OBJECT FILE PROGRAM RECOMBINATION OF EXISTING SOFTWARE PROGRAMS USING GENETIC ALGORITHMS

by

Blair Carleton Foster Jr.

Submitted in partial fulfillment of the
requirements for the degree of
Master of Computer Science

at

Carleton University
Ottawa, Ontario
February 2011

© Copyright by Blair Carleton Foster Jr., 2011

CARLETON UNIVERSITY

FACULTY OF COMPUTER SCIENCE

The undersigned hereby certify that they have read and recommend to the Faculty of Graduate Studies for acceptance a thesis entitled "OBJECT FILE PROGRAM RECOMBINATION OF EXISTING SOFTWARE PROGRAMS USING GENETIC ALGORITHMS" by Blair Carleton Foster Jr. in partial fulfillment of the requirements for the degree of Master of Computer Science.

Dated: February 28, 2011

Supervisor: ______________________________
Dr. Anil Somayaji

Readers: ______________________________
Dr. Dwight Deugo

____
Dr. Liam Peyton

CARLETON UNIVERSITY

DATE: February 28, 2011

AUTHOR: Blair Carleton Foster Jr.

TITLE: OBJECT FILE PROGRAM RECOMBINATION OF EXISTING SOFTWARE PROGRAMS USING GENETIC ALGORITHMS

DEPARTMENT OR SCHOOL: Faculty of Computer Science

DEGREE: M.C.Sc. CONVOCATION: June YEAR: 2011

Permission is herewith granted to Carleton University to circulate and to have copied for non-commercial purposes, at its discretion, the above title upon the request of individuals or institutions.

Signature of Author

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

The author attests that permission has been obtained for the use of any copyrighted material appearing in the thesis (other than brief excerpts requiring only proper acknowledgement in scholarly writing) and that all such use is clearly acknowledged.

Table of Contents

List of Figures . . . . **viii**

Abstract . . . . **x**

Acknowledgements . . . . **xi**

Chapter 1 Introduction **1**

1.1 Software Recombination 1

1.2 ObjRecomGA 2

1.3 Contributions 3

1.4 Thesis Organization 3

Chapter 2 Background and Related Work **5**

2.1 Genetic Algorithms 5

2.2 Genetic Programming 6

2.2.1 Program Representation 7

2.3 Genetic Programming Extensions 8

2.3.1 Automatically Defined Functions 8

2.3.2 Stack Based and Strongly Typed GP 8

2.3.3 Grammar-Based GP 9

2.3.4 Genetic Program Seeding 9

2.3.5 Co-Evolution in Genetic Programming 10

2.3.6 Evolving Existing Software Programs using Genetic Programming 10

2.4 Automated Software Engineering 11

2.4.1 Software Evolution via Software Engineering 12

2.4.2 Software Recombination thorough Automated Software Reuse and Adaptation 12

2.5 Component Based Software Engineering 13

2.6 Artificial Life 13

2.7 Summary 14

Chapter 3 Program Recombination **16**
3.1 Recombination in Genetic Programming 16
3.2 Species of a Software Program 18
3.3 Object File Recombination 21
3.3.1 Object File Overview 22
3.3.2 Object Files as Genes During Crossover 25
3.3.3 Object Files as Chromosomes During Crossover 27
3.4 Object File Recombination Problems 27
3.4.1 Linking Problems 27
3.4.2 Runtime Problems 29
3.5 Summary 32

Chapter 4 ObjRecombGA **33**
4.1 Correcting Linking Problems 33
4.1.1 Processing the Object Files 34
4.1.2 Manipulating the Object Files 35
4.2 Addressing Runtime Problems 36
4.2.1 Identifying Problems at Runtime 36
4.3 Implementation Details 37
4.3.1 Pre-processing the Object Files 38
4.3.2 Program Variant Generation 39
4.3.3 Program Variant Decoding Function 40
4.3.4 Fitness Function 42
4.3.5 Crossover & Selection Functions 43
4.4 Implementation Challenges 45
4.4.1 Common Block Symbols 45
4.4.2 Library Version Conflicts 46
4.4.3 Testing of Recombined Variants 47
4.5 Summary 47

Chapter 5 Results **49**
5.1 Selecting Candidate Programs 49

5.2 GNU-sed . . . 50
5.2.1 Fitness Calculation . . . 51
5.2.2 Versions Tested . . . 52
5.2.3 Analysis . . . 55
5.3 Dillo . . . 55
5.3.1 Fitness Calculation . . . 57
5.3.2 Versions Tested . . . 58
5.3.3 Analysis . . . 63
5.4 Quake . . . 64
5.4.1 Fitness Calculation . . . 65
5.4.2 Versions Tested . . . 67
5.4.3 Analysis . . . 73
5.5 Summary . . . 74

Chapter 6 Discussion . . . 77
6.1 Observations . . . 77
6.2 Contributions . . . 78
6.3 Limitations . . . 79
6.3.1 Closely Related Programs . . . 79
6.3.2 Object File Recombination . . . 79
6.3.3 Object Files in Object-Oriented Programs . . . 80
6.3.4 C Programs with ELF Objects . . . 80
6.3.5 Linux Runtime Environment . . . 81
6.4 Future Work . . . 81

Chapter 7 Conclusion . . . 83

Bibliography . . . 84

Appendix A Source Code for Program Version Alpha . . . 92

Appendix B Source Code for Program Version Beta . . . 93

Appendix C **Fitness Script used for calculating the fitness of GNU-sed program variants** **94**

Appendix D **Fitness Script used for calculating the fitness of Dillo program variants** **97**

Appendix E **Fitness Script used for calculating the fitness of Quake program variants** **99**

List of Figures

Figure 2.1 GP recombination of program A and program B 7

Figure 3.1 Phylogenetic tree depicting the relationship between several closely related species of whale 19

Figure 3.2 An example program revision control graph doubling as an evolutionary tree. 19

Figure 3.3 The GNU/Linux distribution timeline 20

Figure 3.4 Chromosomal crossover 25

Figure 3.5 Unequal chromosomal crossover 26

Figure 3.6 Object Files from versions *Alpha* and *Beta* 28

Figure 4.1 Recombining object *A* from *Alpha* and object *B* from *Beta* . . 34

Figure 4.2 The ObjRecombGA user interface 38

Figure 4.3 Single-Point Crossover 44

Figure 4.4 Two-Point Crossover 44

Figure 5.1 Average fitness at each Generation 54

Figure 5.2 Number of stable program variants at each generation. 54

Figure 5.3 The Dillo web browser - Version 0.8.5 56

Figure 5.4 Dillo average fitness at each Generation 60

Figure 5.5 # of Stable Dillo variants at each generation. 60

Figure 5.6 Dillo screen shots from Test 1 & Test 2 61

Figure 5.7 Dillo screen shots from Test 3 & Test 4 62

Figure 5.8 Typical screen shot result of Dillo variants(0.8.3 and 0.8.0) . . 63

Figure 5.9 Typical screen shot result of Dillo variants (0.8.5 and 0.8.0) . . 64

Figure 5.10 Typical screen shot result of Dillo variants (0.8.0 and 0.7.3) . . 65

Figure 5.11 A Dillo variant generated from versions 0.8.4 and 0.8.1 66

Figure 5.12 A Dillo variant generated from versions 0.8.4 and 0.8.1 67

Figure 5.13 A Dillo variant generated from versions 0.8.4 and 0.8.1 68
Figure 5.14 ID Software's Quake - Version 1.09 69
Figure 5.15 Various Quake forks 70
Figure 5.16 TyrQuake - Version 0.38 71
Figure 5.17 SDLQuake - Version 1.09 71
Figure 5.18 MakaQu - Version 0.2 71
Figure 5.19 ProQuake - Version 3.60 71
Figure 5.20 FishEyeQuake - Version 1.09 72
Figure 5.21 Average fitness at each Generation 74
Figure 5.22 Screenshots of hung Quake variants 75
Figure 5.23 Example screen shots of Quake variants 75
Figure 5.24 Screenshots of SDLQuake recombined with MakaQu 76
Figure 5.25 Screenshot of FisheyeQuake recombined with MakaQu 76
Figure 5.26 Screenshots of SDLQuake recombined with FishEyeQuake . . 76

Abstract

Software program recombination is a standard part of a software development toolbox. Software functionality and features in the form of source code are frequently taken from one program and merged into the existing body of source code of another program. This manually intensive recombination process is hampered by the fact that source code is brittle and prone to errors during compilation. This research presents a new, biologically inspired, approach to software program recombination by automatically recombining the object files of two closely related C programs in order to recombine their features and functionality. In much the same way biologist classify closely related species using phylogenetic trees, a pair of programs can be classified as closely related if they share a common development history or have evolved from a common base program. A software program, ObjRecombGA, automates the object file recombination process by using a genetic algorithm to search the space of all possible object file sets between the two closely related C programs. The results show that object file recombination of closely related programs is not only possible, but that it can even be applied to large and complex software programs. This recombination process can successfully discover new combinations of functionality derived from both parent programs. Moreover, the use of object files makes this approach applicable to existing software programs that generally cannot satisfy the program design and source code constraints required by existing recombination approaches.

Acknowledgements

I wish to extend my utmost gratitude to my graduate advisor, Dr. Anil Somayaji, for his acceptance, guidance, and support during the course of my graduate studies.

Special thanks are also expressed to my friends and family for their support and encouragement. I would also like to thank my co-workers for their insight and advice even though most of them told me that object file recombination would just crash programs. They were *almost* right.

Lastly, I would like to thank my wife Stacy for her love, inspiration, and support. Without her in my corner, this work could not have been completed.

Chapter 1

Introduction

Biology has taught us that organisms evolve and adapt over time to their surrounding environments. This evolutionary process is accomplished through means such as behavioral modification, genetic mutation brought about by environmental conditions, and genetic recombination with other organisms through reproductive means. Genetic recombination, the exchange and passing of genetic information on to offspring, is a critical process in the creation of a new species [29].

Computer scientists have developed several evolutionary and machine learning techniques inspired by biological evolution. Early work in this field focused on artificial intelligence and evolving algorithms with the ability to predict environment changes [25]. The introduction of Genetic Algorithms (GA)[38] brought forth the concept of creating evolved solutions to various computational complex problems. Shortly thereafter, Genetic Programming (GP) [46] further reduced biological-computational evolutionary gaps by leveraging GAs to create computer programs through evolution and program recombination. However, much of the existing research in GP focuses on small well-structured computer programs of limited scope and complexity which are tailored to very specific problems. Additionally, this work strictly focuses on the creation and recombination of code for *new* software programs. Only recently has a small handful of research in the area of computer security [91, 92] directly addressed the problem of evolving the source code of an *existing* software program.

1.1 Software Recombination

The manual process of recombination and evolution of existing software programs is not new. Frequently, source code from one program is recombined with the source code of another program in the form of code reuse, code adaptation, or code rewrites. Software developers perform these types recombinations when fixing software bugs and porting functionality from one software program to another. This manual process

of porting functionality, however, is prone to human error, may take a very long time to complete, often requiring the developer to be intimately familiar with the source code of both programs.

This research presents an approach for automating the recombination of functionality from existing C programs using evolutionary computational methods. More specifically, this approach makes use of biologically inspired techniques to automatically recombine features and functionality of existing software programs using a genetic algorithm (GA). The key insight here is that recombination is possible between different existing software programs so long as they are closely related and can be shown to derive from a common code base. Much as a dog and a camel cannot mate successfully because they are different species, we cannot expect the recombination of a web browser and a word processor to be successful because they are vastly different software programs which likely have very little relation.

Unlike previous related work [5, 1, 91], the presented approach accomplishes program recombination without modifying any existing program source code. Instead, the sets of correctly compiled object files from each C program are recombined and the interaction between these binary components is evolved and recombined. The use object files, rather than source code, avoids nearly all of the complexities of source code transformation and recombination faced in many previous techniques. Moreover, this approach works on existing programs which often have no formalized specifications; no implicit program design or framework; and no source code annotations that are required by existing recombination approaches. Using object files for program recombination will be shown to be not only feasible, but scalable to large and very complex software programs. The end result of this process is the creation of many software program variants which contain new combinations of functionality and characteristics from both original program versions.

1.2 ObjRecomGA

To automate the proposed software recombination technique a software program, titled ObjRecombGA, has been developed. ObjRecombGA performs this software recombination by manipulating the objects files of two distinct versions (or forks) of an existing C program. Through the analysis and modification of the object files from

the two program versions, ObjRecombGA is able to re-link a population of object file sets to create new program variants, each with varying levels of functionality and usability. At each generation of the GA, these program variants are executed against a series of tests to evaluate their fitness in terms of stability and functionality. The program variants which exhibit the highest levels of fitness against the test suite are more likely to be selected for recombination into subsequent generations. Once all generations have been completed, several program variants will be analyzed and compared to the parent versions to determine how the recombination process may have altered or combined the functionality of the original parent programs.

1.3 Contributions

The recombination technique presented here, and implemented by ObjRecombGA, is the first known method for recombining functionality from existing software programs that are closely related. Moreover, this recombination technique is novel in that it is accomplished without modification to either programs' underlying source code, thereby using the correctly compiled components and functionality of the software programs as is. The process for the selection and recombination of software programs was largely inspired by the selection and recombination processes seen in living organisms. The results of this research show that this form of software program recombination allows for the generation of many recombined unique program variants. In addition, several of these variants are effectively new software programs that incorporate functionality and features from both original parent programs.

ObjRecombGA was previously presented at The Genetic and Evolutionary Computation Conference (GECCO) 2010 in a research paper title "Object-Level Recombination of Commodity Applications".

1.4 Thesis Organization

Chapter 1 has provided an introduction to the research objectives of this thesis. In Chapter 2, an overview of relevant background material and related work is presented. Chapter 3 presents a more detailed view of the research objectives and how they were inspired by several biological concepts. Chapter 4 describes the design and

implementation details of the ObjRecombGA software program, including how the research objectives were addressed. Chapter 5 offers the results obtained from running the ObjRecombGA software against a set of existing software programs and provides a brief discussion of these results. Major thesis contributions, and future research ideas are provided in Chapter 6. Chapter 7 serves as a conclusion to this thesis.

Chapter 2

Background and Related Work

This chapter presents an overview of relevant background material and existing work related to software recombination and evolution. The first two sections of this chapter give an introduction to genetic algorithms and genetic programming; two evolutionary computation techniques that are the backbone of much of the existing work being presented. The next section examines previous research efforts that are specifically related to the evolution and recombination of software programs.

2.1 Genetic Algorithms

The term Genetic Algorithms (GAs) is used to describe a biologically inspired algorithm that attempts to mimic evolutionary processes, specifically natural selection and genetic recombination, in order to evolve potential solutions to a given problem. The expectation is that with each iteration of the algorithm the solutions will move closer and closer toward an optimal solution. While there is no single agreed upon definition of a GA, there are a few defining elements which generally lead to an algorithm being classified as a Genetic Algorithm [64, p. 8-12]:

- A population of potential solutions with a clearly defined *encoding*
- A method to measure the *fitness* or value of a solution
- A selection method to select solutions, typically based on their fitness measure, to be evolved into the next generation
- A *crossover* function which exchanges portions of the selected solutions to produce offspring solutions

Where GA implementations tend to vary are in the method(s) used for selecting solutions, their fitness calculation function, and how they decode a solution for evaluation by this fitness function. Regardless, a typical GA implementation performs

the following actions [64, p. 8-12]:

1. Creates an initial population of potential solutions randomly
2. Calculates the fitness value of each solution in the population
3. *Selects* a pair of solutions in the population
4. Performs the crossover function on the selected pair to produce offspring solutions for a new population
5. Advance to the next iteration by replacing the existing population with the new population
6. Repeat the above steps until the desired number of iterations have been completed or some termination condition has been met

Many genetic algorithm implementations use binary encoding (bit strings) as their solution encoding. This simplifies the implementation of the crossover and mutation functions and makes them adaptable for many problem sets. However, there is a specific type of GA which is able to evolve programs known as Genetic Programming.

2.2 Genetic Programming

Genetic Programming (GP) [46] is often defined as a specific encoding of a GA which is adapted to generate, recombine, and evolve computer programs to solve a given problem or objective function. Algorithmically, GP behaves like GA and follows the same iterative evolution described above. As originally introduced [46], programs generated by GP are composed from a set of functions (arithmetic operators, programming operations, etc.) and a set of terminals (boolean values, integer values, etc.). These programs are then evaluated against a specified fitness or objective function with the best programs being recombined to produce new programs for the next generation. GP has been shown to solve many difficult problems in domains such as image and signal processing[76], financial data modeling, and computer generated art [76][52]. In some cases, GP has been able to generate new programs and recombine them to produce programs which offer improvements over existing methods [76][52].

Early work in GP focused on the creation and evolution of simple programs to solve well defined problems (often mathematical equations or approximations) which could be represented as mathematical expressions [46]. However, many modifications and extensions have been made to GP which has enabled it to evolve and recombine much more complex and sophisticated programs for a variety of problem sets.

2.2.1 Program Representation

Popular GP literature [46, 47, 48] prefer to represent GP programs as syntax trees with internal nodes chosen from the set of functions and leaf nodes being chosen from the set of terminals. As an example, Figure 2.1 shows a syntax tree representation and recombination of two GP generated programs which approximate the equation $y = x^3$ using a function set $(+, -, *)$ and terminal set $(2, 3, x)$.

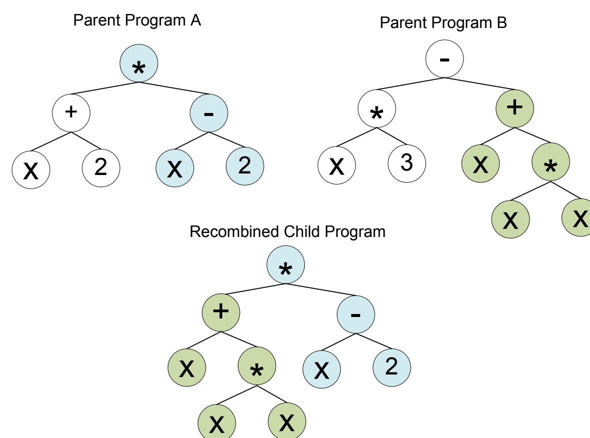

Figure 2.1: A GP recombination of program A $(y = 2x \times (x - 2))$ and program B $(y = 3x - x + x^2)$ which produces a child program $(y = (x + x^2) \times (x - 2))$

This tree-based structure translates easily to symbolic expressions and enables GP to generate and recombine programs in languages such as Lisp, whose source code is represented as symbolic expressions. However, early forms of this representation did not lend themselves to the evolution of programs using other languages. It was not until the debut of Linear GP representations [13, 18] that the creation and evolution of programs written in other languages, such as machine code languages [69] and simple functions in C [43], become more common.

2.3 Genetic Programming Extensions

Genetic Programming has become an indispensable technique in the evolutionary computing community and has played a role in solving many difficult problems [76, 51, 46, 47, 48]. It is commonly accepted, however, that GP alone is not yet capable of replacing software developers for creating or evolving existing software programs [37]. Still, researchers have pushed the boundaries of GPs capabilities by bringing GP program representations closer to that of conventional programs–ultimately leading to the use of GP techniques on existing software programs. These works will be discussed in the following sub-sections.

2.3.1 Automatically Defined Functions

The introduction of Automatically Defined Functions (ADFs)[47] in GP brought forth the notion of well defined sub-trees within a GP program. These sub-trees effectively became viewed as program sub-routines which performed some defined set of operations relevant to the problem. Collections of these sub-routines could be evolved in parallel with the GP program and then incorporated into the GP program generation. Moreover, the sub-routines themselves could be used more than once within the program(s), thereby allowing code reuse and modularity within and across GP programs[47]. In a crude sense, ADFs represent GPs first forte into program code reuse and code recombination.

2.3.2 Stack Based and Strongly Typed GP

The desire to enable GP to evolve programs of more complex structure and representation led to creation of Stack Based GP and [75] Strongly Typed GP [65]. In Stack Based GP all terminals and operation results are pushed and popped from a virtual stack. This behavior more closely mimics that of conventional programs and forces GP to not only evolve the program code, but the program state representation as well. Strongly Typed GP introduced the concept of type checking to GP. This allowed for the generation and evolution of programs that contained multiple data types while still remaining syntactically correct.

2.3.3 Grammar-Based GP

Grammar-Based Genetic Programming [59] makes use of grammar formalizations to place restrictions on the interactions, expressions, and types within the code that is generated by GP. This allows GP to more accurately and more correctly generate programs using very complex representations. Recently [71] strongly typed GP has been paired with Grammer-Based GP to create a framework for evolving syntactically correct Java programs that evolve the state and behavior of the program.

2.3.4 Genetic Program Seeding

While classical GP uses a set of randomly generated programs as the initial population, GP seeding makes use of a set of existing, working, programs as the initial population[50]. This has the effect of focusing the GPs initial search area to that of known working programs, where a randomly initialized GP may take several generations to discover such programs, if at all. One requirement of using seeding in GP is that the fitness function needs to be multi-objective - trying to satisfy more than one set of criteria[50]. This is obvious as the GP needs to evaluate the fitness of the program itself while ensuring that the GP is still exploring the search space effectively and evolving away from the initial seed[76]. One of the first research endeavors[50] to highlight the value of seeding in GP used a set of existing 'good' functions in order to find a set of more generalized functions over a given dataset. Specifically, the research focused on the size of the function as a measure of its generalization - assuming that smaller functions were more generalized. The work concluded that evolving non-random populations is possible due to GPs built-in nature to generalize the programs it evolves [50]. Subsequent research[58] used GP seeding in robotics navigation to develop generalized navigation algorithms. Not only did the research use hand-coded individuals as seeds, but it also used previously discovered GP programs as seeds as well. This is particularly noteworthy as this represents early attempts at recombining more than one existing GP program using, rather than just evolving a single GP program on its own. The results of the research found that seeding can weaken the GPs ability to find better performing algorithms if the initial seeds are too fit[58]. Therefore, having an appropriate initial seed is crucial to the performance of the GP.

2.3.5 Co-Evolution in Genetic Programming

Existing work [4] has attempted to address several fitness limitations of GP by co-evolving programs *and* test cases based on formalized program specifications. This was demonstrated to be successful for creating small programs with a manageable set of specifications, however the success of this approach on "real-world software"[4] was left as an open question. Moreover, the requirement of formalized specifications make this approach unusable on existing software programs. More recently, GP seeding has been combined with the concept of test case co-evolution to evolve semantically equivalent functions given the source code of a single 'good' function as input[3]. In this research, the authors used the initial function as an oracle for evaluating the semantics of the evolved functions with the goal of maximizing efficiency. The co-evolution of the test cases ensured that evolved functions were also correct and optimized over varying sets of input.

2.3.6 Evolving Existing Software Programs using Genetic Programming

Within the past year, research has emerged highlighting GPs potential for evolving existing software programs[1, 5]. Automated software testing and fault repair is one such area where GP has been investigated for use on existing software[2]. This research introduced JAFF[1], a framework for automatically fixing faults in Java programs. JAFF uses GP to evolve programs against a fitness function which measures the number of unit tests that they successfully complete, with at least one unit test exposing a fault. The faulty program is parsed and represented as a syntax tree which enables JAFF to more efficiently search for, and isolate, the faulty area of the program. To evaluate a mutated program's fitness, JAFF converts the syntax tree back to Java source code, compiles it, and re-runs the unit tests. Though not actually tested on existing software programs (primarily due to its lack of full support of the Java language) the research did find some very interesting results. While JAFF was shown to repair some test programs, the repair code often reduced the efficiency of the code and provided no way to ensure that different inputs would not cause a fault in the repair code[2]. Additionally JAFF is limited in the programs that it could possibly evolve as the input programs unit cases were required to be runnable from

[1] JAFF: Java Automatic Fault Fixer [2]

within the JAFF framework.

