This resulted in enabling a part of the framework that had great potential for messy and unclear code to be very readable and understandable by a competent Python programmer.

6.1.3 GEEnvironment

As stated in the high level design chapter (see section 5.1.4), on the whole the changes that need to be made in GEEnvironment to make it suitable for use in GE do not consist of large alterations to the core functionality of the class (with the exception of the constructor-like function); but it is primarily the addition of missing features that are desired for our GE implementation and slight modifications to existing functions to support the passing of new variables, for example. This validates our choice of using PyGA as the basis for our system, as it has helped to reduce the work involved in implementing a GE framework.

The biggest change that needed to be made to an existing function in Environment was to the constructor-like function `__init__()`. In PyGA, this function simply sets up the parameters for the environment, creates a population of individuals if necessary (note that a population can be passed as a parameter to `__init__()`), and does an initial evaluation and report of the population.

We have had to modify `__init__()` to take several GE specific parameters (such as the path to the BNF grammar file, the path to the variables file, duplication rate, etc.). The path for the BNF grammar file and variables file are used to instantiate the associated Grammar object and then read in the contents of the variables file. The file contents are then used with the `eval()` function and with the result stored in an instance variable to be used by GEIndividual objects. The `__init__()` function also sets the global variable used to store the harsh

fitness value to the appropriate value based upon the type of optimisation being used[3]. Finally, `__init__()` will call the constructor of the parent Environment class through a call to the function `super()`.

The `_makepopulation()` function had to be overridden, in order to pass references to the instance variables containing the Grammar object and to the variables to be used in code evaluation to the individuals in the population when they were created. We also chose to add a check to `_makepopulation()` in order to ensure that the type of the class of individual being used is a subclass of GEIndividual. If it is not, then the program will exit gracefully with an error message. Whilst this is not specifically needed for GE, it was felt to be a useful improvement on the PyGA code in terms of robustness.

The only other functions modified from the original PyGA Environment code were `_select()` and `_crossover()`[4]. The `_select()` function was altered to change the default selection method to be roulette wheel, whilst the original `_crossover()` was renamed to `_crossover_elitist()` and support was added for GE's genetic operators (duplication and pruning). The actual `_crossover()` function was overridden to allow the researcher to easily specify the crossover method to be used, defaulting to steady state population update, to be in line with the system requirements.

We shall now move on to discuss the additional functions added to the GEEnvironment class. Firstly, we shall examine the roulette wheel selection function, which is based upon the basic algorithm given in [Goldberg, 1989], and described in the literature review (section 2.1.4). We shall not recap the implementation here as the description in the literature review is sufficient, however we shall mention an alteration that was made in our implementation. Goldberg's method only supports maximisation problems, and given the minimisation nature of many GE test problems (such as Symbolic Regression), it was decided that it was necessary to adapt it to support this. The alterations required were simple:

1. First, provide a check of the type of problem through the `optimization` variable.

[3]Note that this is defined as the class variable `optimization` in the class the researcher will have implemented, inheriting from GEIndividual. This class is passed to the environment class constructor as the variable `kind`, and is also used in the `_makepopulation()` function to create all the individuals in our population of the appropriate type.

[4]The function name "`_crossover()`" is somewhat a misnomer. It should perhaps be called something like `_create_new_generation()` to reflect its actual purpose - the creation of a new population of individuals with a specific population update method, and the application of genetic operators. However, it was decided to retain PyGA's naming convention for consistency, to avoid the workload of re-factoring and checking the code for bugs.

2. If it is a minimisation problem:

   (a) Rather than simply calculate the total score as the sum of the scores of the individuals, calculate the total score as the sum of the inverses of the score of the individuals.

   (b) Similarly, calculate the partial sum as the sum of the inverse score of the individuals.

In order to keep the code clearer, we provide helper functions to calculate the total and inverse total score of the population (`_get_total_score()` and `_get_inverse_total_score()`, respectively).

The steady state population update version of `_crossover` was a simple adaptation of the existing elitist implementation; in the elitist version, the new population is initialised with a copy of the best individual in the population, as follows:

```
1 next_population = [self.best.copy()]
```

Note that `best` is not a variable but a *property* (or accessor function) of GEEnvironment. Now, steady state replacement retains a certain number of the best individuals in a population, which is done by this simple alteration:

```
1 next_population = self.population[0:(len(self.population)-1)\
2                                   - num_replaced]
```

Note that the population will be sorted in fitness order before this is performed. The rest of the function remains the same as the elitist version - a while loop that iterates until the new population reaches the size of the population, applying crossover and the genetic operators as necessary. Once the new population is complete, the current population is replaced with it.

The functions that were added to determine whether the prune or duplication operators should be applied are based on the mutation determination operator from PyGA - a random number generator is used, and the value is compared with a threshold value (the parameters `prune_rate` and `duplication_rate`, respectively).

We should note at this point that the `_prune()` function checks to see if an individual's

gene_offset is equal to 0 before calling the individual's prune() function. If it is equal to 0, we call the function's evaluate() function in order to make sure that the individual *has* been evaluated. Next, the individual's is_wrapped variable is checked to ensure we do not call prune on an individual who has been wrapped. This alleviates two potential problems with using _prune() - if we were to prune an individual's chromosome at position 0, it would destroy the chromosome, and hence the individual. Similarly, if an individual has had to wrap its chromosome to generate its code string, then pruning at any point of the chromosome will result in the destruction of genetic information actually used in the code generation, and hence, the individual will be damaged.

It is worth noting that one further alteration was made to the actual PyGA Environment code - its use of the Python range type was replaced with xrange. The xrange type is an immutable sequence which is commonly used for looping. The advantage of the xrange type is that an xrange object will always take the same amount of memory, no matter the size of the range it represents [van Rossum and Drake, 2006d].

6.1.4 GEIndividual

As stated in the high level design of GEIndividual (section 5.1.4), several changes and additions to the Individual class will have to be made in order to support GE, with the primary focus being around the genotype to phenotype mapping and the execution of the resultant code. We shall now detail the changes that were necessary.

As with GEEnvironment, GEIndividual's __init__() function takes various parameters needed to configure an individual for GE alongside the optional parameter chromosome from PyGA's Individual class. Required parameters are a reference to the Grammar object used in phenotype to genotype mapping, alongside a reference to the variable containing the evaluated contents of the variables file, used in running the individual's generated code. There is also an optional gene offset parameter, commonly used when an individual is copied. Once all the parameters have been assigned to the appropriate instance variables of the individual, the __init__() function of PyGA's Individual is called via a call to super(), passing chromosome as a parameter.

We shall now examine the most critical addition that was made - the genotype to phenotype mapping, and the evaluation of the subsequently generated code. As mentioned in the high level design (section 5.1.4, the calls to initiate the mapping process and the code evaluation,

namely `init_mapper()` and `run_code()`, will be made from the overridden `evaluation()` function.

The genotype to phenotype mapping is a recursive process, started by the `init_mapper()` function. This function obtains the start symbol from the Grammar object, and then proceeds to call the actual recursive function `handle_expansion()`, passing the start symbol and 0 as arguments. The 0 signifies the current recursion depth, which is used in the heuristic to prevent infinitely looping through the BNF grammar when mapping an individual. [5]

Rather than provide a full textual description of the function, we shall present it diagrammatically in Figure 6.1. However, we shall cover a couple of points that need expansion. The check for whether an expansion is a non-terminal or not is simply done by checking whether the string starts and ends with the < > characters, and the expansions for non-terminals are obtained from the Grammar object. Codon values are obtained through the `get_codon()` function, whose behaviour alters depending on whether or not wrapping is enabled. If wrapping *is* enabled, then the function will simply wrap the genome as expected, and return the appropriate codon value. Otherwise, if the end of the genome is reached, then the function returns the value `None`, and `handle_expansion()` will assign the individual a suitably harsh fitness value. Note that `get_codon()` updates the `gene_offset` variable, which will be used next time the function is called, and is also used by prune to remove introns. As there is not a built-in function in Python to convert binary strings to decimal integers, the function `to_decimal()` is used, which is a piece of public domain code. Aside from these points though, the function is a fairly simple implementation, which just recurses through the grammar until it reaches an end condition, be it an error or a fully generated string.

Once the call to `init_mapper()` is complete, `evaluate()` then calls the function `run_code()`. This function is responsible for the running and evaluation of the individual's code, and hence, assigning it a fitness value. This function will typically be overridden by a researcher to fit their needs more closely, as not all code will be set up and executed in the same way. However, the

[5] The value of the limit we use is actually dynamic - Python has a limit on the maximum depth that you can recurse to, and as such, we base our heuristic upon that by offsetting the limit against the result of calling `sys.getrecursionlimit()`, which gives us the current recursion limit of the current Python configuration. It is worth noting that for hypothetical GE problems involving a grammar that features a high depth of recursion, that this limit could prove to be a problem - even if you increase the maximum recursion depth, you are still bound by the capabilities of the machine running GE. Whilst out of the scope of this project due to the constraint that the code should work with a standard Python interpreter, it is worth noting that there is a stackless variant of Python which makes use of continuations that could alleviate this potential shortcoming [Tismer, 2006].

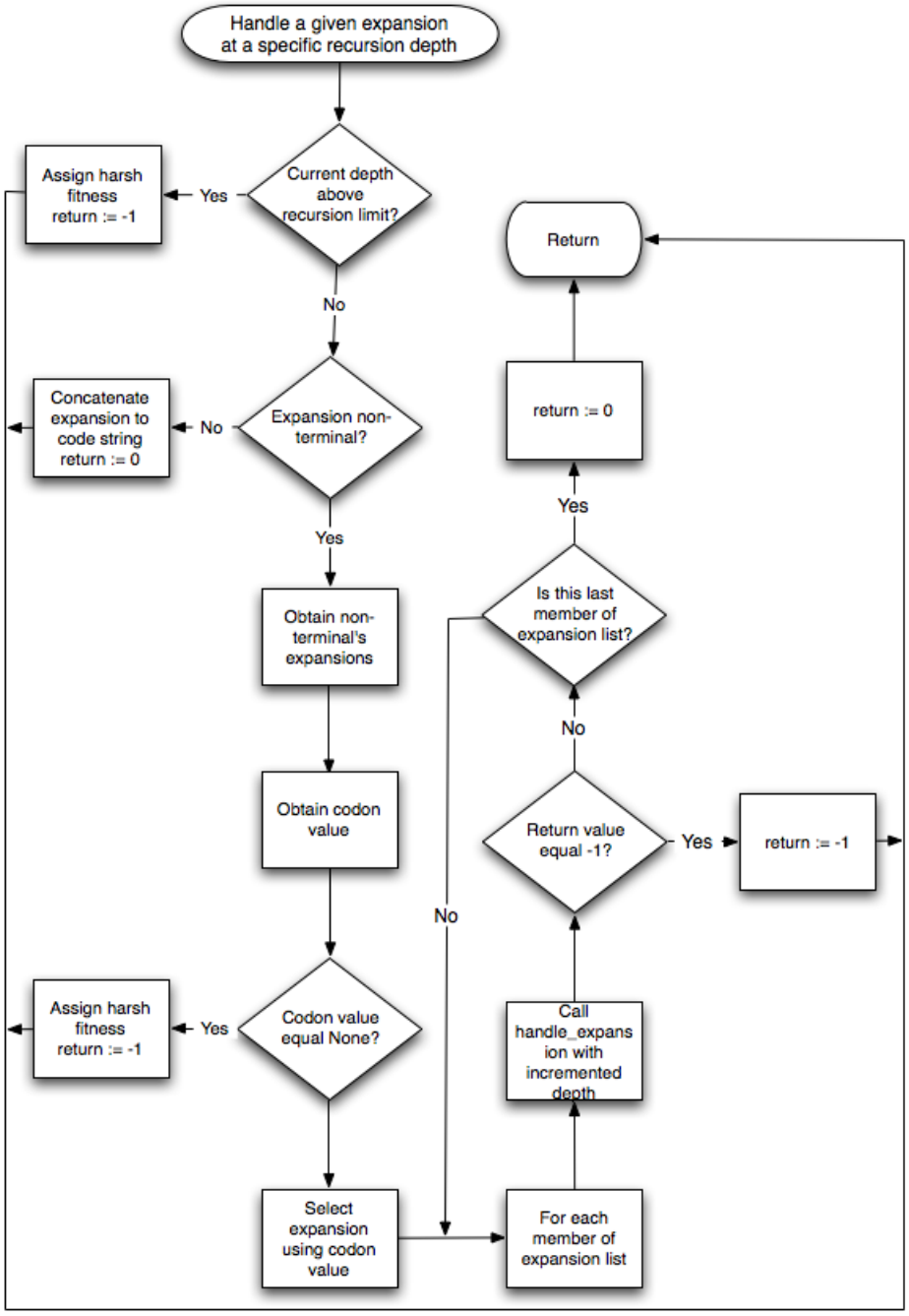

Figure 6.1: handle_expansion() function.

`run_code()` function supplied with the framework is suitable for fairly general purpose applications, and was designed with the standard GE tests of Symbolic Integration and Symbolic Regression in mind (as per the requirements, detailed in section 4.3.2), the fitness is determined as a measure of error between the code being run, compared to a value in the variables file called `ans`. It is clear that from a high level, this function must do three things:

1. Load any variables that are needed in order to evaluate the code
2. Run the code in a "sandbox" like environment, so that it won't stop execution of the GE should there be a problem with the code (e.g. a syntax error)
3. Assign a suitable fitness value to the individual based upon the quality of the code

The issue of providing a sandbox was very simple to implement - we simply wrapped the entire body of `run_code()` in a `try...except` block, which would catch any errors that could occur from either variable loading or code execution[6]. If an exception is raised, the individual is assigned a suitably harsh fitness, and no further attempts to evaluate its code are made as a result.

We shall now describe the code that runs within the `try...except` block. Firstly, specific to our case of fitness being based on error, we will store a running the cumulative total of the error for the different test cases in a local variable `total`. In order to process the code for each combination of variables stored in the variables file, the code is enclosed within a for loop which iterates over the input variables list, storing the current dictionary in the loop in the variable `current_var_dict`. For each of the elements in this list, we must load the contents of the dictionary into the local scope of the `run_code()` function. This is done by a for loop, and the built-in function `keys()` from the list type in Python as follows:

```
1 for current_var in current_var_dict.keys():
2         locals()[current_var] = current_var_dict[current_var]
```

A check is made to see that the individual being evaluated has not already had its score marked harshly so as to be considered invalid. If it has, then we break out of the loop, otherwise,

[6]We note that the use of a `try...except` block without testing for a specific error is often seen as bad programming practice due to the temptation for a developer to not write proper error handling code. However, it was essential in this case as there is no way of determining whether or not a piece of dynamically generated code will cause an exception due to the halting problem [Turing, 1936].

we continue. The individual's code contained is run using Python's `exec()` function, with the result of the execution being stored in the variable `val`. The result is checked using fpconst's `isNaN()`, to ensure that it code string has not produced a `NaN` value (e.g. $\frac{\infty}{\infty}$). If it is indeed a `NaN` value, then we set the individual's score to a suitably harsh value, and concatenate the following comment to the individual's code string to provide the researcher with useful information for debugging purposes: *`# INVALID INDIVIDUAL (NaN)`*. If `isNaN()` returns false, then the difference is calculated as the absolute value of the difference between the code's result `val` and the target value `ans`, and added to the `total` variable[7]. Once the outer for loop is complete, the running total is assigned to the individual's `score` variable, and code evaluation is complete.

We have now described the key modifications to existing code that were required for the GEIndividual class to be suitable for GE. Now, we shall examine the additions that were made in GEIndividual in order to support GE.

Our implementation of one point crossover called `_onepoint()` was based upon PyGA's two point crossover function, `_twopoint()`. The individual that is to be mated with this individual is passed to the `_onepoint()` function as a parameter. A pivot point is chosen as being between 1, and the last binary value in the shortest of the two individuals' chromosomes. An inner function called `mate()` is defined, which actually swaps the genetic information of the two individuals about the pivot point to create a child. This is actually called twice in the return statement of `_onepoint()`, with the individuals being altered in the order they are passed to `mate()`, so as to create two distinct children.

GE's two specialised genetic operators duplicate and prune, were simple to implement. The `duplicate()` function takes the argument `number_to_duplicate` to specify how many randomly selected genes should be copied to the end of the genome (which are selected paying particular attention to gene alignment within the chromosome), and appends them using standard Python list functions. The `prune()` function uses the `gene_offset` variable set during genotype to phenotype mapping in order to determine the location of introns. If `gene_offset` is equal to 0, pruning is not performed (as this will erase the entire genome), otherwise, Python's `del` list operator is called in order to delete all genetic information from the `gene_offset` onwards.

[7]Whilst originally implemented as being the mean square difference as this was felt to be superior, it was decided to use the absolute difference method in order to mirror GE's fitness evaluation in Symbolic Regression and Symbolic Integration

Other Changes

There were a number of changes made to the original PyGA code, for various reasons. The mutation function from PyGA had a bug, which meant that it had to be overridden. It was resulting in a value of `False` being assigned to a bit , rather than 0, which would result in the system exiting with an error. So, we simply created a function which changes a bit's value to the absolute value of itself minus one, i.e. `self.chromosome[bit] = abs(self.chromosome[bit]-1)`.

PyGA's `_twopoint()` and `copy()` functions had to be altered to ensure that the Grammar object and `_vars` variable were copied so that the new individual could evaluate its code. The `gene_offset` variable was also copied, to give an efficiency gain with pruning (it will potentially cause the pre-evaluation for pruning to be avoided).

In order to allow for a variable length genome, length was changed from being a variable to being a property. This was chosen in order to simplify the implementation - rather than having to manually update the length variable every time the chromosome was altered, thus increasing the risk of introducing bugs, length will be calculated "on the fly". Naturally, this does give rise to a slight negative impact upon performance, but the benefits to simplifying implementation (and also code readability) outweigh this. Finally, as with GEEnvironment, the uses of `range` within Individual were replaced in GEIndividual with `xrange` for performance reasons.

