Grammar-Based Object-Oriented Genetic Programming

An Initial Implementation

By

Yandu Oppacher

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of
Master of Computer Science

Ottawa-Carleton Institute for Computer Science
School of Computer Science
Carleton University
Ottawa, Ontario

15 September 2008

Your file Votre référence
ISBN: 978-0-494-44110-7
Our file Notre référence
ISBN: 978-0-494-44110-7

Abstract

Grammar-Based Object Oriented Genetic Programming leverages the Object Oriented programming paradigm in evolutionary computation. The difference between this approach and others is that the resulting solution is a syntactically correct and executable Java program. Candidate solutions are created and tested against a JUnit test suite. Evolved solutions incorporate useful Object Oriented patterns.

To create and compile the Java classes a new framework for Grammar-Based Genetic Programming has been created. The framework is entirely written in Java. By externally compiling and then loading the classes into the framework for testing true object-oriented encapsulation is achieved. For the first time state and behaviour can be passed around as an object.

Acknowledgements

Kristen, for supporting me and, introducing, me, to, commas. My dad, for being the brightest mind I know and always letting me bounce ideas off of him. My mom for always believing in me.

Finally, I would like to thank Dwight for agreeing to supervise me and allowing me to explore this topic. His guidance and patience through the entire process was invaluable.

> *I love deadlines. I like the whooshing sound they make as they fly by.* - Douglas Adams

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Since the mid 1990s, object-oriented programming has become the dominant programming paradigm. Object-oriented programming allows for a high level of abstraction and strict boundaries in applications. By combining state and behaviour into one entity it is possible to delegate work to objects and strictly define what an object can do. This idea can be achieved with procedural programming, but the lines of responsibility become blurred.

Evolutionary computation uses Darwin's Theory for Natural Selection to evolve solutions. A set of candidate solutions are evaluated to see how close the candidate is to the optimal solution. Candidate solutions that are better are combined with other promising solutions. The newly combined are then evaluated. Over several generations the solutions become better at solving a given problem.

Humans benefit from easier abstraction and code reuse when using object-oriented design. Evolutionary computation can also leverage these benefits when evolving a solution. The notion of state and behaviour is somewhat foreign in the evolutionary computation community. However, the benefits of allowing evolutionary algorithms to have access to several entities with well defined roles and an individual state are worth examining by evolutionary computing. Until recently no attempt has been

made to evolve objects, or to use objects in evolutionary computation.

None of the previous attempts to evolve the behaviour of objects have created true objects. The evolved objects are not viewed as true objects by Java. Instead of an encapsulated object with a clearly defined state and behaviour, the objects are a collection of loosely coupled variables and methods.

This thesis presents a new approach called grammar-based object-oriented genetic programming. This new approach ensures that any programs produced are syntactically correct. The generated programs are fully functioning Java classes that can be compiled, loaded and instantiated by the Java VM. By ensuring that generated programs can be executed within Java, no complex interpreter needs to be created to test and run solutions. All constructs and methods associated with Java are available for free.

1.1 Problem

Programs evolved in evolutionary computation have used a strictly procedural representation. No attempts have been made at evolving the behaviour of an object as part of a solution. No implementation of an evolutionary computation framework exists to attempt the evolution of objects. Genetic programming evolves programs, but these programs have no notion of state. Abott et. al. [3] produce an object, but without true encapsulation. Without using encapsulation, any object created simply calls the methods of other objects in a procedural way. The notion and benefit of encapsulation is lost.

Previous approaches used Java's reflection libraries to evolve an objects behaviour. Using reflection does not allow for true encapsulation to be observed. The evolved object must truly exist within the Java VM, compiled and instantiated. The object can not simply be a list of instruction invoking methods on other objects. A well defined class gains the ability to interact within the system in a more direct way.

Using object-oriented constructs such as double dispatching become available to the evolved object. The evolved object does not need to know the internal working of other classes to properly invoke their methods. The evolved object can pass itself as a parameter to other functions.

In order to create a well-defined object that can be treated as any other object, at a minimum the object must be syntactically correct. Evolving syntactically correct programs in a non-trivial grammar is very difficult. Care must be taken to allow all valid possibilities without crippling the system to such an extent that a proper solution cannot be found.

1.2 Goal

The main goal of the thesis is to develop an EC approach that can do the following:

Create syntactically correct Java classes. These Java classes will evolve a single line method that computes a value and returns the result.

Produce more complex behaviour within an evolved method. Multi-line methods that correctly combine and leverage already implemented methods are to be evolved. Objects will be used in a non-trivial way and in an object-oriented manner. Delegation or some other form of clear delineation between the state and behaviour of objects will be observed.

The solutions to the problems presented in experiments in the Results chapter require objects interacting with each other in a specific order to attain a proper solution. No one function call with the correct parameters can generate a complete solution.

To evolve an object interface is beyond the scope of this thesis.

1.3 Objectives

To properly evolve the behaviour of a Java class, a framework will be created. The framework will use a grammar-based approach to evolutionary computation to ensure that the classes are syntactically correct. Instantiating and evaluating the evolved classes will be handled by the framework. The framework will have facilities to evolve the objects in order to guide the search for an optimal solution.

In order to facilitate experimentation and make the framework more flexible, an XML specification will be defined for the grammar. A schema will be defined to allow different grammars to be easily used within the framework. The defined grammars will be general enough to ensure that multiple experiments can be run successfully.

1.4 Thesis Overview

The following section is a high level overview of the upcoming chapters. Each chapter is briefly examined.

1.4.1 Chapter 2

The background chapter examines the biological basis for evolutionary computation. The translation of DNA into a functioning protein is described, highlighting how it is simulated in evolutionary computation. Any biological terms needed throughout the thesis are defined and explained here.

Following the biological primer is an introduction to evolutionary computation. The general life cycle of an evolutionary algorithm, fitness evaluation, and generation of new populations is described. Several streams of evolutionary computation are covered, including the multiple streams that form the basis for grammar-based genetic programming.

1.4.2 Chapter 3

This chapter focuses on the grammar and how the grammar produces a syntactically correct Java class. The framework is not examined here, it is discussed in the following chapter. This chapter also introduces the augmentations to the traditional grammar-based approach that allowed for the evolution of object behaviour. How these new artefacts relate to existing grammar components is reviewed.

The structure and implementation of the parse tree is discussed in detail. How the parse tree defined by a grammar is used to represent the evolved program is looked at. The different constructs within the parse tree and how the tree is formed are also examined.

Finally, a new structure called a Skeleton (which is used as scaffolding to help the evolution) is introduced. Skeletons can be thought of as a wrapper around the evolved method. The relationship between skeletons and the parse tree is also examined. Skeletons are defined using an XML specification.

1.4.3 Chapter 4

The framework used for the experiments is discussed in this chapter. The full life cycle of the evolutionary framework is examined. The generation and compilation of a program, the evaluation of individuals, and the generation of new populations are all touched on.

The way in which a grammar and parse trees are used to generate programs within the the framework is discussed. The notion of a context is also introduced during this chapter. The context is used to store the state of the evolution. What object types and variables can be accessed by the evolution are determined by the context.

JUnit unit tests are used to evaluate the individual programs. The limitations within JUnit and the workarounds are discussed. The issues involved with JUnit stem from the fact that JUnit was not designed to be used as a fitness evaluation tool.

Finally, the evolutionary process used by the framework is reviewed. Particular emphasis is given to the type of crossover used by gramar-based genetic programming. The traditional type of crossover used within grammar-based systems is examined. The issues involved with this crossover style are highlighted. A brand new crossover that addresses some of these issues is analysed and presented.

1.4.4 Chapter 5

In this chapter, we conduct several experiments designed to validate and demonstrate different aspects of the framework and approach. The first experiment reproduces an experiment done by O'Neill and Ryan [17] to show that the framework implemented can properly evolve Java objects. The two types of crossover introduced in the framework chapter are compared.

The next two experiments show object-oriented behaviour being evolved. One experiment evolves a withdrawal transaction for a bank account. The solution requires the use of double dispatching and control structures to be properly implemented. The other experiment involves evolving three methods for a Reverse Polish Notation calculator. The object must interact with its instance variable as well as instantiate objects to correctly simulate a calculator. The two object-oriented experiments both use the same grammar specification, which demonstrates the flexibility of the framework.

Chapter 2

Background

Grammar-Based Object-Oriented Genetic Programming is an emerging field within Evolutionary Computation, that is built upon many different areas of Evolutionary Computation (EC). This chapter examines the various elements of EC that Grammar-Based Object-Oriented Genetic Programming (GBOOGP) draws on, as well as the biological theory that it is based on. Several different streams of EC are examined, showing how they differ from one another and how GBOOGP draws from each stream. Before any discussion of EC can occur it is important to look at how biology and evolutionary theory have influenced the terminology and ideas surrounding it. A brief introduction to biology follows.

2.1 Biology Primer

Grammar-Based Genetic Programming and more over, the entire field of EC, is based on biology and biological systems. When discussing Evolutionary Computation, biology terminology and analogies are always used. Since EC and GBOOGP draw heavily on the genetic foundation of living things, it is important to look at some of the more common terms and a simple example of protein synthesis to understand

them.

2.1.1 DNA and Genetic Foundations

All living things have, at their base, a genetic code for instructing how the organism should be built. All their characteristics are governed by this code. The code is a long polymer chain of nucleic acid known as deoxyribose nucleic acid or simply DNA[1].

> Chemically, DNA consists of two long polymers of simple units called nucleotides, with backbones made of sugars and phosphate groups joined by ester bonds. These two strands run in opposite directions to each other and are therefore anti-parallel. Attached to each sugar is one of four types of molecules called bases. It is the sequence of these four bases along the backbone that encodes information. This information is read using the genetic code, which specifies the sequence of the amino acids within proteins. The code is read by copying stretches of DNA into the related nucleic acid RNA, in a process called transcription. [28]

The DNA of an individual represents its genotype. The realisation of the coded instructions stored within DNA is known as an individual's phenotype. The phenotype is the physical and mental characteristics of the individual, for instance if the DNA codes for blue eyes, then the phenotype is the fully-functioning blue eyes. Individuals' offspring only inherit the genotype of their parents, not the phenotype. The difference between genotype and phenotype is very important for the different approaches in EC.

The genotype is not open to selection, within evolution. The phenotype concretely exists within the world, and is subject to selection. If the phenotype is selected then its genotype is propagated to subsequent generations. For example, if a strand of DNA codes for the length of tail feathers, the strand of DNA is not selected by

[1]Some creatures have a slightly different genetic code known as ribonucleic acid or RNA

potential mates. Rather the phenotype of the long or short tail feathers resulting from the DNA and the bird that they are attached to is open to selection by potential mates. For our discussion it is important to remember. Selection occurs only on the phenotype, not on the genotype.

Any changes to an offspring occur to the genotype, which in turn affects the phenotype. An offspring's genotype can be affected by mutation or recombination. Mutation is simply the change of one or more portions of an individual's nucleotides. Recombination or crossover is how an individual receives genetic information from both of its parents. These two halves of genetic information are combined to give the offspring its own unique genotype.

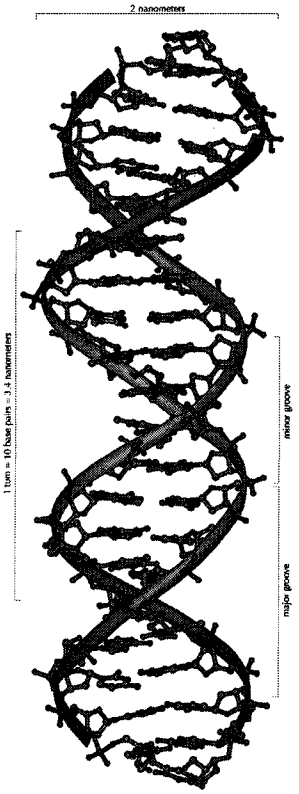


Figure 2.1: DNA double helix [28]

Living organisms can be thought of as bags of proteins. DNA contains the instructions for creating proteins. Proteins perform tasks and are the phenotype of the DNA. The process of translating DNA into a fully functioning protein is known as synthesis. Very generally, synthesis happens in two main stages: transcription, and translation. Transcription is the copying of DNA into ribonucleic acid (RNA). RNA is another type of nucleic acid, the polymer that it forms includes a different base pair. The RNA is then passed to a molecule called a ribesome which translates the codons[2] to produce a polypeptide sequence. A protein can involve

[2]Each protein has a unique encoding of RNA, the sequence of amino acids that make up a protein are determined by the codons present in the RNA. A codon is a set of three bases which codes for a single type of amino acid.

one or more polypeptide sequences in different 3D shapes [5]. Figure 2.2 shows the transcription and translation phases.

2.1.2 Evolution

Charles Darwin's book *The Origin of Species* showed that modern species had evolved into their present form through small incremental changes over many generations. What Darwin had – at the time – a harder time showing was that this evolution was due to natural selection. Each individual was under pressure to reproduce, and those that reproduced more passed more of their positive characteristics to their offspring. Unfortunately for Darwin, he had no mechanism for heredity, no knowledge of DNA, making the argument for descent with modification a difficult one.

For evolution to occur four things are needed: a high fidelity method of copying a genotype, some way of recombining genotypes, a way to introduce random variations, and some sort of pressure on the population. The high fidelity copying of a genotype is taken care of by DNA, which has the ability to reproduce itself without errors. Mutation, where a random error occurs in a DNA sequence allows for some randomness and novelty to be introduced. Finally, if there are no scarce resources to fight over and no hardships faced by individuals, then all individuals are equally fit, i.e. individuals cannot differentiate themselves from one-another.

Population genetics examines the evolution of populations, since evolution can only occur on a population. Through sexual recombination and mutation individuals can introduce variance into a population. If these variances prove beneficial then the frequency of this new variance will tend to increase. Individuals with any advantage will – in a competitive environment – produce more offspring which will inherit the new variation. After many generations the population's genetic makeup will have shifted.

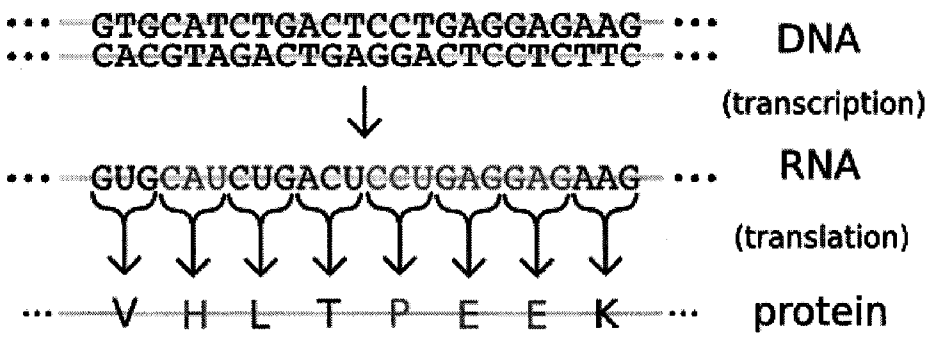


Figure 2.2: Protein Synthesis
[27]

2.2 Evolutionary Computation

"GAs [Genetic Algorithms] work by discovering, emphasizing and recombining good 'building blocks' of solutions in a highly parallel fashion" [15] Although it specifically refers to Genetic Algorithms, the quotation is true for all evolutionary algorithms. EC, in the broadest sense, is using Darwins Theory of Natural Selection to evolve a solution to a problem. The selection pressure on a population causes the better individuals – closer to an optimal solution – to be more successful and propagate their characteristics into the next generation thus causing a rise in the subsequent population's fitness [7]. EC is an exploratory algorithm that walks through a search space. Each individual is a point within the search space. Since each individual in the search space is a point in this multidimensional space, it is possible to have a notion of 'distance' between individuals. The distance is not a Euclidian distance, but a measure of what it means to be neighbours.

Mitchell [15] discuses a real life example of protein synthesis where distance is the number of amino acids differing at each location between two individuals. The individuals' genotype is represented by the amino acids that are coded for by the codons. Two individuals with genotypes $AGGMCGBL$ and $MGGMCGBL$ have a distance of one since the two individuals differ at the first position only. However individuals $AGGMCGBL$ and $LBMPAFGA$ have a distance of eight. Neighbours

typically have some characteristic in common and also more in common with closer neighbours than with more distant ones. Even though two individuals can be close in the solution space, this does not mean that their fitness values are the same. Individuals with similar genotypes can vary wildly.

In order to represent the location of individuals' fitness values, EC uses a *fitness landscape*, first introduced by Sewell Wright. "A fitness landscape is a representation of the space of all possible genotypes along with their fitnesses"[15]. The 'landscape' is shaped by the 'peaks' and 'valleys' that result from graphing the fitness values of individuals. An improvement in fitness can be though of as climbing higher up a 'peak'. Within this landscape any 'peak' that is less than the optimal fitness is called a *local optima.*

It is possible for the population to climb and rest on a local optima and never find the solution because to break away from this 'peak' would cause too much of a degradation in fitness. For EC to work properly there needs to be some gradient to reward small incremental changes that make an individual's fitness better. If the fitness landscape is a step function [3] then the population has no way of moving towards the solution. Either an individual stumbles upon a solution by pure chance or not. In a step function there is no difference in fitness between sub-optimal individuals, regardless of their distance from the optimal solution.

The solution space and fitness landscape are only abstract ideas used to describe what happens during an EC algorithm. In summary, individuals exist as a point within a solution space and are assigned a fitness value. Fitness values are discrete measurements used as a heuristic to properly explore promising solutions. The ability to properly explore promising solutions is dependent on how much noise (sub-optimal peaks) exists within the fitness landscape. How these abstract notions are leveraged is examined in the next section.

[3] the fitness is either optimal or completely sub-optimal

2.2.1 Basic EC algorithm

The basic life cycle for an EC algorithm involves initializing a population of individuals, evaluating the fitness of individuals within the population, selecting individuals for recombination, generating offspring for the next generation, and evaluating the fitness of the individuals within the next generation. These steps, save for the initialization, are repeated until a solution is found or time has run out.

The initialization of individuals is usually done in a randomized fashion. Randomisation is used to get the largest genetic diversity possible from which to begin searching for solutions. Going back to the notion of a landscape, a well initialized population would be scattered rather than clustered together in communities. By having a large amount of genetic diversity the situation where a critical part of the solution is not represented within the population can be avoided. In some cases when the solution space is sufficiently understood, it is possible to use a heuristic while initializing the population. By using a heuristic, individuals representing impossible solutions can be excluded. The individuals within the initial population are used for recombination to produce the subsequent population.

In order to determine which individuals to use for recombination, individuals are assigned a fitness value. This value represents, in a measurable way, how close the individual is to the optimal individual (in other words, how close the solution is to being correct). Figure 2.3 shows the basic EC engine that is used when trying to search for a solution. Individuals with a higher fitness are more likely to be chosen for recombination. Higher fitness is the main inductive bias used in EC. Since a highly fit individual has something 'good' about it, the EC system wants as much of that 'good' thing as possible to be passed on to future generations.

Recombination will take two or more individuals and produce two or more offspring. The aim is that two fit individuals will combine to produce fitter offspring, or, at a minimum the valued traits of an individual are hopefully preserved in the offspring. Typically recombination will not be disruptive. Recombination is performed conservatively enough to keep good parts of the solution together. Splitting an individual

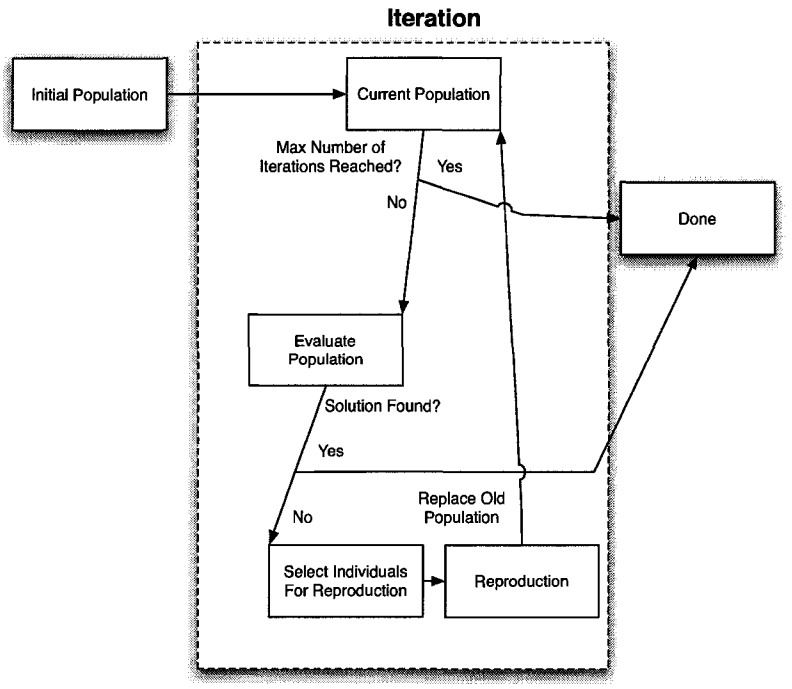


Figure 2.3: Basic Evolutionary Computation Algorithm

into too many small parts will likely cause harm to whatever was good about the solution. Some approaches to recombination, like in Grammar-Based Genetic Programming are very disruptive (this will be examined in more detail in Section 4.3.1). The new population is formed by iteratively selecting individuals from the current population, having them recombine and produce individuals for the next generation. The new population replaces the current population to complete an iteration, as shown in Figure 2.3. The algorithm will terminate if an optimal solution is found or the maximum number of iterations has been reached.