Subsequent research [91, 93] pushed GP as a an automated software repair mechanism one step further. This research used real off-the-shelf C programs and evolved software security patches to fix publicly available vulnerabilities. Similar to JAFF, this research represents the program's source code as a syntax tree and evaluates fitness by compiling and re-testing the evolved variants. The syntax tree was weighted using execution path profiling [68] to enhanced the GPs ability to identify the vulnerable code segments and evolve them. Code readability and consistency was enforced and maintained as all code patches were derived from *existing code* in other parts of the program. Once a successful code patch had been evolved, it was then minimized for efficiency and code bloat. The research [91, 93] successfully demonstrated this technique by evolving highly efficient patches for ten off-the-self C programs.

All of the cited research initiatives are similar and suffer from a few notable limitations. In all cases, it is expected that the negative test case exposing the vulnerable area of the program can be reliably reproduced. This is often not the case on all software programs where the configuration, environment, and timing can influence the manifestation of the defect. The correctness of the evolved software patch could also be considered a limitation. This is because its correctness is heavily reliant on the availability of appropriate and exhaustive unit tests for the software[93]. A more interesting limitation is due to the fact the evolved repair patch is composed of code from within other segments of the program. This implies that no entirely new code segments will be generated[91]. Depending on the program source code and the defect subject to patching, it may not be possible to generate a correct patch.

2.4 Automated Software Engineering

Automated Software Engineering is a broadly growing field which focuses on the automation of any area or activity of software engineering [31]. This includes areas such as: requirements discovery and validation; software testing and verification; software management and deployment; automated software reuse and adaptation; and software evolution[2]. Of these areas, software reuse, software adaptation, and software evolution provide a handful of noteworthy research initiatives.

[2]Evolution, refactoring, re-engineering are often used interchangeably

2.4.1 Software Evolution via Software Engineering

Existing research on software evolution focuses on understanding *how and why* software programs evolve over time as part of the software engineering process[53, 22]. This often involves the analysis of a program's source code across several releases [32, 66, 89, 79] or by simply studying its changing functionality[39]. From these results, models are generated to predict the effects of software evolution and how to better design both current and future software programs[16, 60, 89, 79].

A small handful of research in this field has also considered automating the evolutionary process of software programs [14, 41, 77]. Very little of this research, however, has targeted existing software programs with the remaining research requiring certain design criteria, formal specifications, or other a priori knowledge of the program being evolved.

2.4.2 Software Recombination thorough Automated Software Reuse and Adaptation

Software reuse and adaptation[35] refers to the reuse or adaptation of software components of one software program and integrating them into another software program. This integration or recombination process is often complicated by syntactic or semantic incompatibilities between the interfaces used by the software program components[35]. These incompatibilities are overcome by: modifying one of the programs to understand the components of the other; modifying the desired component itself to make it compatible with the target program; or creating an adapter or wrapper for the component [35]. Each of these processes occur manually when developers decide to reuse existing code or libraries and adapt them to their current work.

The automation of software reuse has been shown to be effective in the reuse of source code[56, 57] as well as the reuse of program designs and models[82]. Identifying potential reuse of source code has been shown to be possible [57, 67] by isolating domain specific sub-routines into collections of libraries, and making them available through well defined specifications. Moreover, automating code reuse by using formal specifications and classifications has also been proven to be successful [73, 98, 40].

Many, if not all, of the automated reuse and adaptation mechanisms[57, 67, 34, 19,

98, 49, 61, 73, 74, 40, 24, 33] in the field of automated software engineering require formal specifications, design patterns, specialized frameworks, source code annotations, specific programming languages, or specific program models to be effective. Because of this, much of the research in this field is ill-suited for existing software programs. Though research into automated reasoning and reverse engineering has attempted to close the gap[30, 63], there still exists no automated system to select and reuse or adapt components from existing software programs as-is.

2.5 Component Based Software Engineering

Research in component based software engineering[42] offers examples of binary software component reuse[20, 44] in running systems. In these systems, components are written based on predetermined formal specifications or interfaces that are compatible with the core system itself. As faults are found or enhancements are made to an existing component, the component can be updated at runtime. These types of systems are interesting because they encourage code recombination at the binary level rather than source code. These binary components, however, must conform to the specifications of the system, and as such, existing software program components would need to be re-factored or re-written before they could be used.

2.6 Artificial Life

Not to be confused with Artificial Intelligence (AI), Artificial Life (ALife) is the study of life by attempting to replicate biological concepts such as evolution and recombination within a computing environment. The introduction of biomorphs [23] represents a subset of early work in this field. A biomorph is a recursively drawn figure representing the phenotype of an underlying genotype. Biomorphs are particularly interesting because they make use of human interaction during their evolutionary process. As the biomorphs evolved, an actual person would evaluate each of the drawn figures by selecting those with desirable characteristics. This evaluation would be fed back into the evolution process to ensure that next generation of biomorphs contained the desired characteristics. The use of human interaction as a feedback mechanism has been subsequently used in several areas of evolutionary computing

[86, 45]. It is particularly useful for recombination of user software programs as it can be leveraged to mimic the prototyping and customer review loop found in software development models [36]. More importantly, certain program components, such as its user interface, are often difficult to evaluate computationally.

More recent work in the ALife field has been using biologically inspired evolutionary techniques to create self evolving and self replicating programs[80]. These *digital organisms* compete against one another in a resource limited simulated environment[78, 81] - a sort of 'survival of the computationally fittest'. Though interesting, this work is primarily focused with understanding the nature of evolution and its mechanisms[95, 70, 96, 62]. It has not yet grown to include the evolution or recombination of programs in an everyday computing environment, let alone existing software programs.

2.7 Summary

The evolution and recombination of software programs has been studied across a handful of fields in computer science. Evolutionary computing holds much of the early work in program evolution and is deeply rooted in GA and GP. However, the program representations commonly used in GP lend themselves to ease of evolution and are generally not well-suited for existing software programs. Though several extensions to GP have tried to increase the complexity of the programs GP is capable of evolving, there exists very few examples where GP has evolved complex software programs based on existing code bases. Within the realm of computer security one such example [93] exists where GP was used to evolve security patches for existing software programs. These patches were created by evolving the source code of the program until a known vulnerability was no longer present. This research, however, did not explore the evolution of the program beyond fixing the vulnerability. GP seeding offers another example [58] of evolving an existing program, however these were not conventional software programs and were in fact GP programs tailored to a specific problem.

With respect to program recombination, research in Automated Software Engineering has explored many avenues [57, 67, 34, 19, 98, 49, 61, 73, 74, 40, 24, 33] . This work has focused on reusing and adapting existing code into domain specific libraries or program designs such that it can be recombined into new programs. Component

Based Software Engineering research [20, 44] has gone one step further and reused existing binary components across software programs rather than just source code. Though automated, all of the techniques in these fields rely heavily on the design and/or the implementation of the existing software that is being reused or adapted. As a result, these approaches are not applicable to the vast majority of the existing software programs in use today.

Sadly, software program recombination used in automated software engineering has not yet crossed paths with genetic programming. There exists no literature which uses genetic programming, or genetic algorithms, to recombine or reuse code or components from more than one existing software program. Similarly, the use of binary components rather than source code, as seen in component based software engineering, has never been attempted using GA or GP techniques.

Chapter 3

Program Recombination

Different processes and techniques in the areas of Genetic Programming and Software Engineering that enable program recombination and evolution have been discussed. However, these techniques have been shown to be too rigid to recombine existing software programs - requiring a priori knowledge of the software or relying on specific design constraints or program representations. This chapter proposes a biologically inspired solution for recombining existing software programs based on re-linking the object files of two closely related programs. Two programs are considered closely related if they share a common development history – effectively classifying these programs as the same, or at least similar, *species*. This technique for recombination is advantageous because it requires no a priori knowledge of the existing software programs and has no software design or program representation requirements. Further more, the use of object files rather than source code means that no source code validation or formal specifications are required. The first section of this chapter contrasts recombination in genetic programming with that of genetic recombination. The next section introduces the notion of identifying the species of a program by leveraging biological classifications of species – an idea which is key to identifying programs that are closely related according to their development history. This is followed an explanation of object files and how they can be recombined to create a new program. The problems surrounding this recombination process are then discussed in detail.

3.1 Recombination in Genetic Programming

Genetic Programming (GP) borrows from two biological processes: the evolutionary process of natural selection and the chromosomal crossover process seen in genetic recombination. The emulation of these processes are what enable GP systems to select and recombine the members of its population to evolve new programs. There

is a fundamental difference, however, between evolution in biological systems and program evolution in GP systems. A GP system assumes that all members of its population are compatible and can be recombined with one another. GP systems assure compatibility by constraining all programs to a common program encoding and a common fitness measure. Moreover, because GP systems ensure a common program encoding and common fitness measure, all generated programs are not only compatible for recombination but they are also functionally compatible meaning they share a common functional purpose. This is unlike biological systems where living organisms live amongst other, incompatible, living organisms in the same environment and do not share a common fitness measure nor a common purpose. Instead, biological systems rely on the living organisms themselves to select and mate (recombine) with others that they are genetically compatible with[29]. This selection of potential organisms to recombine with is known as reproductive isolation and is what defines a species under the biological species concept[29]. Applying this species concept to GP systems would imply that all GP systems are, based on their constraints, capable of evolving only a single specific species of program. This becomes more evident if one considers introducing a new and different program – a new species of program – into a GP systems population. This new program is in no way guaranteed to be encoded correctly or share a common fitness measure with other programs in the GP population. As such, the GP system can no longer assume that all population members are compatible, thereby causing the GP system to fail.

Though the introduction of a new and *different* program will undoubtedly cause problems in a GP system, the same may not be true if this new program is *similar* to the existing population of programs. If the new program were of a very similar encoding, and shared a similar fitness measure, then it may be possible for the GP system to continue to function correctly. In effect, if one were to define the species of a program such that it becomes possible to identify very similar programs that are of the same (or at least similar) species, it is then more reasonable to expect these programs to recombine successfully within a GP system. The next section proposes an approach to identifying groups of software programs that are different but closely related, effectively classifying them as a similar species such that they can be selected for recombination.

3.2 Species of a Software Program

According to the biological concept of species, the species of a living organism can be determined by its reproductive isolation. In other words, a species is defined as members of populations that interbreed in nature or have the potential to interbreed[29]. When considering its applicability with respect to classifying the species of a software program, this definition is not very useful. Reproductive isolation has no validity in the context of software because the interbreeding of living organisms can be witnessed in biology while the interbreeding of software programs cannot. Though it is possible for any component of one program to be recombined with another program given proper adaption and modification[19], it is not something inherent to the behavior of software programs. Thankfully, the biological concept of a species is not the only accepted definition of species. In fact, there are many species concepts all of which define a species according to slightly different criteria such as shared traits, breeding habits, and evolutionary lineages[15]. Because of this, multiple species concepts are often used to identify new species[15].

The phylogenetic species concept is defined by an organism's estimated phylogenetic (evolutionary) tree and how closely related it is to other, more similar, species[29]. Figure 3.1 is a phylogenetic tree showing the relationship between several closely related species of whale. To be considered distinct phylogenetic species two population must have been evolutionarily independent long enough such that synapomorphies have emerged. A synapomorphy is any trait that is shared by two or more members of the population that has been derived from their most recent ancestor, whos own ancestors in turn do not share that trait[29]. In other words, once a new trait becomes present in a population and survives for a generation then a new species can be identified.

In terms of software, newly developed functionality and features are often associated with a specific version of the software program. A version of a software program refers to a particular state in the development process of a given software program[11]. As features or enhancements are added and removed from a software program the version of the program is generally updated to reflect the change[11]. Looking at the evolution of a software program through various development cycles and versions an evolutionary tree of the program emerges. Figure 3.2 is a revision control graph which

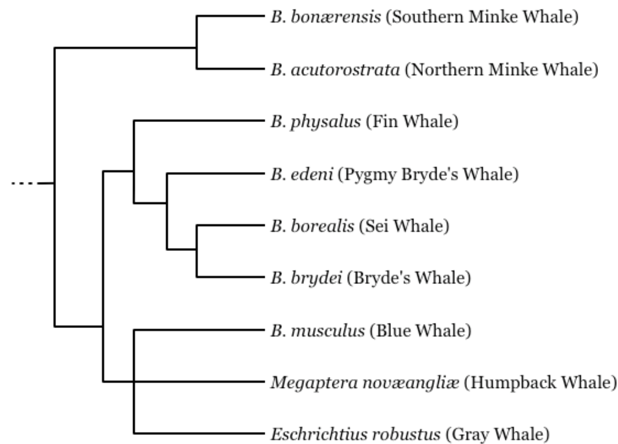

Figure 3.1: A portion of a phylogenetic tree depicting the relationship between several closely related species of whale[8]. Image licensed under the GNU Free Documentation License[27].

illustrates this type of evolutionary tree seen during version releases and development cycles. Further still, if a software program undergoes independent development by a different development team at some point a distinct branch in the evolutionary tree of the program is created. These distinct, independently developed and competing branches are often referred to as program forks[94, 9]. A program fork occurs when a copy of a program, at some version, undergoes separate and independent development thereby creating a distinctly different competing program[94, 9]. Taking a holistic view of all versions and independently developed forks of a software program allows for the construction of a complete evolutionary tree of the program.

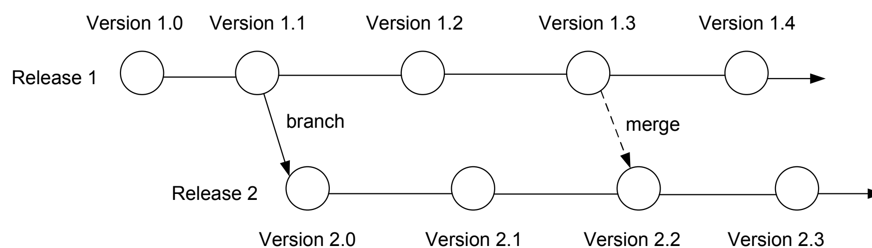

Figure 3.2: An example program revision control graph doubling as an evolutionary tree.

One example of a software program which has undergone many version releases and dozens of forks is the GNU/Linux Operating System. The evolutionary tree of GNU/Linux from 1991 thru 2006 is depicted in Figure 3.3. This form of evolutionary tree is presented much like that of the phylogenetic tree seen in figure 3.1 above.

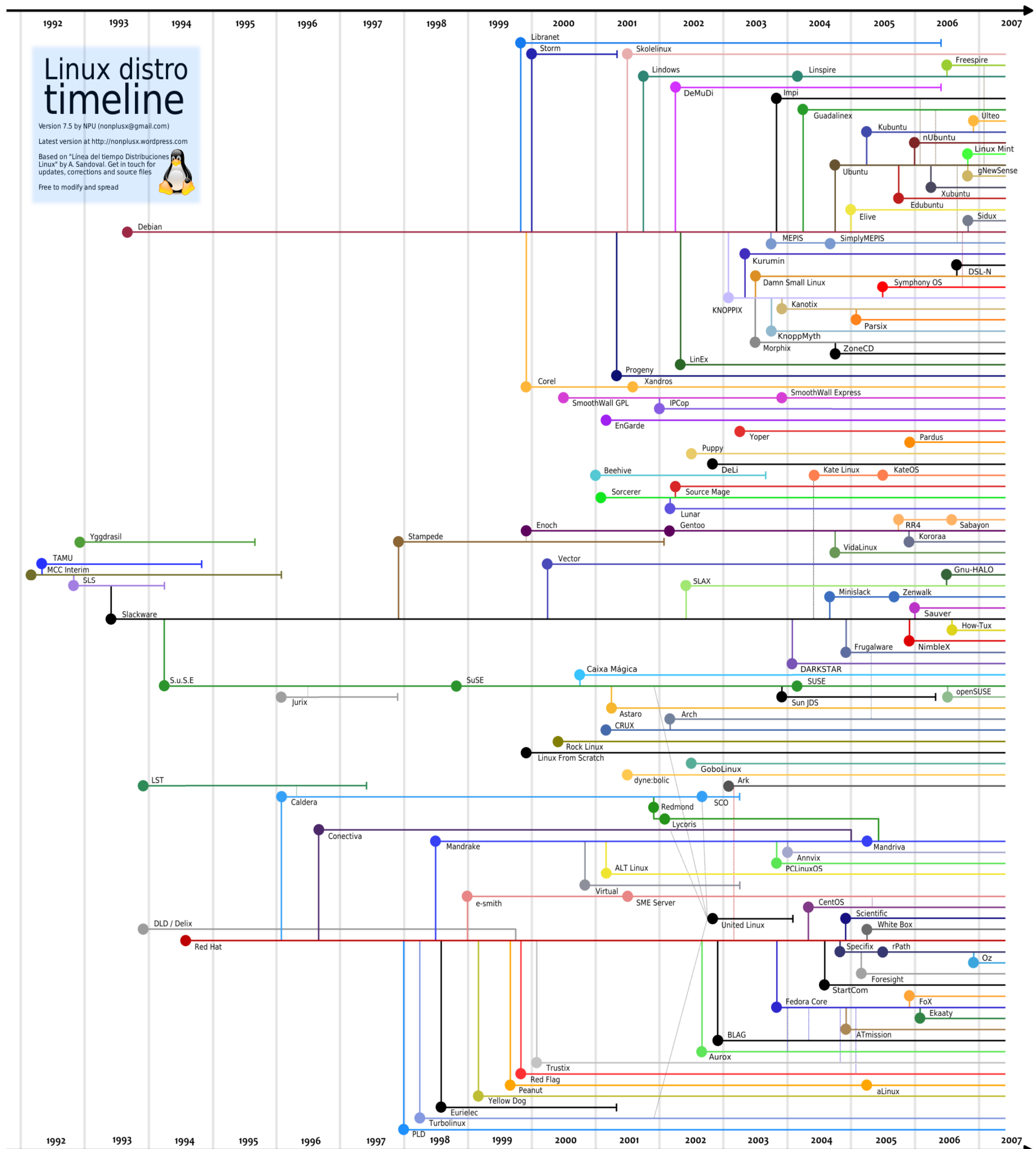

Figure 3.3: The GNU/Linux distribution timeline (v7.5) which depicts the evolutionary tree of the GNU/Linux from 1991 thru 2006 [10]. Image licensed under the GNU Free Documentation License[27]

Pairing a software program's complete evolutionary tree with the phylogenetic species concept provides a mechanism for identifying programs that are closely related and potentially similar or perhaps distinct *species* of programs. In this case, closely related software programs are considered to be the various versions and forks of

a particular software program over its lifetime. For example, three primary forks of GNU/Linux exist: Debian, Slackware, and RedHat[84]. These lineages, though all based on the underlying GNU/Linux code, are highly divergent on the software package management solution that each provide[84]. Therefore, based on the package management system, these three mainstream distributions of GNU/Linux can be classified as different families of GNU/Linux. All further forks of Debian, Slackware, or RedHat have maintained a similar package management system to their respective parent[84] making them closely related but distinct species of their own.

The use of the phylogenetic species concept and a software programs development history creates a program selection and mating criteria which more closely resembles that of living organisms. Though this criteria does not establish reproductive isolation or ensure recombination compatibility, it establishes which software programs are closely related and thus more likely to be to successful at recombination.

3.3 Object File Recombination

Now that the selection criteria of two software programs for recombination has been established, one remaining question for recombination is where the actual action of recombination should occur. Previous software recombination works [34, 19, 98, 49, 61, 73, 74, 40, 24, 33], including manual processes used by developers, have primarily focused on source code when reusing or recombining programs. An inherent problem with recombination using source code is that source code undergoes rigorous lexical, syntactic, and semantic analysis during compilation [97, 17] and even small, simple, changes can break these processes [17, 97]. As such, even when recombining two source files that are significantly similar, the recombination process would still need to adapt and modify segments of the source code before compilation to avoid errors. As a result the recombination process itself becomes a compiler of sorts with complex, language-dependent, validation and error checking rules to ensure that the recombined source code will still compile correctly.

In most compiled languages, such as C, source code is not the only potential place where recombination could occur. The transformation of a software program from its source code to an actual program executable is a multi-step process with two important phases: compilation and linking [97, 17, 54]. During the compilation

of a source file, after all the code analysis is completed, the compiler performs code transformation to convert the source code into intermediate code [1] [97, 17]. This intermediate code is then optimized and converted into machine code and placed inside a file called an object file (commonly given a '.o' file extension) [97, 17]. Once all the object files have been generated, they are provided as input[2] to a program called a *linker* which links code and data within object files together to produce the program executable [54]. Unlike source code, the validation performed on object files by the linker is both minimal and trivial as the linker is primarily concerned with identifying code and data references between object files [54]. The syntax and semantics of the code or data in the object files is of almost no concern to the linker. Given that the amount and complexity of analysis and validation performed on object files is greatly reduced compared to source files, it should potentially be easier to recombine programs at the object level rather than the source code level. It is worth noting that none of the related work in Chapter 2 considers program recombination using object files.

While object files may be easier than source code to recombine given their lack of validation, object files are ultimately just a binary abstraction of the source code they are generated from. More importantly, each object file is a self contained unit of correctly compiled source code. As such, rather than recombining the source code of two programs, using object files allows for the recombination and evolution of the interfaces and interaction between the binary components of two programs. Ultimately, this shifts the attention of program recombination away from finding correctly compiling and behaving programs and instead focuses on how the interaction between correctly behaving binary components of two programs can be recombined to discover new combinations of functionality. The remainder of this section will take a more detailed look at object files and how they can be recombined successfully.

3.3.1 Object File Overview

Understanding object file creation is necessary in order to explore recombining software programs using object files. In modern C compilers, such as GCC, there is a

[1] Generally referred to as Assembly language

[2] A linker can also accept libraries as input, which are simply collections of object files

one-to-one mapping of source files to object files. There are various object file formats in use by modern operating systems today, however the two most widely-adopted are the Executable and Linking Format (ELF) [83] and Common Object File Format (COFF) [54] formats. The ELF format is commonly used in Linux and Unix operating systems variants while the COFF format is used in the creation of Portable Executables (PEs) for use in Microsoft operating systems. Regardless of their format, object files typically contain five distinct pieces of information[3]: header information, code and data elements, relocation information, a symbol table, and debugging information [54]. For the purpose of object file validation, the symbol table and its use during linkage will be discussed in more detail below.

Symbol Table and Symbol Generation

The symbol table within an object file contains the set of symbols generated by the compiler. These symbols act as a named label for the code and data elements within the object file or for code and data elements referenced in other object files. Symbols include the following information: a unique name, the relative location of the element in the object file, the size of the element represented by this symbol, the type of the symbol - code or data[4], the scope or binding of the symbol, and some additional flags [54].

A symbol's name, type, and binding are generated during semantic analysis [97]. The name of the symbol is based strictly on the programmer-supplied name of the function or global data[5] which the symbol represents. The type is simply marked as code or data which corresponds to a function or global data declaration in the source code. In C, the binding of a symbol can be one of three types with the two most common being local binding and global binding[6]. Locally bound symbols are symbols which refer to a function or global data within the object file itself. These symbols cannot be referenced by another object file. Globally bound symbols are symbols which refer to a function or global data available for other object files to reference. Alternatively, globally bound symbols can represent references to functions or global

[3]Not all of this information is essential or always available in an object file, for example debugging information is generally not available in an object file that has been optimized [90, 54]

[4]Other types are possible, however these are not relevant here

[5]Non-static data defined within a function does not have a corresponding symbol

[6]The third type, weak bindings, are not often used and will not be discussed here

data in another object file [54]. The binding of the symbol is determined by the source code declaration of the function or global data which defines the symbol. In C, locally bound symbols are determined by presence of the *static* prefix keyword in the function or global data declaration, such as:

```
static int* func(char arg1, int arg2);
```

or

```
static int data[10];
```

If the *static* prefix keyword is not present, or the data item is defined within a function, the generated symbol will have a global binding.

External References

When the compiler encounters a reference to a function or data element whose function body or data storage does not exist in the current file, a symbol is generated with a global binding that is marked as external[7] in the object file. Taken as a whole, the set of external symbols define the code and data the object file is dependent on from one or more external object files.