6.1.5 ConfigHandler

The primary concerns for designing the configuration system for the framework were that it should:

1. Provide an easy, readable way for researchers to be able to specify commonly altered parameters.
2. In line with the high level design, be able to be easily extended to suit any extra requirements the researcher may have.

Rather than creating a bespoke parser as with the Grammar reader (which was governed by the requirement that the Grammar should be read in a BNF format), it was decided that it would be more suitable to identify an existing Python module that could be used to parse a configuration file written in plain text, and extrapolate the values to be stored in instance variables. There were two primary candidates for this: the `ConfigParser` Python module for parsing

Microsoft Windows INI style configuration files, and the standard XML parsing module.

Whilst both modules offer essentially similar feature sets and offer the end user with a simple, clear configuration file format, it was decided to choose to use XML due to prior experience in developing with XML. This familiarity would mean that less time would have to be spent learning the API of `ConfigParser` and hence, allow for more time to be allocated to areas of the project of higher priority.

An example of a parameter in the XML format specifying the location of the BNF grammar file for a GE run is as follows:

$$< grammar > /labshare/gepy/symb\_int.bnf < /grammar >$$

Python's SAX (Simple API for XML) XML parser is driven by passing a user defined sub-class of the `ContentHandler` class. It is used to define functions that are called by the parser on the appropriate events in the input document [van Rossum and Drake, 2006c]. In our implementation, we had to define the following three functions from `ContentHandler`:

- `startElement()` - which is called when an opening XML tag (e.g. $< grammar >$ is found)
- `endElement()` - which is called when a closing XML tag (e.g. $< /grammar >$ is found)
- `characters()` - which reads CDATA from inside a pair of XML tags (e.g. $/labshare/gepy/symb\_int.bnf$).

We define `startElement()` to assign the value 1 to a variable indicating that a particular tag has been met. This variable is then used by the `characters()` function to assign the CDATA to the correct variable. In the above example, `startElement()` would set the variable `is_grammar` to 1 when the opening tag was found. Next, the function `characters()` would be called, due to CDATA being detected. Within our definition of characters, we have a set of if statements evaluating all of the `is_` variables that have possibly been set by `startElement()`. In our case, `is_grammar` would match, and hence, `characters()` would start copying the CDATA to the appropriate instance variable[8] (`grammar` in this case). Once the parser finds the closing $< /grammar >$ tag, the `is_grammar` variable is set to 0,

[8]Whilst we arguably could have stored parameter values in a dictionary using the parameter name as a key, it was felt that from a basis of trying to maximise the clarity of the code, that the use of a

so that `characters()` stop storing CDATA in `grammar`.

In addition to defining the functions from `ContentHandler`, we define another function of our own called `validate_vars()`, which validates the entries in the configuration file. This function has to be explicitly called. It runs a series of tests on all the variables storing parameters that have been parsed to prevent erroneous parameter entry, and also casts them to the correct types for use within GEPy if necessary:

Note that as Python does not provide functions for checking if a variable is a valid floating point number or if it is a valid path to a file, we had to define our own helper functions. These exploited the `try...except` statement in Python as follows:

```
1 def _isfloat(self,string):
2         try:
3                 float(string); return True
4         except ValueError:
5                 return False
```

The function tries to cast the string to a float - if it cannot, then a `ValueError` exception will raised, and the function will return False. Otherwise, the function returns True. It was felt appropriate to use a `try...except` block in this case as it is a very precise and controlled use of the statement, negating the potential for previously stated problems associated with an unrestricted `try...except` block. Whilst there will be a slight performance hit incurred from casting a variable to a float, as opposed to writing some code to manually verify the format of the variable (e.g., by using regular expressions), it is felt that the fact that as this is not a piece of code that will be used often, this is not a relevant problem. Furthermore, the clarity of code that this provides is preferable to using regular expressions to verify the format of the variable.

Thanks to the simplicity of implementing a SAX parser, and the fact that GEPY is not dependent on the ConfigHandler, a researcher can easily extend or modify the configuration system

dictionary would have just resulted in having messier parameter access (e.g. `config.variable` vs. `config.config_dict['variable']`) for the developer. The only advantage that this approach could provide would be that if we were not concerned with validating the contents of the XML configuration file (and casting the CDATA to the appropriate types as is shown shortly), we could potentially allow the XML parser to store the contents of any tag into the table, thus allowing the format of the XML file be effectively self-defining. As the structure of our configuration file is fixed, this advantage does not concern us.

through alteration of the functions that we have defined, allowing for the addition and/or removal of parameters from the configuration file. Possible improvements they may feel inclined to implement could be for the options in the configuration file to be more detailed (i.e. allow for the alteration of more settings), or perhaps enhance the syntax of the configuration file itself (e.g. allow for the grouping of parameters).

6.1.6 Framework Distribution

GEPy is organised into the Python package structure as shown below:

```
- gepy/
          - PyGA
          - config_handler/
          - grammar/
```

This was done so as to provide a researcher with greater modularity by being able to access the specific parts of GEPy through Python's dotted notation.

For example, given use of the ConfigHandler class is optional in GEPy, a researcher could choose to load it with the statement:

```
1 from gepy.config_handler import ConfigHandler
```

A researcher will typically implement a GE program by writing a "driver" program. This will include the researcher defined subclasses of GEIndividual and GEEnvironment to include the options and specialism they require, and optionally a call to a function provided by GEPy called `init_GE()`. This function simplifies the initialisation of a GE run, by simply requiring the researcher defined subclasses and the location of the XML configuration file to be passed as parameters.

6.2 Parallel Implementation

6.2.1 Development Environment

The key decision that had to be made for the parallel GE implementation was which communications library to use. The two most viable, and widely used candidates were PVM and MPI. Both of these libraries each have their own advantages and disadvantages, which are highlighted in [Geist et al., 1996]. Whilst the advanced features of MPI are not needed for the implementation, the homogeneity and small size of the Department cluster lead to MPI being the ideal choice - PVM's advantages of portability and fault tolerance were minimised as a result, whilst the higher performance of MPI is clearly desirable.

Once the decision to use MPI was made, we had to decide upon how we should use MPI with Python. There are currently two available options - pyMPI and MYMPI. The main difference between pyMPI and MYMPI is that pyMPI is actually a special version of the Python interpreter along with a module. MYMPI on the other hand, is a module only and is used with a normal Python interpreter. Whilst MYMPI only supports around 30 of the 120 MPI calls, this subset is sufficient to develop our parallel implementation of GE [Kaiser, 2006]. Given the requirement for the *framework* that GEPy should work with a standard Python interpreter, it was felt that this should be carried through into the design of the parallel implementation of GE, and as such, MYMPI was selected. It should be noted that unfortunately MYMPI requires the installation of the Numeric module, which has to be specially installed, although this was felt that to be a relatively insignificant issue.

In terms of the actual tools used to write the code, we used the same environment as for the GE framework implementation - vim, with integrated pychecker. As the University's cluster would not be available for the implementation period, a temporary solution had to be found so as to allow for testing of the code as the implementation progressed. A simple 2 node cluster was created, consisting of a Apple PowerBook G4 (1.33GHz G4 PowerPC processor with 768MB of RAM), and a Dell Inspiron 2650 (2.0GHz Intel Pentium M processor with 256MB of RAM). Whilst clearly a cluster of such heterogeneity would be practically useless for providing meaningful experimental results, it was an indispensable tool for development. The cluster will run LAM/MPI, which has been selected over MPICH for its ability to handle heterogeneous clusters.

6.2.2 Parallel GE

It has been shown [Cohoon et al., 1987], that the choice of topology to be used in a parallel GA s not very important in the performance of the parallel GA, as long as it has "high connectivity and a small diameter to ensure adequate mixing as time progresses". One such topology is that of the toroidal mesh. This topology also has the advantage that it is relatively simple to implement, helping to minimise the time needed for implementation.

Hence we shall use a 2-D toroidal mesh in our parallel implementation. The mesh will be organised in an NxN structure, and controlled by an external master node which will be responsible for managing the mesh - logging results if necessary, and shutting down nodes when the need arises. It was decided that due to time constraints, rather than alter the configuration system by extending and modifying the ConfigHandler class, all variables that control the parallel aspect of the implementation should be defined as being class variables. Ideally, if the parallel implementation were to be fully packaged with GEPy then we would seek to extend the format of the configuration file in order to allow the simplify the configuration of the parallel aspect of the framework. However, as we are treating this as an instance of the framework rather than a part of it, and that we shall be the only end users, it is felt this extra clarity is not needed.

Protocol

Before we discuss the implementation of the nodes and the master, we shall first describe the protocol that is used for our implementation. As this is a simple implementation, we shall only require the use of the global communicator, MPI_COMM_WORLD. All messages that will be sent will be of type integer, to the appropriate destination. The different MPI messages we use are identified via different tag numbers, which are identified in table 6.1. An example MYMPI send call is shown below:

```
1 mpi.mpi_send(migrant_chromosome,len(migrant_chromosome),\
2          mpi.MPI_INT,i,MIGRANT,mpi.MPI_COMM_WORLD)
```

A node will check for an incoming message by means of the `mpi_iprobe()` function, which executes an asynchronous probe for a given message type. Messages will be actually received by calling the `mpi_recv()` function. For example:

```
1 count = mpi.mpi_get_count(mpi.MPI_INT)
2 immigrant = mpi.mpi_recv(count,mpi.MPI_INT,mpi.MPI_ANY_SOURCE,\
3         MIGRANT,mpi.MPI_COMM_WORLD)
```

Note the use of `mpi_get_count()` to get the message size (the MIGRANT message contains the binary string chromosome of an individual, represented as an Numeric integer array).

The MIGRANT message is used to transmit an individual between nodes, and contains the chromosome of the individual as its value. The BEST message is used if a node has reached the goal condition to transmit the individual that triggers the goal condition to the master node. The HALT message is sent by the master to instruct nodes to finish execution and shut down. The MAXGEN message is sent by a node to its neighbours and master in the event that its maximum number of generations has been exceeded, and as such, it should no longer be considered for migration events. Note that the node ID of the sender is sent as the message contents, to allow other nodes to disregard it.

Tag Name	**Tag Value**
MIGRANT	0
BEST	1
HALT	2
MAXGEN	3

Table 6.1: Protocol tags

A brief overview of the life cycle of a typical run of parallel GEPy is as follows:

1. The master and island nodes are started using the `mpirun` command or equivalent. The islands create their populations, and then begin evolving them as per sequential GEPy. The master node enters a loop listening for BEST or MAXGEN messages.

2. The island nodes will then continue evolution asynchronously, until a migration interval is reached. At this point, an island will check emigrate a predefined number of individuals to its neighbours, and check for any immigrants, importing them into its population as necessary.

3. This process will continue until either:

   (a) An island reaches the maximum allowed number of generations, at which point it notifies its neighbours not to send migrants to it anymore,and notifies the mas-

ter that it is no longer functioning. Both of these tasks are carried out using the MAXGEN message. After this, the GEPy process on the island exits.

(b) An island reaches the goal condition. In this case, it notifies the master node with a BEST message, which in turn instructs all the islands to shut down with a HALT message. Once it has done this, the master node exits.

Note that it is entirely possible for an island to find an individual satisfying the goal condition, and hence triggering the sending of a BEST message to the master node before its first migration interval occurs.

Master Node

The code for the master node is very simple, consisting of a while loop that simply checks whether a BEST message has been received, a MAXGEN message has been received, and then if all the nodes have exceeded the maximum number of generations. If a BEST message is received, then the chromosome for the best individual is printed to standard output, HALT messages are sent to all the nodes, and the master then calls `mpi_finalize()` and exits. If a MAXGEN message is received, the variable `num_exceeded` is incremented, which is used for the check for if all the nodes have exceeded the maximum number of generations. If this is the case, then `mpi_finalize()` is called, and the master exits. The master code is purely procedural, consisting of no OO code, as the program is so simple it was felt to be "overkill" to clutter the implementation with an unnecessary class definition.

Migration Functions

There was one key design issue that overlapped both the `immigrate()` and `emigrate()` functions. An appropriate method of serialising the MPIGEIndividual objects for transmission between nodes had to be identified. The first method that was considered was to use one of the standard Python serialisation modules, `pickle`, which provides the ability to convert a Python object into a byte stream and back again. The second method considered was to simply pass just the chromosome of a migratory individual between nodes, and re-instantiate the MPIGEIndividual object when a chromosome is received. Whilst the pickling method would at first glance provide a simple and neat solution of simply passing the individuals between nodes, it quickly came apparent that it would be a less than ideal solution when compared to merely migrating the chromosome data. Firstly, it is less efficient on both a computational and communications level - rather than just passing the contents of an instance variable that is

typically small, an entire object would have to be converted into a byte stream, resulting in data to be transmitted that is often larger than the length of a chromosome. This serialised object would then also have the computational inefficiency of having to be de-serialised upon being received by a node. In contrast, by transmitting the chromosome, the computational cost by a transmitting node is reduced, along with the communications cost. Although there is work involved in instantiating a MPIGEIndividual object by a receiving node, it is typically much less than for de-serialising an existing object. With these considerations in mind, it was elected to transmit just the chromosome of an individual.

In order to allow for the easy alteration of the selection of individuals to be emigrated and to be replaced in a population, we provide a layer of abstraction akin to the `crossover()` and `mutate()` functions in PyGA's Individual class. This is found in the `_emigration_select()` and `_immigration_select()` functions, which simply return a call to the desired selection method. For example, the default emigration selection method is tournament selection, and is called as follows:

```
1 def _emigration_select(self):
2         return self._tournament()
```

The `emigrate()` function is the simpler of the two in their operation. It simply calls `emigrate_select()`, and for each of the returned individuals, it sends a MIGRANT message to the node's neighbours with the chromosome as the message contents.

The `immigrate()` function first receives a copy of the current population minus the individual that is to be replaced from the `_immigrate_select()` function. It then effectively casts the immigrant chromosome (which is passed as a parameter to the function) from a Numeric array to a standard Python array. This chromosome is used to instantiate the MPIGEIndividual object of the new individual, which is then evaluated and added to the node's population. Note that this function deals with a single immigrant at a time - it is the responsibility of the code calling this function to ensure that this function is called for every waiting immigrant at each migration interval

Implementation with GEPy

All of the changes in functionality, in order to implement a parallel implementation of GE, were contained within a class inherited from GEEnvironment, appropriately named MPIGEEnvi-

ronment. We do however, create a corresponding class that inherits from GEIndividual called MPIGEIndividual, although this will not actually alter any of the functionality of GEIndividual, but merely specify options such as which crossover method to use.

The `__init__()` function of MPIGEEnvironment is not a radical departure from the original `__init__()` function in GEEnvironment. It obtains the particular node's rank and the number of nodes in the current mesh and stores them in local variables. These values, along with a class variable `width`, are used by the `find_neighbours` function to calculate the node's neighbours in the mesh (note that this is possible in MPI, as nodes are assigned sequential numbers on spawning). The `__init__()` function of GEEnvironment is then called via a call to `super()`. The other modifications that were made were encapsulated within the `run()`, `step()` and `_goal()` functions.

The `run()` function consists of a loop checking for HALT and MAXGEN messages, and also whether the goal condition has been met, or the maximum allowed number of generations has been exceeded. If a HALT message is encountered, then `mpi_finalize()` is called and the program exits. If any MAXGEN messages have been received, then the senders of these messages are removed from this island's list of neighbours (note that the contents of a MAXGEN message is the rank of the sending island).

The `_goal()` function which is called by `run()` has been modified to separate out the check for whether the maximum number of generations has been found, thus allowing to treat these conditions differently. If `_goal()` returns true, then we broadcast a BEST message to the master node containing the chromosome of the best individual in the population[9]. Should the maximum number of generations be exceeded, then the island's neighbours and the master node are informed with a MAXGEN message, `mpi_finalize()` is called and the island exits. If however, the goal condition has not been met and the current number of generations is acceptable, `step()` is called.

The `step()` function is initially the same as the original version; it sorts the population, and calls the `_crossover()` function in order to create a new generation of individuals. However, we modify its behaviour by adding the functionality to handle the migration of individuals into and out of the population. All migratory functions are performed according to the migration

[9] At this point, we also set a variable that stops the checking of whether the goal condition has been met, if the maximum number of generations has been exceeded, and also to prevent further evolution of the population by preventing `step()` from being called. The island then waits for a HALT message from the master node to exit.

interval. This is a variable that used in conjunction with the modulo operator as follows:

```
1 if self.generation % self.migration_interval == 0 and\
2          self.generation != 0:
```

i.e., migration will be performed at every `migration_interval` generation, and the initial generation is skipped as migrating at the point in time would be fruitless. If migration is to be performed, the `emigrate()` function is called. Then, a check for any waiting immigrants is made by calling a while loop on a `mpi_iprobe()` call for MIGRANT messages. Whilst there is one present, we perform a further check for a waiting HALT message is made (in order to prevent a possible race condition). If one is present, we call `mpi_finalize()` and exit the program. Otherwise, we get the chromosome of the immigrant from the MIGRANT message through the appropriate MPI calls. The chromosome is then used to call the function `immigrate()`.

Finally, once all migratory tasks are complete, the `step()` function returns to its original behaviour, incrementing the number of generations performed, and calling the `report()` function to generate any reporting data.

Chapter 7

Testing

This chapter firstly describes the decision process behind choosing an appropriate testing strategy to be used in the implementation of the project. We then describe the planning that was required for the testing, and finally cover the results of the integration and system testing, discuss their implications.

7.1 Choice of Testing

As stated previously, the choice of testing methodology was something that would have an important influence on the design life cycle that was chosen. We shall now illustrate the decision making process and reasoning that lead to unit testing being selected for this project. As identified from the analysis of libGE and PyGA for the requirements gathering process, two key observations that directly related to testing were made:

1. GE is an inherently iterative process.
2. PyGA, whilst simple from a high level design point of view, is a highly interdependent framework.