2.2.2 How The Solution Space is Searched

The solution space is searched through a process called descent with modification. Each generation's genetic makeup is slightly different from previous and subsequent

generations. As each generation's individuals recombine the average individual in generation n should be fitter than the average individual in generation *n-1*. The average fitness increase only as a rule of thumb and is not strictly speaking always the case. With each passing generation the "good" traits are encouraged to flourish, while those with detrimental effects are weeded out. Any traits that have neither a positive nor detrimental effect on an individual's fitness are not open to selection, as they do not affect an individual's fitness. These neutral traits are propagated into future generations based on the non-neutral traits that they are associated with. In EC, traits are normally associated spatially.

The two key factors needed for descent with modification are some form of heredity, and some way of introducing novel characteristics or traits. The heredity involves high-fidelity copying and exchange of traits between individuals. The exchange of traits to produce offspring is known as recombination. Recombination allows for a novel combination of traits to be produced but has no way of introducing new traits.

When discussing the basic EC algorithm it was noted that the starting population was randomly initialised. If the random initialisation did not produce one of or a set of key traits required for the solution, or if a specific key traits was lost somewhere during previous generations, then no combination of the existing traits will find the optimal solution. In order to introduce or in some cases reintroduce these traits random noise must be introduced. Mutation is the operator used for introducing random traits into the evolution.

Heredity and General Recombination in an EC System

For heredity to properly work in EC there are two basic requirements that must be in place. First, a method to produce high fidelity copies of the parents genetic information, and second a way to recombine the genetic material. The sections of genetic information that are exchanged are copied exactly from one individual to another. By reproducing these sections exactly, no noise is added into the individual. Most EC

programs will simulate how living organisms pass along genetic information.

The most common approach for recombination is sexual reproduction, with two individuals recombining their genotypes to produce offspring. The genotype is the internal representation of the individual. For humans and other living creatures the genotype is DNA or RNA. For Genetic Algorithms it is an array of bits that needs to be copied and passed to future generations. Figure 2.4 shows a simple example of sexual reproduction. Genetically unique offspring is what is desired/required. The specific mechanics involved with recombination is domain specific and unimportant for now.

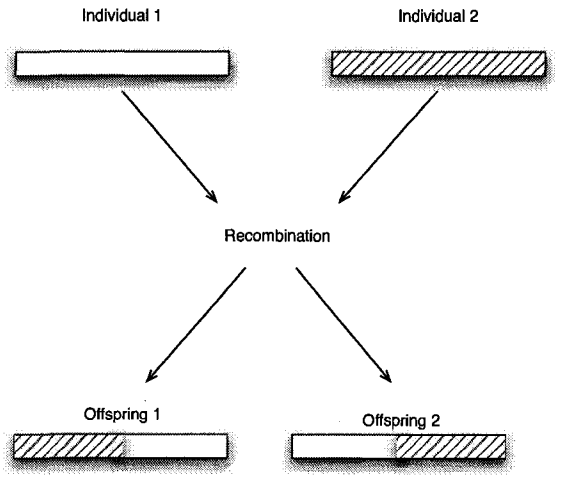


Figure 2.4: Basic Sexual Reproduction

Mutation

Mutation is key for exploration within the search space, it can introduce new beneficial traits or reintroduce lost ones. When picturing a landscape of all possible solutions with individuals being a point within that landscape, recombination and heredity can be thought of as a walk along a path. Mutation allows an individual to 'jump' from the slope of one peak to another, or along the current path. By

jumping around the landscape, new paths open up to the individual that were never previously an option.

The jump caused by a mutation can be beneficial or detrimental, the key is that it can't be too large. In his book *The Genetical Theory of Natural Selection*, R. A. Fisher showed that the probability of a mutation being beneficial decreases in relation to the severity of the mutation. So, if an individual is sitting somewhere in the solution space and a single random change is introduced into its genotype, the chance that it is beneficial is about 50 percent. Either the distance between the individual and the optimal solution is closed by one step or it moves away by one step. However if n random changes occur at once then the chances of the mutation being beneficial overall is 0.5^n, a 50 percent chance of being detrimental or beneficial per random change.

2.3 EC Flavours

EC comes in many flavours: Genetic Algorithms, Evolutionary Strategies, Evolutionary Programming, Classifier Systems, and Genetic Programming. Grammar-Based Genetic Programming borrows heavily from both Genetic Algorithms and Genetic Programming.

2.3.1 Genetic Algorithms

Genetic Algorithms (GA) were first conceived by John Holland to study adaptation in an artificial system. The individuals' genotypes are represented as a binary array. The genotype of GAs must be translated into the phenotype. The translation is specific to the domain. The representation remains domain independent.

GA Example

The following is an example of a simple GA and how it is represented and evaluated. The problem is the *Max-Ones* problem. The fitness is determined by the number of 1's present within the bit-array. Optimally the array consists of all ones, the position of the ones is unimportant. For example, the arrays 0110 and 1001 both have a fitness of 2.

Table 2.1: Initialisation and Fitness Evaluation

Individual No.	Initial Population	Fitness	Expected Contribution	Actual Contribution
1	010100	2	0.5	1
2	101100	3	1.13	2
3	000001	1	0.25	1
4	000011	2	0.5	0

Table 5.4 shows an initial population of individuals. The maximum fitness possible for these individuals is six, the size of the array. Individual 2 is the most fit individual of the population with a fitness of 3, followed by individuals 1 and 3. The least fit is individual 3.

The expected contribution (how many offspring they are expected to have) is proprotional to the individual's relative fitness. The selection process is probabilistic and indicates how often an individual is expected reproduce. In practice the observed number of times an individual is selected can vary from the expected. Table 2.2 shows that the fittest individual does breed twice, as expected. Of note is the fact that individual 4 does not breed even though its expected contribution is higher than individual 3. In a larger population one would expect that on average individual 4 would produce twice as many offspring as individual 3.

The results of the selection and recombination are shown in Table 2.2. The only offspring to increase its fitness was number 3. The fitness' of the remaining offspring either remained unchanged or was made worse by the crossover. The average fitness

Table 2.2: Mating and Crossover of Initial Population

Mating Pair	Individual Genotype	Crossover Point	Offspring Genotype	Offspring Fitness	Change
1	01—0100	2	**10**0100	2	-
2	10—1100	2	**01**1100	3	-
2	10110—0	5	10110**1**	4	+1
3	00000—1	5	00000**0**	0	-1

for individuals in Table 2.2 is higher than the average fitness in Table 5.4.

Table 2.3 demonstrates the mutation operator affecting some of the individuals in the population. The mutation operator is a bit flip. The value of a bit within the array is changed from a 0 to a 1, or from a 1 to a 0.

Table 2.3: Mutation Operator on Two individuals

Individual No.	Initial Genotype	Fitness	Mutation Location	Genotype After
1	100100	2	3	10**1**100
4	000000	0	1	**1**00000

The representation in the above example uses an array of bits. An array of bits is ideal for the max-ones problem examined in the above example. Choosing the appropriate representation is the most important part when setting up a GA. Since the genotype and phenotype are separate domains a representation that can map properly into the problem domain is key. If there are certain areas within the fitness landscape that cannot be reached by the current representation, or its translation into the problem domain, then the GA may fail to find a solution. Genetic programming avoids these potential pitfalls by keeping the genotype and phenotype the same, no translation is needed.

2.3.2 Genetic Programming

Genetic programming (GP) also uses Darwins theory of natural selection to evolve solutions to problems. Through evolution, the candidate solutions to a problem explore the solution space by creating an executable program that attempts to solve the problem. The difference between a GP approach and a genetic algorithm is that the genotype and phenotype are the same. That is to say that there is no translation from the hereditary information that is being evolved – the genotype –, to the outer description of the actual solution – the phenotype –. Evolution occurs directly on what is exposed to selection. So, GP is unique in that the genotype is exposed to selection, and the phenotype is exposed to evolutionary process. In a GA, by contrast, the evolution occurs on the internal representation that is not open to selection.

GP was initially implemented using Lisp and its function-based paradigm allowing the program to be visualized as a tree [11]. The tree represents both the genotype and phenotype, and this tree structure is what makes it possible to perform the evolutionary operations directly without any translation. Recombination occurs by swapping subtrees from two parents to create two new offspring. The newly formed offspring are genetically different. All instructions apart from the exchanged subtree remain preserved. The individual's offspring to retain any good subtrees not involved in the recombination, allowing the individual to incrementally evolve towards a solution. Each incremental benefit is kept by the individual and passed onto its offspring.

Table 2.4: Example Terminal and Functional Set

Functional Set	$\{+, -, /, *\}$
Terminal Set	$\mathbb{Z} \cup \{x, y\}$

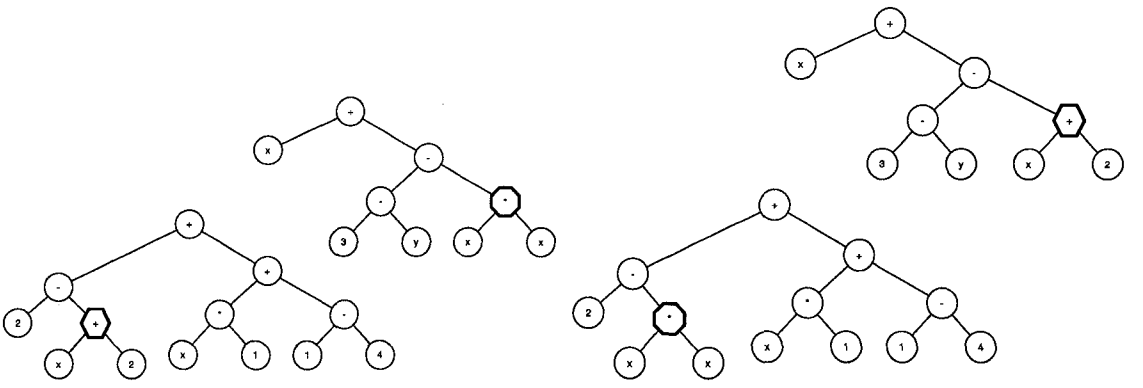
The syntax of the trees are defined using a function set and a terminal set. Elements of the functional set are internal nodes and can never be leaf nodes; whereas elements of the terminal set must be leaf nodes, and cannot be internal. Table 2.4 shows the

functional and terminal sets necessary for generating the trees found in Figure 2.5 on page 31. More formally the functional sets can be defined as follows:

- All elements of the terminal set T are correct expressions.
- If $f \in F$ is a function symbol with arity n and $e_1, ..., e_n$ are correct expressions, then so is $f(e_1, ..., e_n)$.
- There are no other forms of correct expressions.

[7]

All operators and operands must be compatible, otherwise the individual created cannot be evaluated due to syntactical errors. Having a system that is fully interoperable is known as the closure property. For instance, an operator that performs a function on two Strings cannot work if it is given an integer. The search space of all valid programs is a very small subset of all possible combinations of instructions. Therefore, one needs to ensure that either all combinations of instructions are valid, that invalid combinations of operators and operands are handled in some way, or that only valid combinations are ever generated. The latter method is what Grammar-Based Genetic Programming does.



(a) Parents with Crossover Points Chosen (b) Resulting Offspring

Figure 2.5: Crossover in Genetic Programming

Subtrees can be thought of as a well defined, discrete set of instructions. Passing around these subtrees is similar to passing around functions. The output produced

by the subtree is dependent on inputs but the instructions are preserved. The syntax or structure of the subtree is context independent. A unique subtree in one individual will represent the same subtree (structurally) in any individual. There is no chance that the structure of subtree that is passed between individuals will change its structure. Since there is no translation in GP, there is no chance for reinterpretation. However having no translation can lead to issues of its own. There is the possibility that invalid individuals can result due to the crossover.

Any tree that can be specified with the syntax is valid, because all operators are interchangeable. The input of any function can be the output of any other function. The drawback with the closure property is that for any new function, the new function must accept all existing functions as input. All existing functions must now accept as input the new functions output. Interoperability is required due to the closure property and will cause GP to generate a large amount of individuals that cannot possibly be correct. The search space is increased much more than necessary due to the interoperability requirements.

In order to get around the closure property we must introduce some way of reigning in what can be done through crossover or what needs to occur to fix the incorrect expression. Montana introduced the notion of *Strongly Typed* GP [16] to get around the issue of closure.

The main difference with Strongly Typed Genetic Programming (STGP) is the restrictions that it places on construction of the initial random population and the crossover operator. When building the initial generation, the tree must adhere to certain rules, i.e. the output of a subtree must be compatible with the input of its parent. So at each node only a subset of functions and terminals may be chosen, rather than the entire set. These restrictions remain in effect for the crossover operator. Only subtrees of similar or compatible roots may be exchanged. The manner in which Montana [16] encodes is quite domain specific, with all valid subsets being hardcoded. Montana's approach does not expand well since for a larger syntax the number of subsets could become very large. Another approach to ensure syntactically

correct individuals is to use a grammar-based approach.

2.3.3 Grammar-Based Genetic Programming

In Grammar-Based Genetic Programming (GBGP) the genotype and phenotype are two separate entities. The genotype is translated into a program. The grammar ensures that the translation results in a program that is syntactically correct. GBGP does not have any restrictions on crossover, the crossover is similar to a GA, where the crossover can occur anywhere along the genotype. It is in the translation from genotype to phenotype that grammar is enforced. At the centre of this approach is the grammar, an example grammar can be seen on page 2.1.

A grammar consists of non-terminals, terminals and production rules. Non-terminals are the same thing as functions in GP. GP's notion of terminals is also the same in GBGP. The difference between GP and GBGP lies in the production rules. Production rules are how syntactic correctness is enforced. Any production rule that can be chosen will always work towards a syntactically correct individual. In other words there is one possible combination of production rules that will result in a syntactically incorrect individual. Each grammar has a well defined set of terminals and non-terminals. The non-terminal set in the sample grammar consists of, *expr*, *op*, and *var*. The symbols x, 1.0, $+$, $-$, $/$, $*$, (and) represent terminals. The sample grammar in Implementation Detail 2.1 is shown in Backus-Naur Form (BNF) [17].

BNF represents a grammar by showing each non-terminal and its production rules. Each non-terminal is on the left, with all possible production rules on the right side of the equals sign. For example the $<$ *expr* $>$ non-terminal has three possible production rules. The *expr* non-terminal can be replaced by *expr*, *op*, *expr*. The *op* non-terminal can be replaced with $+$, or one of the other mathematical operators. Production rules consist of one or more non-terminals and or terminals. Syntactic correctness is enforced by strictly controlling what production rules can be applied

$$
\begin{aligned}
&< expr > ::=< expr >< op >< expr > && (0)\\
&\quad |(< expr >< op >< expr >) && (1)\\
&\quad | < var > && (2)
\end{aligned}
$$

$$
\begin{aligned}
&< op > ::= + && (0)\\
&\quad |- && (1)\\
&\quad |/ && (2)\\
&\quad |* && (3)
\end{aligned}
$$

$$
\begin{aligned}
&< var > ::= x && (0)\\
&\quad |1.0 && (1)
\end{aligned}
$$

Implementation Detail 2.1: Example Grammar in Backus-Naur Form

to which non-terminal. The replacement of non-terminals with production rules goes from left to right, the left most non-terminal is always replaced. Once all non-terminals have been replaced the program is completed.

For example, starting with the non-terminal $< expr >$ and applying the production rule $(< expr >< op >< expr >)$ the result is $(< expr >< op >< expr >)$. Now the left most token is a terminal (so we cannot apply a production rule to it. The left-most non-terminal is an $< expr >$ node again, so this is what is replaced. Applying the third production rule of the $< expr >$ non-terminal the resulting equation is $(< var >< op >< expr >)$. $< var >$ is still a non-terminal so the next production rule must be applied to it. Non-terminal $< var >$ is replaced by a terminal x this time, leaving an equation of $(x < op >< epxr >)$. Now the left-most non-terminal is $< op >$ that is replaced by another terminal $+$, after $< op >$ has been replaced then the left-most and last remaining non-terminal is $< expr >$, this is replaced in turn by $< var >$ and then by the terminal 1.0. The finished program looks like $(x + 1.0)$, where all tokens are now terminals. The application of the rules assumes that there is some deterministic function which decides what production rules are applied and when. For grammar-based genetic programming the genotype determines what production rules to apply.

In GBGP the individual's genotype is represented by an n-array of integers, each integer in the array codes for a production rule. The exact length of the array is dependent on the problem domain. If the array is too small then it is possible that the proper solution will not be found, but if the array is too big then it might be hard for the program to converge on the correct solution. The algorithm will have to search a much larger search space than necessary.

The integer array is translated into a program using the production rules to ensure syntactic correctness. The resulting list of terminals represents the individual's phenotype. It is this phenotype which then is evaluated. The phenotype is a fully realised syntactically correct and executable program.

It should be noted that it is possible that not all leaf nodes are terminals. If there

are any non-terminals remaining within the tree after all n integers have been read and translated then default production rules are used. Default production rules are applied and typically will be the least amount of production rules needed to reach a terminal. This approach is used in order to grow the tree only as much as is absolutely necessary. Also by using a defined set of default production rules, an individual's phenotype will always be the same. No randomness is introduced that may produce one good individual for one generation only.

Creating this very clear definition between genotype and phenotype, also creates a very clear distinction between the search space and the solution space, the solution space being inhabited by the fully realised program. O'Neill and Ryan point out that, "...separation of search and solution spaces can result in benefits such as unconstrained search of the genotype while still ensuring validity/legality of the program's output"[17].

GBGP borrows heavily from how proteins are produced in living systems, Figure 2.6 illustrates the similarities between protein production and the generation of a program using Grammar Based GP.

GBGP can be thought of as GAs with an augmented translation algorithm. The translation algorithm is the production rules within the grammar. The grammar consists of a set of terminals and non-terminals. These are analogous to the function and terminal sets discussed in GP. Implementation Detail 2.1 a simple grammar in Backus-Naur Form, in brackets is the production rule number.

Abstractly, when discussing these grammars and how they produce phenotypes, the translation involves one non-terminal being replaced by another or by a terminal. In the implementation a parse tree is used. The parse tree is very similar to the program tree used in GP. However the parse tree in GBGP is what is known as a *concrete syntax tree.* Another distinction between GP and GBGP is that only the terminals affect the program, they are what define the program's instructions in GBGP. The non-terminals are intermediate steps in the creation of the instructions. In GP the non-terminals represent functions and are active in the computation.

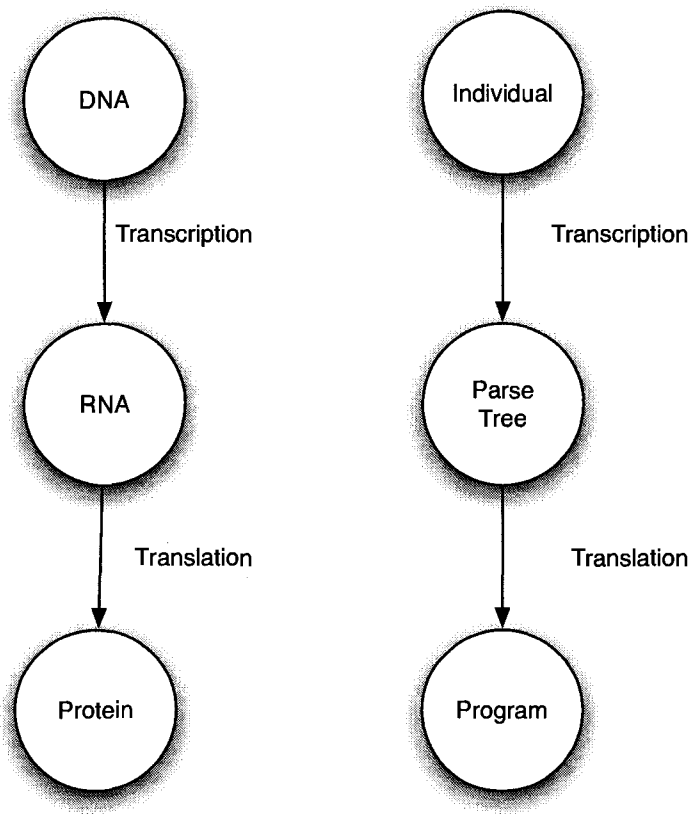


Figure 2.6: Comparison Between DNA Transcription and Program Creation

The sample translation of the grammar does not show how an individual's genotype is transformed into its phenotype, or how the genotype influences the phenotype. As stated earlier the individual's genotype is represented as an array of integers. These integers determine which production rule replaces the current non-teminal. Since these integers can be any number, there needs to be a way to ensure that only a valid production rule is chosen. For example the integer values used in the experiments detailed in Chapter 5 ranged anywhere from 0-10. None of non-terminals had more than 5 production rules, so to prevent choosing production rule 7 when it did not exist the modulus operator is used.