Linking and Executable Creation

After compilation, all the produced object files are passed onto a linker to produce an executable program. The linker will use the symbol information within object files to determine how the set of object files to interact with each other and which pieces of code and data need to be shared between them. This symbol information is validated by the linker simply by looking at each external symbol in a given object file and pairing it up with a matching global symbol in another object file [54]. If for any reason, a matching global symbol is missing or is present in more than one object file, the linker will report an error and fail to create the executable.

[7] Some literature may refer to this as undefined, which is synonymous in this context

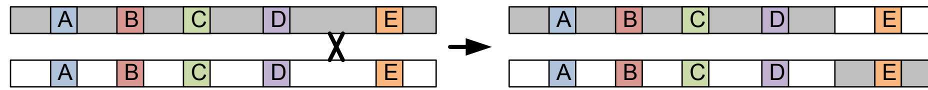

Figure 3.4: Chromosomal crossover. A-E signify gene locations along the homologous chromosomes.

3.3.2 Object Files as Genes During Crossover

In biology an organisms genetic makeup determines its physical and behavioral traits [29]. These specific traits are encoded within each organism's genes, with a single trait being represented by one or more genes [87]. This is not unlike software programs where the underlying source code of the program determines its functional traits. Moreover, the source code of a program is typically modular with a single source file containing several sub-routines related to a given functional module[72]. For example, the GNU implementation of the standard C library uses a single C file for each exported sub-routine or sets of related exported sub-routines of a module [55]. As mentioned earlier, each source file corresponds to exactly one object file in most compiled languages. Therefore, one or more object files will correspond to a given functional trait of a software program.

The genes of a living organism are arranged linearly along chromosomes with specific genes being found at specific locations[87]. Alternate forms of a particular gene are known as alleles[87]. All living organisms contain at least one chromosome, with more complex organisms containing many[87]. The exchange of the genetic material of two organisms occurs during genetic recombination in a process known as chromosomal crossover[8] (illustrated in Figure 3.4). During this process, homologous chromosomes – chromosomes which are similar both in structure and length – line up and crossover to exchange genetic material[87]. The crossover location(s) for exchange between these chromosomes are such that whole genes, and thus genetic traits, are kept intact[87].

Given that the traits of living organisms and software programs are encoded within the genes and object files respectively, a potential strategy for program recombination is to consider object files as genes during crossover. Under this strategy a program can be thought of as a single chromosome with its set of object files becoming the

[8] This is true for eukaryotic organisms, however prokaryotic organisms exchange genetic material differently.

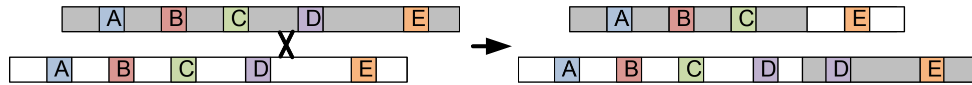

Figure 3.5: Unequal chromosomal crossover. A-E signify gene locations, however after recombination gene D is duplicated in the recombined chromosome.

genes along that chromosome. Two closely related software programs, however, can contain slightly differing sets of object files and, as such, are not obviously compatible (homologous).

In order to perform crossover between two software programs, it is necessary establish a common object file set (their chromosome) and only perform crossover within that set. For example, consider two software program versions with two slightly different sets of object files: $\{a.o_x, b.o_x, c.o_x, d.o_x\}$ and $\{b.o_y, c.o_y, d.o_y, e.o_y\}$. From these versions, the common set of object files available for recombination is $b.o, c.o, d.o$, with each program having an allele of this set. All remaining object files outside of these homologous sets are used as needed to avoid linking errors. One possible recombination of the software program chromosomes could be $\{a.o_x, b.o_x, c.o_y, d.o_x, e.o_y\}$, with object files $a.o_x$ and $e.o_y$ added to the resulting crossover in order to ensure that linking can be successful. Additionally, should there be significant changes such as code removal or code additions between the two related software programs, it may be necessary to include both alleles of the same object file in order to satisfy linking.

Using both versions of an object file containing slightly differing functionality to successfully link the object files is not unlike unequal chromosomal crossover. Unequal chromosomal crossover, illustrated in Figure 3.5, occurs when chromosomes do not pair correctly leading to non-homologous crossover points[29]. This leads to gene duplication in the resulting recombined chromosome however only one gene is actually used – the other is redundant[29]. This redundant gene may mutate, however, leading to the creation of a new gene rather than just remaining a copy of an existing one[29]. The same is true for the use of both alleles of an object file. Though both object files are being used for the purpose of linking, one of the object files will likely be redundant and benign or it could potentially lead to some unknown behavior.

Using this approach for recombination, it should be clear to see that, given two closely related software programs with n common object files, the search space for all possible recombinations becomes (2^n).

3.3.3 Object Files as Chromosomes During Crossover

An alternative analogy to object files as genes during crossover is to instead consider the object files as whole chromosomes. In this case the functions and data within the object files themselves are considered the genes to be exchanged. However, removing or adding functions and data from an object file is inherently problematic for several reasons. First, the compiler often optimizes code such that two or more functions can share segments of the same generated code within an object file. Therefore, removing code from one function may end up destroying another function[17]. Second, object files are considered to be highly cohesive. This means that code and data items that are internal to the object file are not easily identifiable and various compiler options may discard information that would otherwise make these elements identifiable[17]. Lastly, once all functions in all object files are considered for recombination the search space of possible recombinations greatly increases over using just object files alone. If one considers two related software programs with n object files (chromosomes), each with an average of m functions and data elements (genes), then the search space for all possible recombinations of the programs becomes $(n \times 2^m)$. For these reasons, using object files as chromosomes during crossover will not be explored in this thesis.

3.4 Object File Recombination Problems

With respect to object file recombination, there are two problems areas: These are problems that can occur during linking, and problems at runtime that will cause the recombined software program to terminate abnormally due to incorrect execution. These two problem areas will be explained further, including details of how even small changes between closely related software programs can cause them.

3.4.1 Linking Problems

As previously mentioned, a linker enforces two validations rules when linking object files: every symbol reference must be resolvable and symbols cannot be defined more than once[54] A symbol reference is considered resolvable if some other object file[9] defines that symbol. However, each symbol file can only be defined once, otherwise

[9] Or library file, which is simply a collection of object files

the linker will not know which symbol to choose when resolving symbol references.

Consider a simple program consisting of only two object files, A and B. In version *alpha* of the program (source code listed in Appendix A), object A contains two functions with symbols f_An and f_Am, along with two data items with symbols d_Ax and d_Ay. Object B also has two functions with symbols f_Bn and f_Bm, in addition to a single data item with symbol d_Bx. In version *beta* (source code listed in Appendix B), function f_Am has been removed from object A. Figure 3.6 illustrates objects A and B from both versions with arrows indicating symbol references between the two object files. Here we can see that all symbols within each version are uniquely defined and therefore the object files of each version will successfully link together.

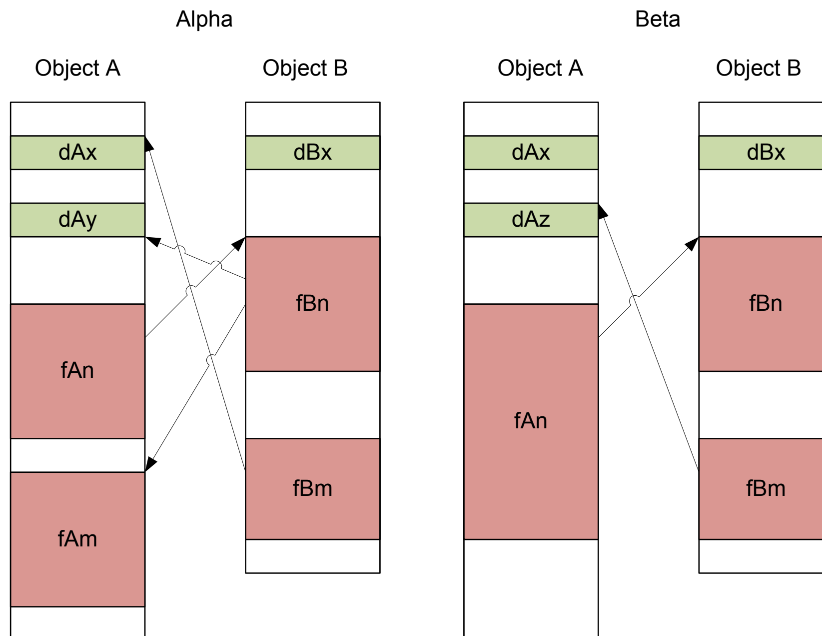

Figure 3.6: Object Files from versions *Alpha* and *Beta*

Examining the symbol references in figure 3.6 a bit more closely it becomes clear that attempting to recombine and link object A from version *alpha* with object B from version *beta* will be problematic. This is because function fBm from object B has an unsatisfied symbol reference to a global data item dAz, which is not present in object A of version *alpha*. Likewise, attempting to recombine and link object B from version *alpha* with object A from versions *beta* will fail because object B will have and unsatisfied symbol reference to function fAm.

3.4.2 Runtime Problems

Though object files under go less validation than source code, ultimately object files are merely an abstraction of the source code they were generated from. As such, source code changes between program versions can introduce runtime issues in the recombined program that cannot be overcome during the linking. In fact, even the smallest of changes can cause disastrous runtime problems. These source code changes can be classified into one of two distinct categories: syntactic changes and state changes.

Syntactic Changes

Syntactic changes to externally accessible functions, global data, and common data types can have catastrophic effects on any recombined program which will likely cause the program to terminate abnormally at runtime. Because syntactic changes are validated by the compiler, the ability to link the object files and produce an executable is not effected because the symbols are generally not effected. Consider a function, shared data, and data type defined in C as:

```
int* func(char arg1, int arg2);

int data[10];

typedef struct _DATATYPE {
    int member1;
    int member2;
    char member3;
 } DATATYPE, *PDATATYPE;
```

These items can be changed between software program versions and software program forks in many ways. Listed below are several examples of syntactic changes to these items which will undoubtedly lead to undesired behavior or abnormal termination of a recombined program.

1. Adding or removing additional function arguments, thereby changing the required function input arguments between versions. This will cause a calling function to pass arbitrary values as input. Consider a change such as:

```
int* func(char arg1, int arg2, int arg3);
```

When this changed function attempts to use the last argument, it will be accessing a previous 4-byte value in memory. In all likelihood, this value will not be valid for the function and will cause undesired behavior.

2. Re-arranging function arguments, which changes the order of the arguments between versions. This means that a calling function could pass the correct input, but in the incorrect order. Consider a change such as:

   ```
   int* func(int arg2, char arg1);
   ```

   Depending how the function uses the input arguments, this may work correctly; but in the general case, a change such as this will cause undesired effects.

3. Modifying the return type, causing a calling function from a different version to possibly misinterpret the returned data. Consider a change such as:

   ```
   int func(char arg1, int arg2);
   ```

   If the calling function is expecting a pointer to an integer as a return value, and instead receives and integer, any attempt to dereference that integer will likely cause undesired effects.

4. Modifying the type or size of the global data, causing any code from a different version which uses the data to misread and / or misinterpret. Consider changes such as:

   ```
   char data[10];
   ```

   or

   ```
   int data[5];
   ```

   Changing the data type or size alters the semantics of the data and the amount of memory required to store the data. Any function not aware of this change will likely misinterpret the data, or even read and write past the end of the data therefore using and creating invalid data.

5. Changing the mutability of the data, which causes invalid write operations during the execution of the recombined program. Consider a change such as:

   ```
   const int data[] = {0,1,2,3,4,5,6,7,8,9};
   ```

   In this case, the type and size of the data has not changed. Instead the compiler will mark the data as read-only, where previously it was writable. This will cause an access violation if any unaware function attempts to write to the data.

6. Altering the data type definition by adding or removing members, such as:

   ```
   typedef struct _DATATYPE {
       int member1;
       char member3;
    } DATATYPE, *PDATATYPE;
   ```

   In this case, because a member was removed, all subsequent members now reside at different offsets in the data type. Any code which was compiled to use the previous definition of this data type will inadvertently access $member3$ while attempting to $member2$. Similarly, any attempt to access $member3$ will result in accessing invalid data beyond the size of the new data type leading to undefined behavior.

State Changes

State changes are any changes that alter expected internal or external state between object files. These can include changes to the format or location of files, changes to the number or type of system resources used, or changes to the internal understanding of how data is being processed within the program. When these changes occur between program versions or forks, they will likely lead to runtime errors when those changes affect the interaction and exchange of data between object files as sub-routines are invoked. Similar to syntactic changes, state changes will not affect the ability to link the program's object files because no symbols are changed. Unlike syntactic changes, however, state changes may not be as catastrophic and may not abnormally terminate the running program on their own. In general it is more reasonable to

expect undesired behavior during the execution of the recombined program because well-designed and well-written sub-routines should be hardened against invalid or erroneous data.

3.5 Summary

This chapter has presented a biologically inspired approach for selecting software programs to be recombined and where the act of recombination between the programs should occur.

Based upon the species concepts which exist in biology, existing software programs selected for recombination should be ones which are identified to be closely related. To be more specific, the various versions and forks of a similar software program should be considered as the candidate software programs for recombination. This criteria for selection ensures that the software programs are functionally similar and increases the likelihood that the programs are compatible for recombination without any of the requirements used in previous techniques.

While previous attempts at software reuse and recombination have focused on recombining the source code of the programs, this chapter has proposed program recombination using object files. While not without problems, this method of recombination is far less complex than source code recombination with respect to the amount of validation that is performed against the recombined program. Like living organisms, duplicate genes in the form of multiple versions of object files, can possibly occur. It is expected that using multiple object files will either provide similar functionality to that of either parent or provide new combinations functionality from both parents. There are, however, open problems with respect to object file recombination. These problems include linking problems and runtime problems, and they must be addressed.

Chapter 4

ObjRecombGA

This chapter addresses the aforementioned problems surrounding the recombination of programs at the object file level and presents ObjRecombGA, a software program that automates the recombination of closely related GNU/Linux based software programs written in C. The first two sections of this chapter provide solutions to the linking and runtime problems. The last section describes the implementation of these solutions within ObjRecombGA.

4.1 Correcting Linking Problems

Correcting linking problems between a set of object files simply means ensuring that all linking validation rules are satisfied. With respect to recombining the object files of two software programs, it is assumed that all symbol references are resolvable and defined in at least one of the two programs. If this were not the case then either software program on their own would not be able to be linked because each would be missing one ore more required symbols. Therefore, correcting linking problems across the object files of the two programs becomes a matter of ensuring each symbol is defined only once. This will not, however, correct the runtime problems.

Going back to the example in figure 3.6, regardless of how the object files are chosen from each version, they cannot be recombined and linked together. In both cases the linker will report an undefined symbol error and will fail to link the object files together in a conventional manner. However, this is only true when limited to one version of object A and one version of object B. Instead, consider what happens if object A from *alpha* is recombined with object B *and* object A from *beta*. In this case some external symbols, namely dAx and fAn, will be defined more than once. To solve this problem, it is possible to simply rename these symbols in object A from version *beta*. In doing so, a new copy of object A from version *beta*, labeled A', is created with new symbols dAx' and fAn'. Figure 4.1 illustrates the new references

between these three object files. As can be seen, all external references are satisfied, and no symbols are defined more than once. A linker is now able to link these object files and produce a program executable.

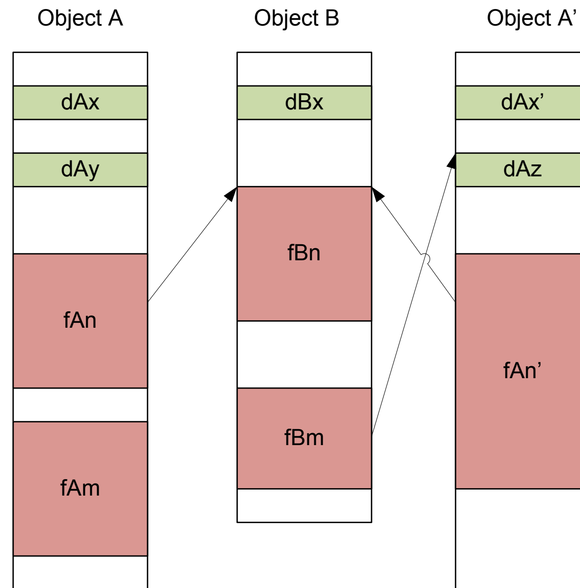

Figure 4.1: Recombining object A from *Alpha* and object B from *Beta* by including object A'.

In order to implement this corrective linking procedure it is necessary to first process the object files to find all symbols and symbol references then manipulate the object file symbols to remove all multiply defined symbols.

4.1.1 Processing the Object Files

In order to determine all of the multiply-defined symbols between the object files of two related software programs, it is necessary to first analyze the set of object files within each software program to record their set of globally bound symbols. Recall that the set of globally bound symbols for an object file identifies the functions or data that the object file is dependent on from another object file, or that are available for other object files to reference. With all object files analyzed, the entire set of globally bound symbols can be used to establish the interdependency between all object files and libraries required to link either software program.

Once a chosen common set of object files is selected from both parent programs (as described in section 3.3.2) the global symbols of each object file in the set are

examined further. If any symbol of an object file in the set cannot be resolved within the set of object files itself, the dependent object files from the corresponding parent program is added to the set to satisfy that symbol resolution. The symbol itself, and object file it resolves to, is recorded. This process continues until all symbols within all object files in the set are resolved or the remaining symbols cannot be found within the object files of either program. These symbols that are not resolved within either program are assumed to be resolved when the actual linking takes place by simply including all of the required libraries for both programs. The final set of all common and dependent object files that has been identified is labeled as the recombination set.

4.1.2 Manipulating the Object Files

After having determined the entire set of required object files, all that remains is to remove any multiply defined symbols. This is done by enumerating the recombined set of object files and validating that each symbol exists only once. To ensure that this is done in a consistent manner, the object files are enumerated in the order in which they appeared in the set. Object files that appear later in the set take lower precedence over those which appear earlier, and as such, will be among the first to have their symbols modified. Recall that the later object files in the set were inserted because they were identified as being required by object files in the 'original' recombined set. Modifying object files with this ordering enforces a tighter symbol resolution and thus linkage between the common set of object files identified for recombination. This encourages the recombination functionality between the two parent software programs.

The symbol modification of an object file is done by first making a copy of the object file, then simply altering the symbol table entry for that symbol by renaming it[1]. Any renaming scheme will work so long as the renamed symbol does not collide with another existing symbol in the set of object files. Once all necessary object file modifications are made, the collection of modified object files can be placed into a library and this library can be added as input when linking the common object files in the recombination set.

[1]Software tools exist to easily rename symbols within object files

4.2 Addressing Runtime Problems

All of the runtime problems described in the previous chapter are a direct result of source code incompatibilities between two software programs. As such, these issues cannot be fully determined without a detailed examination of their source code. Furthermore, these issues cannot be solved without making source code modifications, which themselves cannot be performed during the examination or linking of the object files. As mentioned in chapter 3, avoiding source code modifications was desirable due to the many complexities involved and was the motivating factor for using object files during recombination. Since there is no guarantee that any runtime issues will actually be encountered, a straightforward approach is to identify and discard any combinations of object files which fail to run or exhibit runtime failures. The proposed approach is to use a genetic algorithm to search the space of possible object file recombinations with a fitness function that identifies runtime errors and returns very poor fitness for malfunctioning recombined programs. This identification of runtime errors in the fitness function will steer the genetic algorithm away from using a similar combination of object files in future generations thereby weeding out unstable combinations.

4.2.1 Identifying Problems at Runtime

A program that terminates execution without being instructed to by the user is referred to as an abnormal termination[55]. This behavior can be caused by a number of program runtime problems such as execution of an invalid or privileged instruction, reading or writing to invalid memory, or division by zero[55]. When this occurs, the execution of the program is trapped by the operating system. In the case of the GNU/Linux operating system using the Bash[2] shell, a program termination file called a *core dump file* is generated. This file can be used to diagnose the program errors, and contains the state of the program as it existed in memory at the time that the program termination occurred[90]. In the case of the Windows operating system, a dialog window describing the program termination is generally presented to the user and an application event log entry is generated.

[2] Bourne Again SHell - typically the default shell in GNU/Linux operating environments

Many of the runtime problems identified in chapter 3 are expected to lead to abnormal program termination. As such, providing a GA fitness function which is able to identify these reports and negatively adjust the fitness of the recombined variant that generated the error will encourage the GA to search more stable variants that do not contain runtime problems.

4.3 Implementation Details

ObjRecombGA is a Java-based application that automates the processes described in the previous section to recombine the object files of two-closely related software programs, producing many recombined program variants. As currently implemented, ObjRecombGA is limited to recombining C programs written for the Linux platform and compilable using *gcc*. These limitations were chosen because utilities, specifically the GNU Binutils[3], exist which offer the ability to parse, copy, and manipulate ELF object files created from compiled C programs.

ObjRecombGA performs its software recombination using a two stage process. During the first stage the two previously built program versions are parsed for object files and each program's object files are pre-processed to determine an overlapping set of common object files which are candidates for recombination. Next, all symbols and symbol references within all object files and libraries of each version are cataloged and parsed to determine all interdependencies between object files. During the second stage, ObjRecombGA uses a genetic algorithm (GA) to discover functional recombined program variants of the given software programs. Functional variants are discovered by using the pre-processing results to correct potential linker errors between the selected object file recombination allowing for the creation of a recombined program executable. This executable is then tested using a provided set of tests, called a fitness script, to determine the fitness for the variant.

In addition to the pre-processing results, the GA also requires: a temporary working directory; a population size, s; a desired number of generations, g; a selection method, $S(x)$; a crossover method, $R(x)$, build strings for the program versions being recombined, and the location of each versions object files on disk. The ObjRecombGA user interface and required input parameters are shown in Figure 4.2.

[3]http://www.gnu.org/software/binutils/

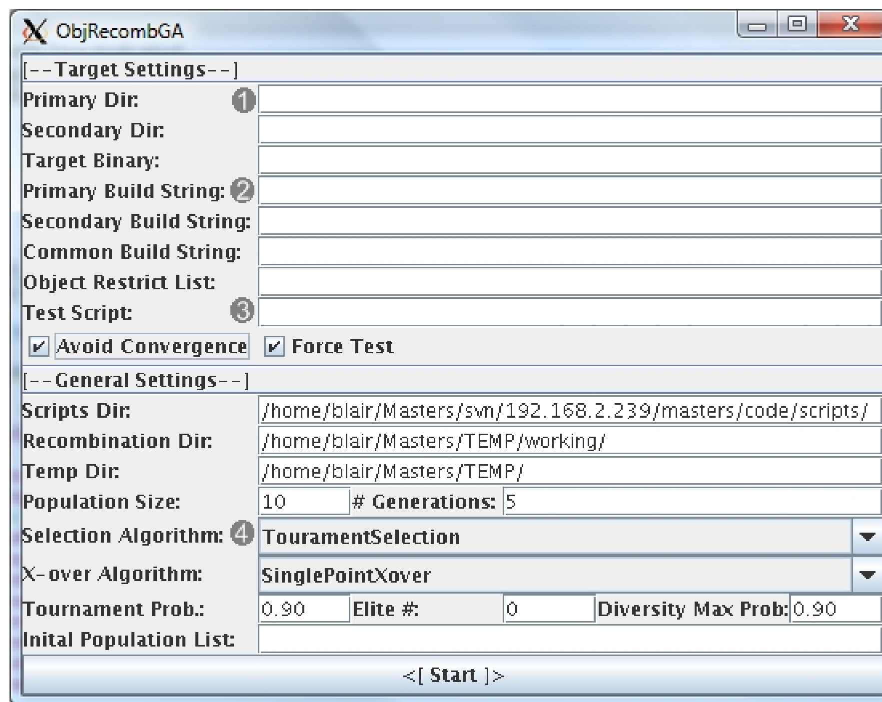

Figure 4.2: The ObjRecombGA user interface showing input for: (1) The program version file locations; (2) the program version builds strings; (3) the program fitness test script; (4) the GA selection and crossover function settings

4.3.1 Pre-processing the Object Files

The pre-processing of the two program versions, *alpha* and *beta*, object files is done as follows:

1. Build the two program versions, *alpha* and *beta*, and extract the final *gcc* build string needed to produce the final program binary for each.