Both of these observations pose particular problems from a testing perspective. The complex nature of the way that the system will work beneath the simple high level design means that it would be prohibitively ineffective to solely test the system as a singular entity (or "big bang" approach). If we were to take such an approach, the lack of focus of the testing would mean that

identifying problems and their causes would become a process of essentially educated guesswork, built upon assumptions hampered by a myriad of possible interactions within the system.

Clearly, a testing methodology of finer granularity is needed. Given the object oriented approach of the project, the unit testing approach is ideal for our needs. Unit testing is essentially writing a set of tests for the units or blocks of code in a system, based upon the requirements specification. By breaking the system down into its component objects and the functions that make up those objects, we are able to pinpoint precisely where there are problems with the code. Furthermore, by writing the unit tests before we actually start the implementation, we will be provided with ongoing assurance that what we have written is what we had set out to do [Sommerville, 2001] by running the tests as we add to the implementation.

A commonly used tool in Java development is the JUnit unit testing framework. This provides developers with a simple, elegant and automated way of testing their code. It allows for the construction of test suites which allows the developer to automatically regression test an entire test suite in one fell swoop, or to test individual parts of a system. Python has its equivalent testing framework called "PyUnit" which is derived from JUnit[1], and shares many features with it. PyUnit is written in pure Python, and has been a standard part of the language since version 2.1. The framework itself is made available through the `testunit` module, and tests are implemented by sub-classing the `TestCase` class, asserting statements about various properties of the items that are to be tested.

It is worth noting that an alternative testing framework known as TestOOB, that extends and integrates with PyUnit is available [2] - this provides many more advanced reporting and debugging features over the standard PyUnit offering. Whilst these are certainly useful, it was felt that given the time constraints and relative lack of complexity of the actual code within the classes, it would not be worth spending valuable project time investigating and understanding the extra features that TestOOB provides.

Now that we have a strategy for the low level testing, we should return to examine the higher level testing - namely integration and system testing. The second observation made regarding PyGA from the requirements gathering process was that it is a highly interdependent framework. This has the consequence that it is very hard to separate the integration and system test

[1] It is interesting to note that JUnit is itself heavily influenced by, and based upon the standard Smalltalk testing framework by Kent Beck, who also co-authored JUnit. Given Smalltalk's influence on the design of Python (dynamic typing and support for non-inheritance based polymorphism), this forms a neat circular relation.

[2] TestOOB is also fully syntactically compatible with PyUnit.

phases - as there are so few components of the system, as soon as you start integrating them, you end up with complete system. Hence, top-down and bottom-up testing are irrelevant for our system, and we will have to use an overarching "big bang" testing approach. Whilst this has the obvious drawbacks mentioned before, it is felt that the use of unit testing would help to minimise the impact of this, and allow for a successful and useful testing stage.

With all of this in mind, it allowed us to feed the decisions made for testing back into the design life cycle model decision, resulting in the V-Model as described in section 5.1.2, although it should be noted that the overlapping nature of the System and Integration Testing phases is not shown on the diagram in order to maintain clarity.

7.2 Test Planning

The first key task in implementing unit testing is to identify the various test cases that will be used. As can be seen in Figure 5.1 the Unit Testing stage is contingent on the Low Level Design phase, and intends to validate that the statements made in the Low Level Design are met in the actual implementation. However, we should note that the planning should incorporate knowledge gained from all previous stages of the project, given that the Requirements stage will influence the High Level Design, as such, indirectly (and in some cases directly) influence the Low Level Design stage.

Whilst we shall not document all the tests that are to be done here, a the test plan and source code for an example unit test is available in Appendix C.

It is impossible for us to exhaustively test our system through unit testing, due to the inherently stochastic nature of GE. For example, if we are to test the `_mutate()` function of GEEnvironment, as its functionality is based upon a random number generator there is simply no way of guaranteeing that it will exactly reproduce the same behaviour between successive tests, even if the same random seed is used. In order to deal with this problem, we adopt a strategy of equivalence partitioning [Sommerville, 2001] - testing on specific values that will give known outcomes. For example with the `_mutate` function, we can set the `mutation_rate` variable to be 0.0 or 1.0, thus forcing a mutation to either definitely not occur, or to definitely occur, respectively. Similarly, for functions such as `run_code()` in GEIndividual, we can use a chromosome with a known code expansion, and hence, use an assert statement on the resultant score of that individual. Limiting ourselves to testing in this way, whilst it cannot guarantee

that there might be some cases in which the code will fail, it does provide us with sufficient confidence that the implementation is valid.

In order to provide a system/integration test, it will be necessary to determine that the system is performing GE as expected. In order to do this, we shall take some of the standard tests from the GE papers (namely Symbolic Integration an Symbolic Regression), and by using the available parameters for their runs, attempt to produce similar results. We should note that given the stochastic nature of GE, and as not all parameters for the GE tests were made available in the published papers, in line with the requirements described in section 4.3.2, it is unreasonable to expect to exactly reproduce the results published, although the system *should* be able to produce valid solutions to GE problems.

7.3 Results of Testing

The unit testing proved to be an invaluable tool when it came to the implementation work. It allowed us to catch bugs in the code earlier, and provided us with immediate feedback when changes to the code base were made. This had the effect of speeding up development - bugs were not able to progress to become "deep" bugs that are fundamentally hard to identify (which would have significantly slowed the development process). Furthermore it provided motivation for development - by essentially giving us a "check list" of things to do, it allowed the implementation of GEPy to be broken up into bite-sized chunks, providing us with more attainable goals rather than an overarching goal of "get GEPy working". The only disadvantage was that not all the functions in GEPy could be unit tested, as a result of the functions needing an integrated system in order to work. However, this only affected the minority of functions within the framework.

Whilst not all the parameters for the testing of the original GE system have been made publicly available, we configured GEPy using a combination of known parameters and educated guesses. The configuration for the first test run we made is presented in the GE Tableau format shown in Table 7.1.

The initial system/integration testing provided us with mixed results, which was anticipated from the requirements analysis. Despite numerous attempts at altering parameters, crossover and selection techniques, we could not regularly find the target solution for the standard GE Symbolic Regression test, which was to find the function $X^4 + X^3 + X^2 + X$ (in fact, despite

Objective	Find a function of one independent variable and one dependent variable, in symbolic form given a sample of 20 (x_i, y_i) data points, where the target function is the quartic polynomial $X^4 + X^3 + X^2 + X$
Terminal Operands	X (the independent variable)
Terminal Operators	The binary operators $+$,$*$,$-$ and $/$. The unary operators Sin, Cos, Exp and Log
Fitness cases	The given sample of 20 data points in the interval $[-1, +1]$
Raw Fitness	The sum, taken over the 20 fitness cases, of the error
Standardised Fitness	Same as raw fitness
Hits	The number of fitness cases for which the error is less than 0.01
Parameters	Population size = 500, Maximum generations = 51, Crossover rate = 0.01, Mutation rate = 0.01, Maximum Genes Duplicated = 3, Prune rate = 0.01, Genotype length = 160

Table 7.1: Grammatical Evolution Tableau for Symbolic Regression

many hundreds of attempted runs, the solution was only found once). Numerous parameters were missing which contributed to the inability to find a solution. For example, the papers published on GE state that the input for X are a sample of 20 data points in the interval $[-1, +1]$, yet they do not specify exactly what these values are (note that it does not state that these are a linear range of samples).

The fact that we were able to obtain one valid result shows that it is an achievable goal using our system, although it clearly is suffering from severe problems. It is possible that this valid result is caused by our GA breaking out of a local optima that it tends to fall into and that the GA used in the original implementation of GE achieves the global optima more frequently. We are using a simple GA which the authors of GE also claim to have used, however the discrepancy in results indicate there maybe more to the GA which they are using than they have described in published papers.

When this problem was reached, we took the step of analysing the original GE code. Unfortunately, this was incomplete, and was lacking the GA needed to make it work. Clearly, this prevented us from getting a true picture of the conditions under which the GE tests were run, and the time constraints of the project prevented us from contacting and discussing this matter with the authors of GE. Furthermore, given that this very same problem has also been experienced by others trying to implement GE ([Griffith, 2002] and [Painter, 2006]), it is felt that this is an acceptable level of performance which matches the requirements laid out in section 4.3.2. It is certainly worth considering releasing GEPy into the public domain so as to allow others to investigate this issue further.

Further from this initial test problem it was decided to investigate the second problem that was listed in the requirements: the Symbolic Integration test with a target function of $Sin(X) + X + X^2$. This resulted in much a improved success rate compared to the Symbolic Regression problem. From exhaustive testing of this particular GE problem (involving nearly 800 GE runs, resulting in over 2 weeks of computation time on a 2.0GHz Intel Pentium M processor), we were able to produce detailed statistics.

The same parameters were used as in the Symbolic Regression experiment shown in Table 7.1, however the objective was now to find the target function $Sin(X) + X + X^2$.

Initially, the suggested GA configuration from the Literature Review was used, comprising of steady state population update, roulette wheel selection and one point crossover. However,

it quickly became apparent that steady state selection hindered our attempts to find a solution - in fact, we were not able to find a solution at all using this method. It is possible that this particular problem is highly dependent on the use of the duplication operator and the steady state population update method will inherently result in less individuals in the population having duplication applied to them. Some basic experimental work to determine this, by altering the duplication rate was made, however due to time constraints on the project, we were not able to investigate this fully and as such the results were inconclusive. Hence, in order to improve the performance of the GA, elitist population update was used, which has often been found to produce favourable results [Mitchell, 1996].

From empirical testing and comparison of the tournament and roulette wheel selection it became apparent that the difference in performance between them from a fitness point of view seemed to be negligible in this case, whilst the computational efficiency of the tournament method was vastly superior. From this finding, it was decided that it would be of more benefit to use tournament selection in order to provide us with the ability to gain more results in the limited timeframe available to us.

As can be seen by comparing our results with in Figure 7.1 with the original GE results (from [Ryan and O'Neill, 1998]) in Figure 7.2, the plots of cumulative frequency of success rate show the clear discrepancy between the published performance of GE with steady state population update and our implementation using elitist population update and tournament selection. Unfortunately our implementation has a success rate far below the published success rate of 87% for steady state GE - we have only achieved a success rate of 30.1% of GE runs finding the solution within 50 generations (and a 54.4% success rate within 120 generations).

This discrepancy can again be attributed to the result of too many unknown parameters of the GA that the authors had used. It is worth noting, however that it is an improvement upon the published 22% success rate of the original generational GE implementation (which can clearly be seen on the graphs), which supports the traditional notion of performance gains from using elitist population update and tournament selection with a GA [Mitchell, 1996].

Whilst it is not immediately apparent why this test problem worked, it is certainly worth considering that this may just simply be a simpler problem to solve than the Symbolic Regression problem, thus allowing an answer to be found. On the basis that this provides us with a useful working example problem for GE, we shall take this result forward into our experimental work to be used as the test problem and test parameters for investigating parallel GE (although

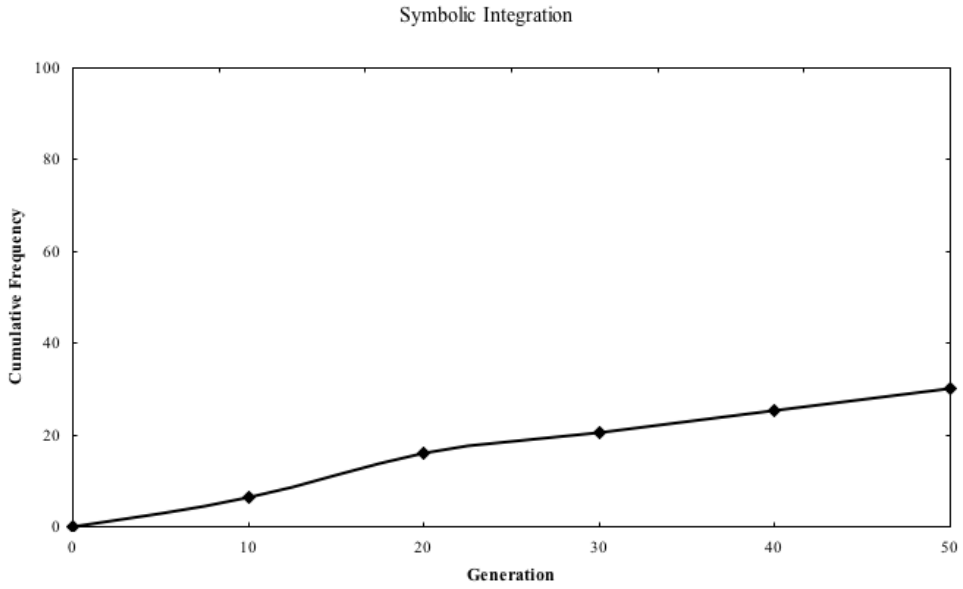

Figure 7.1: GEPy Symbolic Integration cumulative frequency measure.

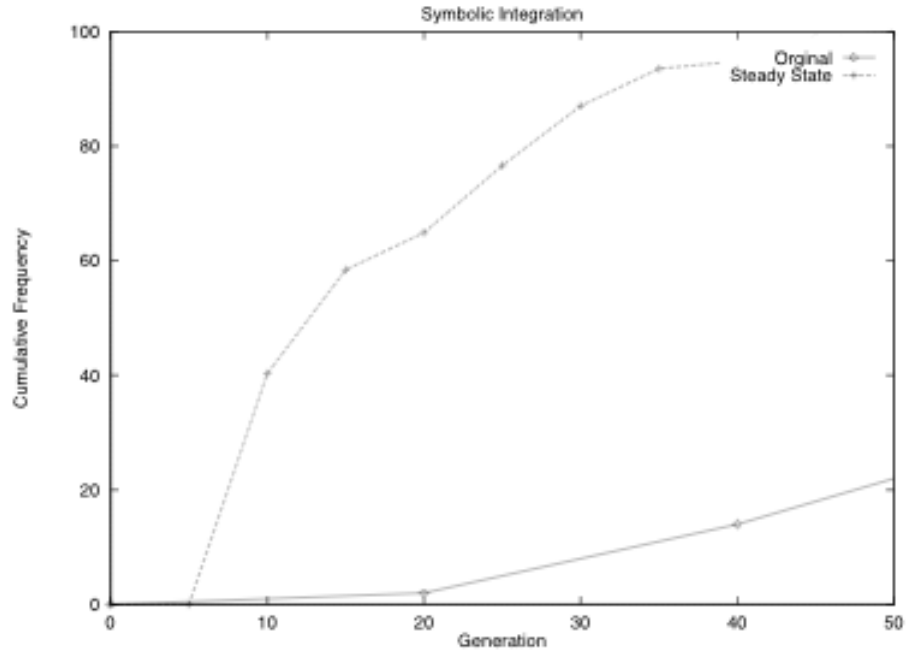

Figure 7.2: GE Symbolic Integration cumulative frequency measure.

the apparent simplicity of the problem can have consequences for a parallel implementation, which we shall cover in the Experimental Work chapter).

Chapter 8

Experimental Work

In this chapter, we shall describe the experimental work done to investigate the effects of parallelism on GE. We shall first form some hypotheses and describe the method to be used in the experiments. We shall then present the results from our experimental work, and then present a conclusion and discussion based upon these results.

8.1 Hypothesis and Experimental Method

As stated in the introduction to this dissertation, one of the aims was to investigate the applicability of parallel techniques to try to over come the performance penalty incurred through using the Python language. Various techniques have been used over the years to parallelise GAs, which are described in section 2.1.8. We have implemented a parallel version of GEPy, based upon a coarse grained parallel GA (or island model) approach. This utilises a 2-D mesh topology, with nodes organised in an NxN structure. The nodes (or islands) will be identical in terms of parameterisation, but will be initialised and will evolve asynchronously.

From this basis, we shall state our experimental hypothesis to be that a parallel version of GEPy will provide a distinct performance gain over sequential GEPy, finding the optimal solution in a shorter period of time. Furthermore, we form the sub-hypothesis that a side effect of using a parallel GA will result in an improved success rate of finding the optimal solution within the maximum allowed number of generations. These hypotheses are formed on the basis of prior work with parallel GAs (see section 2.1.8). There has been no previous attempt to implement a parallel version of GE and whilst we can form a conjecture that this will help, it

could be the case that it will not bring performance benefits. For example, a parallel GA can be configured in such a way that it actually hinders performance in comparison to a sequential version [Cantú-Paz, 1997]. Conversely, given the relatively low bandwidth required to perform parallel GE, and the relatively high bandwidth connections between islands (detailed later), it is a possibility that super-linear speedup could occur.

In order to test this, we will clearly need to form a comparison between our standard sequential version of GEPy with our parallel implementation. In order to test our hypothesis and sub-hypothesis, two measurements will have to be made:

1. The time taken for a GE run to complete
2. Whether or not the run will find the optimal solution within the maximum permitted number of generations

Timing will be performed using the standard Linux command `time` (this is so we fully include the start up and shutdown time of the Python interpreter, as opposed to using the Python `timeit` module). Given the stochastic nature of GE, it will be necessary to perform multiple runs and obtain an average for each configuration. Whilst we will not have the time available to fully explore the "tuning" of the parameters controlling the parallel implementation (migration rate, migration interval, etc.), and given that there is not a full understanding of how to tune a parallel GA (see section 2.1.8), we shall have to limit the range of parameters for the parallel GA that we alter. From examining previous work, it was discovered that the migration interval (how often migration takes place) [Skolicki and Jong, 2005] is perhaps one of the most dominating factors on the performance of an island model GA (especially when compared to the migration rate or size - the number of individuals migrated). Hence we shall focus our efforts on tuning the parallel implementation through this parameter.