The next few paragraphs re-examine the previous example with an emphasis on how

the individual is represented and how the production rules are applied.

The individual's genotype is: $1, 2, 0, 0, 2, 1, 5, 5, 2, 0, 1, ...$ and our starting non-terminal is $< expr >$. So, to begin producing the program we need to apply a production rule to $< expr >$. For this we take the first integer value in the array (1) and mod it by the number of production rules for the current non-terminal (3), $1\%3 = 1$ so we apply production rule 1 (we are using a 0-based index). Figure 2.7 shows a detailed diagram of what occurs in the parse tree. The resulting tree that produces the output $(x + 1)$ can be seen in Figure 2.8.

The mod function was introduced by O'Neill and Ryan [17] as a way of making the production of a parse tree more flexible. By using the mod operator no special calculations are required if the integer in the individual's phenotype is larger than the number of production rules for a particular non-terminal. A valid production rule will always be chosen.

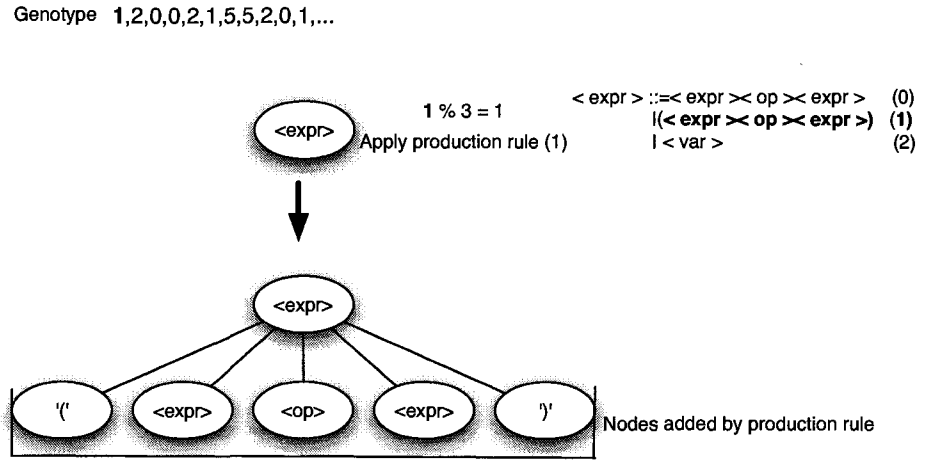


Figure 2.7: First Application of A Production Rule

The resulting tree shows the difference between a GP tree and the parse tree in GBGP. In GP, non-terminals or functions are a part of the program that gets executed. In GBGP only terminals are a part of the program. The non-terminals of a GBGP tree are simply intermediate steps that have no affect on how the program

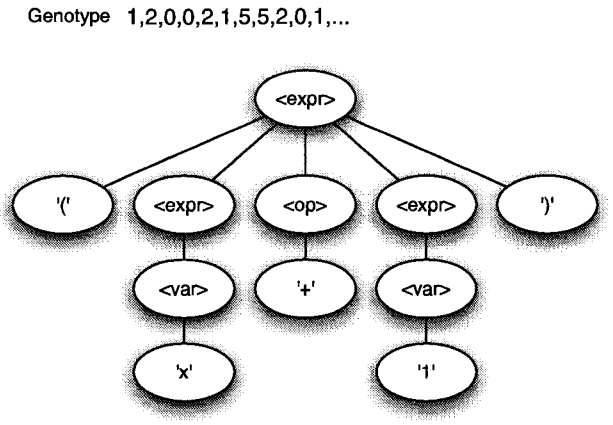


Figure 2.8: Resulting Parse Tree

ends. It needs to be clear that the parse tree produced during the translation step in GBGP is not the phenotype of the individual. The parse tree is not evaluated it is the program that the parse tree represents that is the phenotype. Once the parse tree has been completed, an in-order traversal of the tree is performed and the text of each terminal node is outputted. The outputted nodes form the executable program.

Using this approach it is possible to generate programs without having to deal with invalid syntax. Also, this approach leads to a much more flexible way of generating solutions to a wide variety of programs which may require a problem specific language. All that needs to be plugged in is an xml specified grammar and a different language can be used to search for a solution. The program can be written in C, C++, Java or some other domain specific language. The method of evolution and underlying representation of an individual does not change, only the translation of the individual does.

For simple programs the grammar can be fully specified, which ensures that all programs are syntactically correct. A fully specified grammar implies that there is no combination of methods and arguments that cannot be fully specified by the grammar. Requiring a full specification amounts to the closure property in GP, which

can only work for very small grammars where all combinations can be enumerated. The work done by O'Neill and Ryan[17] falls into this category, where the grammar is simple and fully defined. More difficult examples would require larger grammars or an approach similar to STGP. A grammar based system incorporating type checking is discussed in detail in Section 2.3.5.

2.3.4 Object-Oriented Genetic Programming

Object-Oriented Genetic Programming (OOGP) is a relatively new approach to standard GP, and allows GP to have the same type of abstraction that humans enjoy with object-oriented programming. Allowing GP to store and pass around state is a useful addition. OOGP borrows strongly from STGP, since objects themselves are types and can only understand certain messages. In fact, Agapitos and Lucas employ Montana's approach when creating recursive functions for Fibonacci and Exponential problems [4]. Their approach, similar to Abbott's [2] use objects as simple data structure holders.

Agapitos and Lucas defined a set of wrappers around the objects being used so that the GP tree produced could be interpreted. For each possible instruction there is a corresponding object wrapper/node for the tree to use. The inputs and outputs of these nodes are defined similarly to those in Montana's [16] paper to ensure that the produced trees were syntactically correct. The major difference between Monatana's approach and Agapitos and Lucas' is that instead of function calls, some instructions were messages passed to objects.

Agapitos and Lucas do say that their evolved methods "should be able to use its own objects to invoke methods on."[4] It is not clear what the method has access to and under which circumstances it can call methods. All of their solutions simply pass around objects as data structures but do not take advantage of them. How this differs from STGP is unclear since there did not seem to be a way of passing the evolved method's class to any other objects. They do make extensive use of

reflection to catalogue and use available methods at runtime, which is key to creating an extensible system for object-oriented evolution.

The results detailed by Abbott for evolving a sorting program [3] are similar, in that the only thing which is OO about the program is that the program is allowed to send messages to objects. Abbott's approach is much less flexible since it dictates what functions are available to use. How the methods should be called, even the condition in the if statement is hardcoded. Abbott also goes to great lengths to analyse the different logic sections of the problem. He breaks down the sorting function into a set of smaller sub-problems. These smaller sub-problems are then evolved separately.

While writing small self contained function is a good way of developing code, it is very guided from of evolution. One nice thing about Abbott's attempt is that each subsequent method had access to the previously evolved methods. Having access to previously evolved solutions within a currently evolving object is interesting and useful. Although useful having access to previously evolved solutions does not deal with the fact that an object cannot pass itself around, using the 'this' or some other method of self reference.

Some initial attempts have been made to incorporate objects into EC. The clear separation between interface and implementation is a very desirable trait. Since programmers use objects as a type of abstraction to allow them to handle more complex systems, it stands to reason that EC could benefit from this as well. The experiments done by Agapitos, Lucas and Abbott begin to leverage the fact that objects can be asked to perform tasks and there is no need to know how those tasks are done. All of their experiments could also be done in a procedural way, passing in data structures. None of their attempts leveraged object-oriented design for the methods that were being evolved. With all of the experiments done by Agapitos, Lucas and Abbott it is not clearly defined what is consider to be a class and where the method is being evolved, it is difficult to know how that object can be moved around. Having a strictly defined notion of an object's state and behaviour and

utilising this encapsulation to delegate is a key level of abstraction in OO.

2.3.5 Grammar-Based Object-Oriented Genetic Programming

Grammar-Based Object-Oriented Genetic Programming (GBOOGP) attempts to generate syntactically correct OO programs. No work other than this thesis has been done in this area, only programs using procedures have been evolved. The benefit of using a grammar-based approach is that the entire richness of the language is available. Where other approaches could only use a crippled subset of the syntax available, in GBOOGP all construct of the language can be accessed.

GBOOGP as the name suggests is the combination of GBGP and OOGP. Using a grammar based approach side steps the issue of requiring a sophisticated interpreter to execute an individual's phenotype. The individual's phenotype is a syntactically correct program in a given programming language. All that is needed is to compile and run the program. Because the phenotype is simply an executable program there is no need or requirement to produce a domain specific interpreter as there is in other approaches. There is no need to fake existing mechanics or features of a language since they are available for free.

Not being a compiled program prevents the evolved method from having a state of its own, i.e. without being inside of a well defined class, it has no class. By having no class it has none of the benefits such as an encapsulated state or a well defined interface. Since the generated program is a stand alone class it can take advantage of these things, like passing itself to other objects. Being able to perform delegation allows for simpler functions to be evolved, since the evolved class can now rely on other objects to do the work for it.

2.4 Summary

The thesis work on Grammar-Based Object-Oriented Genetic Programming draws heavily on both Genetic Algorithms and Genetic Programming. The representation of an individual's genotype in GBOOGP is the same as that in a GA. Also, the idea of separating the search space and the solution space is taken from GAs. The phenotypic representation of an individual mirrors the tree based idea used in GP. Type checking and syntactic correctness are provided by both the grammar being used, and the work done in OOGP.

All of these different types of EC are used together to give a very close emulation of protein synthesis described in Section 2.1. The translation and creation of an individual's phenotype in GBOOGP is based on the process of DNA $\rightarrow$ RNA $\rightarrow$ a fully functioning protein. Using the protein synthesis as a biological basis our goal is that GBOOGP produces fully functional Java objects.

Chapter 3

Grammar Based Genetic Programming Approach

3.1 Introduction

This chapter examines the implementation details involved with the translation of an individual's integer array into a fully defined parse tree. The different components used in the grammar implementation are examined. How the grammar drives the generation of a parse tree is also discussed. The Skeleton class is introduced in Section 3.5. Skeletons are used as a type of scaffolding to help ensure that a syntactically correct Java class can be generated.

The grammar and parse tree are distinct but interconnected structures. When developing a Java class using a grammar based approach, the grammar acts as a way of describing how a program can be formed. The parse tree represents the program. The grammar is used to form the parse tree but does not define a class itself.

The grammar exists only to produce a syntactically correct program. The program is represented by a concrete parse tree. All leaves in the completed tree are terminals, and are written to the java file. The leaves are traversed in order from left to right.

When the traversal is complete, a syntactically correct code-snippet has been formed. The tree is grown by adding children to a node, based on the applied production rule for that non-terminal node.

The generation of a new program is done by filling in the missing information. The content surrounding the missing information is a type of scaffolding called skeletons. Skeletons are a way of keeping static information within the program. A parse tree representing the generated code can be placed inside a skeleton. The skeleton acts as a type of wrapper around the parse tree.

For clarity, we define the following terms:

- Program: a Java class produced by using the grammar based approach.
- Framework: the implementation of the grammar based approach.

Stated another way, any mention of a program refers to something that can be produced by the framework. The framework is discussed in detail in Chapter 4.

3.2 Overview

In order to put the different parts of the system described in this chapter into context, Implementation Detail 3.1 shows a high level pseudo code representation of the algorithm used to generate a parse tree. The algorithm begins by getting the root node of the parse tree and the initial integer of an individual's genotype. The integer represents which production rule is to be applied.

```
node := parseTree->getRoot
rulNumber := curIndividual->getStartingInteger

while (parseTree is not complete) do:
  grammar->applyProductionRuleTo(ruleNumber, node)
  node := parseTree->nextNodeToFill
  ruleNumber := curIndividual->getNextInteger
```

Implementation Detail 3.1: Algorithm for Generating A Parse Tree

The grammar then applies the proper production rule to the current node. The components of the production rule are added as children to the current node. Once the children have been added to the current node, the parse tree is asked to for the next available node within the tree. The individual is asked for the next production rule to be applied.

While the tree is still incomplete, i.e. there is at least one leaf node that is not a terminal, the process continues. At the end of the while loop a fully defined parse tree is produced. The parse tree's terminals outline an executable program. The first major part of this process is the grammar.

3.3 Grammar

The grammar is represented in Backus-Naur Form Grammar as outlined in Section 2.3.3. Within the framework, an XML representation of the grammar is used in order to parse the grammar more easily. The grammar consists of a set of Terminals, Non-Terminals and Rules. These grammar artefacts are referred to as components. These components do not actually produce the program, they merely represent how the program can be formed in a syntactically correct way. A production rule is defined as a set of components which replace an existing component. One component is replaced by the components of a production rule. This is continued until either no production rules can be applied, or the array representing the genotype has been exhausted.

The grammar defined in Section 2.1 has been reproduced here for easier reference. Any examples involving a grammar will be referring to the grammar below.

$$
\begin{aligned}
&< expr > ::=< expr >< op >< expr > && (0)\\
&\quad |(< expr >< op >< expr >) && (1)\\
&\quad | < var > && (2)
\end{aligned}
$$

$$
\begin{aligned}
&< op > ::= + && (0)\\
&\quad |- && (1)\\
&\quad |/ && (2)\\
&\quad |* && (3)
\end{aligned}
$$

$$
\begin{aligned}
&< var > ::= x && (0)\\
&\quad |1.0 && (1)
\end{aligned}
$$

The notion of components, terminals, non-terminals, and production rules are specific to the grammar only. Components exist with no notion of context, the context is only given to a component when it is in a parse tree.

Component

A *Component* is the base class for all artefacts within the grammar. Terminals, non-terminals and production rules are all subclasses of components. For instance $+$, $< expr >$, and $< expr >< op >< expr >$ are all components. The components represent a terminal, non-terminal and production rule respectively.

NonTerminal

The *NonTerminal* class is a representation of the non-terminal element defined in a BNF grammar. Since a non-terminal is not an end state, within the production of a program it must be replaced by another component. The NonTerminal class maintains a list of all Components that it can be replaced by, ensuring a syntactically correct individual. These Components can be of type Terminal, NonTerminal or Rule. For example, in the sample grammar $< expr >$ is a NonTerminal.

Terminal

Terminals are a subclass of Component and represent the terminal element from the BNF grammar. Terminals are the simplest type of Component and only contain a string indicating what the output to the file will be. Since a terminal can not be replaced it maintains no list of Components. $+$ and x are both examples of Terminal objects. The Terminal for x would simply write x out to a file. Terminals are the end result of applying production rules to NonTerminals. The Java class is entirely specified by the Terminals of the generated parse tree.

Rule

Production rules within a BNF grammar do not actually exist in the same way as terminals and non-terminals do. Production rules indicate how a specific non-terminal can be replaced. A *Rule* is simply a list of Components. Rules only exist as a way of bundling a set of Terminals and NonTerminals so that they can be added to the production rule list that a NonTerminal maintains. For example a Rule would be $< expr >< op >< expr >$ and this Rule could then be added to the NonTerminal $< expr >$. It is a way of discretizing how a NonTerminal can be replaced.

A production rule for a BNF grammar is anything that the non-terminal can be replaced with. Within the context of the framework the scope of a Rule is much

more narrow, it does not represent all possible production rules, only those that are made up of multiple components. As mentioned above a Rule would be $< expr >< op >< expr >$ and represent a production rule. Whereas $< var >$ is also a production rule of $< expr >$ but it is not represented by a Rule object. The element $< var >$ is represented as a NonTerminal object.

Rules can be thought of as meta-components, simply a high level way of describing how components are connected. When looking at intermediate steps of how a program is formed, it is possible to see where a rule was applied but not the rule itself. For example, if the NonTerminal $< expr >$ were replaced with the Rule $< expr >< op >< expr >$, all that is left is three NonTerminal objects.

Component, Terminal, NonTerminal, and Rule objects are used to specify the grammar as defined by O'Neill and Ryan [17]. This grammar needs to be fully specified. All terminals, non-terminals and production rules must be specified within the grammar. In order to produce programs that could send messages to other objects this grammar must to be augmented.

3.3.1 Object-Oriented Grammar

The grammar defined by O'Neill and Ryan had to be augmented in order to properly produce syntactically correct OO programs. The requirement that every method call and parameter sequence be explicitly spelled out is unreasonable. Any program that uses objects becomes intractable very quickly. Any grammar that fully defines the possible combinations of object instances, methods and parameter combinations will become extremely large. In an OO space, evolution is no longer happening in a homogeneous environment, (i.e. everything is not of the same or compatible types). In order to avoid this limitation, types needed to be introduced into the grammar.

Rather than enumerating a special production rule for each unique type allowed, the component indicates that the framework requires a type to be selected. By adding

the type component it is now possible to define a grammar for the general case. The types within the evolution are determined by the problem domain. By keeping the type information separate the same grammar can be used for multiple and differing problems.

The two BNF grammar snippets in Implementation Details 3.2 and 3.3 illustrate the difference between enumerating the types and keeping the type information separate. In the first grammar snippet the $< method\_call >$ non-terminal must explicitly list all possible method calls that the program can use. Where as the second BNF snippet lists how the method call can be correctly constructed. In the second instance the $< method\_call >$ non-terminal is replaced by the $< type >$ non-terminal. The values that $< type >$ can take has been removed from the grammar and can now be defined elsewhere.

$$
\begin{aligned}
< method\_call > ::= & \; objectA.foo(a, b) && (0)\\
| & objectA.foo(b, a) && (1)\\
| & objectB.bar(a, b) && (2)\\
| & objectB.bar(b, a) && (3)
\end{aligned}
$$

Implementation Detail 3.2: Grammar With Enumerated Methods

$$
< method\_call > ::=< type > . < method > \qquad (0)
$$

Implementation Detail 3.3: Grammar With Dynamic Type Look-up

A grammar used to define a programming language does not enumerate all possible types. The types and variables that are available within a program depend on how the solution is implemented. The grammar of a programming language can remain fixed regardless of what the solution is. The programming language never enumerates

all valid types. The ability to dynamically allow types to be represented is what is lacking in O'Neill and Ryan's work. Types are required in order to properly leverage the notion of encapsulation within object-oriented design. The idea of encapsulation allows several different types of objects/classes to collaborate on a solution, since they each have their own state and behaviour.

In order to create a functioning object-oriented grammar, type information is very important. When creating a program, it is imperative to know which messages can be sent to what objects. Without type information, all messages would need to be compatible with all objects. Having no type information would require a homogeneous system. It is possible to artificially create a homogeneous environment where type information is fixed and unecessary, through the use of interfaces. The benefits of defining an interface (i.e. defining the interface bot the implementation) would be lost if the classes did not honour this contract. For example, if an interface *Foo* were defined for a solution and had methods *a*, *b* and *c* then there would be no way of telling what those methods are intended to do, let alone what the interface is meant to represent. The names *Foo*, *a*, *b* and *c* chosen are clearly poorly named. However, the fact remains that unless the classes implementing an interface share a common theme no method name will be accurate or descriptive. It is possible to overcome obtuse interface and method names with descriptive variable names as hints indicating what a class is actually being called and, more importantly, what the methods are doing.

By limiting the solution to a strictly homogeneous environment from which to create the program, only one interface can be used. Using one interface ensures any available object can receive any available message, resulting in the solution only having access to the methods defined within that single interface. By adding methods to the interface the solution can have access to more methods. The downside to adding methods is that some classes are required to add empty methods for the sole purpose of satisfying the extended interface. For example, if the solution requires two completely distinct classes in both state and behaviour, then the interface will need to have the union of the messages that both classes can receive. In this case, both

classes will need to implement extraneous methods. A more minor issue is how to create new objects if no type information is available. The issue of instance creation can be overcome by implementing a factory class which would simply return a new instance of a certain class. The types would then be defined within the factory and determined in the same way as any other rule.

The solutions mentioned above to artificially create a homogeneous system all cover up the fact that a fully specified grammar is undesirable. The resulting grammar and outputted solution will become unclear. The underlying issue is that in a strongly typed language, types are required not optional. To address this issue we introduce a Type component.

In our approach, there are two other components that work with the type component: the variable and method component. Both are extensions we made to O'Neill and Ryan's original grammar. The variable component determines what actual instance of a type will receive a message. The actual message that is sent to the variable is determined by the method component. How Type, Variable and Method production rules are determined is discussed in detail in Chapter 4. They are introduced here to show the extensions that are required.

Type

A *Type* component indicates what the receiving class of a message call will be. A Type can be any class that the framework has knowledge of. For instance String, Integer, Stack are all classes that could be chosen. The Type component is a subclass of NonTerminal. However, the production rules associated with Type are not defined within the grammar. The production rules are determined by the problem domain and defined separately.

Variable

Variable is also a subclass of NonTerminal. Once a type has been determined a Variable can be selected. The Variable selected will be an instance of the class that was chosen by the Type's production rule. Again the number of instances available for each type can vary, depending on the problem and how the evolution has progressed.

Method

The third component introduced is the *Method* component. The production rules for the Method component are dependent on the class selected. Any instance method associated with a particular class is a production rule.

3.3.2 Grammar XML Specification

For the grammar to be properly parsed and represented within the program, XML was chosen as the document format[1]. For simple grammars the only requirement was to have tags for Terminal, NonTerminal and Rule components. Implementation Detail 3.4 shows the xml specification of the grammar shown in Section 2.3.3.