2. Identify one program version, *alpha* or *beta*, as the *primary* recombination version and the other as the *secondary* version. This is strictly used for internal bookkeeping purposes.

3. Create a common build string by merging the two program build strings together and removing any duplicate files, flags, or libraries. This common string contains only common linker flags and external library files required for both versions. It also contains a *magic* string that ObjRecombGA will replace with the list of common object files to be included during linkage.

4. Using both program version paths, the common build string, and the default compiler libraries, create a list, Z, of all *external* object files and library names $(z_1, z_2, ..., z_p)$. These are object file and libraries that are not part of either program but are required to link either program versions.

5. $\forall z_i \in Z$, use *objdump* (a software tool found within GNU Binutils) to create a set z_iE of all exported symbol names $(z_1E, z_2E, ..., z_pE)$

6. Create a global export set $ZE = (z_1E \bigcup z_2E \bigcup ...z_pE)$

7. Create two sets, X and Y, of all the object file and library filenames $(x_1, x_2, ..., x_n)$ and $(y_1, y_2, ..., y_m)$ for versions *alpha* and *beta* respectively. These object files and library names are identified by simply parsing the build command for each version respectively.

8. $\forall x_i \in X$, use *objdump* to create a set x_iE of all exported symbol names $(x_1E, x_2E, ..., x_qE)$ and set x_iI of all import symbol names $(x_1I, x_2I, ..., x_rI)$ such that $x_iI \notin ZE$

9. $\forall y_i \in Y$, use *objdump* to create a set y_iE of all exported symbol names $(y_iE, y_iE, ..., y_sE)$ and set y_iI imported symbol names $(y_1I, y_2I, ..., y_tI)$ such that $y_iI \notin ZE$

10. Create the *common* set, C, such that $\forall c_i \in C, c_i \in X \cap Y$. This represents the matching filenames of the objects and libraries that are available for recombination between the two program versions.

It is worth noting that the first three steps - building the program versions, choosing a primary and secondary version, and creating a common build string - are done manually before running ObjRecombGA and are simply provided as input.

4.3.2 Program Variant Generation

Once the pre-processing of the object files has been completed, ObjRecombGA discovers recombined program variants using a genetic algorithm as follows:

1. Create an initial program variant population set, P, of size s, where each member is a random bit string of size $\mid C \mid$ and is associated with a unique identifier.

2. While $g > 0$, repeat the following:

   i. $\forall p \in P$, decode p_i using the program variant decoding function $D(p)$ defined below.

   ii. $\forall p \in P$, calculate the fitness $p_i f$ using the fitness function $F(p)$ defined below.

   iii. Create a new, empty population set, P'.

   iv. Select and crossover two program variants, $\{p_i, p_j \in P\}$, using a selection function $S(P)$ and a crossover function $R(p_x, p_y)$, to produce a two new program variants p'_{ij} and p'_{ji}

   v. Insert p'_{ij} and p'_{ji} into P'.

   vi. Repeat steps (i. - v.) until $\mid P' \mid = \mid P \mid$.

   vii. Replace P with P' and reduce g by 1.

The program variant decoding (which resolves linker problems), the fitness function, selection functions, and crossover functions used by the GA are now described in more detail.

4.3.3 Program Variant Decoding Function

Decoding of a program variant is performed by determining which object files from the common set are represented by the program variants bit string and then resolving all their symbol references. This process is complicated by the fact that these object files may have external references to other object files which exist outside the common set. The decoding function must, therefore, examine all external references from the results of the pre-processing stage to determine all object files required to successfully link the program variant. The program variant decoding function copies any required object files and modifies any symbols in those object files which may cause linker errors. These copied and modified object files are then placed in a library and the common build string is modified to include this new library. More formally, the program variant decoding function, $D(p)$, processes the solution as follows:

1. $\forall p_i$ bits in the bit string, $if p_i = 1$, randomize program variant p and call $D(p)$ (this is to avoid convergence on the *primary* program version).

2. $\forall p_i$ bits in the bit string, $if p_i = 0$, randomize program variant p and call $D(p)$ (this is to avoid convergence on the *secondary* program version)

3. Create a new unique directory for the program variant based on its unique id.

4. Create an empty list, O, which will contain the object files $(o_1, o_2, ..., o_m)$ required outside the common set.

5. Create an empty list, S, which will contain the required symbols names $(s_1, s_2, ..., s_m)$ from the object files in list O.

6. Create an empty list, E, which will contain the export symbols $(e_1, e_2, ..., e_m)$ from the common set.

7. $\forall p_i$ bits in the bit string identify all export symbols from the common set by performing the following:

   i. if $p_i = 1$: find j such that $X_j = C_i$. Next, $\forall X_jE_l \in X_jE$ add symbol X_jE_l to list E.

   ii. if $p_i = 0$: find j such that $Y_j = C_i$. Next, $\forall Y_jE_l \in Y_jE$ add symbol Y_jE_l to list E.

8. $\forall p_i$ bits in the bit string identify the corresponding object file and all its dependents based on its symbol references by performing the following:

   i. if $p_i = 1$: find j such that $X_j = C_i$. Copy file C_i from the primary version to the program variant directory. Next, $\forall X_jI_l \in X_jI and X_jI_l \notin E$, $\forall X_kE(k = 0 \rightarrow \mid X \mid)$, if $X_jI_l \in X_kE$ then: if $X_k \notin O$ then add X_k to list O; add symbol X_jI_l to list S. Recurse this process on X_k.

   ii. if $p_i = 0$: find j such that $Y_j = C_i$. Copy file C_i from the secondary version to the program variant directory. Next, $\forall Y_jI_l \in Y_jI and Y_jI_l \notin E$, $\forall Y_kE(k = 0 \rightarrow \mid Y \mid)$, if $Y_jI_l \in Y_kE$ then: if $Y_k \notin O$ then add Y_k to list O; add symbol Y_jI_l to list S. Recurse this process on Y_k.

9. Create an empty list, O', which will contain the file names of the modified object and library files.

10. For each $o_i \in O$ perform the following:

    i. Create a list O_iS to contain the symbols from S corresponding to object file o_i.

    ii. $\forall o_j \in O(i > j)$, if $o_i = o_j$ insert s_j into O_iS and remove o_j from O and s_j from S.

    iii. Copy file o_i to the *temporary directory* .

    iv. Using the *objcopy* utility, for each symbol, t in file o_i, if $t \notin O_iS$, rename t to t'[4].

    v. Add o_i to list O'.

11. Using the *ar*[5] utility, create a library of all files in O'.

12. Move the newly created library to the program variant directory and add it to the build string for the program variant.

13. Remove all files from the *temporary directory*.

4.3.4 Fitness Function

In order to determine if ObjRecombGA has generated a usable program variant, two conditions must be met: the program variant must successfully link to produce a program executable and the program variant must execute successfully without terminating abnormally. Both of these conditions can be validated by attempting to link the object files and running the program variant if linking was successful. If the program variant fails to link correctly, then no binary will be produced. If the program variant fails to run properly, or terminates abnormally, then it is expected that a core dump file will be generated.

If the program variant does in fact run successfully, then ObjRecombGA will run a user provided fitness script to test the program variant further. This fitness script is expected to be a Linux (*bash*) shell script which accepts the path to the program variant executable and the unique program variant directory as its only two

[4]Each renamed symbol is simply prefixed and suffixed with an underscore. This renaming scheme was chosen arbitrarily.

[5]Part of the GNU Binutils package

inputs. Additionally, the fitness script is expected to return an integer value which will augment the fitness value of the program variant being evaluated.

Evaluating the fitness of a population member, p, is done using the fitness function $F(p)$ defined as follows:

1. Set the fitness value for the program variant, f, to 0.

2. Run the modified build string (as determined by the decoding function) to produce the program variant executable.

3. Check to see if the build was successful, incrementing f by 1 if a program variant executable was produced, otherwise return f as the fitness value for this program variant.

4. Execute the program variant. After 5 seconds of execution, terminate the program variant if it is still executing.

5. Examine the program variant directory for a *core dump file*, incrementing f by 1 if one does not exist, otherwise return f as the fitness value for this solution.

6. If a fitness script was provided by the user, run the script and add the returned value to f.

7. Return f as the fitness value for this program variant.

4.3.5 Crossover & Selection Functions

ObjRecombGA includes an implementation of two crossover functions and one fitness-based selection function. The crossover functions being used by ObjRecombGA are the *Single Point Crossover* and *Two Point Crossover* methods.

The Single-Point Crossover method recombines solutions by selecting a single position in the solution at random and swapping the segments, bounded by this position and the last position, between the two parents[64, p. 171-172]. Figure 4.3 illustrates the Single-Point Crossover method with position 8 being chosen at random.

The Two-Point Crossover method recombines solutions by selecting two positions in the solution at random and swapping the segments, bounded by these two

	1	2	3	4	5	6	7	8	9	10	11	12	13
Parent 1	1	1	0	0	1	0	0	1	0	0	1	0	1
Parent 2	0	0	1	0	0	1	1	1	0	1	0	1	0
Child 1	1	1	0	0	1	0	0	1	0	1	0	1	0
Child 2	0	0	1	0	0	1	1	1	0	0	1	0	1

Figure 4.3: Single-Point Crossover swapping the segment $(8, 13)$ between *Parent 1* and *Parent 2* to produce *Child 1* and *Child 2*

positions, between the two parents[64, p. 171-172]. Figure 4.4 illustrates the Two-Point Crossover method with positions 5 and 11 being chosen at random. Two-Point Crossover is advantageous when the size of a solution is large and solutions may contain large segments in the solution which are valuable. Single-Point crossover is more likely do destroy these large segments [64, p. 171-172].

	1	2	3	4	5	6	7	8	9	10	11	12	13
Parent 1	1	1	0	0	1	0	0	1	0	0	1	0	1
Parent 2	0	0	1	0	0	1	1	1	0	1	0	1	0
Child 1	1	1	0	0	0	1	1	1	0	1	0	0	1
Child 2	0	0	1	0	1	0	0	1	0	0	1	1	0

Figure 4.4: Two-Point Crossover swapping the segment $(5, 11)$ between *Parent 1* and *Parent 2* to produce *Child 1* and *Child 2*

The fitness-based selection function used by ObjRecombGA is the *Tournament Selection* method. This selection method involves randomly selecting n individuals from the population to form a tournament group. A tournament parameter, k where $(0 \leq k \leq 1)$, is compared against a randomly generated number, r, between 0 and 1. If $r < k$, the fittest member of the tournament group is chosen as a parent, otherwise a parent from the tournament group is chosen at random. All members of the tournament group are placed back into the population and the process is repeated to find the second parent [64, p.170-171]. The selection parameters, such as the tournament probability and the number of solutions selected for "Elitism", are user-defined at runtime. "Elitism" is an extension of the selection method which ensures that a portion of the fittest population members are retained at each generation. This

generally improves the GAs performance because each new generation is now seeded with desirable population members[64, p. 168].

4.4 Implementation Challenges

While implementing ObjRecombGA, several challenges were encountered and overcome. These include dealing with object files that contain common block symbols and how to best test the recombined variants created by ObjRecombGA.

4.4.1 Common Block Symbols

As previously discussed, when examining object files together, the linker is primarily concerned with resolving symbols across object files such that no symbols are multiply-defined and all undefined symbols can be resolved. However, object files generated from C code can have a specific type of symbol known as common block symbols[54][6]. Common block symbols are symbols that represent uninitialized data items within an object file and they cannot be linked against as they have no associated storage space within the object file. When the linker encounters a common block symbol; it reserves storage space within the final binary and all other matching common block symbols from other object files will also refer to this reserved storage space[54]. In effect, the symbol that represents this data item can be declared multiple times so long as it is always declared uninitialized and the linker will happily combine all these uninitialized instances into a single instance with only one storage space. If, however, any source file provides an initialized instance of this variable then a global symbol with storage will be created by the compiler and this symbol will collide with all common block symbols causing the symbol to be multiply defined.

During the development and testing of ObjRecombGA there were a handful of instances where the variables declared in a given program had been modified from being uninitialized to being initialized across different versions. Recombining the object files between these versions resulted in multiply-defined symbols, and thus linkage failure between the common block symbols in earlier versions and the actual symbols in later versions. In order to deal with these situations, limitations were put

[6]Also known as tentative symbols. Originally introduced in Fortran

into ObjRecombGA such that common block symbols are treated as normal symbols when any non-common block symbol with the same name is identified. This means that common block symbols may end up being renamed such that they no longer point to a common storage location. This will ultimately introduce runtime issues into the resulting program because these symbols, which should all be pointing at the same storage location, are pointing to unique storage locations. However, common block symbols are rare and this limitation had minimal impact on the ability of ObjRecombGA to identify and create successful recombinations of object files during testing.

4.4.2 Library Version Conflicts

The object file manipulation performed by ObjRecombGA is localized to the object files from the parent software programs being used for recombination. This means that any additional libraries needed to build either parent program are not subject to manipulation. Should a particular library dependency change between the parent versions or forks, linker errors may be introduced that would limit the success of the recombination.

For instance, consider a library that removes a handful of functions in its latest version such that it is no longer compatible with the previous versions. If the first parent program uses the older library version and the second parent program uses the newer library version it will not be possible to link against both library versions because this will introduce multiply-defined symbol linker errors. Similarly, in some sets of object files it may not be possible to link against either version of the library because this will introduce undefined symbol linker errors.

ObjrecombGA provides no solution to this problem. Instead, the build string which specifies the libraries to use are provided as input to ObjrecombGA and it does the best it can to find successful variants. Should a library version conflict arise for a given set of object files, the linker will simply fail to link these object files together. This will result in a very poor fitness score such that the object file set will not be considered in subsequent generations of the GA.

4.4.3 Testing of Recombined Variants

Though ObjRecombGA provides the ability to automatically run and test the recombined variants it generates, determining an appropriate set of tests for the variants is difficult. It is difficult because the recombined variants will likely contain broken functionality or functionality that is a combination of both programs, creating potentially inconsistent output results from the tests that are run. Moreover, each parent program may have significantly different functionality such that a single test is unable to execute properly without causing runtime problems or crashes. As such, the purpose of using testing as a fitness measure is not to exhaustively test all functionality of each variant within the GA but rather test some small subset of expected functionality such that the variants can be shown to function in some capacity. Determining how to measure this functional capacity is also difficult because once the functionality of either parent program is altered, it will effect the expected output of any tests run. Therefore when examining any test results it is important to expect some variation or use of fuzzy matching.

4.5 Summary

This chapter has presented two classes of problems that can occur when recombining the object files of two closely related software programs, namely linking problems and runtime problems. Solutions and approaches to dealing with these problems have also been presented and implemented within a software program entitled ObjRecombGA.

Linking problems between sets of related object files are solved by manipulating the symbolic information within those object files such that they can be successfully linked together. ObjRecombGA solves this by cataloging all object files and their symbolic information such that dependencies between object files can be identified. When presented with one possible combination of object files, ObjRecombGA resolves all dependent objet files and symbol information, then modifies any duplicate symbols such that linking and creating a recombined software program variant will be successful.

Runtime problems, unlike linking problems, cannot be identified or solved through object file analysis or modification. Instead, ObjRecombGA simply tries to avoid

recombined variants in the search space that have runtime problems by using a genetic algorithm paired with an appropriate fitness function which can run the program through a series of tests. When runtime problems are identified or suspected for a given variant, the GA fitness function will give that variant a low scoring fitness value such that the particular combination of object files of that variant is less likely to survive to subsequent generations of the GA.

Chapter 5

Results

This chapter discusses the results obtained from running the ObjRecombGA software on a group of candidate programs. First, reasons for which these program were selected will be discussed. Following this, a brief overview of each program and a detailed set of results are presented and discussed.

The desired results of the ObjRecombGA testing are threefold. First, to show that the recombination of closely related software programs using object file manipulation is feasible; second, that using object file as a recombination method scales to even large complex software programs with many object files; and lastly, that new previously unseen combinations of functionality can be discovered.

5.1 Selecting Candidate Programs

Selection of the candidate programs for recombination is based on the following factors and limitations of the ObjRecombGA software:

- The candidate programs must have several versions or forks available to test. This allows for more recombination scenarios using multiple combinations of program versions.

- The candidate programs should make minimal use of signal handling, so that abnormal program termination will result in a core dump file being generated. ObjRecombGA uses the presence of core dump files to identify recombined program variants that have critical runtime failures.

- All versions of the candidate program must be able to be built with the same compiler version. This ensures consistency across the object files and avoids the complications of supporting multiple compilers. Specifically, compiler optimization adjustments between various compiler versions may introduce more runtime issues than those examined in Chapter 3.

Quake	Dillo	GNU-sed
libc-2.7.1	libc-2.7.1	libc-2.7.1
libX11-6.2.0	libX11-6.2.0	
libXext-6.4.0	libXext-6.4.0	
libSDL-1.2	libpng-1.2	
libXxf86dga-1.0.0	libpthread-2.7.1	
	libgtk-1.2	
	libgdk-1.2	
	libgmodule-1.2	
	libXi-6.0	
	libjpeg-6.2	

Table 5.1: Additional libraries required to successfully link the selected versions of Quake, Dillo, and GNU-sed.

- All versions of the candidate program, once compiled, should exist as a single executable binary. This simplifies recombination and testing because there will only need to be one build command to execute and one program binary to test.
- The program executable should accept command line arguments for program input. This is necessary in order to write a fitness script to exercise the functionality or alternate code paths of the recombined program variants and provide a good fitness measurement.

For testing, three different programs with vastly different functionality and code base sizes were used. Each of these programs fit the criteria outlined above and allowed for testing the recombination of software programs with varying size and complexity. The three chosen programs were: GNU-sed [26]; Dillo [88]; and Quake [85]. The testing these programs with ObjRecombGA was done on an Intel x86 based PC running Ubuntu Linux 8.04. GNU Make version 3.81 and GCC version 4.2.4 build tools were used in addition to several specific libraries listed in table 5.1 were required to successfully compile and build all versions of GNU-sed, Dillo, and Quake.

5.2 GNU-sed

GNU-sed (sed) is a stream editor which is used to perform text-based transformations on an input stream. GNU-sed is considered to be an efficient editor due to the fact that all text transformations occur using only a single pass over the input stream [28].

Text transformations using sed are performed by using a sequence of sed commands. These commands can be grouped together in a file which is referred to as a sed program or sed script. When the sed executable is invoked, the script file is provided to sed via an input argument and the commands within the script file are executed against the desired input stream. Although there are only a limited number of sed commands, these commands can be grouped together to enable very complex text transformations [28].

GNU-sed was chosen due to its small number of object files and the fact that it has a source code base that does not undergo a large number of changes between program versions. GNU-sed is also a command-line based tool , thereby making it easy to test any recombined program variants using a shell script. Though not many exciting results were expected, the results gathered were encouraging enough to consider larger and more complex software programs.

5.2.1 Fitness Calculation

To calculate the fitness of each GNU-sed program variant, a fitness script (listed in Appendix A) was written to perform a set of six tests. The fitness script executes a set of sed scripts, taken from the Sed Script Archive [7], which exercises a wide range of functionality provided by sed. The scripts are executed in sequence as follows:

1. `head.sed` - output only the first 10 lines of text from the input stream

2. `remccomms1.sed` - output a C program file (provided as the input stream) with all C-style comments (those contained within matching /* and */ comment delimiters) removed

3. `indentls.sed` - Output a path depth indented copy of the input stream provided by a recursive directory listing (created using the "ls -lr" command)

4. `revchr_1.sed` - output the entire input stream with each line reversed

5. `cflword5.sed` - output the entire input stream with the first letter of the first word in each line capitalized

6. `sierpinski3.sed` - output a text representation of a Sierpinski Triangle[1] based on a pre-formatted input stream which determines the triangle size

Each of these scripts were run against all tested GNU-sed versions to verify that each version is capable of executing each script. In order to test the output from the sed scripts as they were run against recombined program variants, each script was run using the latest version (4.1.5) of GNU-sed and each script's output was captured to a file as reference.

After each of the sed scripts have been executed by a program variant, the fitness script will validate the output against the reference output. This is accomplished by capturing all output to a file, then performing a checksum calculation and comparing the captured output to the reference output. If the checksums match, the fitness script increments the script return value (R) by 1 and continues to the next sed script. If the checksums do not match or there are no remaining sed scripts, the current value of R is returned. The maximum value which can be returned by this fitness script is 6, which is then paired with an addition score of 2 if the program variant being tested links correctly and can be run without crashing (as discussed in Chapter 4). This provides a total fitness range of 0 to 8 for all generated GNU-sed variants.

5.2.2 Versions Tested

At the time of this research, 11 versions (3.01, 3.02, 4.0.6 - 4.0.9, and 4.1.1 - 4.1.5) of GNU-sed were available and could be built using the test environment. The object files available for recombination in each version of GNU-sed are listed in Table 5.2. Each possible version pair of GNU-sed has a minimum of 4 matching object or library files available for recombination. Therefore, ObjRecombGA's search space is bounded by a minimum of 14 ($2^4 - 2$) and a maximum of 126 ($2^7 - 2$) possible recombinations[2] for the GNU-sed version pairs tested.

Several pairs of versions were chosen as input for the ObjRecombGA software. These pairs were tested using single-point crossover and tournament selection with a population size of 12 over 8 generations, a tournament size of 7 with a 90% probability, and using an elitism value of 4. Using these parameters allows ObjRecombGA to

[1]A fractal named after mathematician Waclaw Sierpinski

[2]Recall, the solution of all 0's and all 1's is ignored.

Versions	Object Files
All	sed.o, compile.o, execute.o, libsed.a
4.0.6 - 4.0.8	regex.o
4.0.6 - 4.1.5	fmt.o
4.0.9 - 4.1.5	regexp.o
4.1.1 - 4.1.5	mbcs.o

Table 5.2: GNU-sed object files available for recombination.

Version Pair	# Files for Recombination	#Variants Generated	#Unique	#Unstable
4.0.6 - 3.02	4	68	12	9
4.0.9 - 3.02	4	68	12	19
4.0.9 - 4.0.6	5	68	21	0
4.1.2 - 4.0.6	5	68	21	15
4.1.3 - 4.0.7	5	68	15	4
4.1.4 - 4.0.8	5	68	20	10
4.1.5 - 3.01	4	68	14	12
4.1.5 - 4.1.1	7	68	22	0

Table 5.3: Uniques and stability results for the tested GNU-sed pairs.

produce a maximum of 68 recombined program variants for each tested pair. Due to the limited number of possible solutions, larger population and generations were not considered practical. Additionally, Single-Point Crossover was chosen over Two-Point Crossover because there was a small number of matching object files between any two versions.

Table 5.3 lists each tested pair, the number of matching object or library files available for recombination, the number of total and unique variants produced, and the number of unstable variants generated. Variant uniqueness was determined by comparing the bitstrings of all variants from each tested pair. Unstable variants are variants which at some point during the fitness evaluation or the fitness script were abnormally terminated, resulting in a core dump file being generated. In all tested pairs, ObjRecombGA was able to recombine and successfully link all 68 created variants. This result shows that the manipulation of the object files is working correctly to produce the recombined program executable.