Whilst it was originally planned to test the parallel implementation on the Department's cluster, various technical problems[1] resulted in this not being possible. Thankfully, due to the assistance of a member of staff at the Department of Computer Science at the University of Malta, we were able to gain access to an alternative cluster. This consisted of 17 machines,

[1] We were unable to install some of the required libraries on the cluster, such as Numeric. Whilst a workaround was attempted by placing the associated shared object library files onto the NFS share, this would not work. The operating system would try to map the shared object in memory, however NFS does not support such operations (as with any IPC structures over NFS), and would simply result in a permission denied error when the library was loaded.

each equipped with a Dual Core 2.8Ghz Intel Pentium 4 processor and 1GB of RAM, connected via a 100MBps switched Ethernet network. The machines will be running Fedora Core 2 Linux, and the MPI library shall be LAM/MPI 7.0.3. This will mean that we will be able to test 2-D toroidal meshes up to a width of 4 (as we will have a 4x4 mesh, complete with a master node).

We shall be testing the Symbolic Integration problem that was found to be successful in the testing stage of GEPy. We shall use the same parameters for the configuration of GEPy as were used for the Symbolic Integration problem in the testing stage (see Table 7.1; note that the objective is the target function $Sin(X) + X + X^2$), and we shall use the following parameters for the parallel implementation configuration:

Parameter	**Values**
Migration Size	1
Migration Interval	2,5,10,15,20
Mesh Width	1,2,3,4

Table 8.1: Parallel Parameters

Migration is to be performed using a "best-worst" policy - we emigrate the best individual from a population, and replace the worst member of a population with an immigrant. The reasoning behind is two-fold; firstly, by using the best individual, we will maximise the chances of migrating "good" building blocks between populations. Secondly, by replacing the worst individual in a population, we are minimising the immediate impact on the population's gene pool, as the worst individual will be replaced through crossover anyway.

We shall not provide a theoretical prediction of speedup, as due to the stochastic nature of GE (and GAs in particular), T_{comp} will vary greatly (and thus the time intervals of communications between neighbouring islands will vary greatly). Whilst it is possible to form a prediction of speedup by factoring in the stochastic variance (also, see the work done on GA performance prediction in [Louis and Rawlins, 1992] and [Cantú-Paz and Goldberg, 1997]), this is beyond the scope of these experiments (and indeed project), and would not add substantially to the value of the work being done.

Processors	Rank of speedup				
	1	2	3	4	5
4	Mi 10	Mi 20	Mi 5	Mi 15	Mi 2
9	Mi 10	Mi 2	Mi 15	Mi 5	Mi 20
16	Mi 5	Mi 15	Mi 20	Mi 2	Mi 10

Figure 8.1: Table showing ranking of speedup for various migration intervals at different numbers of processors.

8.2 Experimental Results

The experimental results for the tests relating to the main hypothesis are shown in Figure 8.2 [2], with the various migration intervals being identified by the key Mi*N*, with N being the migration interval being tested (i.e. Mi2 is the series with a migration interval of 2). There are several features of this graph that are notable. Firstly, all of the different configurations resulted in a sub-linear speedup, with the most economical speedup occurring for a mesh of width 2.

Secondly, we should examine the different plots of migration intervals in order to try to determine any trends. Theoretically, a small migration interval will result in the individual islands acting more as a single population, and conversely, a high migration interval will result in them working more independently. The ideal trade off between these should lie somewhere between the two extremes. From our results, it is identify with certainty a trend in the migration intervals we tested. If we examine the graph of the speedup without the inclusion of the linear trend (Figure 8.3), then we can see the effects of the migration intervals more clearly.

At 4 processors, it appears that all migration rates are rising, with the medium to high migration intervals performing somewhat better overall. However, once the number of processors rises to 9, the trend changes, with there being no clear pattern. When the migration interval is set to 10, it is the best performer at 4 and 9 processors. Its advantage seems to diminish rapidly after 9 processors, and at 16 processors, it is the worst performer. By placing the migration intervals in a table showing their ranking at the differing number of processors (see Figure 8.1), one further possible trend is identified. It seems that as the number of processors increases, the performance of migration interval 15 increases.

[2] As it was unfeasible to plot the standard deviations as error bars on the graphs in this section., we have placed individual plots of each series (as number of processors vs. average time) in section D.1 of the Appendix.

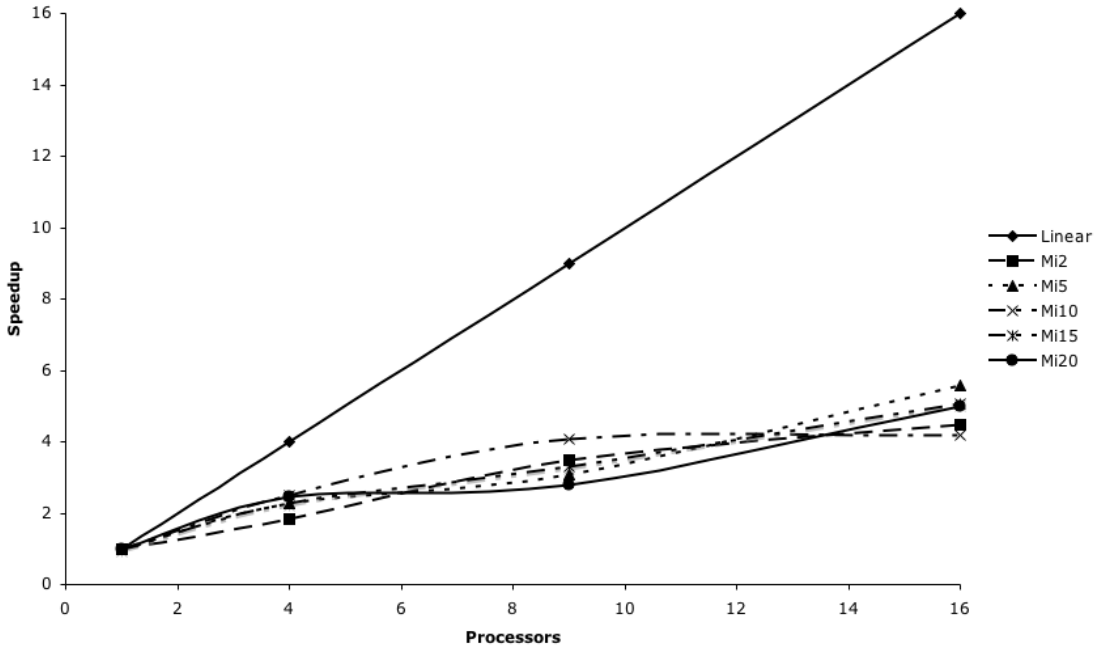

Figure 8.2: Speedup for various mesh widths and migration intervals.

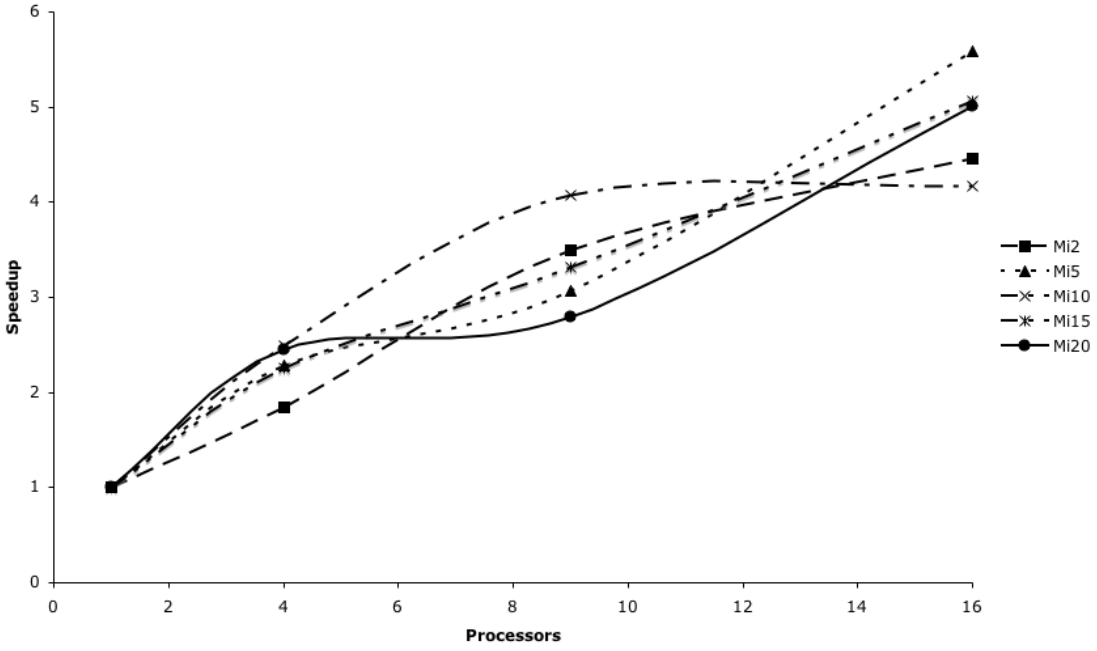

Figure 8.3: Speedup for various mesh widths and migration intervals (without linear plot).

Our results for the success rates of the various configurations are shown in Table 8.2. It should be noted, that apart from the entries in the table all the runs were able to find the optimal solution within the maximum allowed generations limit of 120.

Mesh Width	**Migration Interval**	**Success Rate**
Sequential	N/A	58%
2	2	92%
2	5	98%
2	10	98%
2	15	98%

Table 8.2: Success Rates

8.3 Conclusion and Discussion

From examining Figure 8.2, it is clear that whilst being sub-linear, a parallel implementation of GEPy does provide a distinct performance gain over sequential GEPy. The ideal configuration from an economy point of view is a 2x2 mesh, using a migration interval of 20, which provides a speed up of 2.45. Given the small size of the mesh, we can reason that the larger migration interval helps to prevent the mesh from acting as a single population, and thus allows the run to benefit more from the parallelism.

It is possible that if the Symbolic Integration problem we are testing is relatively simple to solve (as postulated in section 7.3), then we may not be seeing the full speed up that is available from a parallel implementation of GEPy. Time constraints made it impractical to fully pursue this line of inquiry. However, this is an area of investigation that could improve the parallel implementation of GEPy, and there is cause to be optimistic that further research would produce beneficial outcomes.

As indicated in the results, no clear pattern emerges for testing various migration intervals. Whilst a migration interval of 10 is best for 4 and 9 processors, it turns out to be the worst migration interval for 16. A migration interval of 15 steadily increases in performance as the number of the processors in the system increases. From the mesh topology that was used, we know that increasing the number of processors causes the system to act less as a single population. However, we do not have enough data to ascertain the mechanism of interaction between the number of processors and the different migration intervals, and further research is required

to draw firm conclusions regarding this.

The results documented in Table 8.2 show that a parallel implementation of GEPy results in a dramatic increase of the success rate. Even with a minimal 2x2 mesh, a 100% success rate can be achieved by using a migration interval of 20 (which coincidentally also gives us the best speedup performance at this number of processors).

These conclusions confirm the validity of parallel techniques for GE, from both a speedup performance perspective and from also an algorithmic performance perspective. Given the light effort required to create a parallel implementation of GEPy, this is something that is of definite worth and we would recommend further investigation into using parallel techniques in a wider range of GE problems (perhaps sourcing test cases from Genetic Programming literature), especially in the comparison of speedup in problems of varying difficulty.

It interesting to compare the speedup gained from using a just-in-time compiler for Python, called Psyco. By compiling the GEPy and PyGA code (whilst not compiling the Grammar or ConfigHandler code in order to minimise start up time), we were able to observe a speedup of 1.12. Whilst parallel techniques still provide a more substantial performance gain, it would be worth considering the combination of both Psyco and a parallel implementation of GEPy to maximise speedup (we should note that this would only be applicable on Intel platforms, as this is the only processor family that Psyco supports).

Other non-parallel options for speeding up the performance lie in variants of Python that target the Common Language Runtime, compiling Python programs into byte-code that will run on either the Microsoft .NET Platform or the Open Source implementation of this, the Mono Development Platform. There are two prime examples of this, IronPython and Boo (we should note that Boo is not strictly a variant of Python - it is a language in its own right, however many Python programs can be compiled by it in an unaltered state). By compiling to this byte-code rather than to the Python Virtual Machine byte-code, significant performance gains are possible - IronPython quotes a figure giving a speedup of 1.7 for the PyStone benchmark. As with using Psyco, we could combine parallel techniques and these compiler technologies in order to bring about more significant speedup increases to GEPy.[3].

There are however, a couple of caveats that should be considered for these non-sequential

[3]We should note that Python provides the ability to use modules written in C, however due to the need to maintain simplicity of the implementation this shall not be considered

speedup techniques. Firstly, whilst T_{comms} and the Amdahl portion will become more significant under a compiled version of GEPy, the intensive computational nature of GE means that these will still not play a great factor. Also, there is the intrinsic problem that the dynamically generated code of the individuals in the population cannot be compiled, and as such this would not see any performance gain from using such compilation techniques.

Chapter 9

Conclusion

9.1 Critical Evaluation

As stated in the Introduction, the aims of this project were to:

1. Produce a simple, easy to understand and easy to use GE framework in Python.
2. Use this framework to investigate the effect on performance that a parallel implementation of GE would bring.

In this section, we shall discuss the degree to which these aims of the project were met.

GEPy Framework

The requirements described in section 4.3 will be discussed, and more specifically, the degree to which the implementation of the project achieves these will be identified. Some of the requirements are not critical to this section, so those that are most important shall be discussed.

Global Requirements

Readability of code is primarily a qualitative judgement, so it is difficult to provide quantitative analysis. However the framework has a significantly smaller code base (GEPy consists of approximately 900 lines of Python code compared to approximately 3700 lines of C++ code for libGE) and is purposely designed in a manner that is less complicated than libGE.

The framework was successfully implemented in line with the Python style guide [van Rossum and Warsaw, 2001], resulting in the code having a consistent and standard style, which contributes significantly to improved readability. We can conclude, therefore, that we have met the requirement for a clear and easily understood framework.

The system was successfully created using a standard Python interpreter. The burden of installation was minimised by ensuring that any Python modules used could simply be packaged with the framework. Finally, the code was fully documented with pydoc compatible comments which provides researchers with readily accessible useful information on the various functions within the framework. From all of this, the outcomes of the project enable us to can confidently conclude that the key requirement that the implementation should be easy to read and understand has been met.

Genetic Algorithm

The majority of the GA requirements were immediately met through basing the framework on the use of PyGA. The modular approach adopted by PyGA was carried through into our implementation of GEPy, which meant that the requirement that it should be simple to add new functionality of the system was met. We included a configuration system based around XML files, which meets the requirement of a simple configuration of the framework external to the code.

Support for both the Symbolic Integration and Symbolic Regression problems was provided "out of the box". As highlighted in the results of testing (see section 7.3), we could not match the performance or reproduce the results contained in the published GE papers. Since the same problem has been encountered by others implementing GE systems (see [Griffith, 2002] and [Painter, 2006]), some discrepancies between the results in our testing and publishing GE papers [Ryan and O'Neill, 1998] might be anticipated. There is reason to suspect that full disclosure of the parameters and GA used in the original GE system, the performance of GEPy would improve to a point where it matches the original implementation.

Parallel Implementation

With the GEPy framework in place, we were in the position to quickly and easily create a simple island model version of GE to investigate the effects of parallelism. This was used to measure the different levels of speedup and success rate through altering the mesh width and migration interval, the findings of which supported the following hypotheses:

1. A parallel version of GEPy will provide a distinct performance gain over sequential GEPy, finding the optimal solution in a shorter period of time.

2. That a side effect of using a parallel GA will result in an improved success rate of finding the optimal solution within the maximum allowed number of generations.

To conclude, it is felt that using parallel computing techniques brings significant benefits to GE. Whilst the speedup is sub-linear, it is still of a significant level and when combined with the vastly increased success rate that parallelism brings the advantages of adopting this approach can clearly be seen. It would have been desirable to conduct further testing using the parallel version of GEPy in order to identify the full potential for speedup by investigating the effect of different parameter settings. However, time constraints precluded us from doing this.

9.1.1 Conclusion

The outcomes of the project enable to conclude that its two aims were successfully met. Our experience in using the GEPy framework to implement a parallel version of GE gave us confidence in that it was a good and usable product for researchers in the GE community. Given the readability of GEPY's code, it could prove to be a useful education tool for allowing others to gain understanding of GE systems. This gives a sound basis for releasing GEPy into the public domain in order to further identify the needs of researchers and refine GEPy's code base as a result.

9.2 Future Work

Improvements to GEPy

There are two clear focal points for future development work improving GEPy;

1. The need to minimise the handicap of Python performance within the sequential version of GEPy.

2. Add more features to improve the practical range of use of the GEPy framework.

Whilst it is not realistic to expect to improve the performance of GEPy to match that of GE implementation in C++ or Java, it is entirely possible to create substantial gains in performance.

Detailed analysis of the GEPy code could be performed using profiling techniques to identify "hotspots" in the code and focus energy on improving the performance of these. Whilst we conducted some preliminary work into the feasibility of using the Psyco just-in-time compiler for Python (see section 8.3), it is felt that further work in this area, and also investigating the use of IronPython would be highly beneficial. However, techniques that give a performance gain should be evaluated against any additional complexity that using them adds to the code of GEPy.

There are several extra features that would be desirable to add to GEPy. For example, extra crossover methods (e.g. homologous crossover), population update methods and provision of more grammar formats (e.g. XML grammars) would allow for a wider range of configuration options to be at a researcher's disposal.

More advanced reporting options would greatly benefit a researcher. For example, the graphs in the Testing and Experimental Work chapters were produced by logging results to a file in CSV file, then importing this into Microsoft Excel. It would be advantageous if GEPy could reduce the workload on the researcher by integrating graph plotting into a standard reporting function. There are several Python modules and packages that could be used to add such functionality. For example, the `Gnuplot` package, which interfaces with (and requires) the Gnuplot graphing program. Alternatively, the `SciPy` package provides similar features in a standalone package.

Finally, given the favourable results of the parallel implementation of GE it is worth considering the inclusion of such a version within GEPy, to allow researchers to be able to further investigate parallelism with GE "out of the box".

Just as with the potential for implementing changes to GEPy to increase performance, it is important to evaluate the additional complexity that adding new features would bring to its code base. The primary aim of the GEPy was for a simple and easy to use GE framework. Such extensions and modifications to its code should be made with caution, so as not to lose the main benefits of the implementation.