A terminal, since it is simply a set of characters, is represented by a unique name and string value (i.e. what the output of the terminal is). The non-terminal is more complicated, it has a unique name used as an identifier, and it contains all of its production rules. Rules, non-terminals and terminals are listed as part of the non-terminal tag's production rules. A rule is also represented in xml as simply a combination of terminals and non-terminals.

The xml has two distinct uses for each tag. One is the tag's definition, the other is when a tag is referenced. When the tag is defined it is created once, so that it has

[1]The XML schema for the grammar can be seen in Appendix A.1

a unique name within the name space. Any reference to this unique name will be associated with the original definition. Below is the XML specification for the simple grammar from Section 2.3.3.

Some of the components contain an optional attribute that represents the default production rule. This is used to indicate which production rule should be used if no rule is specified. The use of default rules is examined in more detail in Section 4.1.

3.4 Parse Tree

The internal representation of the Java program is in the form of an n-ary tree (since each production rule can contain one or more components,) known as a parse tree or concrete syntax tree. A tree structure was chosen for its ease of manipulation. Using a strictly string-based representation would have resulted in a very error prone and slow application of the production rules.

The parse tree is the syntactical representation of the program, it shows the intermediate production rules. It is more verbose than an abstract syntax tree which only shows the semantically important nodes. Using an abstract syntax tree is not useful for this reason as it would be very destructive during crossover. In fact the subtree preserving crossover method detailed in Section 4.3.2 would be impossible to implement. Using the concrete syntax tree allows for the crossover algorithm to see and have access to the intermediate steps used to create the program. Removing these would cause the one-to-one mapping between the genotype representing the individual and the program being produced to become out of sync. Figure 3.2 shows the abstract syntax tree representation of the parse tree in Figure 3.3. The abstract syntax tree removes all of the intermediate steps and only retains the semanticaly important steps.

The parse tree is represented using an interface called a *Node*, this Node can have

```
<GPGrammar>
    <terminal name="plus">+</terminal>
    <terminal name="minus">-</terminal>
    <terminal name="div">/</terminal>
    <terminal name="mult">*</terminal>
    <terminal name="open_bracket">(</terminal>
    <terminal name="close_bracket">)</terminal>
    <terminal name="const">1.0</terminal>
    <terminal name="x">x</terminal>

<!-- RULES -->
    <rule name="exp_op_exp">
        <nonterminal>exp</nonterminal>
        <nonterminal>op</nonterminal>
        <nonterminal>exp</nonterminal>
    </rule>
    <rule name="b_exp_op_exp_b">
        <terminal name="open_bracket"/>
        <terminal name="exp"/>
        <terminal name="op"/>
        <terminal name="exp"/>
        <terminal name="close_bracket"/>
    </rule>
<!-- NON TERMINALS -->
    <nonterminal name="exp">
        <rule name="exp_op_exp"/>
        <rule name="b_exp_op_exp_b"/>
        <nonterminal name="var" default="true"/>
    </nonterminal>
    <nonterminal name="op">
        <terminal default="true">plus</terminal>
        <terminal>minus</terminal>
        <terminal>div</terminal>
        <terminal>mult</terminal>
    </nonterminal>
    <nonterminal name="var">
        <terminal name="x" default="true"/>
        <terminal name="const"/>
    </nonterminal>
</GPGrammar>
```

Implementation Detail 3.4: Regression Grammar XML Specification

one or more child Nodes. Each Node in the tree contains either a *Terminal* or *NonTerminal* component. Nodes containing a Terminal cannot have children as they are leaf nodes. All internal nodes have a component of type NonTerminal.

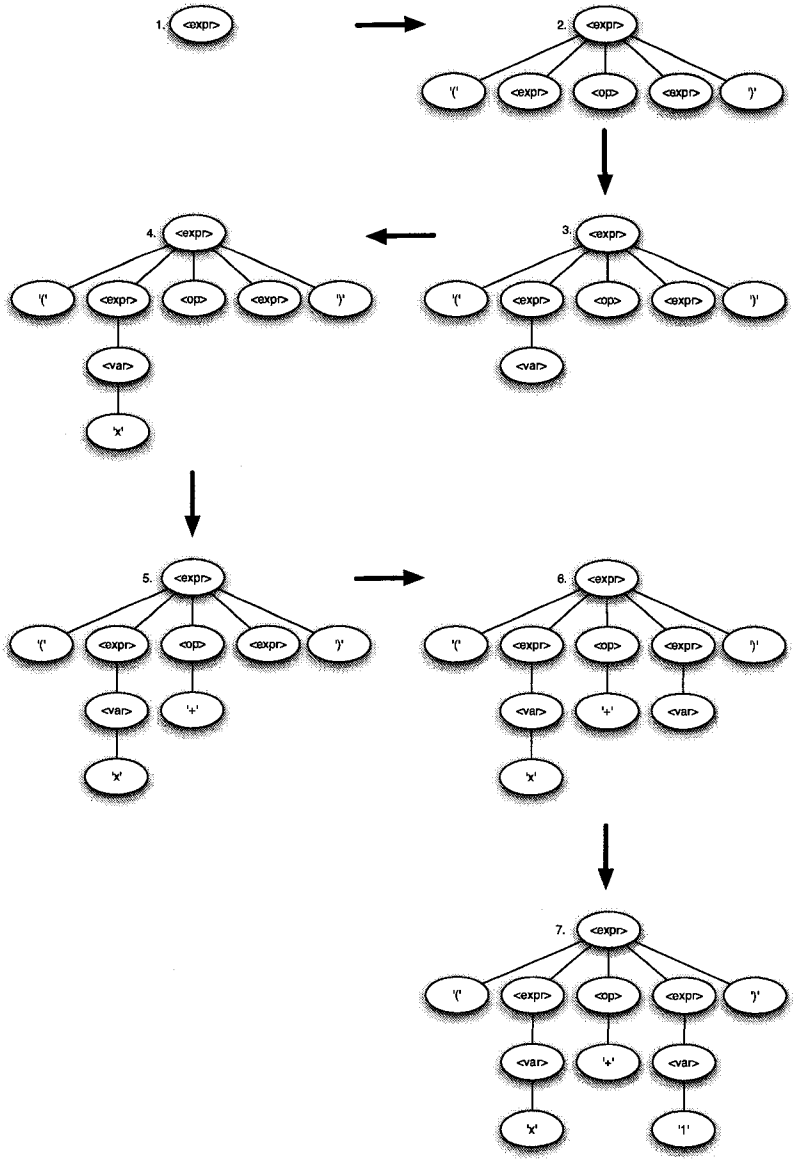


Figure 3.1: Simple Example of The Formation of A Parse Tree

In a completed tree, all leaf nodes are terminals, and the program has been formed. A depth first traversal is then performed where only the tokens from each terminal are written out to a file. No non-terminal nodes are represented in the resulting program, except for special non-terminal nodes known as Skeletons.

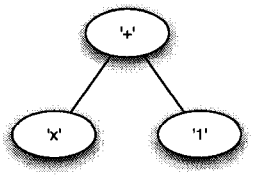


Figure 3.2: Abstract Syntax Tree

Since most solutions have more than one thing in common, the idea of a Skeleton was introduced. A Skeleton is a wrapper around the code generated by the parse tree and is described more in Section 3.5. This wrapper forms the shell or API of the class being evolved. The fully constructed class, i.e. the skeleton plus the evolved code, is written out to a java file and compiled before being loaded for testing.

3.5 Skeleton

Skeletons can be thought of as a scaffolding for the use of the program. They are a way to connect code fragments together to form a cohesive program. By capturing the static information for each problem domain the grammar and creation of objects is made simpler. Any information which is needed by the compiler, but does not affect what the program does, is placed in the skeleton.

If we look at the output produced from the parse tree in Figure 3.1 the output is $(x + 1)$. No Java compiler would accept this as syntactically correct, since the code fragment must be inside of a class and method. Implementation Detail 3.5 shows what the syntactically correct code would look like.

The only code that is evolved in Implementation Detail 3.5 is $(x + 1)$, all remaining

```
public class SimpleClass {
    public int method1(int x) {
        return (x+1);
    }
}
```

Implementation Detail 3.5: Basic Java Class

code is part of the skeleton. In actuality, the final parse tree in Figure 3.1 looks like Figure 3.3. Skeletons are merely a specialised type of node. A *ClassSkeleton* is always the root node of a parse tree and the skeleton will have one or more children. Given that the skeletons act only as a type of scaffolding, they are not represented by the grammar. Skeletons remain the same regardless of what occurs in the evolution. Only nodes that are not skeletons are affected by the grammar.

During the initial regression tests the skeletons were defined programatically to include a method wrapper as well. The class skeleton was subclassed and specialised for each experiment. The class skeleton had one child, a non-terminal, which was the starting token for the grammar. The parse tree was generated from there.

A *MethodSkeleton* was added in later to help accommodate the object-oriented experiments. A method skeleton defined the signature of a method. Method skeletons allowed for a class skeleton to have multiple starting points from where to evolve the program. Each method skeleton is a child of the class skeleton. The root of the parse tree generated by the grammar is now the method skeleton.

For example a class *Foo* has method skeletons *meth1* and *meth2*. The method skeletons meth1 and meth2 are children of the class skeleton Foo, but also the root node from where the evolution begins. The production rules are applied to the subtree rooted at either meth1 or meth2. Meth1 and meth2 are not subject to evolution or the grammar. They are the root of the subtree where the evolution occurs, Figure 3.4 shows at a high level how the parse tree looks.

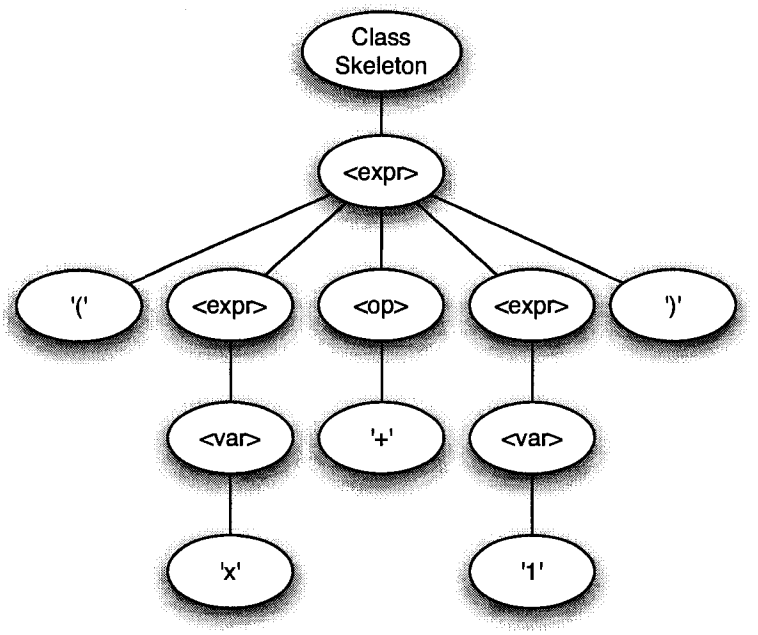


Figure 3.3: Simple Example of A Parse Tree With A Class Skeleton

3.5.1 Skeleton XML Specification

The skeletons are defined using an XML specification[2]. Skeletons are defined by the implementer of an experiment. The XML file indicates the name of the class to be created along with anything that needs to be included for compilation of the class. This includes any import declarations, any method skeletons that are going to be children of the class skeleton, and any variables that the class skeleton will have. The method skeletons referenced from within the class skeleton become the Java class' instance methods. The same is true for the variables which become the instance variables of the produced class.

The XML snippet Implementation Detail 3.6 is an example of what a user would give to the framework to setup an experiments skeleton. The skeleton defines a

[2]The XML schema for a skeleton can be seen in Appendix A.2

```
<ClassSkeleton name="class1">
    <import>org.foo.Foo</import>
    <variablelist>
        <variable name="var1">
            <type>int</type>
            <initialValue>0</initialValue>
        </variable>
    </variablelist>
    <method name="meth1" maxstatements="1">
        <visibility>public</visibility>
        <returnType>void</returnType>
        <parameterlist>
            <variable name="p1">
                <type>String</type>
                <initialValue>\"temp\"</initialValue>
            </variable>
        </parameterlist>
    </method>
</ClassSkeleton>
```

Implementation Detail 3.6: Class Skeleton XML Specification

```
import org.foo.Foo;
public class class1 {
  int var1;
  public void meth1(String p1) {
    <root of parse tree>
  }
}
```

Implementation Detail 3.7: Class Skeleton

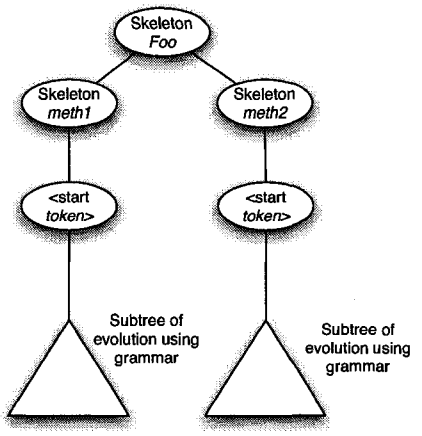


Figure 3.4: Diagram of Skeleton Nodes and Their Evolved Programs

class called *class1*, with an instance variable *var1* and and a public instance method *meth1*. The method has a local variable *p1* which is assigned the value of *temp*. Implementation Detail 3.7 shows the resulting class that is defined by the method skeleton in Implementation Detail 3.6.

3.6 Summary

The grammar was always intended to be a module that can change between experiments with relative ease. The Type, Variable and Method components where added to help increase the grammars flexibility. Being able to dynamically determine types is a valuable contribution to the grammar-based genetic programming.

The parse tree is what is produced by applying production rules from the grammar. The nodes of the parse tree each contain a component. The internal nodes have a non-terminal component while the leaf nodes each have a terminal component. The terminal components are written out to a file and define the executable program.

To overcome Java's rigid syntax defined for classes we created a skeleton node to take

care of any code that did not need to be evolved. The class definition, imports and methods were all handled by skeletons. The skeleton helped to avoid overly complex parse trees that remained static throughout the evolution. Skeletons never affected the behaviour of a class, they simply defined the interface.

Chapter 4

Grammar Based Evolutionary Framework

The extendible framework for Grammar-Based OOGP was designed specifically for evolving objects. Nodes, skeletons and grammars are all connected to produce a new grammar-based genetic programming framework. The framework has three distinct parts: evolution, parse tree creation, and compilation and testing. The basic execution path of the framework is very similar to the basic EC algorithm's execution path. The main difference is the extra compilation step needed before an individual's fitness can be evaluated. Figure 4.1 shows how the individuals are moved around the framework. The 'Generate Parse Tree' and 'Compilation/Class Creation' steps are essentially the translation step within the framework, from genotype to phenotype. The evaluation step occurs on the newly created Java classes rather than an interpreter being used to execute the program.

Once the program has been created and run against a set of tests to determine the fitness of the individual, the selection process begins. When two parents are selected for recombination, the integer arrays are crossed-over, resulting in two new children being created. These new children will again have their genotype translated into a functioning program that attempts to solve the problem.

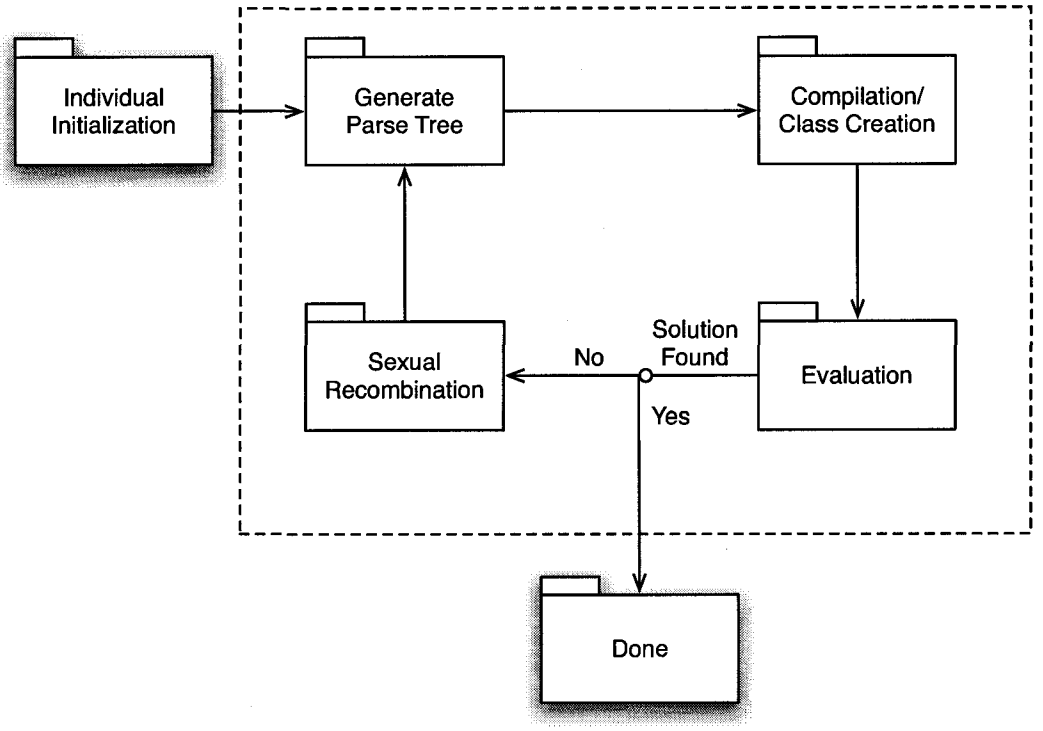


Figure 4.1: Execution Path Within The Framework

4.1 Translation

The translation of an individual takes the integer array of the individual and applies the grammar rules' production rules. The production rules produce the parse tree (as described in Section 2.3.3) and eventually a fully functioning Java class. At the centre of the translation is a static class called *Builder*. The Builder class takes an individual's genotype and applies the production rules based on the grammar that was passed in. By applying each successive rule to the current slot node a parse tree is created. The Builder class has no state of its own and only has static methods.

The individual's integer array is iterated over during the generation of a parse tree. Each value in the array represents a codon. The codon value is an integer in the range of $[0, 10]$. The production rule applied to a particular node is based on the

current codon value.

In order to keep the grammar and the parse tree as separate as possible, the nodes in the parse tree do not know how to apply a rule or how specific non-terminals behave, i.e. what production rules they contain. The "Visitor Pattern" [9] is used, with each component visiting and altering the state of the node. The components dont know how they are altering the state of the node, they simply asked to be added to the node. Figure 4.2 shows the messages passed between the *SlotNode*[1] and *Component* objects. No 'reaching in' is done to access the instance variables of either object in order to avoid tight coupling.

[1]SlotNode implements the Node interface

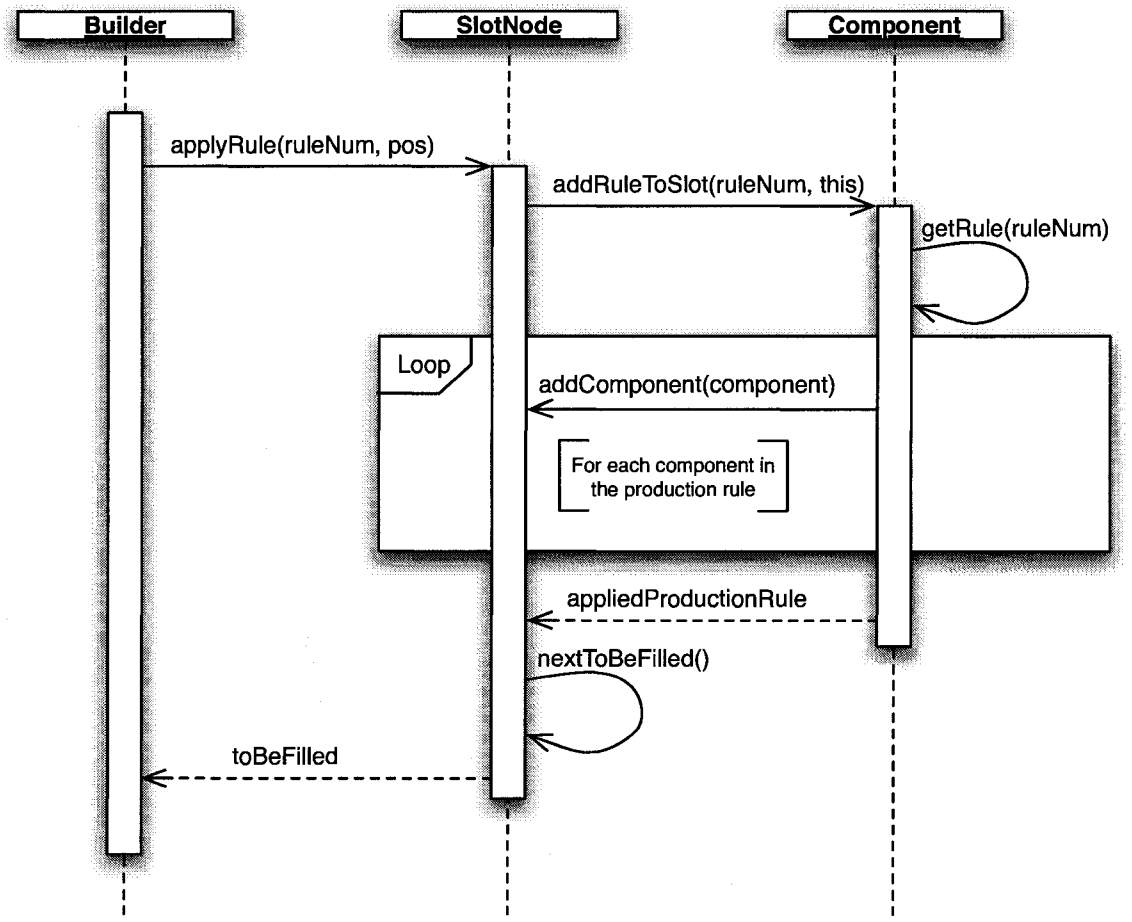


Figure 4.2: Execution Path Within The Framework

4.1.1 Context Object

The notions of dynamically defined types, variables and methods required the creation of a context. The variables available within separate methods that are being evolved are different, so some way of tracking this was required. Also, since new objects could be instantiated within the evolved method, it was possible to gain access to new variables during the evolution of a program. To keep track of all of this

the *Context* class was created. The context stored a list of types and a HashMap of variables associated with their types. A context could have a parent and/or a child. For example the context of a method would be the child of the class' context. A child context would have access to all of its parent's types and variables along with its own.