Figure 5.1 and Figure 5.2 chart the average fitness value and number of stable variants at each generation respectively. A variant is considered stable if it completes its fitness evaluation and fitness script without generating a core dump file from an

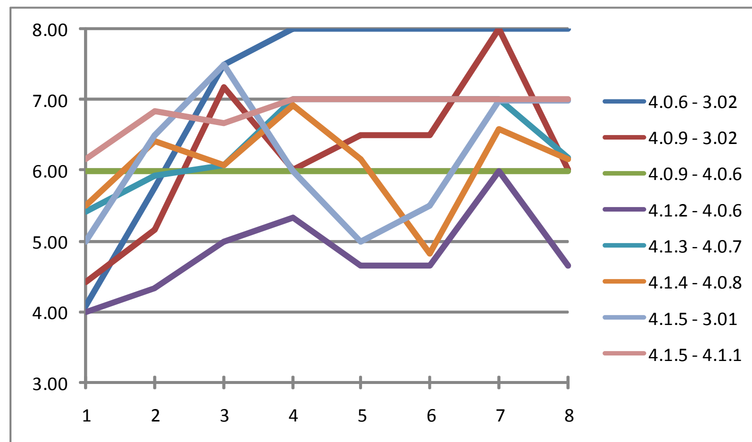

Figure 5.1: Average fitness at each Generation

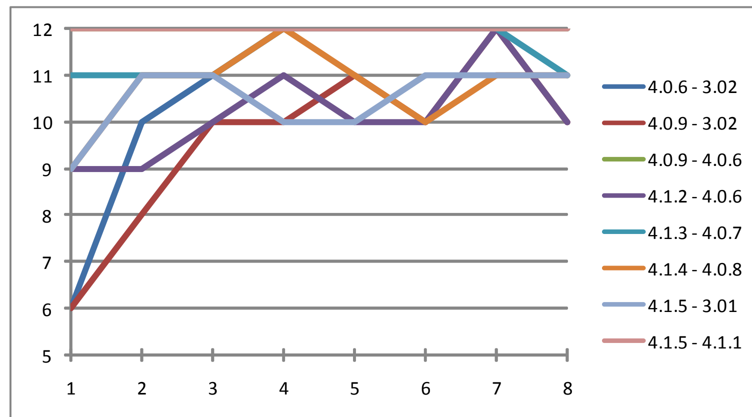

Figure 5.2: Number of stable program variants at each generation.

abnormal program termination.

Looking at the results, it can be seen that the specific versions of GNU-sed selected for pairing have an impact on the fitness and stability of the recombined variants. Specifically, version pairs 4.0.9 with 4.0.6 and 4.1.5 with 4.1.1 showed a consistently high levels of fitness and stability throughout all generations. However, versions pairs 4.0.9 with 3.02 and 4.1.2 with 4.0.6 showed consistent instability and low levels of fitness. In all cases ObjRecombGA was able to quickly weed out many unstable variants with all version pairs tested having over 80% of their population stable by the 3rd generation.

5.2.3 Analysis

From the results, it seems clear that version pairs with a closely related development history in terms of version number are more likely to create stable variants with high fitness while version pairs that have version numbers further apart will be less stable and less fit. This becomes clear when looking at the percentage of stable variants for the closest pair (4.0.9 and 4.0.6) at 100% compared to the furthest pair (4.1.5 and 3.01) at 15%.

Looking at the fitness results, the general trend is that the fitness of the recombined variants improves from the lower fitness values seen in the first generation. This shows that the GA is successfully searching the space of possible program variants. Oddly enough, the fitness averages for many of the version pairs seems to be somewhat erratic. This could be caused by subtle code changes, or semantic changes, between the versions pairs that are being exercised and causing the output from the sed scripts to be slightly off, which given the limited ranges of fitness scores, has a drastic effect on the fitness average for the pair. Ultimately, some adjustment to the fitness script is likely needed to smooth out some of these erratic fitness averages.

Together, both the success in the stability of the recombined variants and the correctness of these variants (according to the selected test scripts) confirms that finding stable and working recombined programs using object files as a recombination approach is possible. Moreover, the results suggest that the more closely related the pairs of software programs are with respect to development history, the more stable the recombined variants become.

5.3 Dillo

The Dillo web browser (seen in Figure 5.3) is an HTML 4.01 compliant web browser consisting of five key functional components: Dillo Widget, Dillo Cache, an HTML Parser, an Image Processor and a Dillo Plugin Interface (DPI) Framework [6].

The Dillo Widget component is responsible for the graphical user interface and window rendering. The Dillo Cache component acts as an input / output engine to abstract all network and file activity required by Dillo. The HTML Parser and image processor work with the Dillo Cache and Dillo widget components to read, process,

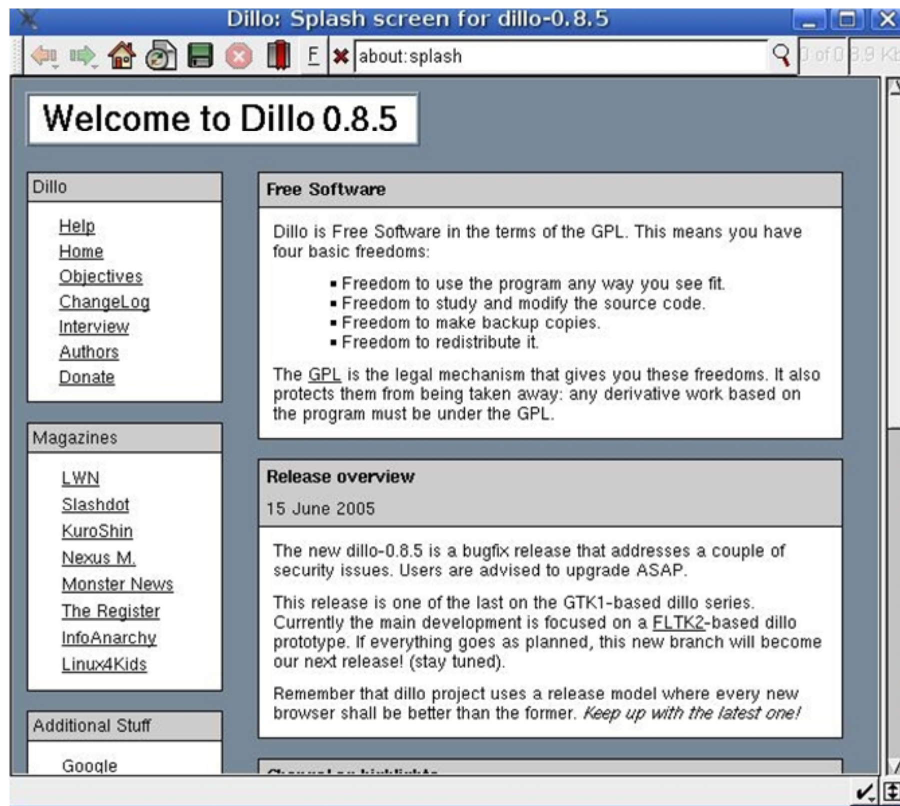

Figure 5.3: The Dillo web browser - Version 0.8.5

and render HTML and images. The DPI Framework is used to allow developers to extend the functionality of Dillo to external programs through a well defined interface [6].

Several pieces of the DPI Framework component are built as separate programs, each with its own linking step, within the Dillo source tree. These pieces are not required to build or run the Dillo web browser program, and as such will not be subject to recombination or testing.

Dillo was selected primarily for its large number of object files and offers a higher level of complexity and less stable source code base compared to GNU-sed. Because Dillo provides a graphical user interface, testing its recombined program variants is much harder to do using a shell script. Manual inspection of stable recombined variants was therefore necessary.

5.3.1 Fitness Calculation

To evaluate the fitness of each Dillo program variant, a fitness script (listed in Appendix B) was written to perform a set of four tests. These tests are:

1. Running the Dillo program variant with no input arguments

2. Running the Dillo program with an input argument requesting that it load an HTML file from the local disk

3. Running the Dillo program variant with an input argument directing it to load the Dillo home page[3]

4. Running the Dillo program variant with an input argument directing it to load the Google home page[4]

Each test was run against each tested version of Dillo to verify that each version is capable of performing them and to capture any existing differences between versions.

These set of tests exercise the variants ability to: simply load without crashing; load and render a simple HTML file from disk; load and render an Internet site containing only HTML; and load and render an Internet site which contains HTML and JavaScript.

After each test is run, the script checks for the presence of a core dump file. If no core dump file is found the script takes a screen shot, terminates the program variant and increments its return value (R) by 1 before proceeding to the next test. In the case of the test which attempts to load a simple HTML file from disk, the script will also verify the file was accessed by assessing the file's *last accessed* time stamp. If the time stamp has been updated, the test script will perform an additional increment of R by 1. Once all tests have completed, or if any test produces a core dump file, the current value of R is returned. A maximum value of 5 can be returned by the Dillo fitness script, giving a possible fitness value range of 0 to 7 for all Dillo program variants.

---

[3]http://www.dillo.org
[4]http://www.google.com

Versions	Dillo Component				
	Dillo Widget	Dillo Cache	HTML Parser	Image Processing	Misc.
All	commands.o, dw.o, dw_bullet.o, dw_aligned_page.o, dw_button.o, dw_container.o, dw_embed_gtk.o, dw_ext_iterator.o, dw_gtk_scrolled_frame.o, dw_gtk_scrolled_window.o, dw_gtk_statuslabel.o, dw_gtk_viewport.o, dw_hruler.o, dw_list_item.o, dw_marshal.o, dw_page.o, dw_style.o, dw_table.o, dw_tooltip.o, dw_widget.o, interface.o, selection.o, progressbar.o, menu.o	cache.o, capi.o, cookies.o, dicache.o, dns.o, libDio.a	html.o, plain.o, colors.o	image.o, jpeg.o, png.o, gif.o, dw_image.o	klist.o, findtext.o, prefs.o, misc.o, bitvec.o, chain.o, url.o, history.o, web.o, nav.o, dillo.o, bookmarks.o[6]
0.8.0 - 0.8.5					strbuf.o
0.8.2 - 0.8.5	gtk_menu_title.o, gtk_ext_menu.o, gtk_ext_menu_item.o, gtk_ext_button.o				
0.8.3 - 0.8.5					dpiapi.o

Table 5.4: Dillo object files available for recombination.

5.3.2 Versions Tested

At the time of this research, 7 versions (0.7.3, 0.8.0, 0.8.1, 0.8.2, 0.8.3, 0.8.4 and 0.8.5) of Dillo were available and could be built using the test environment. Other versions ($< 0.7.3$, 0.8.6, and 2.0) were also available, however they either could not be built successfully in the test environment[5] or they had contained C++ code. The object files available for recombination from each respective Dillo version are listed in Table 5.2. Each version pair of Dillo has a minimum of 51 matching object or library files available for recombination. As such, ObjRecombGA has a minimum search space of $2^{51} - 2$ possible recombinations for every Dillo version pair.

Several pairs of versions were chosen as input for the ObjRecombGA software. These pairs were tested using two-point crossover and tournament selection with a population size of 30 over 20 generations, a tournament size of 7 with a 90% probability, and an elitism value of 8. Using these parameters allows ObjRecombGA to produce

[5] Due to compiler version conflicts, and code which caused compilation errors

Version Pair	# Files for Recombination	#Variants Generated	#Unique	#Unstable
0.8.5 - 0.8.2	56	448	365	0
0.8.5 - 0.8.0	52	448	265	86
0.8.5 - 0.7.3	51	448	233	165
0.8.4 - 0.8.1	52	448	274	145
0.8.3 - 0.8.0	52	448	227	59
0.8.0 - 0.7.3	51	448	348	91

Table 5.5: Uniqueness and stability results for the various tested Dillo pairs.

a maximum of 448 recombined program variants for each tested pair. Larger population and generation sizes were not attempted due to the fact that successful results were found using these parameters.

Table 5.5 lists each tested pair, the number of matching object or library files available for recombination, the number of total and unique variants produced, and the number of unstable variants generated. In all cases, ObjRecombGA was able to recombine and successfully link all 448 created variants; however, not all of these are stable programs. In all but two test pairs, the number of unstable program variants accounted for approximately 20% or more of the total number of variants created. The only exception being version 0.8.5 recombined with version 0.8.2. This particular pair produced the lowest number of unique program variants, however no unstable variants were produced. Interestingly enough, the pair with the widest version gap (version 0.8.5 and version 0.7.3) produced the largest number of unstable variants. This is likely do to the fact that this pair is the furthest apart with respect to version number and meaning they encompass the largest number of code changes between any two pairs in the test group.

Figure 5.4 and Figure 5.5 chart the average fitness value and number of stable variants at each generation respectively. Again, the recombination of version 0.8.5 and 0.8.2 stands out from the rest with the average fitness at every generation being 7, the maximum possible fitness value. Its worth noting that all tested version pairs begin generating stable variants quickly, with all but one version pair (version 0.8.5 and version 0.7.3) having 20 of their 30 population members being stable by the 5th generation. This would suggest that these version pairs are prematurely converging towards a small set of stable program variants with high scoring fitness. Additionally, the version pair 0.8.5 and 0.7.3 highlights that having a wider version gap, with the

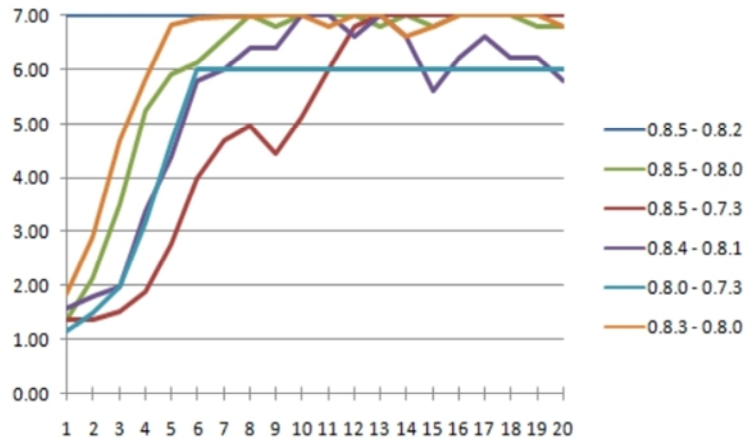

Figure 5.4: Dillo average fitness at each Generation

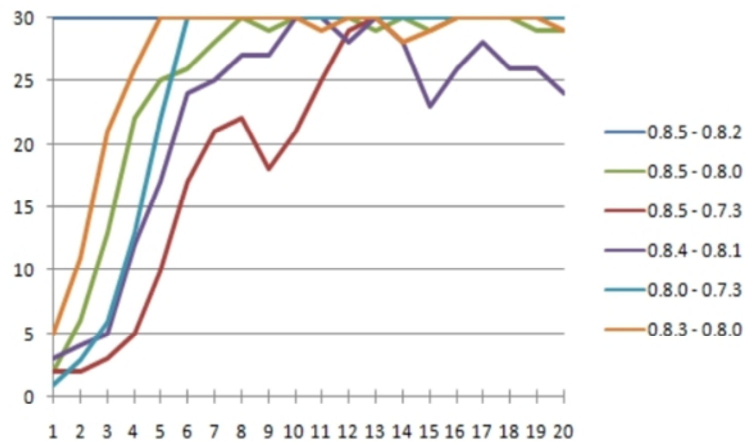

Figure 5.5: # of Stable Dillo variants at each generation.

potential of more code changes, makes it more difficult for the GA to discover a larger set of stable program variants.

Examining the screenshot results captured from the fitness script paints a slightly different picture. Figures 5.6 and 5.7 are the screen shots captured by the Dillo fitness script after running it against version 0.8.5 of Dillo. These will be used as reference when discussing the screenshot results of several noteworthy recombined Dillo variants.

While many of the tested version pairs tend to converge and produce stable program variants quickly, most of the stable variants have difficulty accurately rendering HTML correctly. This is because the fitness script used in the fitness evaluation of the Dillo variants focuses on the stability of the variant rather than the correctness

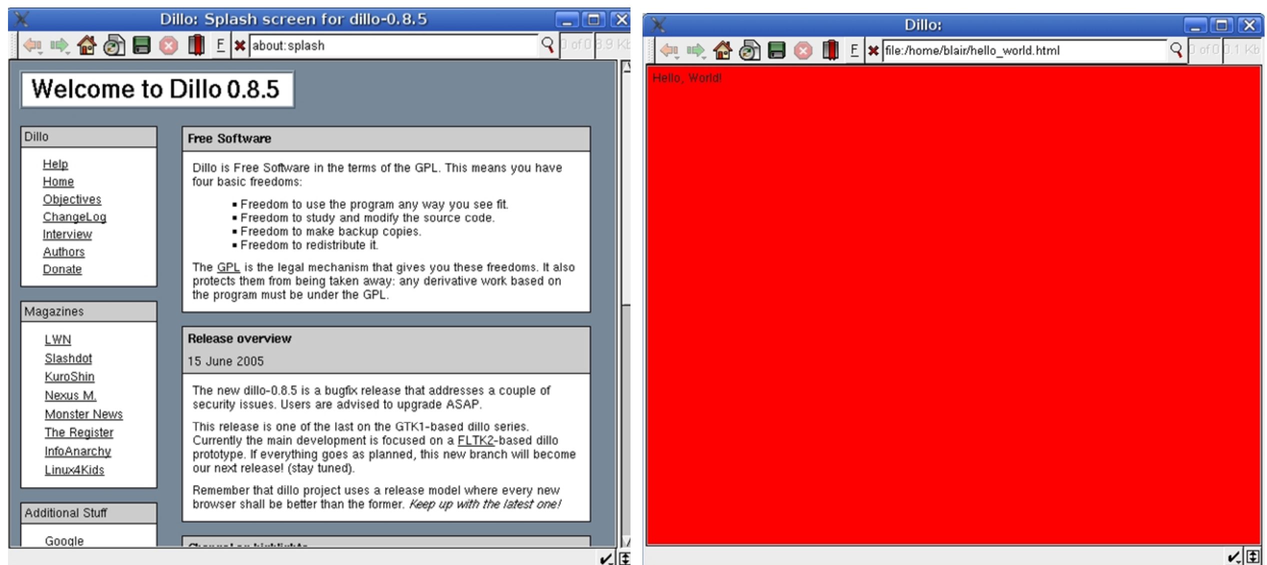

Figure 5.6: Screen shots from Test 1 (left) & Test 2 (right) of the Dillo fitness script against Dillo version 0.8.5

of its output. In this sense the fitness evaluation encourages the exploration of the search space of stable variants as opposed to 'correct' variants. When looking at the results of the most fit recombined variants from version pairs 0.8.3 and 0.8.0, 0.8.5 and 0.7.3, 0.8.5 and 0.8.0, and 0.8.0 and 0.7.3 the shortcomings of the fitness script become apparent. The screenshots in Figures 5.8 - 5.10 are representative of screenshots captured from all of the recombined variants from these pairs. The images clearly illustrate that these variants have difficulty rendering HTML from a file and from a web site. The least usable recombined variants were created from versions 0.8.0 and 0.7.3. Variants generated from these versions present a minimal and unresponsive user interface as seen in Figure 5.10.

Recombined variants from versions pairs 0.8.5 and 0.8.2, and 0.8.4 and 0.8.1 had much more encouraging and interesting results. The screenshot results from *all* recombined variants generated from versions 0.8.5 and 0.8.2 were a mirror image of the reference screenshots. However, while some screen shots results from versions 0.8.4 and 0.8.1 reflected the reference screenshots accurately, others, seen in figures 5.11 - 5.13, presented some interesting behavioral side effects. In Figure 5.11 we see one recombined variant which does not appear to be able to render the HTML. However, upon further inspection, the variant is clearly receiving and parsing the HTML as it

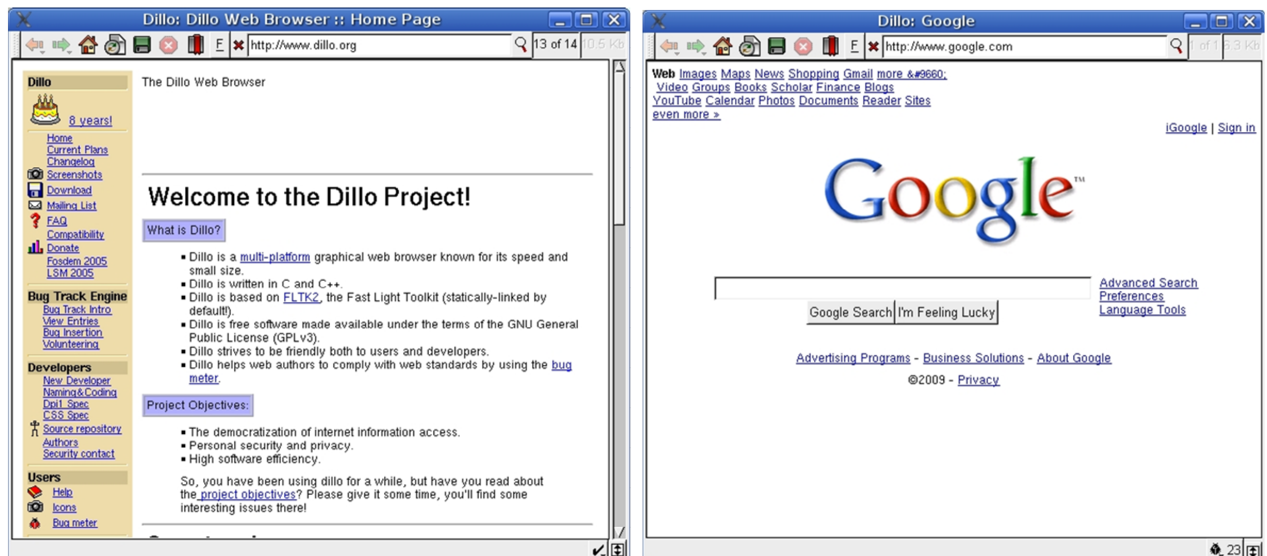

Figure 5.7: Screen shots from Test 3 (left) & Test 4 (right) of the Dillo fitness script against Dillo version 0.8.5

is possible to view the HTML source that the variant is attempting to render. This would imply that only a subset of the HTML parsing component within this variant is malfunctioning.

The screenshot results in figure 5.12 indicate that this particular recombined variant is incapable of completing the HTML rendering once a GIF[7] image is encountered. As can be clearly seen, a partial rendering of the Google website is displayed and only the title of the Dillo website is presented correctly. In both cases this constitutes the entirety of the HTML up to first reference to a GIF image.

Another, more interesting, recombined Dillo variant generated from versions 0.8.4 and 0.8.1 can is seen in figure 5.13. Here we see Dillo homepage rendered without any of the HTML bullet points seen in figure 5.7. Additionally, the underlying HTML table is vastly decreased in width causing the page to appear smaller and somewhat compacted. However, with the exception of these strange characteristics, this variant is accurate and correct in all other respects.

[7] Graphics Interchange Format

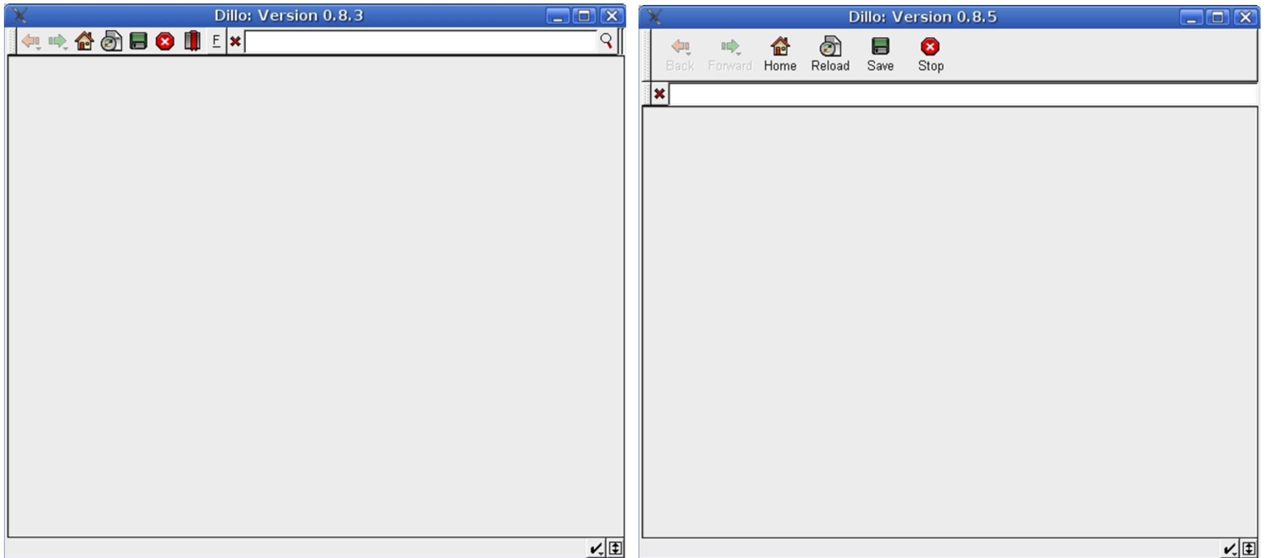

Figure 5.8: Typical screen shot result of Dillo variants generated from versions 0.8.3 and 0.8.0 (left) and versions 0.8.5 and 0.7.3 (right). No HTML is rendered - regardless of source.