Future Research

This project has been the first application of a parallel GA with GE, and its positive outcomes make it a clear area of future research. Our experimental results have confirmed that GE is

highly suitable for parallelism and leave plenty of room for further exploration. For example, whilst we have been able to assert the value of parallel techniques with GE, we have not been able to ascertain the effect of parallelism on genetic diversity for GE.

We recommend that GEPy should be made available as open source software in the public domain. Whilst the performance limitations of Python may be too constrictive for the use of GEPy in "real world" problems, it is worth considering that the simplicity of GEPy may be of considerable benefit to researchers needing a rapid prototyping tool to develop a proof of concept of a GE based system, before rewriting it in, for example, C++ with libGE for computational efficiency.

Furthermore, by open sourcing GEPy it will allow for others to work towards configuring GEPy so as to reproduce the results that have been published for the original GE implementation.

9.3 Personal Reflection

Overall, I am satisfied and happy with how the project went. In GEPy, I feel that we have been able to provide a tool of value to the GE community which, given its relative simplicity and readability in comparison to existing implementations, can hopefully provide newcomers to GE with a shallower learning curve to being able to understand and implement GE based systems. Also, through the experimental work conducted, it has been possible to determine that parallel computing techniques bring significant performance benefits to a GE system.

Earlier access to the Department's cluster would have given me more time for further testing of the parallel implementation of GE. I would have been able to identify the problems that prevented me from using it for testing earlier, and hence made alternative arrangements sooner, thus allowing more testing to be performed. Similarly, had the problems involving the standard GE tests been identified at an earlier stage, more time would have been available to try to work with the GE community in order to ascertain exactly what was missing from our implementation, in order to replicate the published results of GE.

The project allowed me to consolidate and apply skills I have acquired from many of the modules I have studied during my degree. For example, the first year Software Engineering modules gave me the basis to suitably plan the life cycle of the project and design of the

framework, thus allowing for a solid piece of software to be developed. Similarly, I have been able to incorporate prior knowledge from modules such as Compilers (BNF grammars) and Computability and Decidability (regarding the Halting Problem in section 6.1.4). Alongside this, naturally it has also allowed me to apply more general skills I have developed during the course, such as programming, research, scientific analysis and written communication.

The project has allowed me to acquire skills and knowledge in several areas external to the core curriculum of the Degree, which was one of the factors that led to the decision to undertake it. Whilst this resulted in there being a steep learning curve to tackle before I could fully engage with the project, it has been a rewarding process. Perhaps the most distinct area of acquiring external knowledge has been that of studying Biology in order to understand the basis of GAs. It was refreshing to return to an area of science that I have not studied since secondary school, and to use this knowledge in relation to the course. I have often been fascinated by biomimetic systems, and it was enjoyable to actually be able to research and implement one myself. More closely related to the course, I have also been able to acquire knowledge and understanding of parallel computing and genetic algorithms.

I feel it was an interesting and challenging project, which enable me to usefully combine much of the knowledge and skills I have acquired during my degree. It was satisfying to produce a tool that could be used by others in future research, and also to be the first to investigate a parallel implementation of GE.

9.4 Conclusion

To conclude, the project has resulted in the successful development of a usable Python GE framework in GEPy, the value of which has been endorsed by our use of it for our investigation into parallelism and GE. This investigation lead to the conclusion that parallel techniques provide speed up for GE and also vastly increases the success rate of a GE system. The time constraints of the project meant that we could not fully investigate the benefits of parallelism, but this has been identified as an area of possible research for the future. Should GEPy be released into the public domain, we hope that it will prove to be a useful addition to the tools available to the GE community.

Bibliography

B. Alberts et al. *Molecular biology of the cell*. Garland Science, 4th edition, 2002.

G. Amdahl. The Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. *AFIPS Conf. Proc.*, 30:483–485, 1967.

James E. Baker. Adaptive Selection Methods for Genetic Algorithms. In *Proceedings of the 1st International Conference on Genetic Algorithms*, pages 101–111, Mahwah, NJ, USA, 1985. Lawrence Erlbaum Associates, Inc. ISBN 0-8058-0426-9.

James E. Baker. Reducing bias and inefficiency in the selection algorithm. In *Proceedings of the Second International Conference on Genetic Algorithms on Genetic algorithms and their application*, pages 14–21, Mahwah, NJ, USA, 1987. Lawrence Erlbaum Associates, Inc. ISBN 0-8058-0158-8.

Alwyn Barry. Parallel Systems Lecture Notes, 1996. URL `http://www.cs.bath.ac.uk/$\sim$amb/UQC007H3/index.html`.

D. Beasley, D. Bull, and R. Martin. An Overview of Genetic Algorithms:Part I, Fundamentals, 1993. URL `http://overcite.lcs.mit.edu/article/beasley93overview.html`.

E. Cantú-Paz. A Survey of Parallel Genetic Algorithms. *Calculateurs Paralleles, Reseaux et Systemes Repartis*, 10(2):141–171, 1997. URL `http://citeseer.ist.psu.edu/article/cantu-paz97survey.html`.

E. Cantú-Paz. Designing Efficient Master-Slave Parallel Genetic Algorithms. In John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba, and Rick Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, page 455, University of Wisconsin, Madison, Wisconsin, USA, 22-25 1998. Morgan Kaufmann. URL `http://citeseer.ist.psu.edu/36912.html`.

E. Cantú-Paz and D. Goldberg. Predicting Speedups of Idealized Bounding Cases of Parallel Genetic Algorithms, 1997. URL http://citeseer.ist.psu.edu/article/cantu-paz97predicting.html.

J. P. Cohoon, S. U. Hegde, W. N. Martin, and D. Richards. Punctuated equilibria: a parallel genetic algorithm. In *Proceedings of the Second International Conference on Genetic Algorithms on Genetic algorithms and their application*, pages 148–154, Mahwah, NJ, USA, 1987. Lawrence Erlbaum Associates, Inc. ISBN 0-8058-0158-8.

Mike Cotterell and Bob Hughes. *Software Project Management*. McGraw-Hill Publishing Co., 2002. ISBN 007709834X.

Charles Darwin. *On The Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life*. John Murray, London, 1859.

L. J. Fogel, A. J. Owens, and M. J. Walsh. *Artificial Intelligence through Simulated Evolution*. John Wiley & Sons, New York, 1966.

Frank D. Francone, Markus Conrads, Wolfgang Banzhaf, and Peter Nordin. Homologous Crossover in Genetic Programming. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1021–1026, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann. ISBN 1-55860-611-4. URL http://www.cs.bham.ac.uk/$\sim$wbl/biblio/gecco1999/GP-463.pdf.

G. A. Geist, J. A. Kohla, and P. M. Papadopoulos. PVM and MPI: A Comparison of Features. *Calculateurs Paralleles*, 8(2):137–150, 1996. URL http://citeseer.ist.psu.edu/geist96pvm.html.

Kandy D. Baumgardner Gerald D. Elseth. *Principles of Modern Genetics*. West Publishing Company, 1995.

David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989. ISBN 0201157675.

David E. Goldberg and Kalyanmoy Deb. A Comparative Analysis of Selection Schemes Used in Genetic Algorithms. In Gregory J. E. Rawlins, editor, *Foundations of Genetic Algorithms*, pages 69–93. Morgan Kaufmann Publishers, 1991.

Rebecca Griffith. A Java Implementation of Grammatical Evolution. Master's thesis, University of Bath, 2002.

Paul Bryant Grosso. *Computer simulations of genetic adaptation: parallel subcomponent interaction in a multilocus model*. PhD thesis, University of Michigan Computing Center, 1985.

Peter J. B. Hancock. An Empirical Comparison of Selection Methods in Evolutionary Algorithms. In *Selected Papers from AISB Workshop on Evolutionary Computing*, pages 80–94, London, UK, 1994. Springer-Verlag. ISBN 3-540-58483-8.

J. H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Oakland,CA, 1975.

J. H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. MIT Press, Cambridge, MA, USA, 1992. ISBN 0262082136.

Timothy Kaiser. MYMPI, 2006. URL `http://peloton.sdsc.edu/$\sim$tkaiser/mympi/`.

Motoo Kimura. *The neutral theory of molecular evolution*. Cambridge University Press, 1983. ISBN 0-521-23109-4.

John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992. ISBN 0-262-11170-5.

James R. Levenick. Inserting Introns Improves Genetic Algorithm Success Rate: Taking a Cue from Biology. In *ICGA-91*, pages 123–127, 1991.

Harvey F. Lodish, D. Baltimore, A. Berk, S. L. Zipursky, P. Matsudaira, and James E. Darnell. *Molecular Cell Biology*. W. H. Freeman, New York, NY, USA, third edition, 1995. ISBN 0-7167-2380-8.

A. Javier Lopez. ALTERNATIVE SPLICING OF PRE-mRNA: Developmental Consequences and Mechanisms of Regulation. *Annual Review of Genetics*, 32:279–305, 1998.

S. Louis and G. Rawlins. Predicting convergence time for genetic algorithms, 1992. URL `http://citeseer.ist.psu.edu/louis92predicting.html`.

Melanie Mitchell. *An introduction to genetic algorithms*. MIT Press, Cambridge, MA, USA, 1996. ISBN 0-262-13316-4.

Michael O'Neill. Automatic programming with grammatical evolution, 13 July 1999.

Michael O'Neill and Conor Ryan. libGE, 2005. URL http://waldo.csisdmz.ul.ie/libGE/.

Michael O'Neill and Conor Ryan. Evolving Multi-line Compilable C Programs. In *Proceedings of the Second European Workshop on Genetic Programming*, pages 83–92, London, UK, 1999a. Springer-Verlag. ISBN 3-540-65899-8.

Michael O'Neill and Conor Ryan. Genetic Code Degeneracy: Implications for Grammatical Evolution and Beyond. In J. Nicoud D. Floreano and F. Mondada, editors, *ECAL99*, pages 149–153, Berlin, 1999b. Springer-Verlag. URL http://www.isrl.uiuc.edu/$\sim$amag/langev/paper/oneill99geneticCode.html.

Michael O'Neill and Conor Ryan. Crossover in Grammatical Evolution: A Smooth Operator? In Riccardo Poli, Wolfgang Banzhaf, William B. Langdon, Julian F. Miller, Peter Nordin, and Terence C. Fogarty, editors, *Genetic Programming, Proceedings of EuroGP'2000*, volume 1802 of *LNCS*, pages 149–162, Edinburgh, 15-16 April 2000. Springer-Verlag. ISBN 3-540-67339-3.

Michael O'Neill and Conor Ryan. Under the Hood of Grammatical Evolution. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1143–1148, Orlando, Florida, USA, 13-17 July 1999c. Morgan Kaufmann. ISBN 1-55860-611-4. URL http://www.cs.bham.ac.uk/$\sim$wbl/biblio/gecco1999/GP-434.ps.

Michael O'Neill, Conor Ryan, Maarten Keijzer, and Mike Cattolico. Crossover in Grammatical Evolution: The Search Continues. In J. Miller, M. Tomassini, P. L. Lanzi, C. Ryan, A. G. B. Tetamanzi, and W. B. Langdon, editors, *Proceedings of the Fourth European Conference on Genetic Programming (EuroGP-2001)*, volume 2038 of *LNCS*, pages 337–347, Lake Como, Italy, 2001. Springer Verlag. ISBN 3-540-43378-3. URL http://link.springer.de/link/service/series/0558/papers/2038/20380337.pdf.

Timothy Painter. Grammatical Evolution In Python, 2006. Bachelor of Science of the University of Bath dissertation - yet to be published.

Mark Pilgrim. Dive into Python, 2004. URL http://www.diveintopython.org/.

Rob Pooley and Perdita Stevens. *Using UML: Software Engineering with Objects and Components*. Addison-Wesley, 1999.

I. Rechenberg. Cybernetic solution path of an experimental problem, 1965. Royal Aircraft Establishment, Library Translation Number 1122, Farnborough, UK.

Conor Ryan and Michael O'Neill. Grammatical Evolution: A Steady State approach. In John R. Koza, editor, *Late Breaking Papers at the Genetic Programming 1998 Conference*, University of Wisconsin, Madison, Wisconsin, USA, 22-25 1998. Stanford University Bookstore. URL `http://citeseer.ist.psu.edu/article/ryan98grammatical.html`.

Conor Ryan and Michael O'Neill. How to do Anything With Grammars. In Alwyn M. Barry, editor, *GECCO 2002: Proceedings of the Bird of a Feather Workshops, Genetic and Evolutionary Computation Conference*, pages 116–119, New York, 8 July 2002. AAAI.

Conor Ryan, J.J. Collins, and Michael O'Neill. Grammatical Evolution: Evolving Programs for an Arbitrary Language. In W. Banzhaf, R. Poli, M. Schoenauer, and T.C. Fogarty, editors, *Proceedings of the First European Workshop on Genetic Programming (EuroGP'98)*, volume 1391 of *LNCS*, pages 83–96, Paris, France, 1998. Springer Verlag. ISBN 3-540-64360-5. URL `http://link.springer.de/link/service/series/0558/papers/1391/13910083.pdf`.

T. Schwann and MJ. Schleiden. Mikroskopische Untersuchungen uber die bereinstimmung in der Structur und dem Wachstum der Tiere und Pflanzen. *Translation in Sydenham Soc, London*, 12, 1847. Originally published in Berlin, 1839.

Zbigniew Skolicki and Kenneth De Jong. The Influence of Migration Sizes and Intervals on Island Models. In *Proceedings of Genetic and Evolutionary Computation Conference – GECCO-2005*, pages 1295–1302. ACM Press, 2005.

Ian Sommerville. *Software Engineering*. Addison-Wesley, 6th edition, 2001.

Christian Tismer. Stackless Python. `http://www.stackless.com`, 2006.

A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proc. London. Math. Soc.*, 2:230, 1936.

Guido van Rossum and Fred L. Drake. Python Library Reference, 2006a. URL `http://docs.python.org/lib/typesnumeric.html`.

Guido van Rossum and Fred L. Drake. pydoc – Documentation generator and online help system, 2006b. URL http://www.python.org/doc/current/lib/module-pydoc.html.

Guido van Rossum and Fred L. Drake. Python Library Reference, 2006c. URL http://docs.python.org/lib/content-handler-objects.html.

Guido van Rossum and Fred L. Drake. Python Library Reference, 2006d. URL http://docs.python.org/lib/typesseq-xrange.html.

Guido van Rossum and Barry Warsaw. PEP 8 – Style Guide for Python Code, 2001. URL http://www.python.org/dev/peps/pep-0008/.

Michael D. Vose. A critical examination of the schema theorem. Technical Report UT-CS-93-212, University of Tennesse, 1993. URL http://citeseer.ist.psu.edu/vose93critical.html.

Tim Westergren. Pandora, 2005. URL http://www.pandora.com/. Pandora Media Inc. is a commercial spin off from the Music Genome Project.

Wikipedia. Guppy, 2006. URL http://en.wikipedia.org/wiki/Guppy.

Wikipedia. Backus-Naur form, 2005a. URL {http://en.wikipedia.org/wiki/BacksNaurForm}.

Wikipedia. Prokaryote, 2005b. URL {http://en.wikipedia.org/wiki/Prokaryote}.

Appendix A

Existing Systems

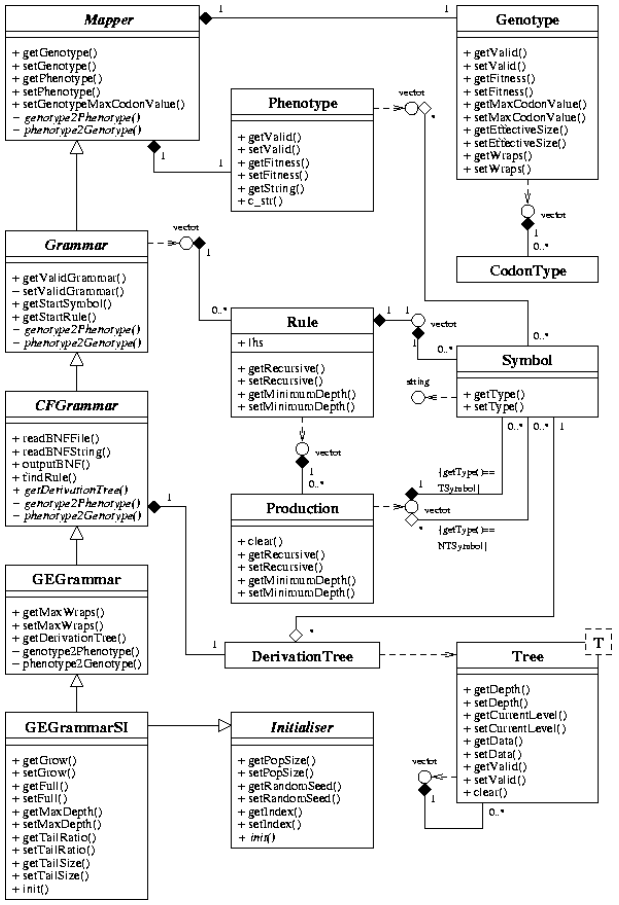

Figure A.1: Class hierarchy diagram for libGE

Appendix B

Requirements Elicitation and Analysis

B.1 Software Requirements Specification

General Concerns of the Framework

1. The framework will be capable of finding a program from a given search space over a BNF grammar that provides the optimal or closest feasible solution to a given problem.
2. The framework will use a Genetic Algorithm (GA) to search over the grammar.
3. The framework should provide an implementation of Grammatical Evolution (GE) that is easy to read and to understand
   (a) The code should adhere to a relevant standard (as highlighted in the Python Style Guide) to sure consistency and readability
4. There are no specific documentation requirements, however the implementation should make use of pydoc comments in order to aid readability.
5. The implementation should work with a standard Python interpreter, from version 2.3 upwards.
6. The framework should be written to minimise the burden of installation - for example, any Python modules that are used should either be included with a standard Python installation, or be simply packaged with the distribution of the framework.
7. The framework should be simple to adapt and extend in all aspects of its implementation.