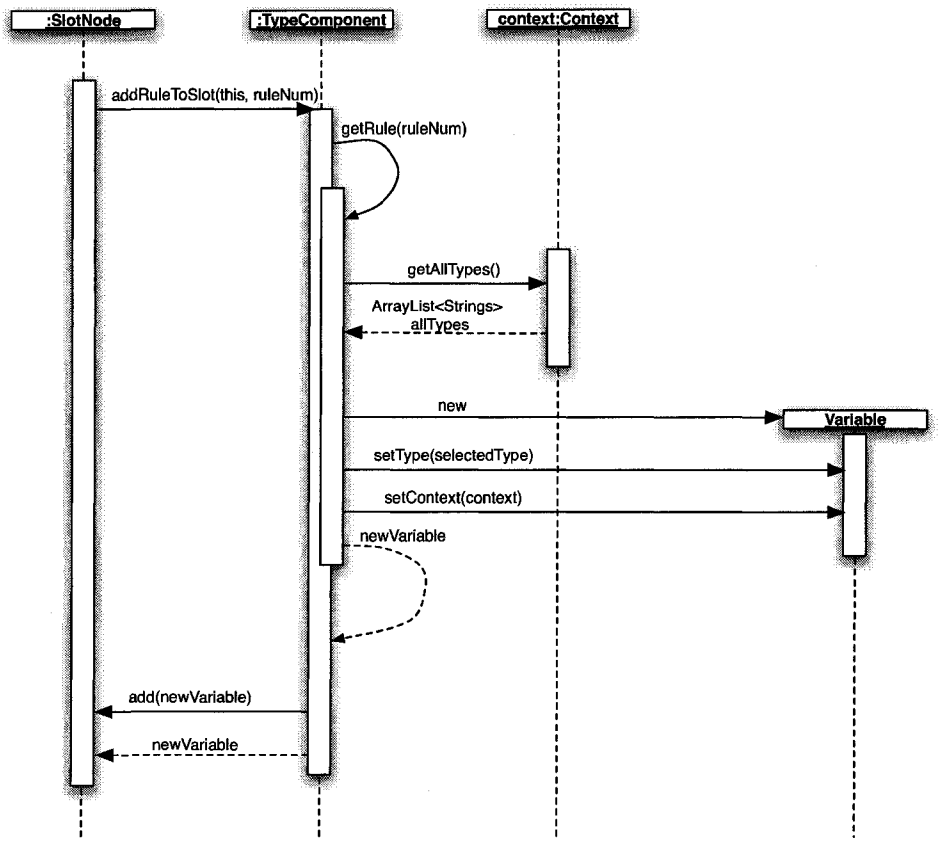


Figure 4.3: Message Sequence Chart For Selecting a Type

Selecting a variable that receives a message is a two step process. First the type of the receiver must be selected, then the instance of the receiver. Once a type has been selected, a valid instance of that type needs to be selected. The two steps occur in two separate components, the Type component and the Variable component. The Type component is responsible for determining the class of the receiver.

Whereas the Variable component must select a valid variable name (i.e. one that the context object knows about) for the particular class determined by the Type component.

The types known by the context object are based on the import declarations of the class skeleton. It is possible to use types not listed in the import declarations. For instance, a method that accepts an object that is a superclass of a known type is allowable. In this case, any valid subclass can be chosen as the type. Selecting a type is similar to applying a production rule to any other non-terminal. The difference is that instead of the production rules being defined by the grammar the production rules are the known types within the context.

For example, if the context knows about three types, String, Integer and Stack then the Type component has three production rules. The method for determining which production rule to use is the same. The modulus of the integer coming by the number of known types is used. So, if the current codon value of the individual is 5 then the production rule used is $5\%3 = 2$, a Stack type.

The production rules of other components contain a list of one or more terminals and non-terminals. This list is what is returned and subsequently added to the current node. The Type component is different, it creates a new Variable component and sets its type. Rather than having a predefined set of components that can be returned the Type component is dynamic. It will always return a Variable component but the type of that variable can change. The newly created Variable is what is returned. Figure 4.3[2] shows how a Variable component is determined and returned.

Figure 4.4 shows the message sequence chart for selecting and returning a variable name. The production rules for Variables are defined in the same way as a Type's production rules. Variables also have their production rules defined by the context. The production rules of a Variable are the number of variables know by the context of a specific type. Recall that the type of the Variable was set when it was created. The

[2]MSC [Message Sequence Chart] is a graphical and textual language for the description and specification of the interactions between system components.[21]

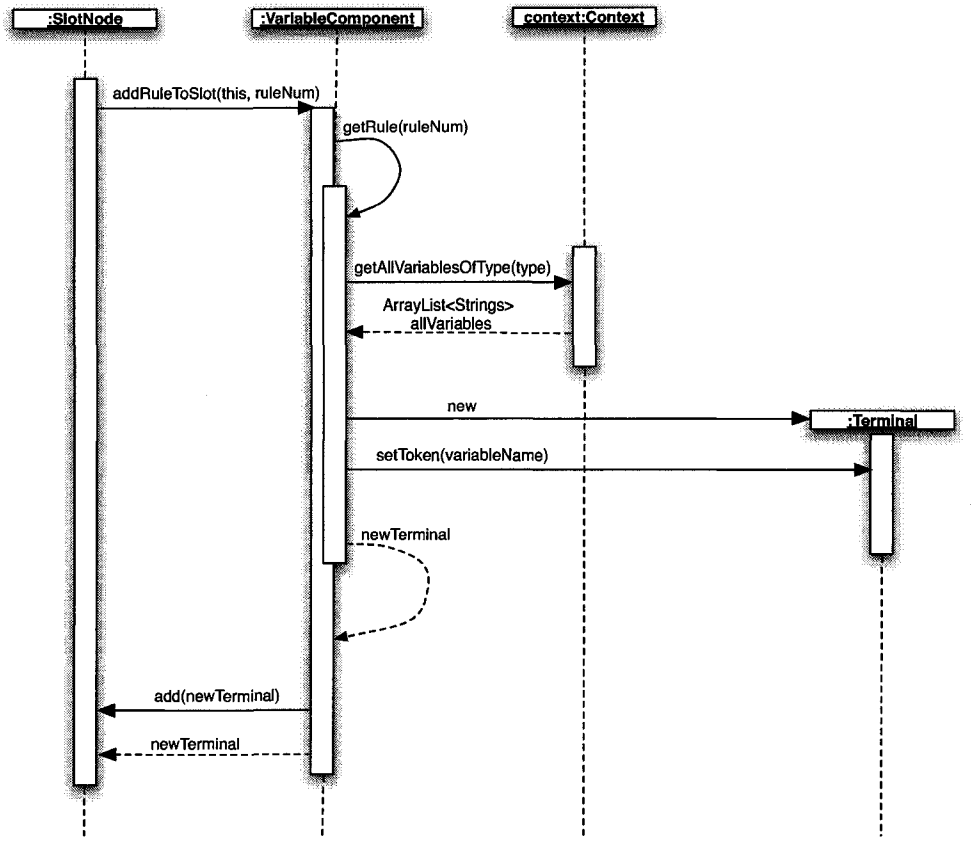


Figure 4.4: Message Sequence Chart For Selecting a Variable Name

number of variables within a given context can grow and shrink. For example a class may have several instance variables and a method may have several local variables. The method has access to both its local variables and the instance variables of the class as potential production rules. All that is left to determine is the name of the variable or to create a new one.

Creating a New Variable

The framework allows for new variables to be created on the fly. Whenever a variable name is selected an extra production rule is added to represent a possible new

variable. Lets say that the context knows of two Stack variables stack1 and stack2. With two known Stack variables the number of production rules is 3^3. To select a production rule the formula is actually $n\%3$. If $n\%3 = 2$ then a new variable is created. This variable is added to the list of known variables of type Stack and available for the remainder of the evolution. If the production rule to create a new variable was not chosen then one of the existing variables is used.

Once a variable name has been chosen, the Variable component creates a new Terminal and sets the string value of that Terminal to be the name of the variable chosen. The newly created Terminals is returned and added to the parse tree. Determining the receiver of a message is a two step process within the grammar: First choosing the type of the variable, second the actual name of the variable.

The final special component used is the Method component. As the name suggests it is used to indicate what method will be invoked. The Method knows what class will be receiving the message. The type of the receiver is set in the context. Using Java's reflection libraries a list of all possible methods is created. Each method is a possible production rule. Once a method has been chosen the Method component creates a new Rule. The Rule is a set of Terminals and NonTerminals that represent the methods signature. The name of the method and opening bracket of the method call are added as Terminals to the rule. For each parameter a Variable component of the parameter's type is added to the rule.

Filling in an Incomplete Tree

During translation it is possible the exhaust the integer array of the individual and still have an incomplete tree, i.e. some leaf nodes are non-terminals. If this occurs then the default production rules are used. For each non-terminal a default production rule can be defined. If none is defined then production rule 0 is automatically used.

[3] 2 for the known variables of type Stack, plus 1 for a possible new variable

The default rules are typically the shortest path from a non-terminal to a terminal. The default rules are applied until the tree is completed. In a complete tree all leaf nodes are terminals. This ensures that all trees will produce a valid program even if their genotype did not manage to complete one.

4.1.2 Java Class Compilation

Once the parse tree has been completed, the classes need to be compiled and instantiated. In Java 6, a new API was introduced to make calling the compiler easier. Before, the compiler had to be called externally through the command line on the underlying operating system. In addition to being required to know the location of the java files and the compiler location, there was no easy way to deal with compiler errors. Now, with this API, the default compiler for the runtime environment can be called programatically using a virtual file system[24]. The virtual file system allows the user to pass around *File* objects instead of needing the physical file locations. Also, since the package *javax.compiler* is merely a set of interfaces, it is easier to write a single piece of code irregardless of VM.

The parse tree outputs its contents to a **.java* file. The content is all the string values from leaf nodes and whatever output the skeletons add. All leaf nodes contain a Terminal component. The Java files are opened and passed to the compiler, producing **.class* files. All Java files have a unique file/class name that is determined by the individual's name. Once compiled, the Java class loader is called to load up the files. The benefit of using a grammar-based solution is it ensures that each class at the very least compiles. Since there is a one-to-one mapping of classes to individuals, each individual is assigned its own unique class.

Implementation Detail 4.1 shows an example of a class produced by an individual during one of the experiments. In order to cut down on the overhead of loading and unloading the compiler for each individual, all files were given to the compiler at once, and all resulting classes were loaded in at once, rather than sequentially as the

*.*java* files were written out.

```
import problem.two_box.defaults.TwoBoxBase;
public class Ind_422_11 extends TwoBoxBase  {
    public double volumeDiff(int p, int q, int r,
                                              int x, int y, int z) {
     return r*r-y-r/x+q+r/1.0-x*z+q*y-1.0*r/1.0-r/z*p/y-p-r*q/y;
   }
}
```

Implementation Detail 4.1: Generated Class for The Two-Box Regression Problem

4.2 Evaluation

Once classes are loaded and assigned to their corresponding individuals it is possible to evaluate their individual fitness. The initial experiments were simple regressions where the fitness of an individual was a distance function. The fitness was the sum of absolute errors between the target function and the evolved function, given a set of inputs. Therefore, the function *volumeDiff* (when looking at the example described in Implementation Detail 4.1) would be called with a series of inputs. For example, *volumeDiff(1,2,3,4,5,6)* represents a possible method call within a test case. The returned value would be subtracted from what the function should return (i.e what the target function does return). The absolute difference is recorded. The sum total of the differences is the individual's fitness value. The lower the fitness value the better, a fitness of 0 is considered ideal.

To create a way of automating these inputs for determining an individual's fitness value, the JUnit testing framework was used. JUnit was used so that nothing would have to be added except some test cases for the evaluation. At a high level, the evaluator simply passes the individual's unique class to a test suite and records the observed difference. JUnit has the notion of a *test suite* and a *test case*. A test suite can contain zero or more test suites in addition to zero or more test cases. A test case is meant to be a single stand alone test, where as the test suite is a logical

collection of these tests. For determining an individual's fitness a single test suite is used containing several (10 or more) test cases.

Unfortunately, JUnit was not intended to be used in this way. It is built for writing unit tests for a particular class. However, for each of the n individuals within the population, there are n unique classes and class names. To overcome this each individual extended a single base class. For example, in the code snippet from Implementation Detail 4.1, the super class is a class called *TwoBoxBase*. By using a single base class, the JUnit tests were able to be written for the base class, and the evolved functions could be called through the base class using polymorphism.

```
package problem.two_box.defaults;
public class TwoBoxBase {
    public double volumeDiff(int p, int q, int r,
                             int x, int y, int z) {
        return -1;
    }
}
```

Implementation Detail 4.2: TwoBoxBase class

Using the base class exposed another unfortunate JUnit limitation – it is not good for testing a specific instance of a class. Most JUnit tests create and populate an instance variable before running the tests on that instance. Simply instantiating a class did not work in this case since we require a specific instance of *TwoBoxBase*, namely an instance of one of the subclasses representing an individual. To overcome this, the "Singleton Pattern" [9] was used, allowing for JUnit to have access to a repository of each individual's class. All individuals of the population were loaded into the *IndividualIterator*. Whenever the test suite was run, it would request the next individual's class from the IndividualIterator. All test cases run from within that test suite would request the current instantiated class. Using the IndividualIterator ensured that each individual (program) was run against the entire test suite and not just one test case or a subset of the test cases. The resulting difference is stored as the current individual under test's fitness. The fitness values are then used for

selecting individuals for breeding.

4.3 Evolution

Initially, the plan was to use ECJ [14] to perform all operations involved with the evolution, including breeding selection, sexual reproduction. The fitness evaluation must still be done as described in Section 4.2. In order to short circuit ECJ's execution path, the framework would need to interface with the ECJ framework, and give ECJ the individuals for evolution. Once ECJ had completed the selection and recombination of the individuals to form a new population, the population would be handed back to the framework. The framework would take the new population and perform the translation, compilation, and fitness evaluation for each individual.

After some investigation, incorporating ECJ proved quite difficult for the relatively simplistic crossovers that were to be implemented. Therefore, it was determined that the best solution was to implement the evolutionary part of it from scratch. Developing the evolutionary engine also made it easier to deal with problems that presented themselves during experimentation.

In order to keep the newly developed framework as flexible as possible, interfaces are used for the major components. Interfaces for fitness, selection and crossover were defined. The fitness interface described three methods: first, a way to set the fitness value; second, a way to get the fitness value; and finally, a comparison operator for determining whether a given fitness was better than another. The selection interface was just as simple, it allowed for a population to be set and, when prompted, would return a breeding set of individuals. In addition to this, the selection class could also be passed a crossover interface and asked to select a breeding set and return the newly formed offspring.

The crossover interface took only two individuals and returned an array of the resulting offspring. There is no special interface or sequence specifically set aside for

mutation. When mutation occurs, it occurs during the crossover step, after the new individuals have been formed. To avoid duplicate code, an abstract class was created which had a mutation function, where the mutation would be applied to any individual that was passed in. The mutation operation was applied randomly to zero or more integer values within the array. The integer values would be either incremented or decremented by one.

The abstract class did not implement any of the crossover interface. Initially only one-point and two-point crossover were going to be implemented for sexual recombination. However, simple GA style crossover had severe limitations, which we outline in the following sections.

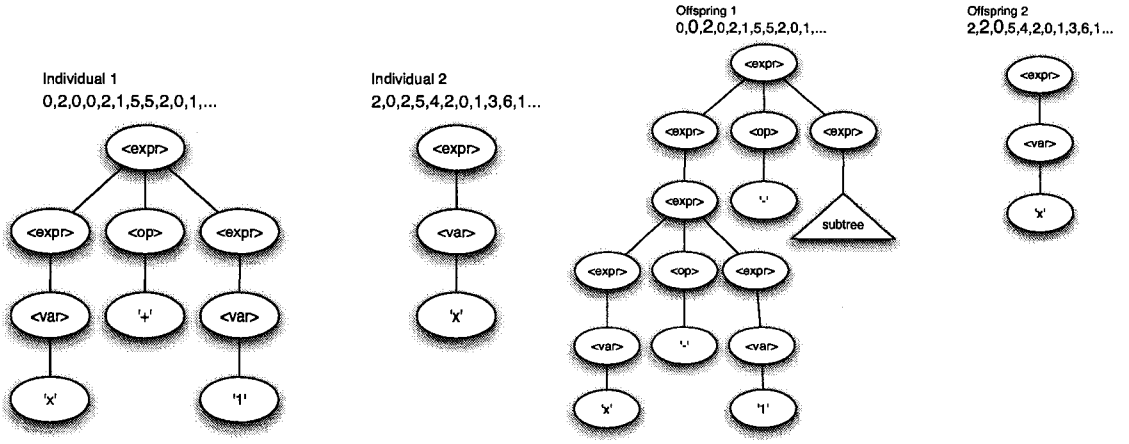
4.3.1 Traditional Crossover in a Grammar-Based System

As mentioned earlier in Section 2.3.3, one of the useful aspects of Grammar-Based Genetic Programming is that crossover is the same as crossover in Genetic Algorithms (Section 2.3.1). Parts of parents' genotype are exchanged with one another to produce the offspring. Through translation the new class is formed for each of the offsprings. The reason this translation step is necessary is no compiler would understand a string of bits or what they represent, and to ensure a syntactically correct individual.

With Genetic Programming (Section 2.3.2), there is no guarantee that a syntactically correct individual would be produced. The crossover could introduce incompatible types or strange constructs of multiple operators in a row (i.e 1++2). Unfortunately, the traditional method of crossing over the genotypes in a Grammar-Based System fails to ensure one of the most important aspects of heredity, not destroying the beneficial characteristics of the parents.

The genotype in a Grammar-Based System is different from most other Genetic Algorithms in that the translation is spatially dependent. The output of a particular location within the genome depends not only on the current value in that location,

but also all previous values.



(a) Parse Trees of Parents (b) Parse Tree of Offspring

Figure 4.5: Resulting Parse Trees for GA Style Crossover: Crossover location between 2 and 3 inclusive

Figure 4.5 shows the parse trees of two individuals selected for breeding, and their two offsprings. The genotype of each individual is printed across the top. The major draw back of using the GA style of crossover can be seen in Figure 4.5b, none of the subtrees from *Individual1* are preserved. For example, if the later part of *Individual1* were a good trait (i.e the +1 part) then *Offspring1* does not receive any of that benefit. As a matter of fact, that entire trait is now lost. At least with GP, during crossover only the subtrees that are exchanged are affected. The neighbouring subtrees do not become structurally different. The program or semantics of what those subtrees do may change, but the overall 'good' structure remains intact. The shape and structure of Figure 4.5b has been completely changed, the only parts of the tree that remain unchanged are those that came before the crossover point.

When attempting to converge upon a solution, traditional crossover is too disruptive. The change of a single node can completely alter the structure and semantics of the entire tree. There is no way to localize the effects. There exists the possibility that

a genotype has all of the correct values except for the starting location. To get around this a subtree preserving type of crossover was created, based very loosely on a crossover mentioned in passing by O'Neil and Ryan [17].

4.3.2 Sub-Tree Preserving Crossover

The idea behind a subtree preserving crossover came from the inability of allowing good subtrees to be passed from generation to generation. The actual sequence of integers could be passed down from generation to generation, but there was no way of ensuring that the sequence would be expressed in future generations. Since in an array $1...n$ the value (or production rule used) for position i is completely dependent on the values (production rules) of $1...(i-1)$, and i influences the values (production rules) of $(i+1)...n$. Because of the context sensitivity, code fragments (the phenotype) only remain through chance. Any useful code fragments that are produced after a specific position in the genotype are only guaranteed to remain there if all the positions before it remain the same.