5.3.3 Analysis

Similar to GNU-sed, it would appear that closer version pairs are more likely to create stable variants with higher fitness than version pairs that have more distant version numbers. This correlation is almost certainly due to the increased number of code changes that are being introduced as each version is incremented. Regardless, even the widest apart version pair (0.8.5 and 0.7.3) that had the greatest number of unstable variants was eventually able to attain a high fitness as it moved further away from these unstable variants at each generation.

Testing Dillo was much more complex than GNU-sed, but the results were encouraging as they clearly showed again that ObjRecombGA is capable of discovering successful object file recombinations with even a limited set of tests. Judging by the numerous screenshots analyzed from these results, the usability of many of these Dillo variants is questionable, however. None of the generated variants were able to accurately recreate the very simple reference shots (figures 5.6 and 5.7) meaning that none of the variants were "fully functional" for all test cases. However, this is largely an effect of the test case validation within the fitness script. Because the fitness script here is only concerned with stable variants – those that do not crash and produce a core dump file – that was precisely what the ObjRecombGA managed to find. As

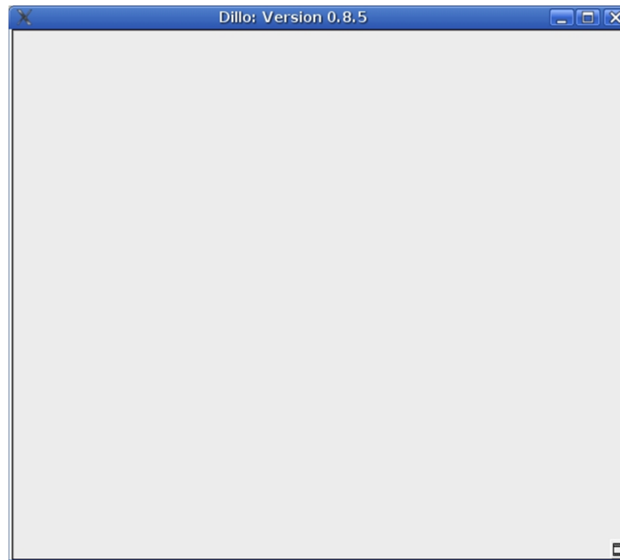

Figure 5.9: Typical screen shot result of Dillo variants generated from versions 0.8.5 and 0.8.0. The user interface is not displayed and no HTML is rendered.

such, neither ObjRecombGA nor the object file recombination methods were to blame for the poor usability of the recombined variants.

Though perhaps not as successful as the GNU-sed results, the limited success of recombining Dillo does highlight the fact that more complex programs are harder to recombine correctly because of the large search space and the difficulty in creating an adequate fitness script to identify correctly behaving variants.

5.4 Quake

Quake is a popular 3D first person shooter video game created by Id Software in 1996. Quake (seen in figure 5.14) popularized the use of the OpenGL 3D programming API and pushed the limits of video game graphics when it was introduced. The source code for the game was released in 1999 under the GNU GPL Lincese and has led to many forked versions of the game containing various enhancements and extended functionality. Figure 5.15 depicts a partial evolutionary tree for Quake showing the large number of forks that have been created.

Having a large number of forks makes Quake an excellent candidate for testing as all of these forks are derived from the original Quake source. Additionally, because each fork is its own unique version (unlike the sequential versions of GNU-Sed and

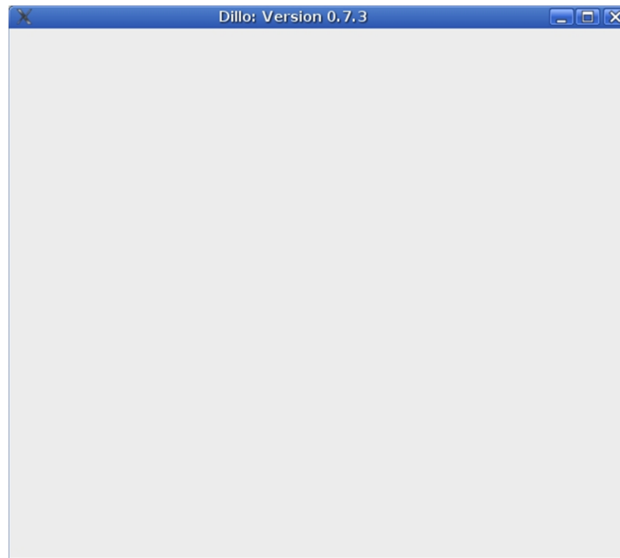

Figure 5.10: Typical screen shot result of Dillo variants generated from versions 0.8.0 and 0.7.3. The user interface is nearly non-existent and unresponsive.

Dillo) it provides a more likely scenario for recombining different functionality between two software programs. Quake, however, is also a much larger and more complex program than either GNU-sed or Dillo. Properly testing a recombined Quake variant to determine the success of the recombination is extremely difficult to accomplish with a simple shell script, therefore manual testing was used to analyze many of the stable recombined variants.

5.4.1 Fitness Calculation

Fitness evaluation for each Quake program variant was done using a simple fitness script (listed in Appendix C). This script performs three simple and very similar tests.

1. Running the Quake program and loading the basic startup map.

2. Running the Quake program and running the default startup demo .

3. Running the Quake program and running the default startup demo with a larger default program heap, CD support disabled, and without sound. This test specifically exercises alternate code paths by altering the resources being used.

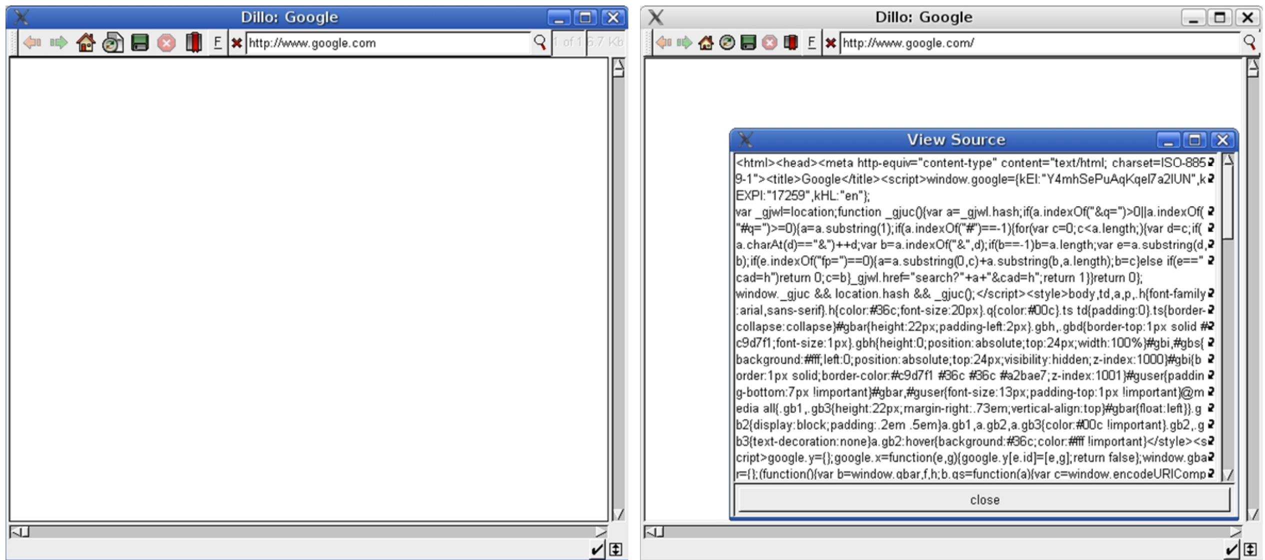

Figure 5.11: A Dillo variant generated from versions 0.8.4 and 0.8.1. No HTML is rendered (left) yet the HTML code is visible when viewing the actual HTML source (right)

The above tests do nothing more than run each recombined Quake variant under slightly different constraints. Test 1 validates that a recombined variant can load a simple game map and test 2 runs a pre-recorded game demo (a default demo is started when the game is run without any other input). Test 3 attempts to run the same pre-recorded demo using different system and environment constraints which alter the flow of execution. With tests 2 and 3 the console output from running the variant is captured for evaluation.

As with Dillo and GNU-Sed, after each test is run the script checks for the presence of a core dump file. If no core dump file is found, the script terminates the program variant and adjusts the fitness return value (R). A value of 10 is added to R if test 1 runs without a core dump. If tests 2 and 3 do not generate a core dump, the script compares the resulting console output against a baseline console output taken from the original Quake 1.09. The number of matching lines from these comparisons is then added to the return value and testing continues. If any test generates a core dump the script immediately exits and returns the current value of R. Based on the number of lines in the baseline console output a maximum value of 240 can be returned by the Quake fitness script, giving a possible fitness value range of 0 to 240 for any single recombined Quake variant.

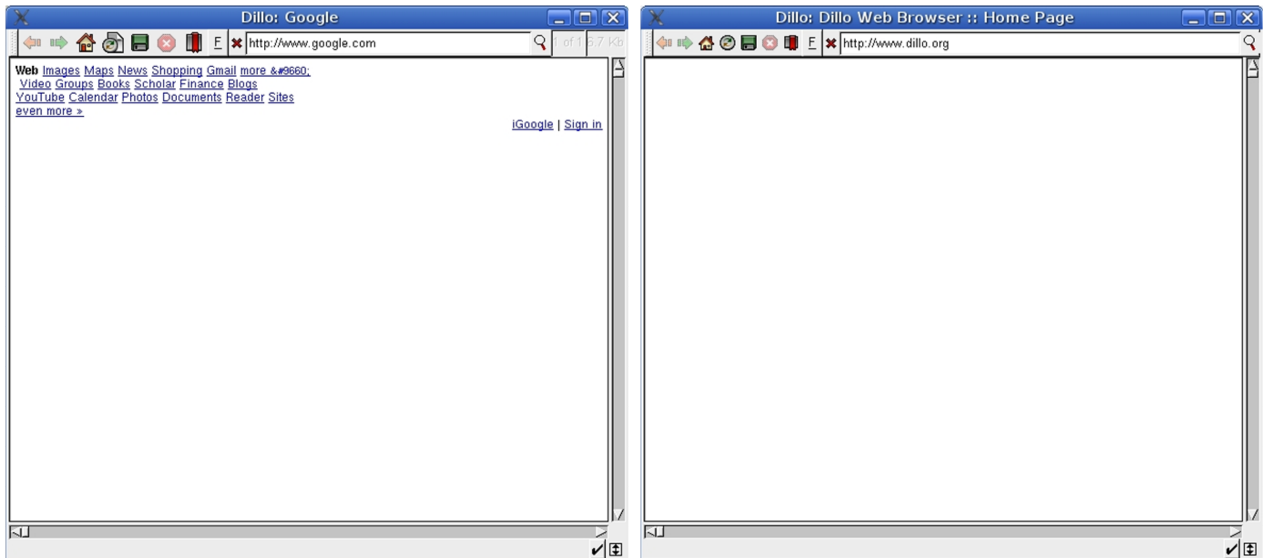

Figure 5.12: A Dillo variant generated from versions 0.8.4 and 0.8.1, which offers a partial display of Google's web site from Test 4 (left) and only the title of the Dillo web site from Test 3 (right).

5.4.2 Versions Tested

Though there are dozens of forks of the original Quake source code, only five were chosen for testing. These include:

- TyrQuake - a conservative fork of Quake primarily focusing on bug fixes and code cleanup.
- SDLQuake - a re-factoring of the Quake code to make it compatible with the SDL (Simple DirectMedia Layer) API.
- MakaQu - includes many graphical enhancements to Quake such as improved menus, refined status bar, etc.
- ProQuake - uses anti-cheat technology and other competitive play enhancements.
- FishEyeQuake - uses an interesting 'fish-eye' view to render the game.

These forks were chosen specifically because they include software rendered builds (though most also included OpenGL builds) and were easy to obtain and build in the test environment. Focusing on forked versions of Quake which contain software render

Figure 5.13: A Dillo variant generated from versions 0.8.4 and 0.8.1, which ignores HTML bullet points and the width parameter of an HTML table when rendering the Dillo homepage from Test 3.

builds, reduces the hardware requirements for the test environment and reduces the number of 3rd party libraries required - such as OpenGL. One exception to this is MakaQu, which is primarily comprised of enhancements for the OpenGL build of Quake, but does include a software build with a reduced set of these enhancements. Figures 5.16 - 5.20 are screenshots of the above forks of Quake after loading the default startup map. These figures are used as reference when describing some of the results obtained. Of particular note from these figures is the apparent lack of model rendering (no gun or zombies are displayed) in MakaQu and the lack of a heads-up display (HUD) in the FishEyeQuake screenshots. The model rendering issue in MakaQu is due to the fact that much of the model rendering code changes are specific to OpenGL and as a side effect causes a bug in the software rendering. The lack of a HUD in the FishEyeQuake fork is due to minor code changes with respect to the default screen size and field of view. By increasing the default screen size, the HUD is intentionally removed from view making it difficult for a player to know their current status indicators (health, armor, ammo, etc.). It is also worth noting the unique default aspect ratio seen in the reference screenshot of SDLQuake. This was not present in any of the other versions.

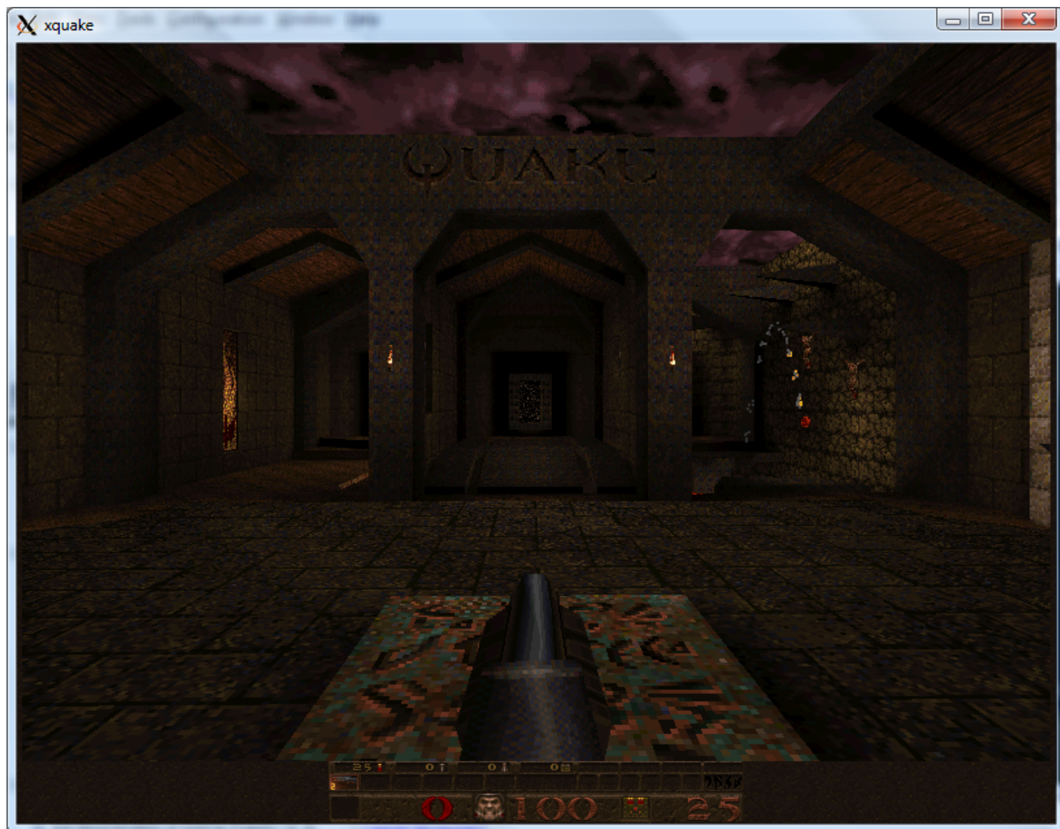

Figure 5.14: ID Software's Quake - Version 1.09

Table 5.6 lists the number of object files available for each Quake version, the number of exported symbols from those files, and the number of differing exported symbols compared to the *original* Quake 1.09 version.

All possible version pairs were chosen as input for the ObjRecombGA software. These pairs were tested using Two-Point Crossover and Tournament Selection with a population size of 50 over 20 generations, a tournament size of 7 with 90% probability, and using an elitism value of 10. Using these parameters allowed ObjRecombGA to produce a maximum of 810 recombined program variants for each tested pair. Larger population and generations sizes were not attempted due to the fact that successful results were found using these parameters.

Table 5.7 lists each tested pair, the number of matching object or library files available for recombination, the number of total and unique variants produced, and the number of unstable variants. Also included is the number of differing symbols between the pairs of forks. This was added to give some indication of how closely

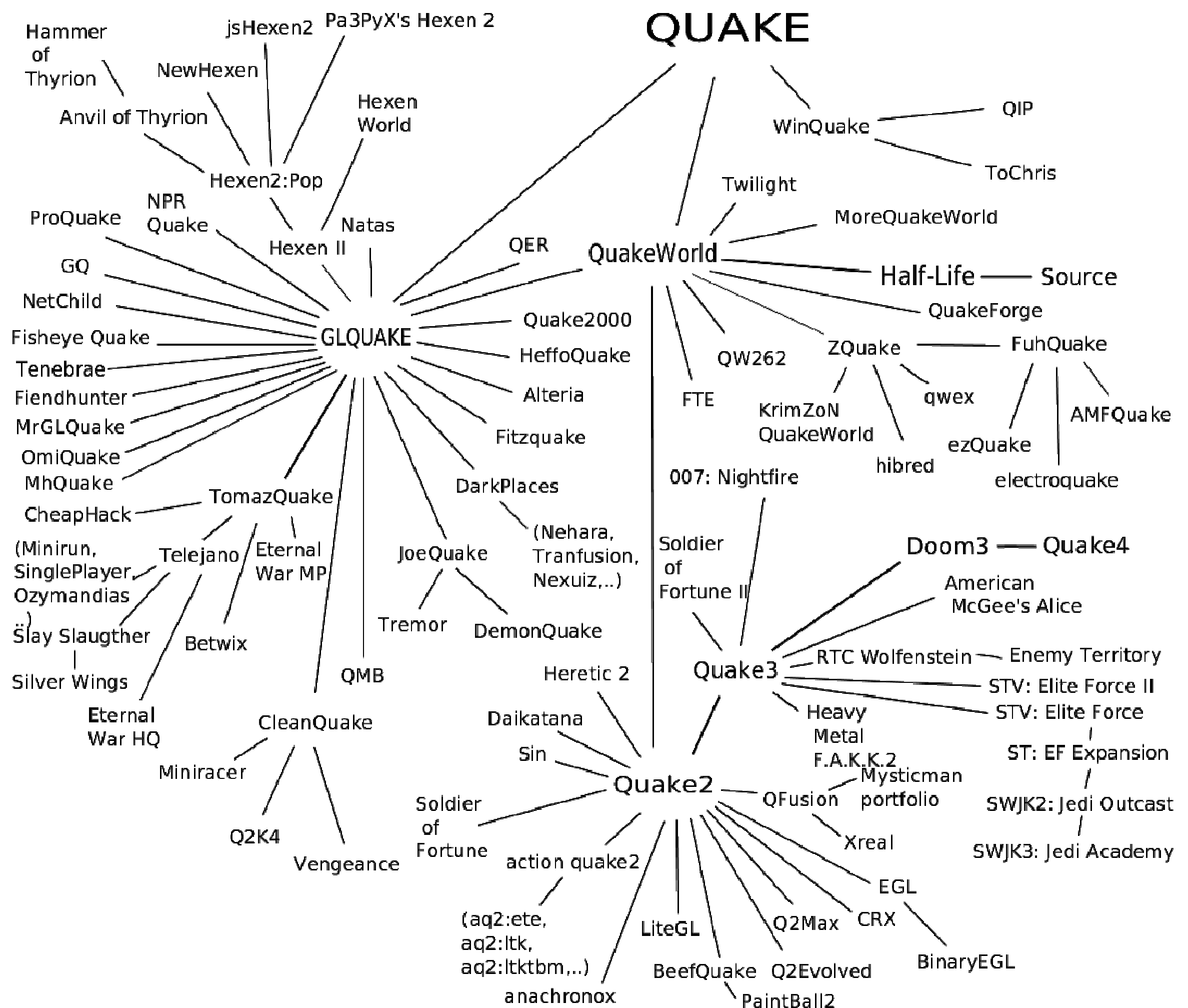

Figure 5.15: A partial evolutionary tree showing various Quake forks from 1999 thru 2006[12]. Image licensed under the Creative Commons Attribution 2.0 Generic license[21].

related the pairs of forks are because, unlike the GNU-sed and Dillo tests which used sequential versions, the versions of these forks have no obvious distance metric.

In all cases, ObjRecombGA was able to successfully create 810 variants for each version pair. However, some test pairs experienced a high percentage of unstable variants. Specifically, TyrQuake with SDLQuake and TyrQuake with MakaQu both had unstable variants (204 and 292) accounting for 25% or more of their total variants produced. One likely explanation for the instability is the high number of differing symbols between the versions themselves (459 and 671) and between the original Quake version. Contrast those results against the recombination of SDLQuake with

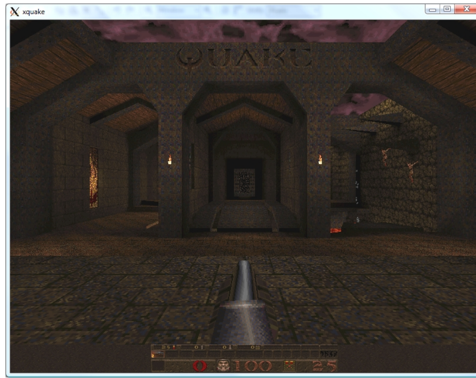

Figure 5.16: TyrQuake - Version 0.38

Figure 5.17: SDLQuake - Version 1.09

Figure 5.18: MakaQu - Version 0.2

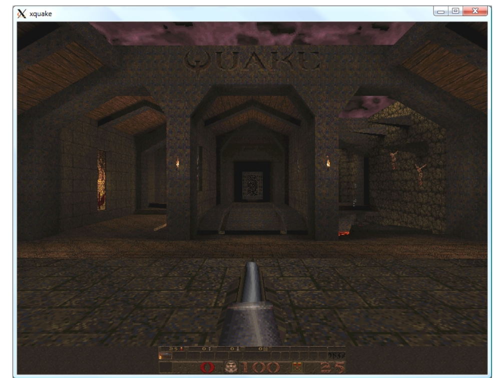

Figure 5.19: ProQuake - Version 3.60

FishEyeQuake or FishEyeQuake with ProQuake where the number of unstable variants (149 and 0) and number of symbol differences (180 and 168) are much lower. This indicates that a high degree of symbolic differences, and thus more divergent versions, are more difficult to recombine successfully. However, version pairs SDLQuake with ProQuake and MakaQu with ProQuake, while still very divergent, had a lower degree of instability.