8. The framework's performance should theoretically match that of GE, however due to the lack of information available regarding the configuration of GE used in papers, there is some leeway with regard to this.
9. No GUI is required, however the framework should be written in a way that does not prevent this.
10. The framework should be able to perform two standard GE tests "out of the box":
    (a) Symbolic Regression
    (b) Symbolic Integration
11. The framework should be easily configured from an external file.

Genetic Algorithm

1. The GA's population size can be specified by a user.
2. The length of genes within the binary string chromosome will be user definable.
3. The maximum number of generations that the GA will operate over will be user definable.
4. When a goal condition is reached, the GA and hence GE system will stop running.
5. When the maximum number of generations has been reached, the GA and hence GE system will stop running.
6. The GA must support a variable length genotype
7. A researcher must be able to specify an initial length of the genotype
8. The GA must be modular, and allow for easy alteration and customisation
9. The GA should at least support the following features:
    (a) One point crossover
    (b) Roulette Wheel selection
    (c) Steady state population update
10. The GA should support the GE genetic operators of duplication and pruning
11. The GA should be configurable through an external file

Code Generation

1. The framework should be able to convert an individual's binary string genotype to a phenotype with the use of a grammar definition
   (a) The binary string will be converted into integer codon values, used to decide which expansions of non-terminals in the grammar should used.
   (b) The grammar should have an easily identified start symbol in order to initiate the mapping process
2. The grammar should be in Backus-Naur Form, and stored in an external text file.
3. Genotype wrapping should be easily turned on and off
4. The system should include a heuristic to prevent infinite wrapping of the genotype.

Fitness Evaluation

1. The evaluation mechanism should be set up to interpret Python code by default.
2. The evaluation should support zero or more variables, each with one or more values.
   (a) These variables should be stored in an external text file .
3. The fitness evaluation mechanism should be modular, easy to adapt and replace.
4. By default, the fitness evaluation system should be set up for the standard GE tests of Symbolic Integration and Symbolic Regression "out of the box".

Constraints

1. Time - perhaps the greatest constraint on the project, the project must be handed in by 12pm on Monday the 8^{th} of May 2006. This means that the functionality and extent of the project will have a finite limit.
2. Finance - there shall be no financial support for the project, and as such, any tools to be used should be freely available.
3. Cluster computing facilities - the project will have the Department's 20 node Linux cluster at its disposal for the testing and experimentation of the parallel implementation of GE, therefore the configuration and design of the parallel code should take this into consideration.

4. Use of standard Python - the code should use a standard version of the Python interpreter, compatible with version 2.3 of Python in order to allow for use on the Department's cluster. This means that any special features added in version 2.4 of Python must be omitted from implementation.

5. Personnel - only one developer is available for the entire project, and hence, as per the Time constraint, this limits the scope and extent of the project.

6. Readability of code - one of the key aims of the project is to produce a GE implementation that is written in clear, simple to understand code. Hence, caution should be applied when selecting features from the requirements elicitation process so that they don't result in extraneous and overcomplicated code.

7. Cross-platform compatibility - The system shall be developed using Mac OS X 10.4, however it must be able to work on Linux in order for the parallel experimentation to be carried out on the Department's cluster.

Parallel Implementation

1. The parallel implementation should be a simple extension of GEPy.

2. The parallel implementation should be based around the island model of parallel GAs

3. The parallel implementation should allow for the alteration of settings and parameters with regard to parallelism

4. The parallel implementation should try and maintain platform agnosticism

B.2 Diagrams

Rather than provide all of the Event Scenario and Sequence diagrams that were used in the requirements elicitation process, a selection has been provided that indicate their use.

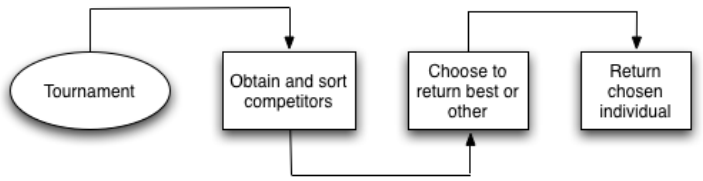

Figure B.1: Event Scenario diagram for `_tournament()` function

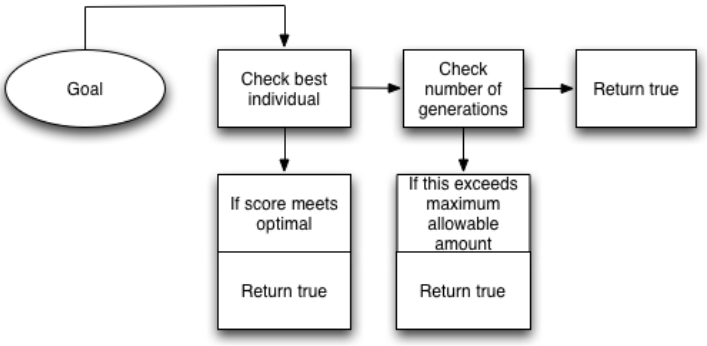

Figure B.2: Event Scenario diagram for `_goal()` function

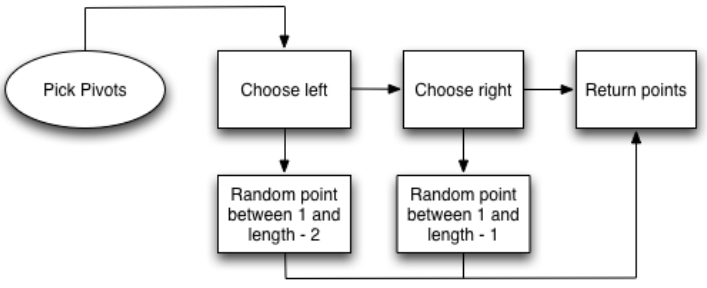

Figure B.3: Event Scenario diagram for `_pickpivots()` function

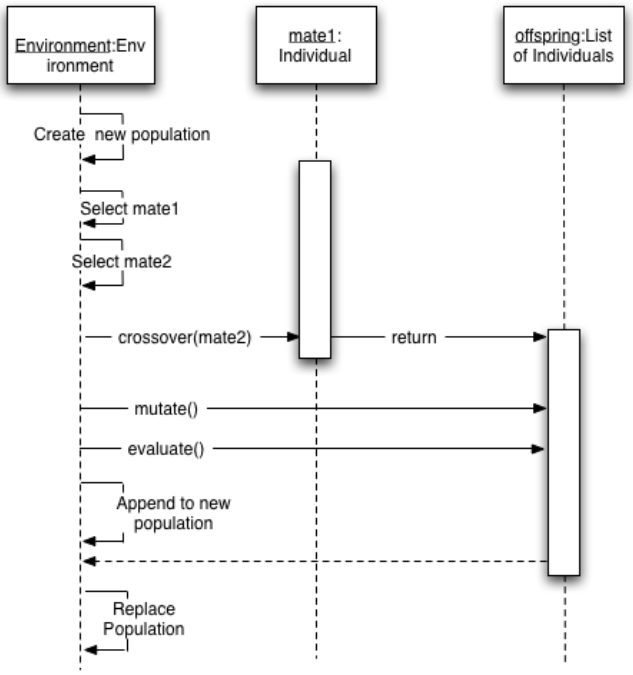

Figure B.4: Sequence diagram for `_crossover()` function

Appendix C

Testing

C.1 Test Plan

We will only list one formal test plan (for GEIndividual), as this will provide a sufficient overview of the structure and content required for them, and the plan can be simply derived from the content of the unit test code.

C.1.1 GEIndividual Test Plan

This test plan covers the functions that are possible to test via unit testing - i.e., those which do not require an integrated system (such as `_crossover()`).

`evaluate()` test

We should assert that calling `evaluate()` on a known chromosome will result in the correct code string being called, and the correct fitness being assigned to the individual.

Furthermore, we should assert that a chromosome that results in infinite wrapping will stop the mapping process due to the heuristic, assign the individual a suitably harsh score and append the individual's code string with the debug message.

`mutate()` test

We should test the functionality of this function by mutating a bit on a known chromosome, and comparing it to a version that is known to be the correct mutated version.

`duplicate()` test

We should test the functionality of `duplicate()` is working by first calling duplicate with a number of genes to duplicate of 0, and asserting the length of the genotype is the same. We should then check by calling `duplicate()` with a number of genes to duplicate of 1, and assert that the length of the genotype has increased by one gene.

`prune()` test

We should test the functionality of `prune()` by setting the individual's `gene_offset` to a known value, ensuring that it's instance variable `wrapped` is set to 0, and then calling `prune()`. We should then assert that the length of the genotype is now equal to the value of `gene_offset`

`get_codon()` test

We should first test that `get_codon()` works with wrapping enabled. We shall set the instance variable `gene_offset` to be a value that will cause wrapping to occur, and note the value that `get_codon()` should return. We shall then call `get_codon()` and assert that this returns the value that is equal to the correct value.

We shall than disable wrapping, and perform exactly the same test, however this time we shall check that `get_codon()` returns the value `None`.

`copy()` test

We should assert that the Individual object returned by calling `copy()` is equal to the individual itself.

`to_decimal()` test

We should assert that `to_decimal()` returns the correct value for a given binary string.

C.2 Unit Tests

As the source code for the unit tests is available on the CD attached to this document, we will only list the source code for one unit test (for GEIndividual), as this will provide a sufficient overview of the implementation of all of them.

```
import unittest
import gepy
from fixedpoint import *
from fpconst import *

class GEIndividualTest(unittest.TestCase):
        pop_size = 10
        gene_offset = 100
        gramfile="test_files/gram.bnf"
        varfile="test_files/input.var"
        grammar = gepy.grammar.Grammar(gramfile)
        vars_file_handle = open(varfile)
        vars_file_contents = vars_file_handle.read()
        vars = eval(vars_file_contents)

        chromosome = [0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1,
            1, 0, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0,
            1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0,
            0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1,
            0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0,
            1, 0, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 1,
            0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0,
            0, 0, 1, 1, 1, 0]

        def setUp(self):
        # self.testGrammar = gepy.grammar.Grammar(self.gramFile)
                self.ind = gepy.GEIndividual(chromosome=self.chromosome,\
                        grammar=self.grammar,vars=self.vars)
```

```
29
30          def tearDown(self):
31                  self.ind = None
32
33          def test_evaluate(self):
34                  self.ind.evaluate()
35                  self.assertEqual(self.ind.code_string,"val=X*X+(math.sin(X)+X)")
36                  self.assertEqual(FixedPoint(self.ind.score,10),0.0)
37
38                  self.ind.code_string = ""
39                  self.ind.chromosome = [0 for i in xrange(self.ind.length)]
40                  self.ind.evaluate()
41                  self.assertEqual(self.ind.code_string,"# INVALID INDIVIDUAL (recursion)")
42                  self.assertEqual(self.ind.score,PosInf)
43
44          def test_mutate(self):
45                  chromosome_copy = self.ind.chromosome
46                  chromosome_copy[0] = 1
47
48                  self.ind.mutate(0)
49                  self.assertEqual(self.ind.chromosome,chromosome_copy)
50
51          def test_duplicate(self):
52                  pre_length = self.ind.length
53                  self.ind.duplicate(0)
54                  self.assertEqual(pre_length,self.ind.length)
55
56                  self.ind.duplicate(1)
```

```
57                  self.assertEqual(pre_length+self.ind.gene_length,self.ind.length)
58
59          def test_prune(self):
60                  self.ind.wrapped = 0
61                  self.ind.gene_offset = self.gene_offset
62
63                  pre_length = self.ind.length
64
65                  self.ind.prune()
66
67                  self.assertEqual(self.gene_offset,self.ind.length)
68
69          def test_get_codon(self):
70                  self.ind.wrapping_enabled = 1
71                  self.ind.gene_offset = self.ind.length - self.ind.gene_length + 3
72                  self.assertEqual(self.ind.get_codon(),46)
73
74                  self.ind.wrapping_enabled = 0
75                  self.ind.gene_offset = self.ind.length - self.ind.gene_length + 3
76                  self.assertEqual(self.ind.get_codon(),None)
77
78          def test_copy(self):
79                  self.assertEqual(self.ind.copy(),self.ind)
80
81          def test_to_decimal(self):
82                  self.assertEqual(self.ind.to_decimal(self.ind.chromosome),203792549487346033337815435125)
83  if __name__ == "__main__":
84      unittest.main()
```

Appendix D

Experimental Results

D.1 Graphs

The graphs in this section show the plots of average execution time in seconds (Y-axis) against number of processors (X-axis) for varying migration rates, complete with error bars.