For example, if an individual has a useful code fragment produced by positions $j...k$, the only way that an offspring is guaranteed to receive that code fragment is if it has an identical genotype as its parent form $1...(j-1)$. Any change to the genotype in the section $1...(j-1)$ may have a ripple effect that can destroy beneficial code fragments. These ripple effects can occur whenever the genotype is crossed-over during sexual recombination. The possibility of disrupting a subsequence by performing the crossover in the middle of a good sequence is no longer a concern. The problem is much worse, in that a crossover anywhere preceding a good subsequence has the possibility to disrupt the good sequence.

What is desired is a crossover that takes the phenotype into account, since the shape of the tree is the most important part for an individuals fitness. This approximates what GP gives you automatically, a way of preserving code fragments that occur after any crossover (see Figure 2.5 for an example). The semantics of the code fragment

may change but at the very least the syntax/structure remains intact. With a little work, it is possible to achieve this in grammar-based genetic programming. The key is to think of the genotype and phenotype together and not as completely separate entities.

Each element within the array can be thought of as a the root of its own subtree. Choosing an element for crossover, is similar to picking the root of a subtree. Replacing one subtree with another will ensure that there is no detrimental affect on any neighbouring subtrees. However, doing exchanging subtrees directly may result in a syntactically incorrect tree (the main draw back of GP). The incorrect syntax is avoided by translating the integer array of an individual using the grammar. As discussed, earlier a series of integers that represent a code fragment in one tree may not represent a fully formed code fragment, i.e. a subtree, in another. It is likely that the codons that form a subtree in one individual will not form a subtree in another.

```
1) xoverPoint1 = crossover point in parent1
2) xoverPoint2 = crossover point in parent2

3) offspring1 = genotype parent1
4) offspring2 = genotype parent2

5) build parse tree for offspring1
6) root1 = find root node of subtree indicated by xoverpoint1

7) remove production rules representing subtree rooted at root1

8) starting at xoverpoint2 read production rules of
     parent2 until root1 is a complete tree
9) add production rules to genotype after xoverPoint1

repeat 5-9 for offspring2
```

Implementation Detail 4.3: Pseudo-Code for Subtree Preserving Crossover

Figure 4.6 shows steps 1-6 of the code fragment in Implementation Detail 4.3. The

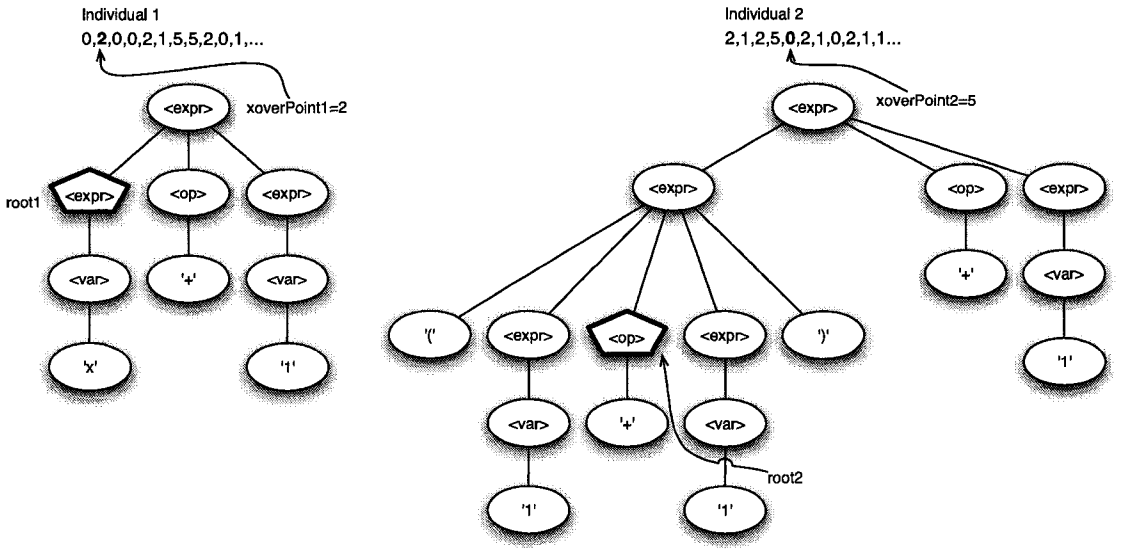


Figure 4.6: Subtree Preserving Crossover:Selecting crossover points

root is selected from the parse tree and isolated, the sequence for producing that subtree is removed from the genotype. Steps 1-6 can be seen in Figure 4.7. The next step is to build a new subtree rooted at the crossover point using the individual2's genotype. The integers used in producing the new subtree are added to the offspring's genotype after the crossover point (see Figure 4.8). Note that all subtrees around the crossover point have been preserved in contrast to the tree in Figure 4.5b.

With this approach it is possible to choose a crossover point to far to the right of the integer array. Once all values of the integer array have been read and the new subtree is still not complete (i.e. at least one leaf node is still a non-terminal) then the integers that were read are inserted into the new offspring's genotype. The attempt at creating a completed parse tree is aborted. So, when the parse tree is generated from the new integer array the subtrees from the parent are not preserved but a syntactically correct program is still produced.

One thing that subtree preserving crossover does fail to do, is allow for good subtrees to be exchanged. Since the production rules are so context sensitive, it is very

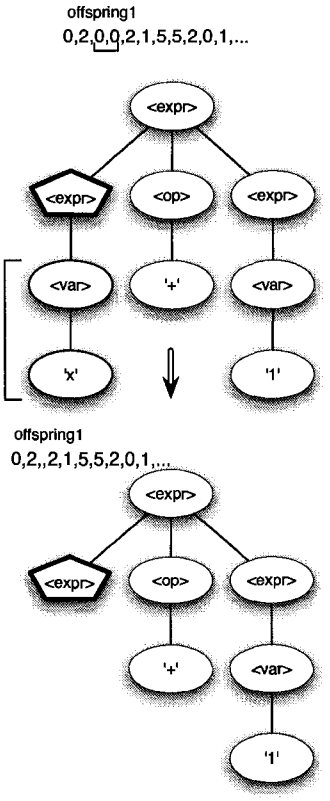


Figure 4.7: Subtree Preserving Crossover:Isolating Subtree Root and Removing Children

improbable that a subtree can be exactly recreated in a new tree. This is unfortunate since all good traits have to come about independently in each individual. With this new crossover it is at least possible for an offspring to retain any good features that its parents had.

4.4 Summary

This chapter presented the framework that implemented to evolve the behaviour of objects. The complete life cycle of an evolutionary run was examined. Each individuals within a population is translated from an integer array into a Java program.

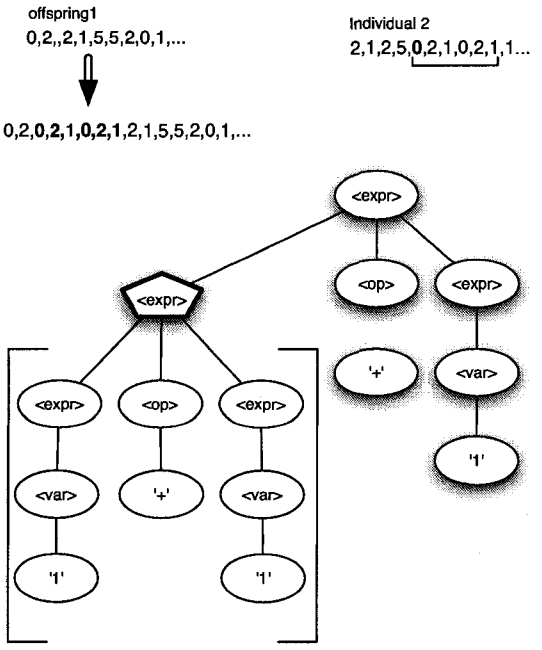


Figure 4.8: Subtree Preserving Crossover:Creating New Subtree

The program is compiled and loaded into the Java VM. The loaded class is evaluated by having a set of JUnit test cases run against it, this evaluation determines the individual's fitness. Once the fitness value of each individual has been determined a new population is generated.

The generation of new offspring involves the recombination of parents to produce a two new individuals. Two types of crossover used in the recombination step were examined. The traditional crossover used by O'Neill and Ryan and the introduction of a new subtree preserving crossover. The limitations with the traditional type of crossover and how it destroyed possibly good subtrees was discussed. Subtree preserving crossover does not have the limitations of the traditional crossover as it ensures that no subtrees are destroyed in the recombination process.

Chapter 5

Results

Three different experiments were conducted using the framework to show that it works to drive grammar-based evolution, and to validate the approach of evolving behaviour within objects. The initial experiment with symbolic regression was used to ensure that the new framework could evolve solutions. The strictly procedural experiment validated the life-cycle of the framework. The experiments to evolve an OO solution show that true object-oriented behaviour can be evolved.

The symbolic regression experiment is similar to the experiments done by O'Neill and Ryan [17] and performed as a baseline to demonstrate that the framework can produce acceptable results. The two-box problem is a non-trivial symbolic regression experiment to ensure that positive results can be obtained. The successful execution of this experiment showed that the framework operates correctly. Also examined in the two-box symbolic regression experiment is the difference between the new subtree preserving crossover and the traditional GA based crossover.

Building on the success of the symbolic regression example, experiments involving objects were performed. The 'Bank Account' and 'Polish Notation Calculator' experiments both involved evolving behaviour within objects as well as the new dynamic

representation of types within the framework. Both experiments used the augmentations to O'Neill and Ryan's grammar to evolve behaviour within objects.

5.1 Symbolic Regression Experiment

Symbolic regression is the attempt to approximate a well defined function using only a few reference points that lie on the curve. The solution space is as complex as the number of mathematical formulas that can be created by the operands given. Within this search space there are many functions that can approximate the curve defined by the target function. In the case of the two-box problem, the target function is $(h1*w1*l1)-(h2*w2*l2)$, the volume of one box minus the volume of another.

Setup

All evolved classes inherited from TwoBoxBase. All of the evolved classes overrode the volumeDiff method which was called from the JUnit tests used in the fitness evaluation. As mentioned earlier in Section 3.5, the class skeleton for the two-box experiment was programatically defined and did not have an xml representation. A grammar was defined and given to the framework. The definition can be seen in Appendix B.

The terminals in the grammar have all of the operations and variables needed by the class to evolve a solution. In the grammar box 1 is represented by x, y and z, and box 2 is represented by p, q and r. The rules, as well as the non-terminals, are identical to the grammar first shown in Section2.1.

Fitness

The two-box experiment is a minimisation problem, the lower the fitness the better. A fitness of 0 is ideal. The experiment had 10 JUnit test cases that were applied

Table 5.1: TwoBox Experiment Parameters

Parameter	Value
Population size	150
Integer array size	300
Num generations	500
Crossover type	Subtree-preserving
Mutation type	Point mutation
Crossover probability	100%
Mutation probability	6%

to each candidate individual. Each test case called the volumeDiff function with different values for each of the six parameters. The result returned was subtracted from the expected value. The absolute difference was added to the individual's fitness value. If the individual caused an illegal arithmetic operation or any other type of exception then that individual was assigned the maximum fitness possible[1]. The Java exceptions were tracked using built in functions within JUnit.

Discussion

Implementation Detail 5.1 shows the successfully evolved solution to the two-box problem. In this case, in generation 283, individual 2 had the correct solution and the program ended.

```
import problem.two_box.defaults.TwoBoxBase;
public class Ind_283_2 extends TwoBoxBase  {
  public double volumeDiff(int p, int q, int r, int x, int y, int z) {
    return r*q*p-x*z*y;
  }
}
```

Implementation Detail 5.1: Two-Box Evolved Solution

Initially, when running the two-box experiment the population was suffering from

[1]Maximum fitness value was Double.MAX_VALUE

premature convergence, i.e. it was finding a poor solution quickly and could not evolve a better one. Premature convergence was due to so called super-individuals. A super-individual is relative and refers to individuals that are locally optimal. In other words they are the best of the current population and take over due to their relatively good fitness. Tournament selection was introduced into the experiment to try and combat the super individuals. In tournament selection, n individuals are chosen at random and the best individual is selected as a parent for reproduction. Another tournament of n individuals is staged to select the second parent for breeding. Individuals for tournaments are chosen at random, the randomness helps avoid having one or a group of individuals dominate the selection. By contrast proportional selection causes an individual that is twice as fit as another individual to be chosen on average twice as often.

After changing the selection algorithm to tournament selection the results where not much better. Relatively little increase in fitness caused the crossover to be examined. Upon examination, the traditional crossover was found to be destructive, as outlined in Section 4.3.1. Once the new subtree preserving crossover (discussed in Section 4.3.2) was implemented convergence occurred more frequently and the average fitness improved.

Over 50 runs, the new subtree preserving crossover outperformed the traditional crossover. As shown in Table 5.2 the performance of the new subtree preserving crossover outperforms traditional crossover in average fitness, average percentage gain, and most importantly, in number of solutions found.

Table 5.2: Performance of Traditional Crossover vs. Subtree Preserving Crossover (Over 50 runs)

Crossover Type	Traditional	Subtree Preserving
Solutions Found	18	27
Avg. Fitness	42.13	31.58
Avg. Percentage Gain	90.27%	94.00%

The average percentage gained was a way of normalising the results of different runs,

since the optimal fitness is fixed at 0 but the worst fitness has no ceiling. In order to properly examine how each run succeeded in finding a solution, we need to take into account where the population started. An experiment where the individuals start with a very poor fitness and improve it significantly is better than one where individuals start out better and make little improvement to their fitness. Since the initial individuals are generated randomly, the crossover methods should be examined based on their performance gain and not based on how well the initial random population was created. Algorithmically percentage gain was calculated as follows $1 - \frac{\text{best of last generation}}{\text{best of first generation}}$

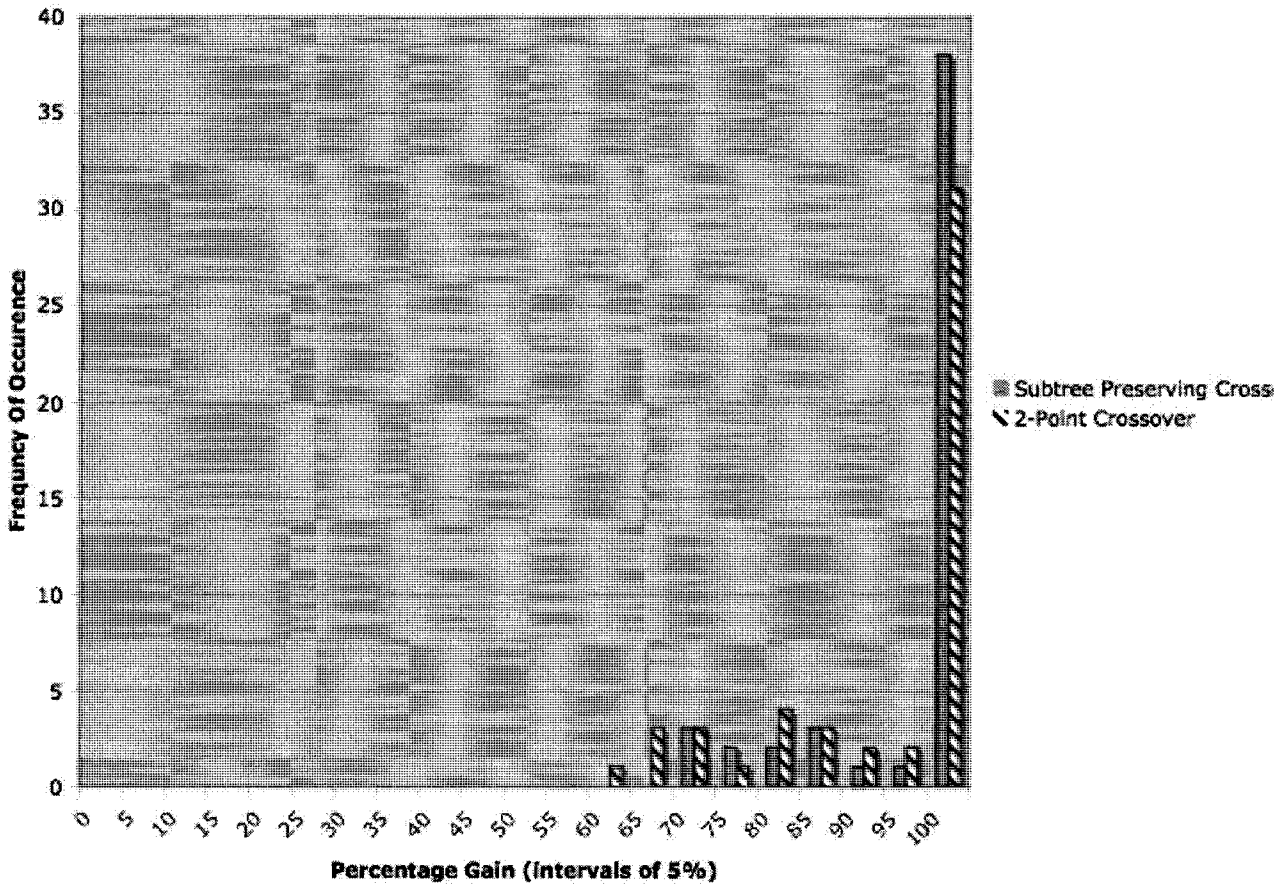
When examining the percentage gain there was no significant difference between the two types of crossover, although experimentally the results looked better. Subtree preserving crossover never did worse than a 66 per cent improvement. With the new subtree preserving crossover, solutions were found 54 per cent, as opposed to 36 per cent of the time with the traditional crossover. To show that the performance of the subtree preserving crossover was better in a statistically significant way, the ranked *Student's t-test*[2] was performed on the percentage gained per run. Subtree preserving crossover was better than the traditional crossover with a significance of 0.001.

Two-Box Regression Summary

The two-box experiment showed that the newly implemented grammar-based evolutionary framework can successfully evolve a solution. The solution was a fully functional, syntactically correct Java class. For the first time a Java class was created compiled and loaded during the process of a evolutionary run.

[2] A t-test is any statistical hypothesis test in which the test statistic has a Student's t distribution if the null hypothesis is true. It is applied when the population is assumed to be normally distributed but the sample sizes are small enough that the statistic on which inference is based is not normally distributed because it relies on an uncertain estimate of standard deviation rather than on a precisely known value.[6]

Graph 5.1 Comparison of Percentage Gain



Also highlighted was the improved performance gained by using the subtree preserving crossover. This new crossover method evolved a correct solution more often than the traditional style of crossover used in grammar-based evolution. In addition to finding the solution more often, the new crossover also was better at improving the initial population than traditional crossover.

5.2 Object-Oriented Experiments

The framework was extended with new grammar components [3] and a new XML specification for skeletons [4] from the regression experiments to allow for object-oriented functionality within the grammar. The use of objects had never been implemented within a grammar-based framework, and certainly never with the extensive use of reflection to allow for the grammar to be dynamic. There has been some work with using strongly typed genetic-programming for evolving objects [20] . However this work is to test objects, not to leverage objects for solutions. The following experiments enable GP to actually create objects and evolve their behaviour, not just use other objects.

The approach discussed in both "Evolutionary unit testing of object-oriented software using strongly-typed genetic programming"[25] and "A Genetic Programming Approach to Automated Test Generation for Object-Oriented Software " [20] does not easily expand to allow for other control structures to be introduced. For example the tests done by Seesing and Gross are simply a list of instructions that are meant to cover the system under test. There are no multiple code paths, only the single one generated.

The paper written by Abbott et al "Guided Genetic Programming" [2] has the most impressive results for generating objects. Abbot et al manage to create a sort program acting on a Java *ArrayList*. The evolution is done entirely in memory using

[3] Detailed in Section 3.3.1
[4] Defined in Section 3.5.1

reflection. The problem with reflection is that it is extremely slow[5] and very unintuitive to perform complex function calls. For instance it would not be very easy to invoke nested methods. Implementation Details 5.2 and 5.3 highlight the differences.

```
foo.method1(anObject.toString(),anotherObject.twoInts(1,2));
```

Implementation Detail 5.2: Regular Java Code: Method Call

```
ArrayList<Object> operands = new ArrayList<Object>();
operands.add(toStringMethod.invoke(anObject,
                necessaryParamsArray1));
operands.add(twoIntsMethod.invoke(anotherObject,
                necessaryParamsArray2));
fooMethod.invoke(foo, operands);
```

Implementation Detail 5.3: Java Reflection Code: Method Call

The approach used by the extended framework was to write a syntactically correct Java program and compile it into byte code that could then be loaded and executed by the VM. Compiling and loading the Java source code has a high overhead, but once compiled, it will run as fast as any other Java code. Another issue not tackled in any of the literature is using objects in a 'true' object-oriented fashion. Currently all attempts have not used objects to their fullest capacity. That is, either they are being used as simply a conduit of access to a method (simply calling the method) or, as was the case with Abbott, treating the object as if it has global instance variables. There is no thought given to encapsulation or the powerful ability to pass both the state and behaviour of objects around.