Examining the average fitness at each generation for the various Quake version pairs, Figure 5.21, further solidifies the observation that versions which are more divergent generally have more difficulty recombining successfully. Of all the versions tested, TyrQuake was the most divergent with all other versions leading to a high degree of instability and very low fitness averages in nearly every test. Many of the variants generated through recombination with TyrQuake resulted in very strange behaviors. Some crashed outright, others hung after loading the game, and some were

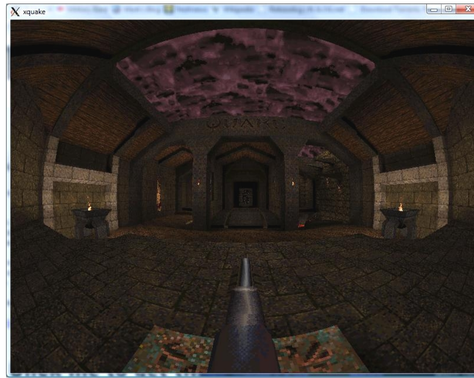

Figure 5.20: FishEyeQuake - Version 1.09

Quake Version	Object Files	# Symbols	# Symbols Diff.
TyrQuake	92	1807	375
SDLQuake	92	1962	168
FishEyeQuake	89	2104	12
MakaQu	89	2160	450
ProQuake	93	2246	156

Table 5.6: A comparison of the versions of Quake used

caught in infinite loops negatively affecting the fitness value. Figure 5.22 illustrates some of these findings.

MakaQu was also highly divergent compared to other versions but was able to achieve much better fitness averages across all tests. However, the fitness results for MakaQu are misleading as manual inspection of the recombination results showed that many variants would start correctly and produce significant correct output before silently failing or freezing without producing a core dump file. This behavior can be seen in figure 5.23.

However, not all results from recombinations involving MakaQu were unsuccessful. A handful of variants generated from FishQuake with MakaQu and SDLQuake with MakaQu showed promise. Figures 5.24 & 5.25 highlight some of the recombined functionality found during manual inspection. Specifically, the menu selection system from MakaQu was successfully integrated with the unique aspect ratio of SDLQuake in one variant, while another variant integrated the HUD from MakaQu into SDLQuake. Even more promising was the successful recombination of the fish-eye view from FishEyeQuake with the HUD from MakaQu.

Version Pair	Files	Total	Unique	Unstable	Symbols
Tyr × SDL	84	810	727	204	459
Tyr × FishEye	88	810	738	138	387
Tyr × MakaQu	88	810	679	292	671
Tyr × Pro	88	810	624	63	529
SDL × FishEye	85	810	676	149	180
SDL × MakaQu	85	810	476	64	428
SDL × Pro	85	810	529	60	324
FishEye × MakaQu	89	810	728	93	462
FishEye × Pro	89	810	528	0	168
MakaQu × Pro	89	810	570	58	604

Table 5.7: Results for the tested Quake version pairs

Additional positive results can be seen when looking at other, highly fit, recombinations. Figure 5.26 provides two screenshots from two different variants produced from the recombination of SDLQuake with FishEyeQuake. Here we can see the fish-eye view from FishEyeQuake integrated into SDLQuake; one without any HUD and the other with the default HUD being displayed. Interestingly enough, displaying the HUD causes minor graphical glitches on the top and bottom of the fish-eye view.

5.4.3 Analysis

There are several aspects of the Quake results which make them stand apart from the results seen in both GNU-sed and Dillo. First, the successful recombinations of Quake versions show that software program forks, not just versions, are entirely capable of being recombined using object file recombination. Second, not only do these recombined variants of Quake run, but they actually exhibit a combined functional behavior that does not exist in either of the parent programs. This is immediately obvious when looking at figures 5.25 - 5.26 and comparing them against several of the reference figures from the parent versions. Lastly, Quake is a highly complex program with an enormous search space for possible object file recombinations, yet both stable and fully playable recombined variants of Quake can be easily discovered with the presented approach.

Though the fitness script for Quake is fairly simplistic given that it merely checks console output and the presence of core dump files, it provides a drastic improvement over the fitness script used for Dillo. By checking for both stability and correct output, the majority of the pairs were able to produce stable, highly fit, and usable recombined

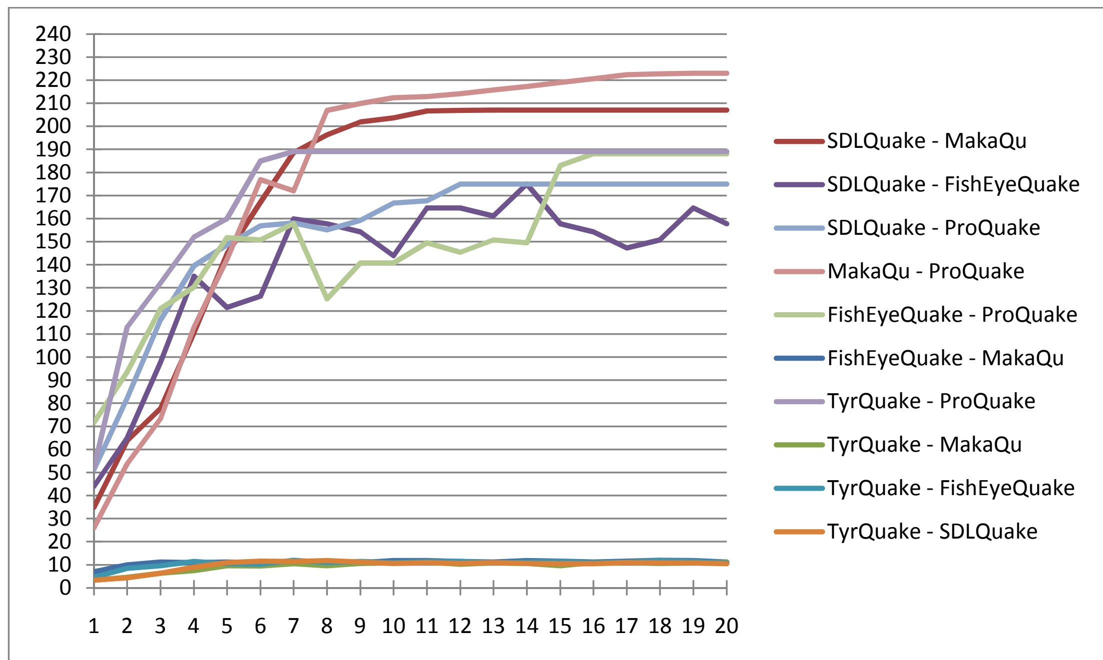

Figure 5.21: Average fitness at each Generation

variants. However, recombinations involving the TyrQuake fork, in particular, seemed to consistently generate unstable and poorly fit variants. This particular fork had one of the highest number of symbol differences compared to the original Quake and the fewest number of symbols in total. These numbers suggest that the code in TyrQuake is a heavily re-factored with many new functions being created (more differing symbols) and with each of those functions containing much more code (fewer symbols in total). As such, TyrQuake is the most divergent of all the forks tested; and its poor recombination results with many of the other forks reinforce this hypothesis. Like Dillo and GNU-sed, the fewer the changes between the Quake forks, the more successful ObjRecombGA is at creating stable and highly fit variants. The difference here is that scope of the changes between forks is estimated by the number of symbol differences against the original Quake version rather the version numbers of each fork.

5.5 Summary

ObjRecombGA was tested using three well known software programs with varying degrees of success. It has been shown that the size and complexity of the software program has an obvious impact on the ability of ObjRecombGA to find and create

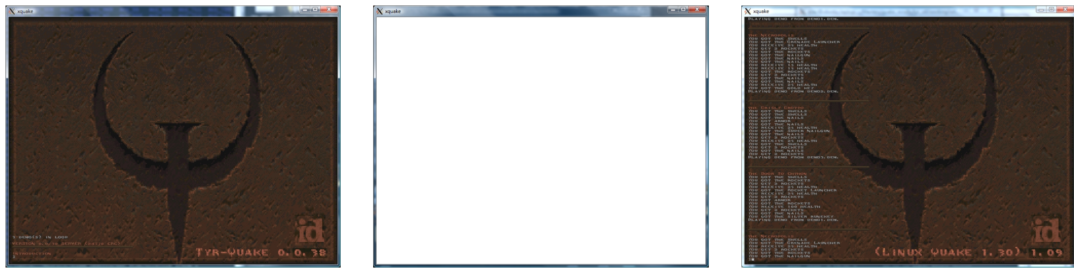

Figure 5.22: Screenshots of hung variants produced during recombination of TyrQuake with MakaQu (left) and ProQuake and MakaQu (center) and variants caught in infinite loops during the recombination of TyrQuake with SDLQuake (right).

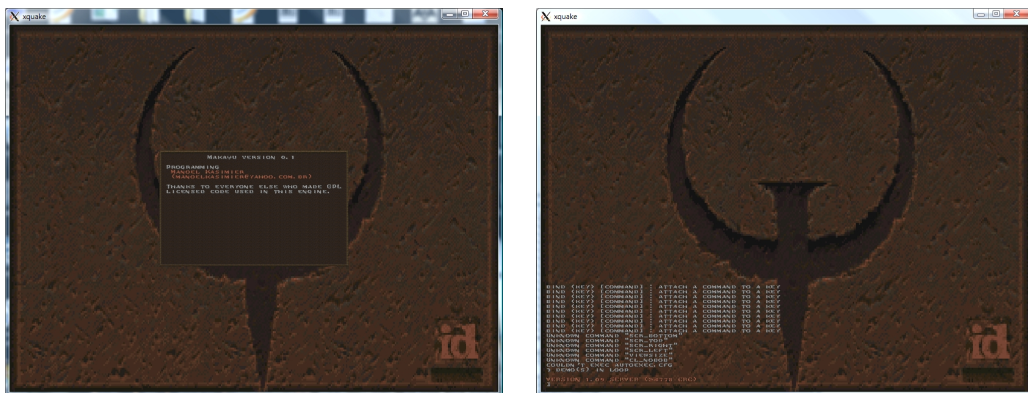

Figure 5.23: Example screen shots of Quake variants generated from recombining FishQuake with MakaQu (left) and SDLQuake with MakaQu (right). In both screenshots, the game has hung silently without terminating. Manual inspection showed many variants generated by these version pairs had similar issues.

successful recombined software program variants. Smaller software programs, such as GNU-sed, which has limited complexity that are incrementally changed over time, show a higher degree of success at object level program recombination. When tested against a larger more complex program, such as Dillo and Quake, the GA search space of possible recombined variants to be discovered by ObjRecombGA makes finding successful recombinations much more difficult - though still possible. The recombination results of Quake are particularly interesting as ObjRecombGA was able to successfully recombine different Quake forks consisting of, in some cases, significantly divergent code bases.

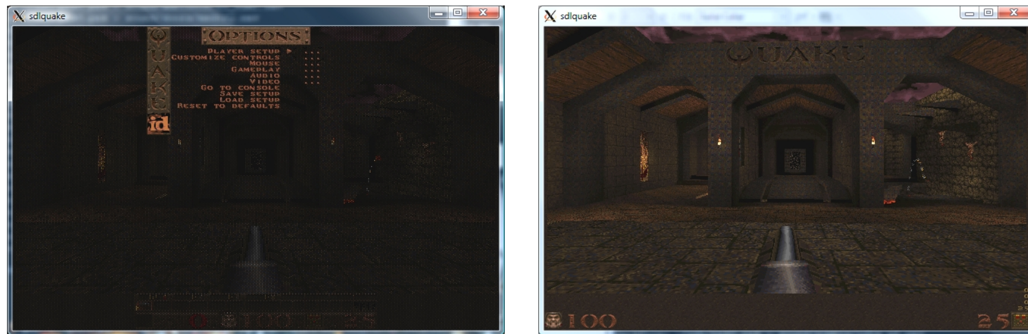

Figure 5.24: Screenshots of SDLQuake recombined with MakaQu. The left screenshot shows the menu system from MakaQu integrated within SDLQuake. The right screenshot shows the HUD from MakaQu in place of the HUD from SDLQuake.

Figure 5.25: Screenshot of FisheyeQuake recombined with MakaQu. Of note here is the HUD from MakaQu integrated with the FishEye view from FishEyeQuake. This solves the problem of the missing HUD in FishEyeQuake and the missing models in MakaQu.

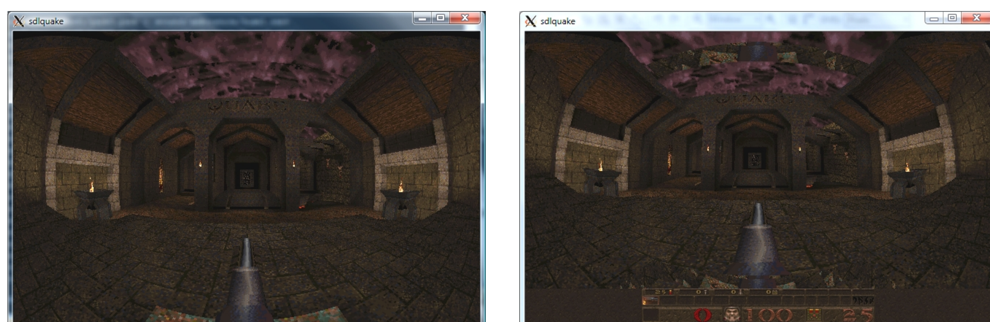

Figure 5.26: Screenshots of SDLQuake recombined with FishEyeQuake. Both variants achieved high scoring fitness and were very playable during manual testing.

Chapter 6

Discussion

This chapter highlights some observations obtained from the experiments in the previous chapter. Major contributions of this research, along with known limitations and suggested enhancements, are also identified in this chapter. Lastly, ideas for some potential future work and improvements to the recombination approach are briefly discussed.

6.1 Observations

Examining the results in the previous chapter leads to a number of observations regarding the use of object files for recombination. The first observation is that software recombination using object files will generally break the functionality of complex software programs. For example, of the 8100 recombined program variants generated from the various pairs of Quake, 1121(15%) were unstable and lead to a program crash. Furthermore, four pairs of forks failed to produce any noteworthy (based on fitness) recombined programs variants. In fact, many of these variants simply hung without crashing, effectively making them unstable also. These four pairs account for an additional 3240(40%) of the total number of variants generated. Therefore, a total of 4361(54%) program variants were generated that were unstable. Though Dillo and GNU-sed showed a slightly lower percentage (21% and 13% respectively) of unusable variants these programs also contained fewer object files and are be considered less complex than Quake. This lack of stability is almost certainly a result of the inability to correct potential runtime problems that are the result of source code changes between the program versions. In most cases, however, the GA was able to work past these runtime issues so that a larger number of stable recombined program variants could be found.

Another observation is that this approach to recombination can actually create software programs that exhibit recombined functionality or functionality that cannot

be seen in either of the original software programs. Recombined functionality can be seen within the Quake results where a few pairs of versions managed to produce variants with combined graphical effects. In particular, the recombination of MakaQu with FishEyeQuake and SDLQuake with FishEyeQuake were particularly promising, with graphical enhancements from both versions recombining to provide a never before seen combination of graphical enhancements (see Figures 5.25 and 5.26). In terms of unique functionality that exists in neither parent program, the recombination of Dillo versions 0.8.4 and 0.8.1 provided the most interesting results (see Figure 5.13). Here we saw that a recombined Dillo browser that was not capable of displaying HTML bullets and did not respect the width parameter of the HTML tables. This particular effect could not be reproduced with either of the original versions.

6.2 Contributions

The research presented in this work makes significant contributions to the domain of evolutionary computing, particularly in the areas of program recombination and genetic programming. The results in the previous chapter provide evidence that emulating selection and recombination as it exists in nature can lead to the successful recombination of existing software programs. Limiting the initial selection to software programs that are closely related, according to their development evolution, more accurately reflects the fact that living organism generally mate with organisms that they are evolutionarly close to. This selection criteria was paired with object file recombination which better represents the whole gene recombination seen during chromosomal crossover in biology, and requires no source code modification of the programs. Both of these design choices represent a fundamentally different approach than those seen in any previous evolutionary computing, genetic programming, or software reuse literature.

Much of the previously related research has focused on program evolution and the creation of new software programs with very little mention of how to deal with existing software programs. Though a few research efforts have tried to address existing software programs, they have all either focused on source code modification to evolve a single software program or relied on formal specifications and code annotations in order to recombine software programs. The recombination approach presented here,

however, works correctly on existing software programs without the need for formal specifications, code annotations, or any other a priori knowledge of the program. Moreover, no source code analysis or modification is required.

6.3 Limitations

The following sections discuss known limitations to not only the presented software program recombination approach in general, but also to the ObjRecombGA implementation itself.

6.3.1 Closely Related Programs

As with genetic recombination, program recombination as presented here will only work correctly when the parents are closely related. Though this is a defining argument of this research, it imposes severe limitations when selecting software programs for recombination. As defined, the concept of closely related programs limits selection to programs that share a common development history, which is essentially program versions and program forks. This means that programs that are closely related in other ways, such as using many of the same libraries or having large sections of shared code, do not immediately fit this selection criteria. Such programs, however, may also be suitable for object file recombination if a substantial amount of the object files pair up correctly.

6.3.2 Object File Recombination

While object file recombination is a novel technique aimed at mimicking genes during chromosomal crossover, it does come with its own drawbacks. For example, short of modifying symbol names, there is very little modification that can be performed on an object file. While it is possible to modify or insert addition code instructions into an object file, great care is needed to ensure that the structure of the object file remains intact and usable by the linker. Changing code within an object file will cause many relative offsets within the file to be miscalculated and, if left uncorrected, will lead to undefined behavior. A second drawback to using object files is that, unlike source code, there is minimal validation occurring during linking. This is because the linker

assumes that the compiler has validated all of the source code and that the resulting object files are compatible. Therefore, in bypassing the compilation step all of the validation and code compatibility checks are also bypassed. While forcing two object files to link together is entirely possible due to this lack of validation, it is very difficult, without source code analysis, to determine how the resulting linked binary will behave at runtime. In the case of ObjRecombGA, the GAs fitness function is expected to weed out object file combinations that have runtime problems. The shortcomings with this approach is that identifying runtime problems may be difficult for many programs. As such, writing an exhaustive fitness script to accurately describe the program behavior the GA should be searching for is also difficult which will ultimately affect the ability of ObjRecombGA to find interesting recombinations of functionality.

6.3.3 Object Files in Object-Oriented Programs

The recombination of object files for C programs is largely feasible because object files generated from C source code are relatively simple and only contain code and data symbol references to one another. Moving beyond object files from source code and looking at the actual *objects* generated by object-oriented languages such as C++ brings an increasing set of issues to light. While the recombination of the object files in a C++ program is similar to a C program, the impact this will have on the objects that the recombined program will generate is much harder to determine. This is because the actual generation of objects and their properties within the program is determined by the source code and the dynamic runtime behavior of the program. Object-oriented paradigm behaviors such as operator overloading, virtual functions, and reflective program properties all introduce new runtime issues which cannot be predicted or overcome during object file linking. However, assuming that a fitness script can be written to steer ObjRecombGA away from such runtime issues, then finding a usable and interesting recombined variants of C++ programs should be possible.

6.3.4 C Programs with ELF Objects

In its current state ObjRecombGA will only successfully recombine programs written in C and compiled by GCC to produce ELF object files. This is purely a superficial

limitation and extending ObjRecombGA to work with programs written in other languages such as C++ (object oriented limitations aside) or Objective-C which are also compiled into object code files should be possible. Adding support for additional compilers and object files, such as Microsofts Visual C++ compiler and COFF object files, should also be possible but will require some work.

6.3.5 Linux Runtime Environment

Though ObjRecombGA was written in Java, it makes use of several common GNU utilities typically found within any Linux operating environment. These utilities include: find, grep, and gawk. All of these tools have been ported to other operating systems, such as Windows[1]; however, they are not installed by default. Additionally, because ObjRecombGA relies on the Bash shell to instantiate these utilities, small modifications are be required to support a different shell environment.

6.4 Future Work

As mentioned above, recombining program versions using object files without regard to the underlying source code changes can lead to many runtime problems which will cause high instability in the recombined variants. Adding an additional source code preprocessing step prior to the object file preprocessing would allow for the identification of potentially incompatible symbols. A large number of symbol incompatibilities can be identified by looking at the associated data and function definitions that they represent within the source code. For example, changing the number of input parameters to a particular subroutine across program versions will cause changes in the program stack alignment and will cause argument mismatches when calling a particular version of the subroutine. These types of changes, however, do not alter the symbol which represents this subroutine. Therefore, attempting to link to this different object file version will succeed but ultimately lead to runtime problems. Preprocessing the source code allows for runtime issues such as these can be avoided by ensuring that symbols identified to be incompatible are not linked together.

There are a number of minor enhancements which can be made to ObjRecombGA itself which could greatly improve its efficiency and robustness when creating

[1]GNU Win32 - http://gnuwin32.sourceforge.net/

recombined program variants. One such enhancement includes verifying that the object files and libraries used for recombination have actually changed between the two target versions. Presently, ObjRecombGA assumes that all matching object and library files between the versions are different. In many of the GNU-sed and Dillo tested pairs, this was not the case; At least a handful of the matching object files were identical across both versions. This assumption lead to an unnecessary inflation of the GA search space. Another minor enhancement that can be made is to omit object files names for object file pairing. Though it was not witnessed with GNU-sed, Dillo, or Quake, object file names (based on the source file names) can easily change across program versions. This name change leads to incorrectly pairing up object files, or failing to pair up object files at all, thereby further inflating the search space. Pairing object files based the number of matching symbols may be better approach in situations where many object files names are different between the program versions being recombined.

Chapter 7

Conclusion

Though various approaches for software program recombination exist, these approaches have entirely focused on using source code as the recombination vector. Moreover, approaches such as genetic programming and automated software engineering have leveraged formal language specifications, design constraints, or other a priori knowledge of the software programs in order to enable successful source code recombination. These requirements, however, cannot be satisfied by existing software programs which have no formal language specifications, design constraints, or source code annotations.

This thesis has shown that it is possible to recombine functionality from existing software programs without the need for specific requirements on the implementation or design of the software programs as long as the two software programs are closely related. This was accomplished by recombining the object files of the two software programs rather than their source code. This recombination process was automated into ObjRecombGA, a software program which is capable of resolving symbolic dependencies between the object files of the two target C programs and manipulating these object files such that they could be linked to create many program variants. A genetic algorithm was used to search the space of all possible object file combinations using the testing results of each combination as a fitness measurement. While object file recombination can introduce instability and runtime issues into the program, the results have shown that it is successful on even large, complex, software programs so long as the testing results are able to correctly steer the genetic algorithm. Furthermore, this approach to program recombination is capable of discovering combinations of functionality from both parent programs that has never been observed before.

Bibliography

[1] Andrea Arcuri. On the automation of fixing software bugs. In *In the Doctoral Symposium of the IEEE International Conference on Software Engineering (ICSE)*, pages 1003–1006, 2008.

[2] Andrea Arcuri. Evolutionary repair of faulty software. Technical Report CSR-09-02, University of Birmingham, 2009.

[3] Andrea Arcuri, David Robert White, John Clark, and Xin Yao. Multi-objective improvement of software using co-evolution and smart seeding. In *SEAL '08: Proceedings of the 7th International Conference on Simulated Evolution and Learning*, pages 61–70. Springer-Verlag, 2008.

[4] Andrea Arcuri and Xin Yao. Coevolving programs and unit tests from their specifications. In *IEEE: International Conference on Automated Software Engineering (ASE)*, pages 397–400, 2007.

[5] Andrea Arcuri and Xin Yao. A novel co-evolutionary approach to automatic software bug fixing. In *In the IEEE Congress on Evolutionary Computation (CEC)*, pages 162–168, 2008.

[6] Jorge Arellano. Dillo program overview. http://www.dillo.org/download/dillo-0.8.5.tar.bz2, 2005. Dillo 0.8.5 Source - /doc/Dillo.txt (last retrieved April 17, 2011).

[7] Various Authors. Seder's script archive. http://sed.sourceforge.net/grabbag/scripts/. (last retrieved April 17, 2011).

[8] Collaborative Authorship. File:rorqual phylogenetic tree.png. http://upload.wikimedia.org/wikipedia/commons/7/79/Rorqual_phylogenetic_tree.PNG. (dated June 25, 2006).