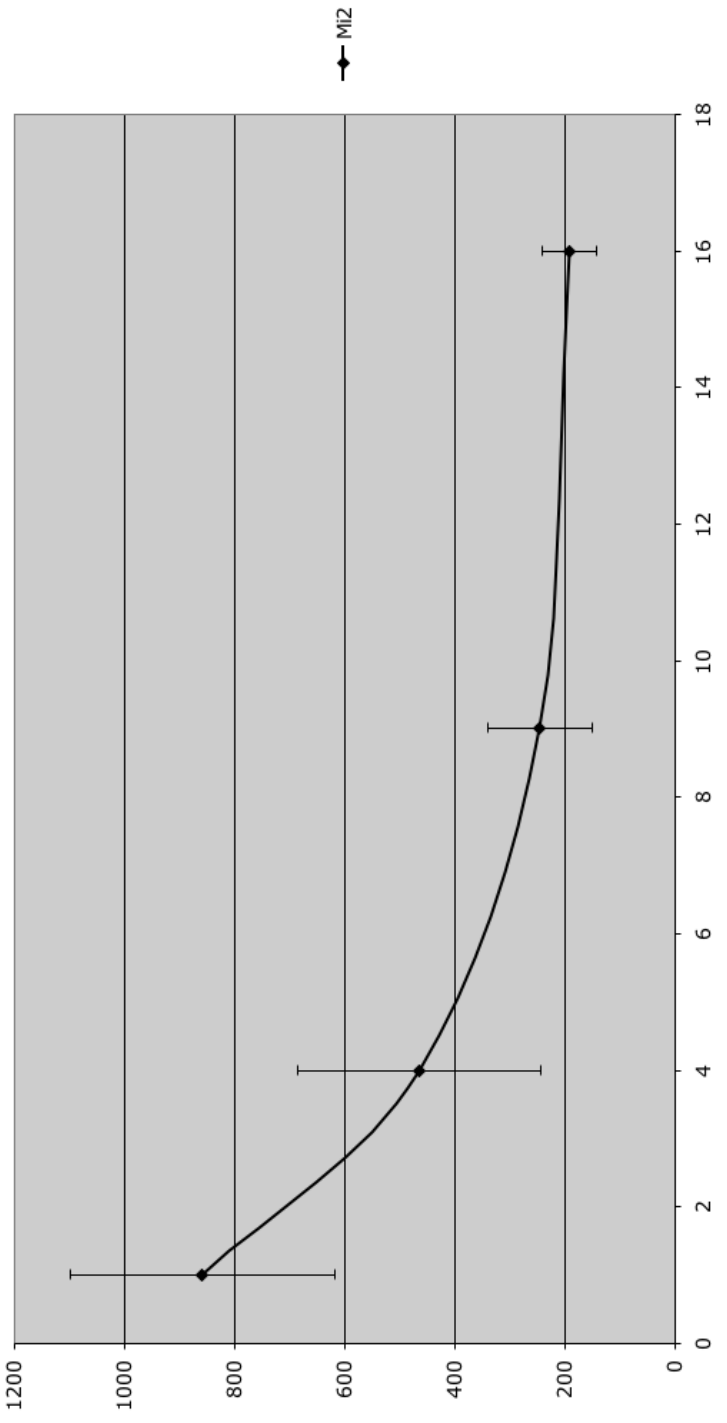

Figure D.1: Migration Interval of 2

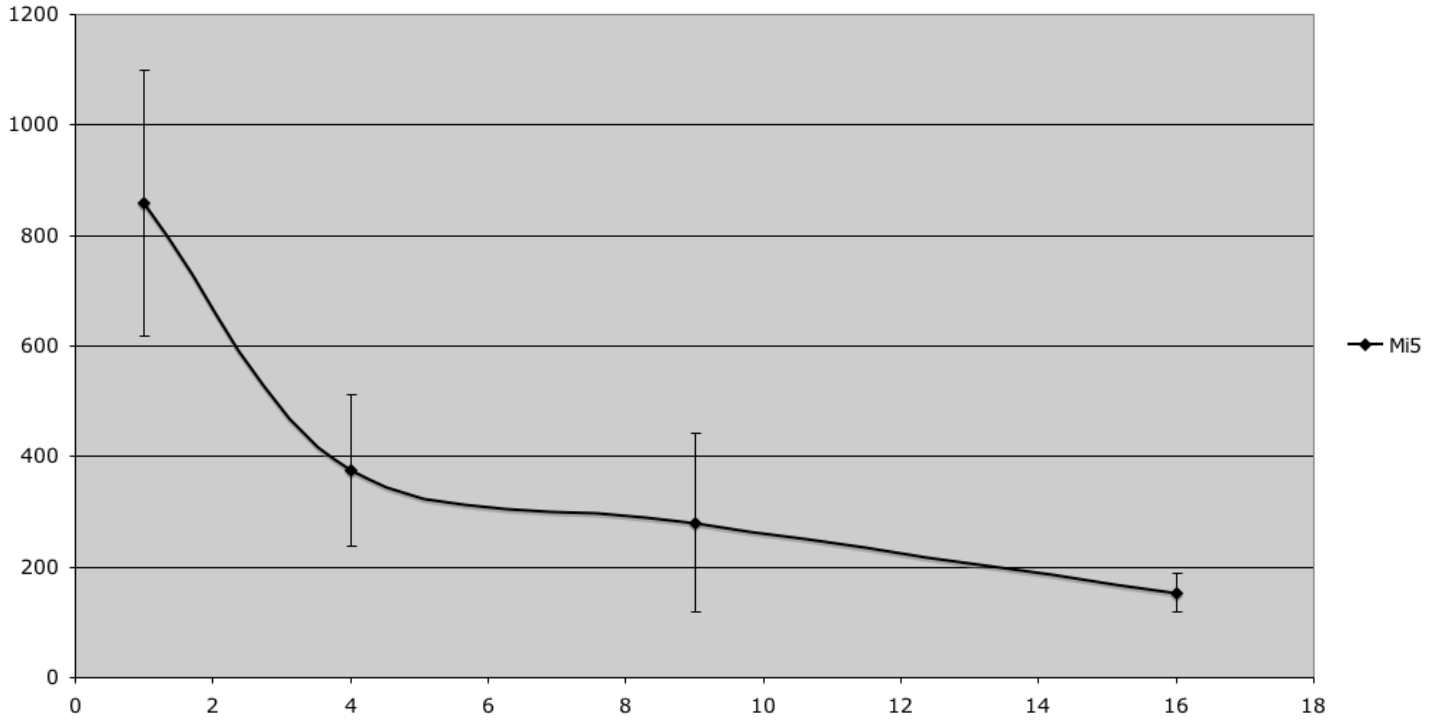

Figure D.2: Migration Interval of 5

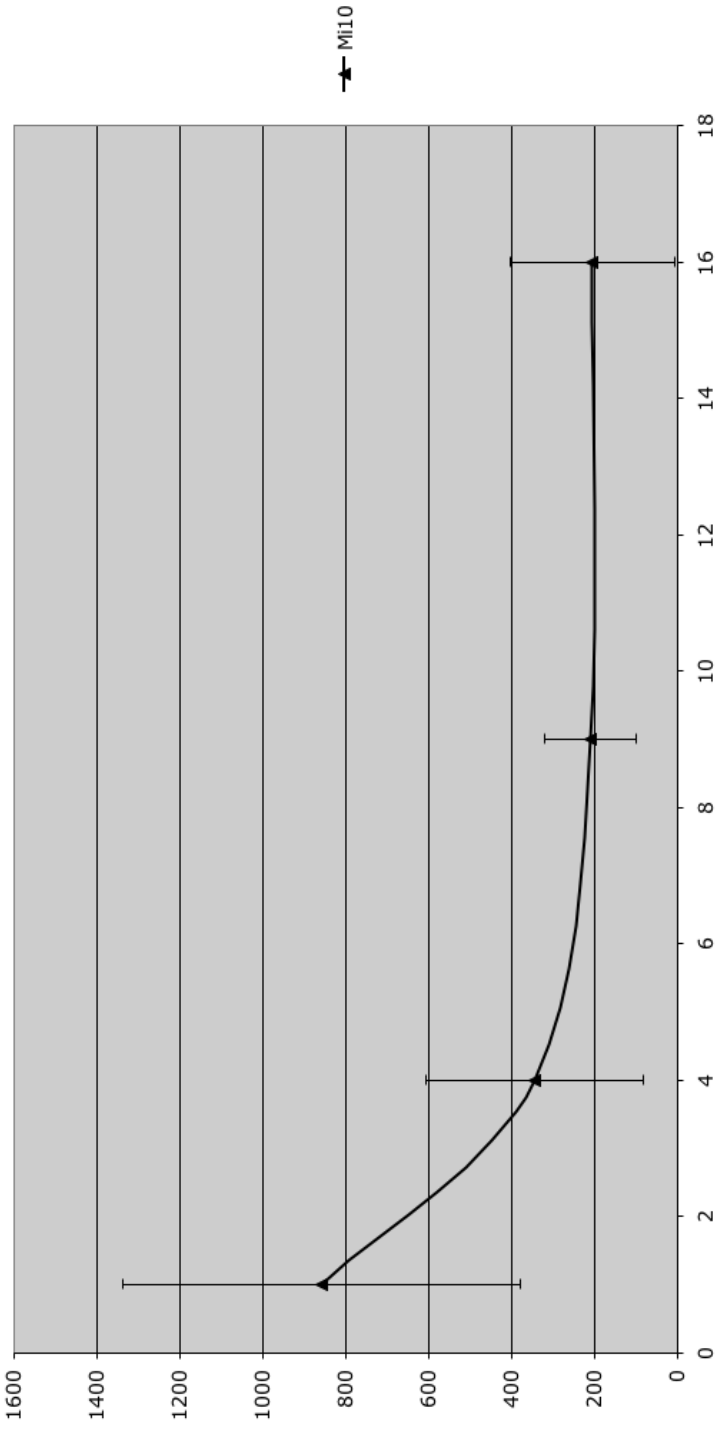

Figure D.3: Migration Interval of 10

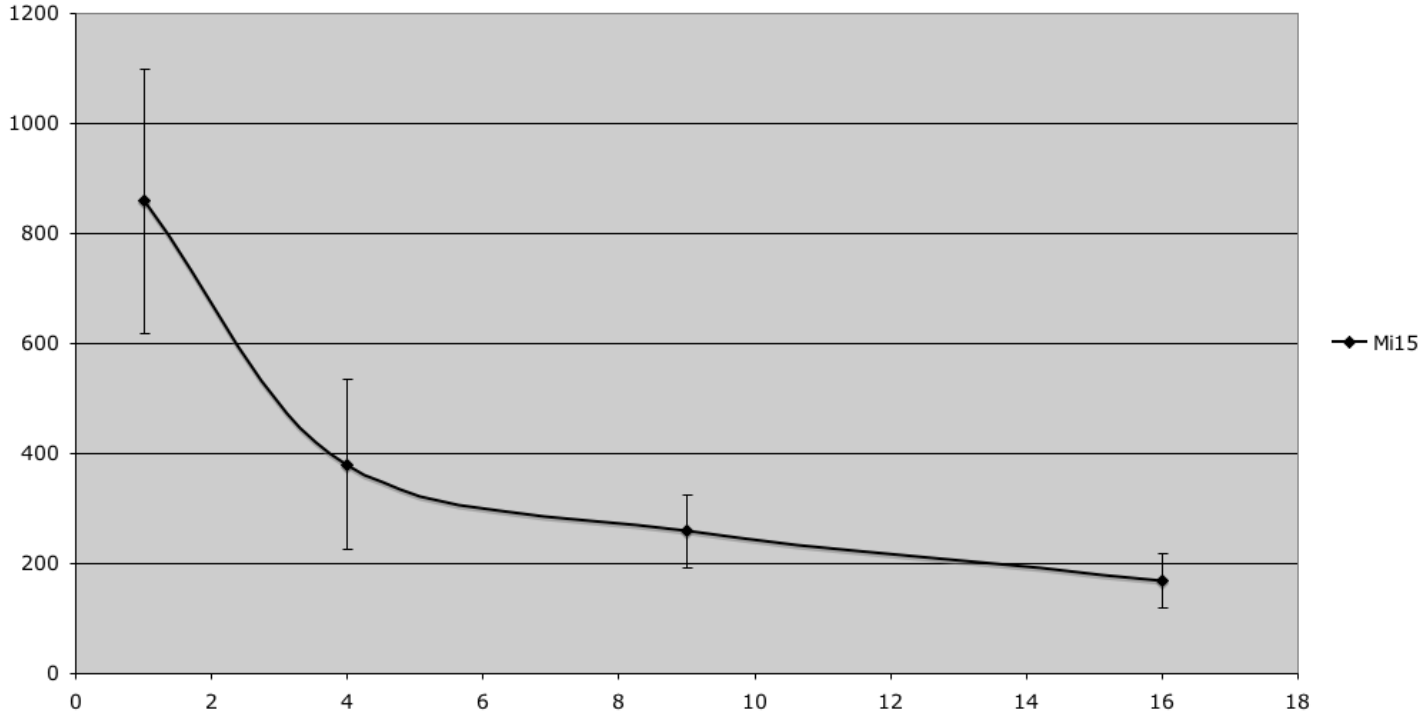

Figure D.4: Migration Interval of 15

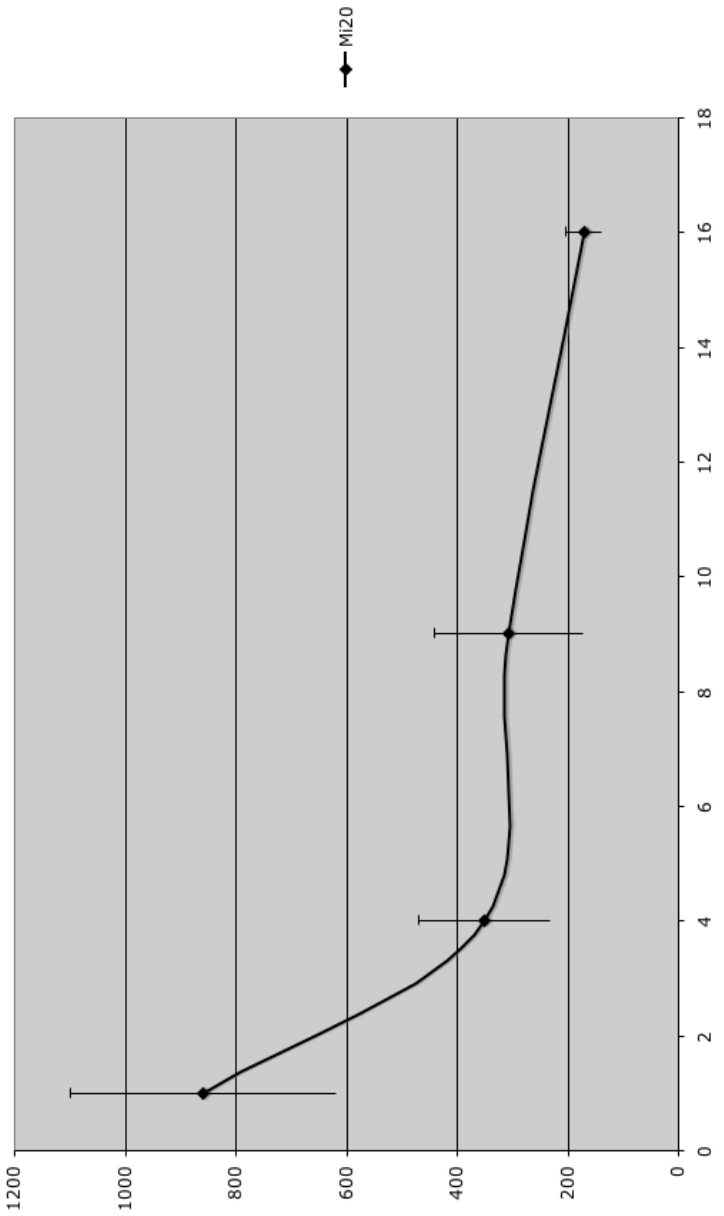

Figure D.5: Migration Interval of 20

Appendix E

Source Code

E.1 GEPY

As the source code is available on the CD provided with this document, we will only list the main code for GEIndividual and GEEnvironment here.

```
1  import pyga
2  import grammar
3  import config_handler
4
5  import sys
6  import math # make dynamic ?
7  import random
8  from fpconst import * # needed for nan detection
9  from xml.sax import make_parser
10 import pickle
11
12 INVALID = PosInf
13 RECURSION_LIMIT = sys.getrecursionlimit() - 20
14
15 class GEIndividual(pyga.Individual):
16         optimization = pyga.MINIMIZE
17         code_string = "val="
18         gene_offset = 0
19         gene_length = 8
20         adjusted_score = 0
21         wrapping_enabled = 1
22         wrapped = 0
23         chromosome = None
24         start_symbol = None
25         _grammar = None
26
27         def __init__(self, chromosome=None,vars=None,grammar=None,\
28                         gene_offset = None,wrapping_enabled=1):
```

```
29                "Constructor-like function for GEIndividual"
30                if grammar != None:
31                        self._grammar = grammar
32                        self.start_symbol = self._grammar.get_start_symbol()
33                else:
34                        print "Error - missing grammar??"
35                        #sys.exit()
36                if vars != None:
37                        self._vars = vars
38                else:
39                        print "Error - missing vars??"
40                        #sys.exit()
41                if gene_offset != None:
42                        self.gene_offset = gene_offset
43                self.wrapping_enabled = wrapping_enabled
44
45                super(GEIndividual,self).__init__(chromosome)
46
47        def evaluate(self, optimum=None):
48                "calls the functions to map our genotype, and then \
49                evaluate the resultant code"
50                self.init_mapper()
51                self.run_code()
52
53        def _onepoint(self,other):
54                "Performs one point crossover using this GEIndividual\
55                 and another, which is passed as a parameter"
56                pivot = random.randrange(1,min(self.length,other.length)-1)
```

```
57                  def mate(p0,p1):
58                          chromosome = p0.chromosome[:]
59                          chromosome[pivot:] = p1.chromosome[pivot:]
60                          child = p0.__class__(chromosome,grammar=self._grammar,\
61                                          vars=self._vars,\
62                                          wrapping_enabled=self.wrapping_enabled)
63                          return child
64                  return mate(self,other), mate(other,self)
65
66          # sample crossover method
67          def _twopoint(self, other):
68                  "creates offspring via two-point crossover between mates.\
69                  Overridden to copy grammar and vars."
70                  left, right = self._pickpivots()
71                  def mate(p0, p1):
72                          chromosome = p0.chromosome[:]
73                          chromosome[left:right] = p1.chromosome[left:right]
74                          child = p0.__class__(chromosome,grammar=self._grammar,\
75                                  vars=self._vars,wrapping_enabled=self.wrapping_enabled)
76                          child._repair(p0, p1)
77                          return child
78                  return mate(self, other), mate(other, self)
79
80          def mutate(self, bit):
81                  "replacement mutation function that corrects a bug in PyGA"
82                  self.chromosome[bit] = abs(self.chromosome[bit]-1) # bit flip
83
84          def noisy_duplicate(self,number_to_duplicate):
```

```
85                "Duplicates number_to_duplicate genes, NOT gene aligned, and\
86                 appends to the end of the chromosome"
87                print "noisy duplicating", number_to_duplicate, "genes..."
88                for i in xrange(number_to_duplicate):
89                        random_offset = random.randint(0,(\
90                                self.length - self.gene_length - 1))
91                        self.chromosome = self.chromosome +\
92                                 self.chromosome[random_offset:random_offset+self.gene_length]
93                print "\tnew length: ", self.length
94
95
96        def duplicate(self,number_to_duplicate):
97                "Duplicates number_to_duplicate genes and appends to the\
98                 end of the chromosome"
99                for i in xrange(number_to_duplicate):
100                       random_offset = self.gene_length*random.randint(0,\
101                               (self.length/self.gene_length) - self.gene_length)
102                       self.chromosome = self.chromosome + \
103                       self.chromosome[random_offset:random_offset+self.gene_length]
104
105       def prune(self):
106               "Deletes all introns from the chromosome"
107               if self.gene_offset != 0: # avoid erasing the entire genotype
108                       del(self.chromosome[self.gene_offset:])
109
110       def init_mapper(self):
111               "obtains start symbol and handle_expansion on it"
112               start = self.start_symbol
```

```
113                 self.handle_expansion(start,0)
114
115         def run_code(self):
116                 "runs generated code, and sets the appropriate fitness value.\
117                 You will probably wish to modify or override this function to suit your needs."
118                 try:
119                         total = 0
120                         for current_var_dict in self._vars:
121                                 for current_var in current_var_dict.keys():
122                                         locals()[current_var] = current_var_dict[current_var]
123
124                                 val = 0
125                                 if self.score != INVALID:
126                                         exec(self.code_string)
127                                         if isNaN(val):
128                                                 self.score = INVALID
129                                                 self.code_string += " # INVALID INDIVIDUAL (NaN)"
130                                         else:
131                                                 partial = abs(ans-val)
132                                                 total = total + partial
133
134                                 else:
135                                         total = INVALID
136                                         break
137                         self.score = total
138
139                 except:
140                         self.score = INVALID
```

```
141
142
143         def handle_expansion(self,expansion,depth):
144                 "Chooses an appropriate expansion of a non-terminal based upon the genotype"
145                 if depth > RECURSION_LIMIT:
146                         self.score = INVALID
147                         self.code_string = "# INVALID INDIVIDUAL (recursion)"
148                         return -1
149
150                 if expansion.startswith("<") and expansion.endswith(">"):
151                         newExpansion = self._grammar.get_expansions(expansion)
152                         codon_value = self.get_codon()
153                         if codon_value == None:
154                                 self.score = INVALID
155                                 self.code_string = "# INVALID INDIVIDUAL (wrapping)"
156                                 return -1
157                         else:
158                                 select = codon_value % len(newExpansion)
159                         for currentExpansion in newExpansion[select]:
160                                 if(self.handle_expansion(currentExpansion,depth+1) == -1):
161                                         return -1 # i.e., we've recursed past 500 levels
162                         return 0
163                 else:
164                         self.code_string = self.code_string + expansion
165                         return 0
166
167         def to_decimal(self,x):
168                 "converts a binary string to a decimal integer value, by Seb Bacon, \
```

```
169                 http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/219300"
170                 return sum(map(lambda z: int(x[z]) and 2**(len(x) - z - 1),\
171                 xrange(len(x)-1, -1, -1)))
172
173         def get_codon(self):
174                 "returns a codon value, or the value None if wrapping is \
175                 disabled and the end of the genome is reached"
176                 overflowCheck = self.gene_offset + self.gene_length - 1
177                 if overflowCheck > self.length:
178                         if self.wrapping_enabled == 1:
179                                 remainingAlleles = self.chromosome[self.gene_offset:]
180                                 wrapAlleles = self.chromosome[0:(self.gene_length - len(remainingAlleles))]
181                                 codonValue = self.to_decimal(wrapAlleles + remainingAlleles)
182
183                                 self.gene_offset = self.gene_length - len(remainingAlleles)
184                                 self.wrapped = 1
185                         else:
186                                 return None # if wrapping is not enabled, return None to indicate an error..
187                 else:
188                         codonValue = self.to_decimal(self.chromosome[self.gene_offset:\
189                                 (self.gene_offset+self.gene_length-1)])
190                         self.gene_offset = self.gene_offset + self.gene_length
191
192                 return codonValue
193
194
195         def copy(self):
196                 "Overridden to copy the code string, grammar and vars as\
```

```
197                  well as the chromosome, etc. We also copy gene_offset for efficieny."
198                 twin = self.__class__(self.chromosome[:],grammar=self._grammar,\
199                 vars=self._vars,gene_offset=self.gene_offset,wrapping_enabled=self.wrapping_enabled)
200                 twin.score = self.score
201                 twin.code_string = self.code_string
202                 return twin
203
204 class GEEnvironment(pyga.Environment):
205         total_score = 0
206         grammar = None
207         _vars = None
208         start_symbol = None
209         optimization = None
210         duplication_rate = 0.0
211         duplication_range = 0
212         prune_rate = 0.0
213
214         def __init__(self,kind, population=None, size=100, maxgenerations=100, \
215                 crossover_rate=0.90, mutation_rate=0.01, duplication_rate=0.01, \
216                 max_genes_duplicated = 1, prune_rate = 0.0, \
217                 optimum=None,gramfile=None,varfile=None,wrapping_enabled=1):
218                 "Constructor-like function for GEEnvironment"
219                 if gramfile != None:
220                         self._grammar = grammar.Grammar(gramfile)
221                 else:
222                         print "Error - no grammar file specified!"
223                         sys.exit()
224
```

```
225                 if varfile != None:
226                         try:
227                                 vars_file_handle = open(varfile)
228                                 vars_file_contents = vars_file_handle.read()
229                         except IOError:
230                                 print "Error reading variables file ", varfile, " !"
231                                 sys.exit()
232                         try:
233                                 self._vars = eval(vars_file_contents)
234
235                         except:
236                                 print "There was an error parsing the contents of your variables file.\
237                                  Please check and try again."
238                                 sys.exit()
239                         # we need to set up the GE specific operators...
240                         self.duplication_rate = duplication_rate
241                         self.duplication_range = max_genes_duplicated
242                         self.prune_rate = prune_rate
243                         self.wrapping_enabled = wrapping_enabled
244                 else:
245                         print "Error - no variables file specified!"
246                         sys.exit()
247
248
249
250                 # TODO: MOVE TO INDIVIDUAL?!? WHAT?!?
251                 self.optimization = kind.optimization
252
```

```
253                 if self.optimization == pyga.MINIMIZE:
254                         globals()['INVALID'] = PosInf
255                 else:
256                         globas()['INVALID'] = 0
257 
258                 super(GEEnvironment, self).__init__(kind,population,size,maxgenerations,\
259                         crossover_rate,mutation_rate,optimum)
260 
261 
262         def _makepopulation(self):
263                 "overrridden to support the use of GEIndividuals. Also contains a subclass sanity check"
264                 if issubclass(self.kind,GEIndividual):
265                         return [self.kind(grammar = self._grammar,  \
266                                 vars=self._vars,wrapping_enabled=self.wrapping_enabled) for individual in xrange(s
267                 else:
268                         print "You did not supply an individual that is a subclass of GEIndividual, sorry!"
269                         sys.exit()
270 
271 
272         def report(self):
273                 "Reports basic information on the progress of a GE run"
274                 print "="*70
275                 print "generation: ", self.generation
276                 print "best:       ", self.best
277                 print "code:       ", self.best.code_string
278                 print "length diff: ", self.best.length - self.best.init_length
279 
280 
```

```
281         def _select(self):
282                 "Abstraction function to allow for choice of selection method. \
283                 Default is Roulette Wheel."
284                 return self._roulette()
285 
286         def _roulette(self):
287                 "Provides a simple Roulette Wheel selection method."
288                 partial_sum = 0.0
289                 index = 0
290                 if self.optimization == pyga.MAXIMIZE:
291                         self._get_total_score()
292                 else:
293                         self._get_inverse_total_score()
294 
295                 choice = random.random() * self.total_score
296 
297                 while index < self.size:
298                         if self.optimization == pyga.MAXIMIZE:
299                                 partial_sum += self.population[index].score
300                         else:
301                                 partial_sum += 1/self.population[index].score
302 
303                         if partial_sum >= choice:
304                                 return self.population[index]
305                         index += 1
306 
307 
308         def _crossover(self):
```

```
309                 "Abstraction function to allow choice of population update method. \
310                 Default is Steady State."
311                 return self._crossover_steadystate()
312
313         def _crossover_steadystate(self,num_replaced=5):
314                 "Provides a Steady State updated method"
315                 next_population = self.population[0:(len(self.population)-1)-num_replaced]
316
317                 while len(next_population) < self.size:
318                         mate1 = self._select()
319                         if random.random() < self.crossover_rate:
320                                 mate2 = self._select()
321                                 offspring = mate1.crossover(mate2)
322                         else:
323                                 offspring = [mate1.copy()]
324                         for individual in offspring:
325                                 self._mutate(individual)
326                                 self._prune(individual)
327                                 self._duplicate(individual)
328                                 individual.evaluate(self.optimum)
329                                 next_population.append(individual)
330                 self.population = next_population[:self.size]
331
332         def _crossover_elitist(self):
333                 "Provides an Elitist population update method"
334                 next_population = [self.best.copy()]
335                 while len(next_population) < self.size:
336                         mate1 = self._select()
```

```
                        if random.random() < self.crossover_rate:
                                mate2 = self._select()
                                offspring = mate1.crossover(mate2)
                        else:
                                offspring = [mate1.copy()]
                        for individual in offspring:
                                self._mutate(individual)
                                self._prune(individual)
                                self._duplicate(individual)
                                individual.evaluate(self.optimum)
                                next_population.append(individual)
                self.population = next_population[:self.size]

        def _duplicate(self,individual):
                "Decides whether an individual should have duplication applied to it."
                if random.random() < self.duplication_rate:
                        individual.duplicate(random.randint(1,self.duplication_range))


        def _prune(self,individual):
                "Decides whether an individual should have pruning applied to it."
                if random.random() < self.prune_rate:
                        if individual.gene_offset == 0: # we need to pre-evaluate to \
                                                        # get a gene offset to prune
                                individual.evaluate(self.optimum)
                        if individual.wrapped == 0:
                                individual.prune()