5.2.1 Bank Account Experiment

The bank account experiment creates a problem where true object oriented design can be used. The problem is a basic but illustrative problem for withdrawing funds

[5]Reflection is approximately 100 times slower than a direct method call [22].

from a bank account. Every account can have different types of transactions on it, and each transaction type can have a different set of fees and rules. Implementation Detail 5.4 shows the life-cycle of a transaction. The transaction has to make sure that it is not breaking any, rules and then if valid – apply the transaction.

```
Transaction::doTransaction(bankAccount, amount)
if(transaction rules for this transaction and bankAccount = OK)
    perform transaction of amount on bankAccount
else
    do nothing
```

Implementation Detail 5.4: Transaction Life-cycle

Setup

For the experiment, a bank account class and corresponding withdrawal fees were created. The transaction class is what will be evolved, specifically a withdrawal transaction. The skeleton that defines the withdrawal class can be seen in Apendix C.1. The transaction has a set of rules that determine whether the transaction is valid. The rules are represented by a transaction rules class. Each transaction rule class has a method which takes an account and an amount as parameters, and returns whether the transaction can be applied. There are also fees associated with the transaction that are contained within the bank account itself.

The specific method that is being evolved is doTransaction, which takes in a bank account and the amount of the transaction. The evolved program must somehow perform the life-cycle detailed in Implementation Detail 5.4. More formally, the message sequence chart of the ideal solution is shown in Figure 5.1. This shows the messages that are needed to be called, as well as the conditional statement that insures a withdraw occurs only if the bank account has enough funds to complete the transactions.

The doTransaction method's skeleton does not contain any local variables other than the BankAccount and amount passed in through the method call. The program must

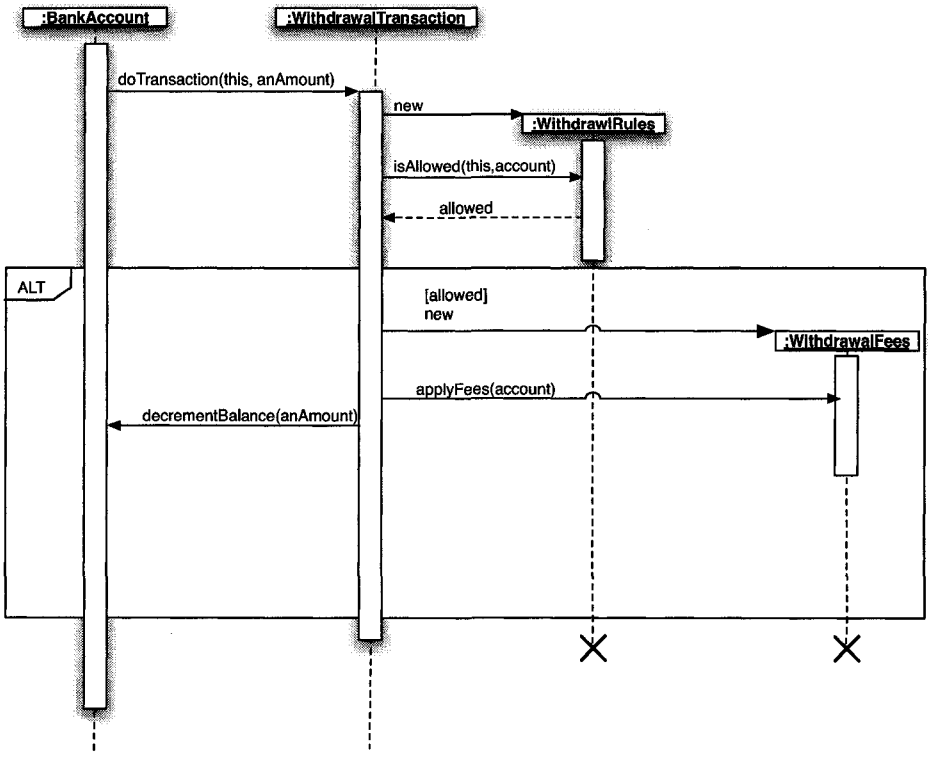


Figure 5.1: Message Sequence Chart For A Bank Account Solution

instantiate both a *WithrawalRules* and a *WithdrawalFees* variables. The variables are then used to aid in the computation. The new variables are stored within the context so that future method calls can use these variables.

Fitness

The fitness function used for the Bank Account Example is similar to the fitness function in the Two-Box experiment. The fitness function again is a set of JUnit test cases. However in this experiment a larger fitness is better. An individual with a fitness value of 10 is better than an individual with a fitness value of 6.

For example, Implementation Detail 5.5 shows one of the tests cases within the

Table 5.3: Bank Account Experiment Parameters

Parameter	Value
Population size	100
Integer array size	300
Num generations	500
Crossover type	Two-point crossover
Mutation type	Point mutation
Crossover probability	60%
Mutation probability	10%

experiment's test suite. The example test case is typical of all of the test cases, a bank account's balance is set and then an attempt is made to withdraw an amount. The amounts and balances for each transaction are different with each test. Some transactions try to remove more than the account's balance.

```
public void test8() throws Exception {
    account.balance = 99;
    account.performWithdrawl(62);

    if(account.balance == 36) {
        InvokationCounter.incrementCount(150);
    } else {
        int diff = Math.abs(36 - account.balance);
        InvokationCounter.incrementCount(-diff);
    }
}
```

Implementation Detail 5.5: Sample Test Case

If the test succeeds and the correct amount is withdrawn from the account and the fees applied, then the fitness is augmented by 150. The individual is penalised for every dollar that the balance differs from the ideal. The absolute difference times two is subtracted from the individual's fitness value. To avoid the fitness being a so-called *step-function* [6] the individual is also rewarded for methods called. Any method that

[6] A step-function is a function that is either correct or incorrect.

affects the state of the bank account and is called will add 1 to the individual's fitness value. This will allow the individual to gain some benefit for calling the state altering method, even if it does not alter the balance correctly. As stated above, there are certain degrees of correctness, but the largest difference in fitness by far occurs when an individual's program calls the decrementBalance function.

Due to the fact that decrementBalance is a function whose implementation remains consistent, there is only one beneficial outcome for an individual, if its program uses the method as intended. If the method is called more than once, the fitness will be heavily impacted. The same is true if it is never called. In order for the individual to receive any benefit from this method call, it must call decrementBalance only once. The small window available for calling the function properly is what causes the 'step' in the fitness function.

Discussion

A sample evolved individual is shown in Implementation Detail 5.6. The sample individual had a fitness value of 1299 out of a possible 1352. The individual has evolved the ability to check whether the transaction is valid, and then decrement the account balance. However, the only thing that the individual is doing incorrectly is applying the banking fees in all cases. The described individual shows something interesting: the second call to apply fees is not to *WithdrawalFees*, but to its base class *BankingFees*. BankingFees does not affect the value of an account balance since it is simply a place holder and has no specific fees associated with it. So, in the majority of test cases where the withdrawal is allowed, this transaction object returns the correct result. Only on transactions where the withdrawal amount is too great does the object produce the incorrect result. Even when the transaction is not allowed, the account balance is only off by a single dollar. The individual has near perfect fitness.

```
import problem.bankaccount.defaults.Withdrawl;
import problem.bankaccount.defaults.BankAccount;
import problem.bankaccount.defaults.WithdrawlFees;
import problem.bankaccount.defaults.WithdrawlRules;
import problem.bankaccount.defaults.BankingFees;
import problem.bankaccount.defaults.BankingRules;
import java.lang.Integer;
public class Ind_350_1 extends Withdrawl {


    public void doTransaction(BankAccount account, Integer anAmount) {
        BankingFees NEW_BankingFees_1 = new BankingFees();
        WithdrawlFees NEW_WithdrawlFees_1 = new WithdrawlFees();
        WithdrawlRules NEW_WithdrawlRules_1 = new WithdrawlRules();
        NEW_WithdrawlFees_1.applyFees(account);
        if(NEW_WithdrawlRules_1.transactionAllowed(this,
                                                    account) == 1){
            NEW_BankingFees_1.applyFees(account);
            account.decrementBalance(anAmount);
        } else {
            ;
        }
    }
}
```

Implementation Detail 5.6: Sample Evolved Transaction Individual

```
...
public void doTransaction(BankAccount account, Integer anAmount) {
    BankingFees NEW_BankingFees_1 = new BankingFees();
    Integer NEW_Integer_2 = 1;
    Integer NEW_Integer_3 = 1;
    Integer NEW_Integer_4 = 1;
    WithdrawlFees NEW_WithdrawlFees_1 = new WithdrawlFees();
    ;
    if(account.decrementBalance(anAmount) == 1){
        ;
        NEW_Integer_2.compareTo(NEW_Integer_3);
        ;
        NEW_BankingFees_1.hashCode();
        NEW_WithdrawlFees_1.hashCode();
    } else {
        ;
        account.decrementBalance(NEW_Integer_4);
   }
}
...
```

Implementation Detail 5.7: First Generation Individual

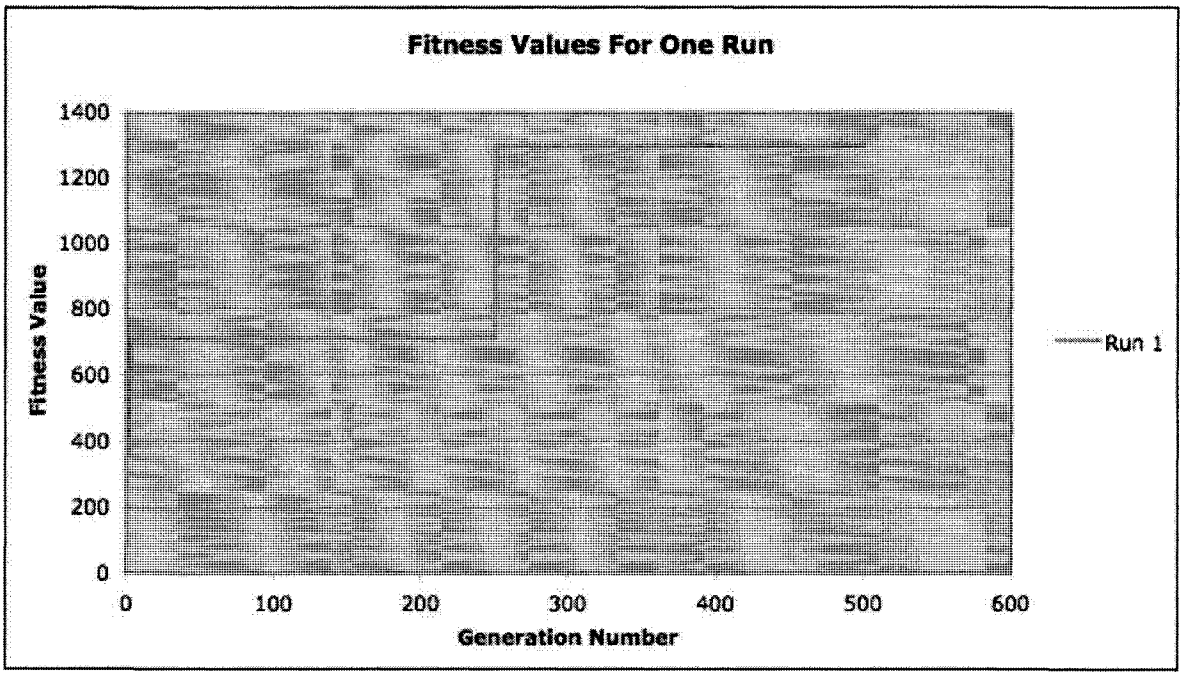
Comparing the successful individual in Implementation Detail 5.6 to an earlier individual (Implementation Detail 5.7[7]) shows a marked improvement between the first generation and an individual from later generations. Of note is the fact that the evolution has the ability to create new instances of any class that it knows about, it is possible to call methods with incorrect parameters. For example, in the else statement, the decrementBalance method call is passed one of the newly created Integer variables. This is illustrative of the complex search space that GBOOGP must navigate. The number of variables can grow and shrink depending on the individual. Some individuals may create many more WithdrawalFee objects than others. The evolution has the ability to greatly increase the number of false parameters that it can use. Yet with all of this potential for noise, the method evolved is a close approximation to the ideal solution presented in Figure 5.1.

Overall, later individuals displayed a higher fitness than those in earlier generations. However, the fitness function did not strictly monotonically increase. Rather than allowing for small incremental improvements to be made towards a local optimum, large jumps occurred. The Graph 5.2 shows an example of the bank account evolution.

The increase in fitness occurs in three distinct steps rather than small incremental changes. The steps are due to the fact that the fitness function did not allow for a gradual improvements. Either the program called decrementBalance or it did not. Once a program calls decrementBalance with the correct parameter, the fitness is greatly increased since most of the test cases the withdrawal amount is valid. Calling decrementBalance greatly reduces the difference between the ideal account balance and the actual. The next large jump is due to applying the account fees. Applying the account fees allows the individual to receive bonuses for getting the correct solution. The individuals shown in Implementation Details in 5.6 and 5.7 have both reached this first incremental step, as they both call decrementBalance passing in *anAmount* to get close to the correct result. The individual in Implementation Detail 5.6 has attained a second step within the function by calling the applyFees method.

[7]Class name and imports removed as they are the same as the previous individual.

Graph 5.2 Fitness Values Over One Run



The bank account experiment was chosen to show that an OO solution could be evolved and that the starting individuals could be improved upon. All but two of the 50 experiments showed improvement between the first and final individuals. The two experiments that showed no improvements began with individuals that had already reached the first fitness step, similar to the individual shown in Implementation Detail 5.6.

Bank Account Summary

The bank account experiment validated the new extensions to the grammar. The addition of the Type, Variable and Method components (discussed in Section 3.3.1) allowed the grammar to be independent of the problem domain. The list of all possible method calls did not need to be enumerated by the grammar. The skeleton specified the classes that could be used and the framework handled producing syntactically correct Java programs.

This experiment was also the first truly object-oriented experiment attempted. The two-box experiment was strictly procedural and did not involve method calls or any objects. The solution to this experiment had to use double-dispatching and several other method calls to produce the correct behaviour. For the first time an evolved object's encapsulation was necessary for a correct solution.

5.2.2 Reverse Polish Notation Calculator

A Revers Polish Notation (RPN) places mathematical operators after all of the calculations operands. For example, '3 + 4' in regular 'infix' notation is written as '3 4 +' in RPN. A RPN calculator naturally lends itself to being stack based. The operands are pushed onto the stack, one at a time. The operator then will pop the required number of operands off that stack and execute the calculation. The result of the calculation is pushed back onto the stack.

The experiment involves evolving three methods for a reverse polish notation calculator. The evolved methods will implement addition and subtraction for a RPN calculator. What is required from the three methods is placing a set of operands onto the calculator's stack and then delegating the addition and subtraction to another class. The 'Strategy Pattern'[9] is implemented through the use of delegation.

Setup

To implement the RPN calculator the same grammar was used as in the previous bank account example. The only new XML file that had to be specified was a Skeleton definition for the RPN calculator class. The skeleton xml file can be seen in Appendix C.2.

Classes were created to perform the addition and subtraction operations. Both classes implemented a Transaction interface. The Transaction interface defines a single method called *performTransaction* and takes a RPN Calculator and a stack as parameters. What occurs within the execute action is specific to the implementer of Transaction.

The class implementing addition takes the stack and adds the two operands before pushing the result onto the stack. Similarly the subtraction transaction will subtract the first operand from the second. The resulting difference is pushed onto the stack.

The three methods evolved within the RPN calculator were *enterNumber*, *add* and *subtract*. Implementation Detail 5.8 shows a proper implementation of the RPN calculator. The enterNumber method pushes the integer parameter onto the calculator's stack. The add and subtract methods pass the stack and the calculator to their respective transactions.

Table 5.4: RPN Calculator Experiment Parameters

Parameter	Value
Population size	100
Integer array size	300
Num generations	500
Crossover type	Two-point crossover
Mutation type	Point mutation
Crossover probability	60%
Mutation probability	10%

Fitness

The fitness was a series of JUnit tests checking the accuracy of the calculations. A larger fitness was considered better for this experiment. Points where awarded depending on the values on the stack. For each correct state within the evolved calculator, the individual's fitness was increased. The fitness test suite had three distinct sections, one for each method evolved. In all 11 test cases were created. Three test cases focused exclusively on enterNumber, while both addition and subtraction had four test cases each.

In order to examine the state of the calculator's stack reflection was used to gain access. Since the stack variable was an actual instance of the evolved class there was no way of accessing it through the base class. Using the *getDeclaredField* method it was possible to get a hold of the instances stack.

The three test cases focusing on the enterNumber function examined the stack. Points were awarded based on the size of the stack and the values on the stack. For example if enterNumber was called passing in an integer value of 3 and then again with a value of 4. The contents of the stack should be 3 4 and the stack should have a size of 2. For a maximum fitness the observed stack had to be exactly what was expected. However, two points were awarded if the stack was the correct size and an additional two points were added to the fitness value if any of the expected

```
public class RPNCalcSolution extends BaseCalculator {
  public CalcStack stack = new CalcStack();


  public void enterNumber(Integer num) {
    stack.calcPush(num);
  }

  public void add() {
    AddTransaction NEW_AddTransaction_1 = new AddTransaction();
    NEW_AddTransaction_1.performTransaction(this,stack);
  }
  public void subtract() {
    SubTransaction NEW_SubTransaction_1 = new SubTransaction();
    NEW_SubTransaction_1.performTransaction(this,stack);}
  }
}
```

Implementation Detail 5.8: RPN Calculator Implementation

values were on the stack.

The test cases for addition and subtraction followed a similar format. Maximum points were awarded if the calculator's stack had the correct value on top. Two points were awarded if the AddTransaction or SubTransaction classes were instantiated. The two points rewarded individuals for being closer to the ideal solution than individuals that did not use or instantiate either of the transaction classes.

Discussion

The grammar based framework successfully evolved individuals for performing addition and subtraction using an RPN calculator. In fact all 50 runs produced an optimal individual. The increased success rate was potentially due to the ability to reward smaller incremental improvements. By focusing specifically on each method it

was possible to give more value to incremental improvements. These small incremental boosts in fitness value allowed for the fitness function to be more monotonically increasing.

Not surprisingly the ability to push values onto a stack was first the first step evolved. The addition and subtraction could not take place without the proper operands on the stack. It was possible for the individuals to begin instantiating transactions before all of the pieces where in place. For example, the individual in Implementation Detail 5.9 managed to interact with the stack but not properly. The stack pushed a newly created integer onto the stack and then immediately called add. The individual received an increased fitness for both putting something on the stack and for instantiating a class that implements the Transaction interface.

An individual with a maximum fitness is shown in Implementation Detail 5.10. The individual properly pushes the passed in integer onto the stack in the *enterNumber*. When either *add* or *subtract* are called, the proper transaction class is instantiated and called. An interesting detail for this individual is the fact that in the subtract method, the *performTransaction* method is called in the else part of the if statement. The subtract method always works because it is highly improbable that the SubTransaction's hash code ever equals 1. Because of this the if statement will in all likelihood be false the else statement is always executed, thus correctly implementing the algorithm.

The RPN calculator used the same grammar as was defined in the bank account experiment. By using the same grammar for both experiments the flexibility of not defining all terminals in the grammar is demonstrated. If no elements had been added to O'Neill and Ryan's original grammar then the grammar would have had to be rewritten to include all possible terminals for the new problem domain. The classes listed in the skeleton file determined what classes were available for the evolution to use. The grammar simply ensured that the implementation was syntactically correct.

```
public class Ind_1_1 extends BaseCalculator {
  public CalcStack stack = new CalcStack();


  public void enterNumber(Integer num) {
    AddTransaction NEW_AddTransaction_1 = new AddTransaction();
    Integer NEW_Integer_2 = 1;
    stack.calcPush(NEW_Integer_2);
    NEW_AddTransaction_1.performTransaction(this,stack);
  }
  public void add() {
    AddTransaction NEW_AddTransaction_1 = new AddTransaction();
    Integer NEW_Integer_1 = 1;
    if(NEW_Integer_1.intValue() == 1){
      ;
      stack.capacity();
    } else {
      this.getResult();
      NEW_AddTransaction_1.hashCode();
    }

  }
  public void subtract() {
    AddTransaction NEW_AddTransaction_1 = new AddTransaction();
    ;
    NEW_AddTransaction_1.hashCode();
    ;;
  }
}
}
```

Implementation Detail 5.9: RPN Calculator Implementation

```
public class Ind_1_1 extends BaseCalculator {
  public CalcStack stack = new CalcStack();


  public void enterNumber(Integer num) {
    if(stack.calcPush(num) == 1){
      this.hashCode();
    } else {
      ;
    }
  }
  public void add() {
    Integer NEW_Integer_1 = 1;
    Integer NEW_Integer_2 = 1;
    AddTransaction NEW_AddTransaction_1 = new AddTransaction();
    NEW_AddTransaction_1.hashCode();
    if(NEW_AddTransaction_1.performTransaction(this,stack) == 1){
      NEW_Integer_1.compareTo(NEW_Integer_2);
    } else {
      ;
      stack.size();
    }
  }
  public void subtract() {
    SubTransaction NEW_SubTransaction_1 = new SubTransaction();
    if(NEW_SubTransaction_1.hashCode() == 1){
      ;
    } else {
      NEW_SubTransaction_1.performTransaction(this,stack);
    }
  }
}
```

Implementation Detail 5.10: RPN Calculator Implementation

RPN Calculator Summary

This experiment showed the flexibility of the entire framework. The same grammar that was used in the bank account experiment was used in this experiment. The only file that had to be created has the skeleton file defining the Calculator class.