[9] Collaborative Authorship. Fork (software development). http://en.wikipedia.org/w/index.php?oldid=381997746. (dated August 31, 2010).

[10] Collaborative Authorship. GNU/linux distro timeline. http://futurist.se/gldt/. (last retrieved on April 17, 2011).

[11] Collaborative Authorship. Software versioning. http://en.wikipedia.org/w/index.php?oldid=382765050. (dated September 3, 2010).

[12] Collaborative Authorship. File:quake - family tree.svg. http://upload.wikimedia.org/wikipedia/commons/archive/6/63/20080925180321!Quake_-_family_tree.svg, 2006. (last retrieved on April 17, 2011).

[13] Wolfgang Banzhaf. Genetic programming for pedestrians. Technical Report 93-03, Mitsubishi Electric Research Labs, 1993.

[14] Don Batory and Lance Tokuda. Automated software evolution via design pattern transformations. Technical Report CS-TR-95-06, University of Texas at Austin, 1995.

[15] C. L. Boggs. *Species and Speciation*, volume 12. Elsevier, 2001.

[16] S.A. Bohner. Impact analysis in the software change process: A year 2000 perspective. *Software Maintenance, IEEE International Conference on*, 0:42, 1996.

[17] Richard Bornat. Understanding and writing compilers - internet edition. http://www.eis.mdx.ac.uk/staffpages/r_bornat/books/compiling.pdf, 2007. (last retrieved April 17, 2011).

[18] Markus Brameier. *On Linear Genetic Programming*. PhD thesis, Universität Dortmund am Fachbereich Informatik, 2004.

[19] C. Canal, J.M. Murillo, and P. Poizat. Software adaptation. *Journal of Universal Computer Science (JUCS)*, 14(13):2107–2109, 2008.

[20] Carlos Canal. On the dynamic adaptation of component behavior. In *In Proceedings of Workshop on Coordination and Adaptation Issues for Software Entities (WCAT)*, pages 81–88. ACM Press, 2004.

[21] Creative Commons. Creative commons - attribution 2.0 generic. http://creativecommons.org/licenses/by/2.0/deed.en. (last retrieved on April 17, 2011).

[22] S Cook, H Ji, and R Harrison. Software evolution and software evolvability. *Assessment*, 25(1):1–12, 2000.

[23] Richard Dawkins. *The Blind Watchmaker*. Penguin Books, 1986.

[24] Mel Cinn Eide and Mel Cinnide. Automated software evolution towards design patterns. In *IWPSE '01: Proceedings of the 4th International Workshop on Principles of Software Evolution*, pages 162–165. ACM Press, 2001.

[25] Lawrence J. Fogel, Alvin J. Owens, and Micheal J. Walsh. *Artificial Intelligence through Simulated Evolution*. John Wiley & Sons, 1966.

[26] Free Software Foundation. GNU sed home page. http://www.gnu.org/software/sed. (last retrieved on February 21, 2009).

[27] Free Software Foundation. GNU free documentation license. http://www.gnu.org/copyleft/fdl.html, 2008. (last retrieved on April 17, 2011).

[28] Free Software Foundation. GNU sed user's manual. http://www.gnu.org/software/sed/manual/, 2008. (last retrieved April 17, 2011).

[29] Scott Freeman and Jon C. Heron. *Evolutionary Analysis - 3rd Edition.* Pearson Prentice Hall, 2004.

[30] Gerald C. Gannod, Yonghao Chen, and Betty H.C. Cheng. An automated approach for supporting software reuse via reverse engineering. In *In Proceedings 13th International Conference on Automated Software Engineering*, pages 79–86. IEEE Computer Society, 1998.

[31] Paul Grunbacher and Yves Ledru. Automated software engineering. *European Research Consortium for Informatics and Mathematics (ERCIM) News*, 58:12–13, 2004.

[32] Ahmed E. Hassan and Richard C. Holt. Studying the evolution of software systems using evolutionary code extractors. In *IWPSE '04: Proceedings of the 7th International Workshop on Principles of Software Evolution*, pages 76–81. IEEE Computer Society, 2004.

[33] David Hearnden, Paul Bailes, Michael Lawley, and Kerry Raymond. Automating software evolution. pages 95–100, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[34] George T. Heineman. A model for designing adaptable software components. In *In 22nd Annual International Computer Software and Applications Conference (COMPSAC-98)*, pages 121–127. ACM Press, 1998.

[35] George T. Heineman. Adaptation of software components. In *2nd Annual Workshop on Component-Based Software Engineering.* IEEE Computer Society Press, 1999.

[36] James A. Highsmith, III. *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems.* Dorset House Publishing Co., Inc., New York, NY, USA, 2000.

[37] Haym Hirsh, Wolfgang Banzhaf, John R. Koza, Conor Ryan, Lee Spector, and Christian Jacob. Genetic programming. *IEEE Intelligent Systems*, 15(3):74–84, 2000.

[38] John H. Holland. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, 1975.

[39] Idris Hsi and Colin Potts. Studying the evolution and enhancement of software features. In *In Proceedings of the 2000 IEEE International Conference on Software Maintenance*, pages 143–151, 2000.

[40] Jun-Jang Jeng and Betty H.C. Cheng. Using automated reasoning techniques to determine software reuse. *International Journal of Software Engineering and Knowledge Engineering*, 2:523–546, 1992.

[41] Yoshio Kataoka, Michael D. Ernst, William G. Griswold, and David Notkin. Automated support for program refactoring using invariants. In *ICSM '01: International Conference on Software Maintenance*, pages 736–743. IEEE Computer Society, 2001.

[42] Iqbaldeep Kaur, Parvinder S. Sandhu, Hardeep Singh, and Vandana Saini. Analytical study of component based software engineering. *World Academy of Science, Engineering and Technology (WASET)*, 50:437–442, 2009.

[43] Robert E. Keller and Wolfgang Banzhaf. Genetic programming using genotype-phenotype mapping from linear genomes into linear phenotypes. In *Proceedings of the First Annual Conference on Genetic Programming*, pages 116–122. MIT Press, 1996.

[44] Abdelmadjid Ketfi, Noureddine Belkhatir, and Pierre yves Cunin. Dynamic updating of component-based applications. In *In Proceedings of Software Engineering Research and Practice (SERP'02)*, 2002.

[45] Alex Kosorukoff and Alex Kosorukoff. Human based genetic algorithm. *IEEE Transactions on Systems, Man, and Cybernetics*, 2001.

[46] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* The MIT Press, 1992.

[47] John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs.* The MIT Press, 1994.

[48] John R. Koza, Forrest H Bennet III, David Andre, and Martin A. Keane. *Genetic Programming III: Darwinian Invention and Problem Solving.* The MIT Press, 1999.

[49] Charles W. Krueger. Software reuse. *ACM Computing Surveys*, 24:131–183, 1992.

[50] W. B. Langdon and J. P. Nordin. Seeding genetic programming populations. In *Proceedings of the European Conference on Genetic Programming*, pages 304–315. Springer-Verlag, 2000.

[51] W. B. Langdon and Riccardo Poli. *Foundations of Genetic Programming.* Springer-Verlag, 2002.

[52] William B. Langdon and Adil Qureshi. Genetic programming – computers using “natural selection“ to generate programs. Technical Report RN/95/76, University College London, 1995.

[53] M. M. Lehman and J. F. Ramil. An approach to a theory of software evolution. In *IWPSE '01: Proceedings of the 4th International Workshop on Principles of Software Evolution*, pages 70–74. ACM, 2001.

[54] John R. Levine. *Linkers & Loaders*. Morgan Kaufmann Publishers, 2000.

[55] Andra Loosemore, Richard Stallman, Roland McGrath, Andrew Oram, and Ulrich Drepper. *The GNU C Library Reference Manual*. Free Software Foundation, 2007.

[56] Michael R. Lowry. Automating software reuse. In *Working Notes, Third Workshop on AI and Software Engineering: Breaking the Toy Mold*. Kluwer Academic Publishers, 1995.

[57] Michael R. Lowry, Andrew Philpot, Thomas Pressburger, and Ian Underwood. A formal approach to domain-oriented software design environments. Technical Report 20020010586, NASA: National Aeronautics and Space Administration, 1994.

[58] Andrew J. Marek, William D. Smart, and Martin C. Martin. Learning visual feature detectors for obstacle avoidance using genetic programming. In *GECCO Late Breaking Papers*. AAAI, 2002.

[59] Robert I. Mckay, Nguyen Xuan Hoai, Peter Alexander Whigham, Yin Shan, and Michael O'Neill. Grammar-based genetic programming: A survey. *Genetic Programming and Evolvable Machines*, 11:365–396, September 2010.

[60] Tom Mens, Kim Mens, and Tom Tourwe. Aspect-oriented software evolution. *European Research Consortium for Informatics and Mathematics (ERCIM) News*, 58:36–37, 2004.

[61] Hafedh Mili, Fatma Mili, and Ali Mili. Reusing software: Issues and research directions. *IEEE Trans. Software. Eng.*, 21:528–562, June 1995.

[62] Dusan Misevic, Charles Ofria, and Richard E Lenski. Sexual reproduction reshapes the genetic architecture of digital organisms. In *Proceedings of the Royal Society of Biological Sciences*. The Royal Society, 2005.

[63] S.K. Mishra, D.S. Kushwaha, and A.K. Misra. Creating reusable software component from object-oriented legacy system through reverse engineering. *Journal of Object Technology*, 8:133–152, 2009.

[64] Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, 1998.

[65] David J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3:199–230, 1995.

[66] Iulian Neamtiu, Jeffrey S. Foster, and Michael Hicks. Understanding source code evolution using abstract syntax tree matching. In *In Proceedings of the 2005 International Workshop on Mining Software Repositories*, pages 1–5. ACM Press, 2005.

[67] James M. Neighbors. Finding reusable software components in large systems. In *Proceedings of the 3rd Working Conference on Reverse Engineering*. IEEE Computer Society, 1996.

[68] ThanhVu Nguyen, Westley Weimer, Claire Le Goues, and Stephanie Forrest. Using execution paths to evolve software patches. In *ICSTW '09: Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops*, pages 152–153, 2009.

[69] Peter Nordin, Wolfgang Banzhaf, and Frank D. Francone. *Efficient Evolution of Machine Code for CISC Architectures using Blocks and Homologous Crossover*, pages 275–299. MIT Press, 1999.

[70] Charles Ofria and Christoph Adami. Evolution of genetic organization in digital organisms. In *Discrete Mathematics and Theoretical Computer Science (DIMACS) workshop - Evolution as Computation*. Springer-Verlag, 1999.

[71] Yandu Oppacher, Franz Oppacher, and Dwight Deugo. Creating objects using genetic programming techniques. In *Proceedings of the 2009 10th ACIS International Conference on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing*, SNPD '09, pages 455–461, Washington, DC, USA, 2009. IEEE Computer Society.

[72] Steven Oualline. *Practical C Programming*. O'Reilly, 3 edition, 1997.

[73] John Penix and Perry Alex. Design representation for automating software component reuse. In *In Proceedings of the First International Workshop on Knowledge-Based Systems for the (Re)use of Program Libraries*. INRIA, 1995.

[74] John Penix and Perry Alex. Toward automated component adaptation. In *In Proceedings of the 9th IEEE International Conference on Software Engineering and Knowledge Engineering*, pages 535–542, 1997.

[75] Timothy Perkis. Stack-based genetic programming. In *in Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, pages 148–153. IEEE Press, 1994.

[76] Richardo Poli, William B. Langdon, and Nicholas F. McPhee. A field guide to genetic programming. http://www.gp-field-guide.org.uk/. (last retrieved April 17, 2011).

[77] Javier Prez. Overview of the refactoring discovering problem. In *European Conference on Object-Oriented Programming (ECOOP) 2006 Doctoral Symposium and PhD Students Workshop*, Nantes (France), July 2006.

[78] The Avidia Project. The avidia digital life platform. http://avida.devosoft.org/. (last retrieved April 17, 2011).

[79] Jacek Ratzinger and Thomas Sigmund. Mining software evolution to predict refactoring. In *In Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE Computer Society, 2007.

[80] Thomas S. Ray. Evolution ecology and optimization of digital organisms. Technical Report 92-08-942, Santa Fe Institute, 1992.

[81] Tom Ray. Tierra digital evolution software. http://life.ou.edu/tierra/. (last retrieved April 17, 2011).

[82] Rodrigo Quites Reis, Carla Alessandra Lima Reis, Carla Aless, Ra Lima Reis, and Daltro Jos Nunes. Automated support for software process reuse: Requirements and early experiences with the apsee model. In *in Proceedings of the 7th International Workshop on Groupware (CRIGW-01) Darmstadt (Germany) September-2001*. IEEE Computer Society, 2001.

[83] The Sanata Cruz Operation (SCO) and AT&T. *System V Application Binary Interface*, 4.1 edition, 1997.

[84] Christopher Smart. The three giants of linux. http://www.linux-mag.com/id/7721/2/, 2010. (last retrieved on April 17, 2011).

[85] ID Software. Id software's quake. http://www.idsoftware.com/games/quake/quake. (last retrieved on May 15, 2010).

[86] Hideyuki Takagi. Interactive evolutionary computation: System optimization based on human subjective evaluation. In *In Proceedings IEEE International Conference on Intelligent Engineering Systems*. IEEE Press, 1998.

[87] Ron Tamarin and Robert Leavit. *Principles of Genetics*. McGraw-Hill, 7 edition, 2001.

[88] Dillo Project Team. Dillo home page. http://www.dillo.org. (last retrieved on February 25, 2009).

[89] Adrian Ulges. Visualizing software evolution. Technical report, University of Kaiserslautern, 2005.

[90] William von Hagen. *The Definitive Guide to GCC - 2nd Edition*. Apress, 2006.

[91] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *Proceedings of he 31st International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2009.

[92] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Closed-loop repair of security vulnerabilities. In *USENIX Security Symposium*, 2009.

[93] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. A genetic programming approach to automated software repair. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation (GECCO)*. ACM, 2009.

[94] David A. Wheeler. Why open source software / free software? look at the numbers! http://www.dwheeler.com/oss_fs_why.html. (last retrieved on April 17, 2011).

[95] Claus O Wilke and Christoph Adami. The biology of digital organisms. *Trends in Ecology & Evolutions*, 17:528–532, 2002.

[96] Claus O. Wilke, Jia Lan Wang, Charles Ofria, Richard E. Lenski, and Christoph Adami. Evolution of digital organisms at high mutation rates leads to survival of the flattest. *Nature*, 412:331–333, 2000.

[97] Niklaus Wirth. *Compiler Construction*. Addison-Wesley, 1996.

[98] Amy Moormann Zaremski and Jeannette M. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology*, 6:333–369, 1996.

Appendix A

Source Code for Program Version Alpha

File: a.c (produces object file a.o)

```
int dAx = 0;
int dAy = 0;

extern void fBn();
void fAn()
{
   fBn();
}

void fAm()
{
   dAx++;
}

int main(int argc, char argv[])
{
   fAn();
}
```

File: b.c (produces object file b.o)

```
int dBx = 0;

extern int dAy;
extern int dAx;
extern void fAm();

void fBm()
{
   dBx++;
   dAx++;
}

void fBn()
{
   dAy++;
   fAm();
   fBm();
}
```

Appendix B

Source Code for Program Version Beta

```
File: a.c (produces object file a.o)

   int dAx = 0;
   int dAz = 0;

   extern void fBn();

   void fAn()
   {
      fBn();
      dAx++;
   }

   int main(int argc, char argv[])
   {
      fAn();
   }


File: b.c (produces object file b.o)

   int dBx = 0;

   extern int dAz;

   void fBm()
   {
      dBx++;
      dAz++;
   }

   void fBn()
   {
      dBx++;
      fBm();
   }
```

Appendix C

Fitness Script used for calculating the fitness of GNU-sed program variants

```
#!/bin/bash

ulimit -Sc unlimited
ulimit -Hc unlimited
ulimit -c unlimited
/bin/rm $2core

SCRIPT_HOME=/home/blair/192.168.1.4/masters/code/scripts/sed_scripts
R=0
###############################
#test 1 - excute 'head.sed' script
###############################

$1 -f $SCRIPT_HOME/head.sed $SCRIPT_HOME/test_input/test.c > ./test_head.c &
/bin/sleep 10
if [ -a ./core ]; then
/usr/bin/killall -q -9 sed
exit $R
fi
#check the output vs. the expected output
checksum1=`/usr/bin/md5sum ./test_head.c | /usr/bin/awk '{print $1}'`
checksum2=`/usr/bin/md5sum $SCRIPT_HOME/test_output/test_head.c | /usr/bin/awk '{print $1}'`
if [ "$checksum1" = "$checksum2" ]
then
let "R += 1"
fi
/usr/bin/killall -q -9 sed

###############################
#test 2 - excute 'remccomms1.sed' script
###############################

$1 -f $SCRIPT_HOME//remccoms1.sed $SCRIPT_HOME/test_input/test.c > ./test_nocomments.c &
/bin/sleep 10
if [ -a ./core ]; then
/usr/bin/killall -q -9 sed
exit $R
fi
checksum1=`/usr/bin/md5sum ./test_nocomments.c | /usr/bin/awk '{print $1}'`
checksum2=`/usr/bin/md5sum $SCRIPT_HOME/test_output/test_nocomments.c | /usr/bin/awk '{print $1}'`
#check the output vs. the expected output
if [ "$checksum1" = "$checksum2" ]
then
let "R += 1"
fi
/usr/bin/killall -q -9 sed

###############################
#test 3 - excute 'indentls.sed' script
###############################

$1 -f $SCRIPT_HOME/indentls.sed $SCRIPT_HOME/test_input/lslr.txt > ./lslr_indent.txt &
```

```
/bin/sleep 10

if [ -a ./core ]; then
/usr/bin/killall -q -9 sed
exit $R
fi
checksum1=`/usr/bin/md5sum ./lslr_indent.txt | /usr/bin/awk '{print $1}'`
checksum2=`/usr/bin/md5sum $SCRIPT_HOME/test_output/lslr_indent.txt | /usr/bin/awk '{print $1}'`
#check the output vs. the expected output
if [ "$checksum1" = "$checksum2" ]
then
let "R += 1"
fi
/usr/bin/killall -q -9 sed

##############################
#test 4 - excute 'revchr_1.sed' script
##############################

$1 -f $SCRIPT_HOME/revchr_1.sed $SCRIPT_HOME/test_input/lslr.txt > ./lslr_rev.txt &
/bin/sleep 10
if [ -a ./core ]; then
/usr/bin/killall -q -9 sed
exit $R
fi
checksum1=`/usr/bin/md5sum ./lslr_rev.txt | /usr/bin/awk '{print $1}'`
checksum2=`/usr/bin/md5sum $SCRIPT_HOME/test_output/lslr_rev.txt | /usr/bin/awk '{print $1}'`
#check the output vs. the expected output
if [ "$checksum1" = "$checksum2" ]
then
let "R += 1"
fi
/usr/bin/killall -q -9 sed

##############################
#test 5 - excute 'cflword5.sed' script
##############################

$1 -f $SCRIPT_HOME/cflword5.sed $SCRIPT_HOME/test_input/test.c > ./test_caps.c &
/bin/sleep 10
if [ -a ./core ]; then
/usr/bin/killall -q -9 sed
exit $R
fi
checksum1=`/usr/bin/md5sum ./test_caps.c | /usr/bin/awk '{print $1}'`
checksum2=`/usr/bin/md5sum $SCRIPT_HOME/test_output/test_caps.c | /usr/bin/awk '{print $1}'`
#check the output vs. the expected output
if [ "$checksum1" = "$checksum2" ]
then
let "R += 1"
fi
/usr/bin/killall -q -9 sed

##############################
#test 6 - excute 'sierpinski3.sed' script
##############################

$1 -f $SCRIPT_HOME/sierpinski3.sed $SCRIPT_HOME/test_input/sp_tri.txt > ./sp_tri_out.txt &
/bin/sleep 10
if [ -a ./core ]; then
/usr/bin/killall -q -9 sed
exit $R
fi
checksum1=`/usr/bin/md5sum ./sp_tri_out.txt | /usr/bin/awk '{print $1}'`
```

```
checksum2=`/usr/bin/md5sum $SCRIPT_HOME/test_output/sp_tri_out.txt | /usr/bin/awk '{print $1}'`
#check the output vs. the expected output
if [ "$checksum1" = "$checksum2" ]
then
let "R += 1"
fi
/usr/bin/killall -q -9 sed

exit $R
```

Appendix D

Fitness Script used for calculating the fitness of Dillo program variants

```
#!/bin/bash
ulimit -Hc unlimited
ulimit -Sc unlimited
ulimit -c unlimited

/bin/rm $2core

R=0
###############################
#test 1 - loading dillo variant
###############################
$1 &
/bin/sleep 5
if [ -f ./core ]; then
        /usr/bin/killall -9 dillo
        exit $R
fi
let "R += 1"
/usr/bin/scrot
/bin/sleep 5
/usr/bin/killall -9 dillo

###############################
#test 2 - local HTML file
###############################
fileaccess=‘stat -c %x /home/blair/hello_world.html
$1 /home/blair/hello_world.html &
/bin/sleep 5
if [ -f ./core ]; then
        /usr/bin/killall -9 dillo
        exit $R
fi
let "R += 1"
fileaccess2=‘stat -c %x /home/blair/hello_world.html
if [ $fileaccess != $fileaccess2 ]
        let "R += 1"
fi
/usr/bin/scrot
/bin/sleep 5
/usr/bin/killall -9 dillo

################################################
#test 3 - Internet site with HTML and JavaScript
################################################
$1 www.google.com &
/bin/sleep 5
if [ -f ./core ]; then
        /usr/bin/killall -9 dillo
        exit $R
fi
let "R += 1"
/usr/bin/scrot
```

```
/bin/sleep 5

/usr/bin/killall -9 dillo

######################################
#test 4 - Internet site with HTML only
######################################
$1 www.dillo.org &
/bin/sleep 5
if [ -f ./core ]; then
        /usr/bin/killall -9 dillo
        exit $R
fi
let "R += 1"
/usr/bin/scrot
/bin/sleep 5
/usr/bin/killall -9 dillo
exit $R
```

Appendix E

Fitness Script used for calculating the fitness of Quake program variants

```
#!/bin/bash

ulimit -Sc unlimited
ulimit -Hc unlimited
ulimit -c unlimited

/bin/rm $2core*

R=0
A=10
B=0
C=0
################
# setup the .pak files
# needed for the game to load
#################
/bin/ln -s /home/blair/Masters/quake_code/ID1 $2id1

###############################
#test 1 - loading quake variant
# simple map only
###############################

$1 +map start 2>$2stdout2.txt 1>$2stdout1.txt &
/bin/sleep 10
if [ -e $2core* ]; then
/bin/rm $2core*
echo "crash" > $2core
exit $R
fi

/usr/bin/killall -9 quake.x11

R=`expr $A + $R`
###############################
#test 2 - loading quake variant
# demo run
###############################

$1 2>$2stdout2.txt 1>$2stdout2.txt &
/bin/sleep 10
if [ -e $2core* ]; then
/bin/rm $2core*
echo "crash" > $2core
exit $R
fi

/usr/bin/killall -9 quake.x11

if [ -e $2stdout2.txt ]; then
B=`/usr/bin/gawk -f lineDiff.awk -v flist1=$2stdout2.txt quake_stdout.txt`
fi
```

```
R=`expr $B + $R`
##############################
#test 3 - loading quake with
#   larger heap,
#   no CD support,
#   no sound support.
#Hoping to increase stability
##############################
$1 -mem 32 -nocd -nosound 2>$2stdout3.txt 1>$2stdout3.txt &
/bin/sleep 10
if [ -e $2core* ]; then
/bin/rm $2core*
echo "crash" > $2core
exit $R
fi

/usr/bin/killall -9 quake.x11

if [ -e $2stdout3.txt ]; then
C=`/usr/bin/gawk -f lineDiff.awk -v flist1=$2stdout3.txt quake_stdout.txt`
fi

R=`expr $R + $C`

exit $R
```