```

```
365         def _get_total_score(self):
366                 "returns the sum of the scores of individuals in the population"
367                 self.total_score = 0.0
368                 for i in self.population:
369                                 self.total_score = self.total_score + i.score
370
371         def _get_inverse_total_score(self):
372                 "returns the sum of (the scores of the individuals)^-1 in the population"
373                 self.total_score = 0.0
374                 for i in self.population:
375                         self.total_score = self.total_score + (1/i.score)
376
377
378 def init_GE(configfile,environment,individual):
379         "Provides a simple way of starting a GE run."
380         parser = make_parser()
381         handler = config_handler.ConfigHandler()
382         parser.setContentHandler(handler)
383         parser.parse(open(configfile))
384         handler.validate_vars()
385
386         env = environment(individual,maxgenerations=handler.maxgen,optimum=handler.optimum,\
387                 size=handler.size,crossover_rate=handler.crossover_rate,mutation_rate=handler.mutation_rate,\
388                 gramfile=handler.grammar,varfile=handler.variables,duplication_rate=handler.duplication_rate,\
389                 prune_rate=handler.prune_rate,max_genes_duplicated=handler.max_genes_duplicated,wrapping_enabled=h
390         env.run()
```

E.2 Parallel Implementation

E.2.1 mpidriver.py

This is the code that was run on the islands in the mesh, providing a parallel version of GEPy.

```
1  import sys
2  import gepy
3  import mpi
4  from Numeric import *
5  from fixedpoint import FixedPoint
6  import sys
7
8  # Tags for MPI messages
9  MIGRANT = 0
10 BEST = 1
11 HALT = 2
12 MAXGEN = 3
13
14 # The rank of the master node
15 mpi_root = 0
16
17 class MPIGEEnvironment(gepy.GEEnvironment):
18         myid = None
19         num_nodes = None
20         neighbours = []
21
22         # Key parameters for experimental work:w
23         width =  4
24         migration_interval = 20
25
26         def __init__(self,kind, population=None, size=100, maxgenerations=100,\
27                 crossover_rate=0.90, mutation_rate=0.01, duplication_rate=0.01,\
28                 max_genes_duplicated = 1, prune_rate = 0.0, optimum=0.0,\
```

```
29                  gramfile=None,varfile=None,wrapping_enabled=1):
30                  self.myid = mpi.mpi_comm_rank(mpi.MPI_COMM_WORLD)
31                  self.num_nodes = mpi.mpi_comm_size(mpi.MPI_COMM_WORLD)
32                  self.neighbours = self.find_neighbours(self.myid,self.width)
33
34                  super(MPIGEEnvironment,self).__init__(kind,population,size,\
35                          maxgenerations,crossover_rate,mutation_rate,\
36                          duplication_rate,max_genes_duplicated,prune_rate,\
37                          optimum,gramfile,varfile,wrapping_enabled=wrapping_enabled)
38
39          def run(self):
40                  "Provides functionality for run to handle incoming HALT and \
41                  MAXGEN messages, and to send BEST and MAXGEN messages when appropriate."
42                  best_found = False
43
44                  while 1:
45                          if(mpi.mpi_iprobe(mpi.MPI_ANY_SOURCE,HALT,\
46                                  mpi.MPI_COMM_WORLD)): # Halt if necessary
47
48                                  mpi.mpi_finalize()
49                                  sys.exit()
50                          if(mpi.mpi_iprobe(mpi.MPI_ANY_SOURCE,MAXGEN,\
51                                  mpi.MPI_COMM_WORLD)): # remove any halted neighbours
52
53                                  while(mpi.mpi_iprobe(mpi.MPI_ANY_SOURCE,\
54                                          MAXGEN,mpi.MPI_COMM_WORLD)):
55
56                                          i = mpi.mpi_recv(1,mpi.MPI_INT,\
```

```
57                                      mpi.MPI_ANY_SOURCE,MAXGEN,\
58                                      mpi.MPI_COMM_WORLD)
59                              self.neighbours = \
60                                      [n for n in self.neighbours if n != i]
61
62              if not best_found:
63                      if self._goal():
64                              mpi.mpi_send(self.best.chromosome,\
65                                      len(self.best.chromosome),\
66                                      mpi.MPI_INT,mpi_root,BEST\
67                                      ,mpi.MPI_COMM_WORLD)    # Notify the master
68                                                              # we have met the goal condition
69
70                              best_found = True
71                      elif self.generation > self.maxgenerations:
72                              mpi.mpi_send(self.myid,1,mpi.MPI_INT,\
73                                      mpi_root,MAXGEN,mpi.MPI_COMM_WORLD) # notify the master node
74                                                                          # we have exeeded maxgens
75
76                              for i in self.neighbours: # notify the neighbours
77                                                        # so they can stop sending to us
78                                      mpi.mpi_send(self.myid,1,\
79                                      mpi.MPI_INT,i,MAXGEN,mpi.MPI_COMM_WORLD)
80
81                              mpi.mpi_finalize()
82                              sys.exit()
83                      else:
84                              self.step()
```

```
85
86
87          def step(self):
88                  "Adds migratory functions to the standard step function"
89                  self.population.sort()
90                  self._crossover()
91                  if self.generation % self.migration_interval == 0 and \
92                          self.generation != 0:
93                          self.emigrate()
94                          if(mpi.mpi_iprobe(mpi.MPI_ANY_SOURCE,MIGRANT,\
95                                  mpi.MPI_COMM_WORLD)): # any immigrants to deal with?
96
97                                  while(mpi.mpi_iprobe(mpi.MPI_ANY_SOURCE,\
98                                          MIGRANT,mpi.MPI_COMM_WORLD)):
99
100                                         if(mpi.mpi_iprobe(mpi.MPI_ANY_SOURCE,\
101                                                 HALT,mpi.MPI_COMM_WORLD)): # avoid race condition
102
103                                                 mpi.mpi_finalize()
104                                                 sys.exit()
105                                         else:
106                                                 count = mpi.mpi_get_count(mpi.MPI_INT)
107                                                 immigrant = mpi.mpi_recv(count,\
108                                                                 mpi.MPI_INT,mpi.MPI_ANY_SOURCE,\
109                                                                 MIGRANT,mpi.MPI_COMM_WORLD) # import them
110                                                                                 # into population
111                                                 self.immigrate(immigrant)
112
```

```
113                 self.generation += 1
114                 self.report()
115
116         def _emigration_select(self):
117                 "Abstraction function to allow for choice of selection method for emigration.\
118                  Defaults to tournament"
119                 return [self._tournament()]
120
121         def _select(self):
122                 "We prefer to use Tournament selection"
123                 return self._tournament()
124
125         def _crossover(self):
126                 "We prefer to use Elitist update"
127                 return self._crossover_elitist()
128
129         def report(self):
130                 "We do not want any mess on STDOUT for parallel runs"
131                 self.population.sort()
132
133         def _immigrate_select(self):
134                 return _worst()
135
136         def _worst(self):
137                 self.population.sort()
138                 self.population.reverse()
139                 return self.population[1:]
140
```

```
141         def immigrate(self,immigrant_chromosome):
142                 self.population = self._immigrate_select()
143                 fixed_chromosome = [i for i in immigrant_chromosome]
144                 new_immigrant = self.kind(term_dict = self.grammar.termDict, \
145                         start_symbol = self.grammar.startSymbol,\
146                         vars=self._vars,chromosome=fixed_chromosome)
147                 new_immigrant.evaluate()
148                 self.population = [new_immigrant] + self.population
149
150         def emigrate(self):
151                 "Selects a number of individuals to emmigrate, and transmits them to neighbours"
152                 migrants = self._emigration_select()
153
154                 for p in migrants:
155                         migrant_chromosome = p.chromosome
156                         for i in self.neighbours:
157                                 if i != 0 or  i != -1:
158                                         mpi.mpi_send(migrant_chromosome,len(migrant_chromosome),\
159                                                 mpi.MPI_INT,i,MIGRANT,mpi.MPI_COMM_WORLD)
160
161         def find_neighbours(self,n,width):
162                 "Calculates the neighbours for a given node n in a nxn 2-D toroidal mesh"
163                 num = width**2
164
165                 north = (n + num - width) % num
166                 if north == 0:
167                          north = num
168                 south = (n + num + width) % num
```

```
169                 if south == 0:
170                         south = num
171                 if n % width == 1:
172                         west = n + (width - 1)
173                 else:
174                         west = n - 1
175
176                 if n % width == 0:
177                         east = n - (width - 1)
178                 else:
179                         east = n + 1
180
181                 neighbours = north,east,south,west
182                 return self.uniq(neighbours)
183
184         def uniq(self,a):
185                 "Given a list, removes duplicate entries"
186                 temp_dict = {}
187                 for i in a:
188                         temp_dict[i] = 1
189                 return temp_dict.keys()
190
191         def _goal(self):
192                 "We have removed maxgen checking, and added checking of the goal to 10 DP"
193                 if FixedPoint(self.best.score,10) == 0.0:
194                         return True
195                 else:
196                         return False
```

```
197
198
199
200 class MPIGEIndividual(gepy.GEIndividual):
201         wrapping_enabled = 1
202         def __init__(self, chromosome=None,vars=None,start_symbol=None,term_dict=None,gene_offset = None,\
203                         wrapping_enabled=1):
204                 super(MPIGEIndividual,self).__init__(chromosome,vars,start_symbol,term_dict,gene_offset\
205                         wrapping_enabled=wrapping_enabled)
206
207         def _crossover(self,other):
208                 "We prefer one point crossover"
209                 return self._onepoint(other)
210
211
212 ##############################################
213
214
215 configfile = "/labshare/gepy/mpi_config.xml"
216
217 if __name__ == "__main__":
218         sys.argv = mpi.mpi_init(len(sys.argv),sys.argv)
219         gepy.init_GE(configfile,MPIGEEnvironment,MPIGEIndividual)
```

E.2.2 master.py

This is the code that provided the functionality of the master node in the mesh.

```
1  import mpi
2  import sys
3  import time
4  import random
5  #from Numeric import array
6
7  # Tags for MPI messages
8  MIGRANT = 0
9  BEST = 1
10 HALT = 2
11 MAXGEN = 3
12
13 mpi_root = 0
14
15 sys.argv = mpi.mpi_init(len(sys.argv),sys.argv)
16
17 myid=mpi.mpi_comm_rank(mpi.MPI_COMM_WORLD)
18 numnodes=mpi.mpi_comm_size(mpi.MPI_COMM_WORLD)
19
20
21 mpi_root = 0
22 num_exceeded = 0
23
24 if myid == mpi_root:
25         print "root here"
26         while 1:
27                 if(mpi.mpi_iprobe(mpi.MPI_ANY_SOURCE,BEST,mpi.MPI_COMM_WORLD)):
28                         count = mpi.mpi_get_count(mpi.MPI_INT)
```

```
29                          i = mpi.mpi_recv(count,mpi.MPI_INT,mpi.MPI_ANY_SOURCE,BEST,mpi.MPI_COMM_WORLD)
30                          print "Found best!", i
31                          for host in xrange(1,numnodes):
32                                  mpi.mpi_send(0,1,mpi.MPI_INT,host,HALT,mpi.MPI_COMM_WORLD)
33
34                          mpi.mpi_finalize()
35                          sys.exit()
36
37                  if(mpi.mpi_iprobe(mpi.MPI_ANY_SOURCE,MAXGEN,mpi.MPI_COMM_WORLD)):
38                          count = mpi.mpi_get_count(mpi.MPI_INT)
39                          i = mpi.mpi_recv(count,mpi.MPI_INT,mpi.MPI_ANY_SOURCE,MAXGEN,mpi.MPI_COMM_WORLD)
40                          num_exceeded = num_exceeded + 1
41
42                  if num_exceeded == (numnodes - 1):
43                          print "All nodes are have exceeded maxgens - exiting!"
44                          mpi.mpi_finalize()
45                          sys.exit()
46
47
48  else:
49          print "Error - not running on root node!"
```