The RPN Calculator evolved the correct behaviour of three methods simultaneously. The add and subtract methods had to instantiate the correct class and then call the correct method to achieve the desired result.

5.3 Summary

The three experiments performed increased in complexity and difficulty. The two-box experiment was a validation that the framework could evolve solutions. The two object-oriented experiments were designed to show that it was possible to evolve true object-oriented behaviour. Both experiments validate the augmented grammar introduced to allow for object evolution.

Both of the discussed crossovers were evaluated during the two-box regression experiment. The new subtree preserving crossover outperformed the traditional method of crossover used by O'Neill and Ryan. The first experiment also validated the framework that was developed. The construction of Java classes, their compilation and instantiation was all validated. JUnit tests were run on the resulting classes to evaluate the programs.

The same infrastructure was used for the two object-oriented experiments. The OO experiments leveraged the new grammar components to dynamically determine method calls. The bank account experiment showed the use of double-dispatching when evolving a solution. The evolved solution had to pass itself as a parameter to another class to produce the correct behaviour.

The final experiment developed the addition and subtraction functionality for a Reverse Polish Notation calculator. The calculator used a stack to keep track of entered operands. This stack had to be passed to addition and subtraction operators by the calculator. The evolved addition and subtraction methods also had to instantiate the correct operator before passing the stack to them.

Through these experiments the framework was validated. The infrastructure in place was shown to be flexible and scalable. Also for the first time an EC system was able to evolve true object-oriented behaviour.

Chapter 6

Conclusion

The extra level of abstraction offered by objects is a benefit to humans. Having partial solutions that can be combined into a complete solution is very beneficial when designing a program. The OO paradigm has not been explored in EC, but can give an evolutionary algorithm the same benefits. Until now nothing has been done to truly leverage OO design and abstraction. The literature involves using objects in a strictly procedural way. The object being evolved cannot be passed around, or use delegation to its advantage. Grammar-based object-oriented genetic programming deals directly with this issue.

Grammar-based object-oriented genetic programming (GBOOGP) is heavily influenced by biological systems. Protein translation is a perfect analogy of how a program is generated using GBOOGP. The translation from DNA $\rightarrow$ RNA $\rightarrow$ Protein is very similar to Integer array $\rightarrow$ Parse tree $\rightarrow$ Executable program. Along with a very strong biological foundation, GBOOGP draws heavily from other areas in evolutionary computation.

Genetic algorithms provide the crossover operators and genotype representation for a GBOOGP individual. The GBOOGP's genotype is an array of integers, the grammar is used to translate this array into a parse tree. The parse tree shows all of the

production rules applied during the translation phase. The terminals of the parse tree are what constitute the phenotype of the individual, the executable program. Crossover is performed on this array rather than the generated parse tree. The traditional crossover used by O'Neill and Ryan [17] failed to preserve the subtrees of parents in the produced offspring. To ensure that beneficial subtrees could be passed from one generation to the next I created a subtree preserving crossover operator. The subtree preserving operator affected a smaller area, only the subtree where the crossover was occurring was altered. All other subtrees remained unchanged.

GBOOGP also borrows heavily from strongly-type genetic programming (STGP) to ensure that the produced programs are indeed syntactically correct. The STGP provides a methodology for looking up type and variable information. Without type information it would not have been possible to produce syntactically correct object-oriented code. The type of an object determines what messages it can understand.

In order to properly implement the generation of parse trees for objects, the initial grammar described by O'Neill and Ryan [17] was augmented. The new artefacts within the grammar allowed for the framework to dynamically create message calls, depending on what objects and classes the evolution was aware of. The new types simplified the grammar and increased the the flexibility. The classes and objects that the evolutionary process knew of was defined within the skeleton. The skeleton was a wrapper around the evolution. Skeleton's were defined using an XML file.

The experiments performed highlighted the fact that it is possible to evolve the behaviour of objects and allow them to use these more advanced OO constructs. For the first time objects are treated as first class citizens. Previous implementation blurred the line between state and behaviour. The encapsulation of an object was non-existent, the object was simply a collection of variables and methods loosely coupled. By compiling and loading evolved Java classes it was possible to obtain full encapsulation. Full encapsulation allowed for an object to be passed around and have more of an ability to delegate without knowledge of the internal workings of

other objects.

6.1 Future Work

The field of Grammar-Based OOGP is very new and has presented a lot of challenges, the majority of which revolve around specifying a non-restrictive grammar that will still yield a syntactically correct program. The implementation of a new framework allowed for some initial experiments to be run. The complexity of the evolved objects could be increased by introducing new control structures.

Only a small part of the Java specification was implemented, if statements and method calls were the two main constructs. The grammar used by the framework could increase in complexity until it supports the entire Java language. Implementing support for for-loops, return statements and primitive data types would allow for much more complex behaviours to be evolved.

Due to the success of the RPN calculator experiment over the Bank account experiment (largely due to the fitness function), more work could be done examining fitness functions. Avoiding step functions is the key to evolving more complex objects. Through more thorough inspection of state and the actual sequence of methods better fitness functions should be attainable.

With a better fitness function it would be possible to identify beneficial subtrees within a parse tree. These parse trees could then be passed down from generation to generation. Similar to some work that is being done with automatically defined functions in GP [12].

Experiments that involve a grammar to define a pattern for the evolving object to use would be interesting. The pattern could be though of as a scaffolding from which the object or objects' behaviour can be evolved. Using patterns could be advantageous to evolving more complex behaviours much in the same way as patterns are useful to humans. Patterns define a language for talking about solutions. If the way that

objects should interact could be defined then the specifics of the solution are that much easier to determine.

Using patterns and evolving more complex behaviours would lead naturally into evolving the interfaces for objects. The idea would allow an object to determine how it wants to interact with the outside world, and its own internal representation. Producing an object that has correct behaviour with little to no direction is difficult, to say the least. The problem domain for what the object would be doing would have to very specific. Another issue would be the fitness evaluation of the object's behaviour.

Bibliography

[1] Evolvable systems group: Automated antenna design. 15 September 2008 http://ti.arc.nasa.gov/projects/esg/research/antenna.htm, 0 2005.

[2] R. Abbott, J. Guo, and B. Parviz. Guided genetic programming. In *The 2003 International Conference on Machine Learning; Models, Technologies and Applications (MLMTA'03)*, las Vegas, 23-26 June 2003. CSREA Press.

[3] R. J. Abbott. Object-oriented genetic programming, an initial implementation. In *Procceedings of the Sixth International Conference on Computational Intelligence and Natural Computing*, Embassy Suites Hotel and Conference Center, Cary, North Carolina USA, Sept. 26-30 2003.

[4] A. Agapitos and S. M. Lucas. Learning recursive functions with object oriented genetic programming. In P. Collet, M. Tomassini, M. Ebner, S. Gustafson, and A. Ekárt, editors, *Proceedings of the 9th European Conference on Genetic Programming*, volume 3905 of *Lecture Notes in Computer Science*, pages 166–177, Budapest, Hungary, 10 - 12 Apr. 2006. Springer.

[5] N. A. Campbell, J. B. Reece, and L. G. Mitchell. *Biology.* Addison-Wesley Longman, Menlo Parks, CA, USA, 5 edition, 1999.

[6] S. Christensen and M. Wineberg. Statistics for ec. Workshop GECCO 2007, July 2007.

[7] A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing.* SpringerVerlag, 2003.

[8] K. Frenken, L. Marengo, and M. Valente. *Computational Techniques For Modelling Learning In Economics*, chapter Interdependencies, Nearly-Decomposibility and Adaptation, pages 145–165. Kluwer Academic Publishers, 1999.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Professional, 1995.

[10] Z. Ivan. Symbolic regression - an overview. 15 September 2008 http://www.it.lut.fi/mat/EcmiNL/ecmi35/node70.html, 07 2005.

[11] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection (Complex Adaptive Systems).* The MIT Press, December 1992.

[12] J. R. Koza, D. Andre, and V. Scholar. Evolution of both the architecture and the sequence of work-performing steps of a computer program using genetic programming with architecture-altering operations. In *in Advances in Genetic Programming II*, pages 50–60. MIT Press, In Press, 1995.

[13] L. Li, N. Krasnogor, and J. Garibaldi. Automated self-assembly programming paradigm: Initial investigations. In *EASE '06: Proceedings of the Third IEEE International Workshop on Engineering of Autonomic & Autonomous Systems*, pages 25–36, Washington, DC, USA, 2006. IEEE Computer Society.

[14] S. Luke, L. Panait, G. Balan, S. Paus, Z. Skolicki, E. Popovici, J. Harrison, J. Bassett, R. Hubley, and A. Chircop. Ecj: A java-based evolutionary computation research system. 15 September 2008 http://cs.gmu.edu/ eclab/projects/ecj/, 09 2007.

[15] M. Mitchell. *An Introduction to Genetic Algorithms.* MIT Press, Cambridge, MA, USA, 1998.

[16] D. J. Montana. Strongly typed genetic programming. Technical Report #7866, 10 Moulton Street, Cambridge, MA 02138, USA, 7 1993.

[17] M. O'Neill and C. Ryan. *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language.* Kluwer Academic Publishers, Norwell, MA, USA, 2003.

[18] N. Pillay. An investigation into using genetic programming as a means of inducing solutions to novice procedural programming problems. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1781–1782, New York, NY, USA, 2005. ACM.

[19] N. Pillay and C. K. A. Chalmers. A hybrid approach to automatic programming for the object-oriented programming paradigm. In *SAICSIT '07: Proceedings of the 2007 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries*, pages 116–124, New York, NY, USA, 2007. ACM.

[20] A. Seesing and H.-G. Gross. A genetic programming approach to automated test generation for object-oriented software. In *1st International Workshop on Evaluation of Novel Approaches to Software Engineering*, Erfurt, Germany, September 2006. NetObject Days 2006.

[21] S. F. Society. What is an msc? 15 September 2008 http://www.sdl-forum.org/MSC/index.htm, 07 2007.

[22] D. Sosnoski. Java programming dynamics, part 2: Introducing reflection. 15 September 2008 http://www.ibm.com/developerworks/java/library/j-dyn0603/, 06 2003.

[23] D. E. Suárez, J. Y. Olarte, and S. A. Rojas. Evolving object oriented agent programs in robocup domain. In *GECCO '05: Proceedings of the 2005 workshops on Genetic and evolutionary computation*, pages 407–410, New York, NY, USA, 2005. ACM.

[24] P. von der Ahe. The java community process(sm) program - jsrs: Java specification requests - detail jsr 199. 21 July 2008 http://jcp.org/en/jsr/detail?id=199, 06 2008.

[25] S. Wappler and J. Wegener. Evolutionary unit testing of object-oriented software using strongly-typed genetic programming. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1925–1932, New York, NY, USA, 2006. ACM.

[26] J. D. Watson and F. H. C. Crick. Molecular structure of nucleic acids: A structure for deoxyribose nucleic acid. *Nature*, 171(4356):737–738, 1953.

[27] Wikipedia. Protein. 15 September 2008 http://en.wikipedia.org/wiki/Protein, 10 2003.

[28] Wikipedia. DNA. 15 September 2008 http://en.wikipedia.org/wiki/DNA, 02 2007.

[29] T. Yu. Polymorphism and genetic programming. In *EuroGP '01: Proceedings of the 4th European Conference on Genetic Programming*, pages 218–233, London, UK, 2001. Springer-Verlag.

Appendix A

XML Schema

A.1 Grammar Schema

```
<?xml version="1.0"?>
<xs:schema>
  <xs:element name="GPGrammar">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="terminal" type="terminal"/>
          <xs:element name="nonterminal" type="nonterminal"/>
          <xs:element name="rule" type="rule"/>
        </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="terminal" type="xs:string" maxOccurs="unbounded">
       <xs:attributeGroup ref="component.attributes" />
  </xs:element>
```

```
<xs:element name="nonterminal" maxOccurs="unbounded">
  <xs:complexType>
    <xs:element name="terminal" type="xs:string"/>
    <xs:element name="nonterminal" type="xs:string"/>
    <xs:element name="rule" type="xs:string"/>
    <xs:attributeGroup ref="component.attributes" />
  </xs:complexType>
</xs:element>

<xs:element name="type" type="xs:string" maxOccurs="unbounded"/>

<xs:element name="variable" type="xs:string" maxOccurs="unbounded"/>

<xs:element name="method" type="xs:string" maxOccurs="unbounded">

<xs:element name="rule" maxOccurs="unbounded">
  <xs:complexType>
     <xs:element name="terminal" type="terminal"/>
     <xs:element name="nonterminal" type="nonterminal"/>
     <xs:element name="type" type="xs:string"/>
     <xs:element name="variable" type="xs:string"/>
     <xs:element name="method" type="xs:string"/>
    <xs:attributeGroup ref="component.attributes" />
  </xs:complexType>
</xs:element>

<xs:attributeGroup name="component.attributes">
  <xs:attribute name="name" type="xs:string" use="required"/>
  <xs:attribute name="default" type="xs:boolean" use="optional"/>
  <xs:attribute name="starttoken" type="xs:boolean" use="optional"/>
</xs:attributeGroup>
```

```
</xs:schema>
```

A.2 Skeleton Schema

```
<xs:schema>
  <xs:element name="ClassSkeleton">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="import" type="xs:string" minOccurs="1"
            maxOccurs="unbounded"/>
          <xs:element ref="variableList"/>
          <xs:element name="methodList">
            <xs:compl
            <xs:element ref="method"/>
          </xs:element>
        </xs:sequence>
        <xs:attribute name="superclass" type="xs:string" use="required"/>
    </xs:complexType>
  </xs:element>


  <xs:element name="method" type="xs:string" maxOccurs="unbounded">
       <xs:complexType>
         <xs:sequence>
           <xs:element name="visibility" type="xs:string"/>
           <xs:element name="returnType" type="xs:string"/>
           <xs:element ref="parameterList">
           <xs:attribute name="name" type="xs:string" use="required"/>
         </xs:sequence>
```

```
      </xs:complexType>
  </xs:element>

  <xs:element name="variableList">
     <xs:element name="variable" type="xs:string" maxOccurs="unbounded">
       <xs:complexType>
         <xs:sequence>
           <xs:element name="import" type="xs:string"/>
           <xs:element name="initialvalue" type="xs:string"/>
           <xs:element name="type" type="xs:string"/>
           <xs:attribute name="name" type="xs:string" use="required"/>
         </xs:sequence>
       </xs:complexType>
    </xs:element>
  </xs:element>

  <xs:element name="parameterList" maxOccurs="1">
     <xs:element name="variable" type="xs:string" maxOccurs="unbounded">
       <xs:complexType>
         <xs:sequence>
           <xs:element name="type" type="xs:string"/>
           <xs:attribute name="name" type="xs:string" use="required"/>
         </xs:sequence>
       </xs:complexType>
    </xs:element>
  </xs:element>
  </xs:schema>
```

Appendix B

XML Grammars

B.1 Two Box Grammar

```
<?xml version="1.0" encoding="UTF-8"?>
<GPGrammar>
  <terminal name="plus">+</terminal>
  <terminal name="minus">-</terminal>
  <terminal name="div">/</terminal>
  <terminal name="mult">*</terminal>
  <terminal name="open_bracket">(</terminal>
  <terminal name="close_bracket">)</terminal>
  <terminal name="const">1.0</terminal>
  <terminal name="x">x</terminal>
  <terminal name="y">y</terminal>
  <terminal name="z">z</terminal>
  <terminal name="p">p</terminal>
  <terminal name="q">q</terminal>
  <terminal name="r">r</terminal>
  <rule name="exp_op_exp">
```

```
      <nonterminal>exp</nonterminal>
    <nonterminal>op</nonterminal>
    <nonterminal>exp</nonterminal>
  </rule>
  <rule name="b_exp_op_exp_b">
    <terminal name="open_bracket"/>
    <terminal name="exp"/>
    <terminal name="op"/>
    <terminal name="exp"/>
    <terminal name="close_bracket"/>
  </rule>
  <nonterminal name="exp">
    <rule name="exp_op_exp"/>
    <rule name="b_exp_op_exp_b"/>
    <nonterminal name="var" default="true"/>
  </nonterminal>
  <nonterminal name="op">
    <terminal default="true">plus</terminal>
    <terminal>minus</terminal>
    <terminal>div</terminal>
    <terminal>mult</terminal>
  </nonterminal>
  <nonterminal name="var">
    <terminal name="x" default="true"/>
    <terminal name="y"/>
    <terminal name="z"/>
    <terminal name="p"/>
    <terminal name="q"/>
    <terminal name="r"/>
    <terminal name="const"/>
  </nonterminal>
```

```
  <nonterminal name="start_exp" starttoken="true">
    <rule name="exp_op_exp"/>
    <rule name="b_exp_op_exp_b"/>
    <nonterminal name="var" default="true"/>
    </nonterminal>
</GPGrammar>
```

B.2 Object-Oriented Grammar

```
<?xml version="1.0" encoding="UTF-8"?>
<GPGrammar>
  <terminal name="dot">.</terminal>
  <terminal name="opencurly">{</terminal>
  <terminal name="closecurly">}</terminal>
  <terminal name="openbracket">(</terminal>
  <terminal name="closebracket">)</terminal>
  <terminal name="semicolon">;</terminal>

  <rule name="statement_expr">
    <nonterminal default="true">statement</nonterminal>
    <nonterminal>expr</nonterminal>
  </rule>

  <ifblock name="if_stmt">
    <nonterminal>methodcall</nonterminal>
    <nonterminal>if_body</nonterminal>
    <nonterminal>if_body</nonterminal>
  </ifblock>

  <rule name="stmt_rule">
```

```
  <nonterminal>methodcall</nonterminal>
  <terminal>semicolon</terminal>
</rule>

<rule name="methcall_rule">
  <type></type>
  <terminal>dot</terminal>
  <method></method>
</rule>

<nonterminal name="if_body">
  <nonterminal default="true">statement</nonterminal>
  <rule>if_expr</rule>
</nonterminal>

<rule name="if_expr">
  <nonterminal>statement</nonterminal>
  <nonterminal>if_body</nonterminal>
</rule>

<nonterminal name="expr" starttoken="true">
  <rule>if_stmt</rule>
  <rule>statement_expr</rule>
  <nonterminal default="true">statement</nonterminal>
</nonterminal>

<nonterminal name="statement">
  <rule>stmt_rule</rule>
  <terminal default="true">semicolon</terminal>
</nonterminal>
```

```
  <nonterminal name="methodcall">
    <rule default="true">methcall_rule</rule>
  </nonterminal>
</GPGrammar>
```

Appendix C

Skeleton XML Specifications

C.1 Bank Account Skeleton

```
<?xml version="1.0" encoding="UTF-8"?>
<ClassSkeleton superclass="problem.bankaccount.defaults.Withdrawl">
  <import>problem.bankaccount.defaults.BankAccount</import>
  <import>problem.bankaccount.defaults.WithdrawlFees</import>
  <import>problem.bankaccount.defaults.WithdrawlRules</import>
  <import>problem.bankaccount.defaults.BankingFees</import>
  <import>problem.bankaccount.defaults.BankingRules</import>
  <import>java.lang.Integer</import>
  <method name="doTransaction">
    <visibility>public</visibility>
    <returnType>void</returnType>
    <parameterlist>
      <variable name="account">
        <type>BankAccount</type>
      </variable>
      <variable name="anAmount">
```

```
        <type>Integer</type>
      </variable>
    </parameterlist>
  </method>
</ClassSkeleton>
```

C.2 RPN Calculator Skeleton

```
<?xml version="1.0" encoding="UTF-8"?>
<ClassSkeleton superclass="problem.calculator.defaults.BaseCalculator">
  <import>problem.calculator.defaults.BaseCalculator</import>
  <import>problem.calculator.defaults.SubTransaction</import>
  <import>problem.calculator.defaults.AddTransaction</import>
  <import>java.lang.Integer</import>
  <variablelist>
    <variable name="stack">
      <importvalue>
        problem.calculator.defaults.classextensions.CalcStack
      </importvalue>
      <initialvalue>new CalcStack()</initialvalue>
      <type>CalcStack</type>
    </variable>
  </variablelist>
  <method name="enterNumber">
    <visibility>public</visibility>
    <returnType>void</returnType>
    <parameterlist>
      <variable name="num">
        <type>Integer</type>
      </variable>
```

```
    </parameterlist>
  </method>
  <method name="add">
    <visibility>public</visibility>
    <returnType>void</returnType>
  </method>
  <method name="subtract">
    <visibility>public</visibility>
    <returnType>void</returnType>
  </method>
</ClassSkeleton>
```