Summer 2012

Evolutionary computing driven search based software testing and correction

Joshua Lee Wilkerson

Department: Computer Science

EVOLUTIONARY COMPUTING DRIVEN SEARCH BASED SOFTWARE TESTING AND CORRECTION

by

JOSHUA LEE WILKERSON

A DISSERTATION

Presented to the Faculty of the Graduate School of the

MISSOURI UNIVERSITY OF SCIENCE AND TECHNOLOGY

in Partial Fulfillment of the Requirements for the Degree

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

2012

Dr. Daniel Tauritz, Advisor
Dr. Thomas Weigert
Dr. Bruce McMillin
Dr. Ali Hurson
Dr. Sahra Sedighsarvestani

ABSTRACT

For a given program, testing, locating the errors identified, and correcting those errors is a critical, yet expensive process. The field of Search Based Software Engineering (SBSE) addresses these phases by formulating them as search problems. This dissertation addresses these challenging problems through the use of two complimentary evolutionary computing based systems. The first one is the Fitness Guided Fault Localization (FGFL) system, which novelly uses a specification based fitness function to perform fault localization. The second is the Coevolutionary Automated Software Correction (CASC) system, which employs a variety of evolutionary computing techniques to perform testing, correction, and verification of software. In support of the real world application of these systems, a practitioner's guide to fitness function design is provided.

For the FGFL system, experimental results are presented that demonstrate the applicability of fitness guided fault localization to automate this important phase of software correction in general, and the potential of the FGFL system in particular. For the fitness function design guide, the performance of a guide generated fitness function is compared to that of an expert designed fitness function demonstrating the competitiveness of the guide generated fitness function. For the CASC system, results are presented that demonstrate the system's abilities on a series of problems of both increasing size as well as number of bugs present. The system presented solutions more than 90% of the time for versions of the programs containing one or two bugs. Additionally, scalability results are presented for the CASC system that indicate that success rate linearly decreases with problem size and that the estimated convergence rate scales *at worst* linearly with problem size.

ACKNOWLEDGMENTS

First, I would like to thank the Missouri University of Science and Technology Intelligent Systems Center for providing financial support for this project.

I would like to thank the members of my Ph.D. committee: Dr. Daniel Tauritz, Dr. Bruce McMillin, Dr. Thomas Weigert, Dr. Sahra Sedighsarvestani, and Dr. Ali Hurson. They all helped both me and this project become what we are today in one way or another, and we both are certainly better for it. I would like to especially thank Dr. Tauritz for teaching me what it is to be a researcher and an academic.

I would also like to thank Clayton Price for all of the advice and support he has given me throughout my time as a student. Clayton has been a good friend and a great mentor over the years, helping me become the teacher I am (for better or worse :)).

I would like to thank all of my family and friends for all the support and advice that they provided me.

Most importantly, I would like to thank my wife Kelley. Kelley has been the encouragement to push through when I was overwhelmed, the motivation to stay focused when I grew tired of it all, the smile I needed when things went wrong, the source of self-confidence I needed when I had none of my own, and so much more. Kelley, you were the light at the end of this tunnel; thank you and I love you.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

NOMENCLATURE

ATC	Abstract Test Case
CASC	Coevolutionary Automated Software Correction
COEA	COEvolutionary Algorithm
CS	Covering test case Set
EA	Evolutionary Algorithm
ES	Evolvable Section
FGFL	Fitness Guided Fault Localization
GP	Genetic Programming
LOC	Lines of Code
MOEA	Multi-Objective Evolutionary Algorithm
MOOP	Multi-Objective OPtimization
MOSP	Multi-Objective Solution Priortization
PSTC	Problem Specific Test Case
SAA	Suspicion Adjustment Amount
SBSE	Search Based Software Engineering
SBST	Search Based Software Testing
SOOP	Single-Objective OPtimization
TBLS	Trend Based Line Suspicion
TSP	Traveling Salesman Problem

1. INTRODUCTION

Testing and correcting a piece of software is a time consuming and expensive task. Software is becoming a more pervasive part of life every day; as such, the need for fast and effective testing tools is higher than ever. Automation of the software testing process is one of the most promising approaches to satisfying this need. Search Based Software Engineering (SBSE) rephrases software engineering tasks as optimization problems, then applies artificial intelligence techniques to them. In SBSE, Search Based Software Testing (SBST) is the most active area of research [39].

This thesis presents a study into the automation of the SBSE tasks of software testing, fault localization, software correction, and software verification. Like many SBSE approaches, automation is achieved through the use of a fitness function to guide the search process. However, the development of an effective fitness function is often a difficult problem in and of itself. This thesis also presents a guide for fitness function design, focused on design for Evolutionary Algorithms (EAs) , since they are the most commonly used search algorithm in SBSE [39].

The overall hypothesis of the presented research is that in order to advance current work in SBST, software specifications must be incorporated into the search algorithms used. This incorporation most obviously fits into the fitness function for the searches used. The rationale behind this hypothesis is that in order to achieve high degrees of optimization in software engineering, the system must be told in some way what defines expected, correct behavior for a piece of software. The most obvious source for this definition is from software specifications, since they, by definition, are this information. To this end, research into the generation, verification, communication, and incorporation of specifications is extremely valuable to SBST. The research in this thesis focuses on the communication and incorporation of specifications into the presented SBST studies.

The most significant contributions of this work are:

- A novel investigation into structuring the fitness function design process. The presented approach organizes the design process into a series of steps, prescribing an ordered thought process to the design procedure; and so is useful for both non-expert practitioners and expert users of EAs.

- The Fitness Guided Fault Localization (FGFL) system. A novel application of specification based fitness functions to automated fault localization. Current automated fault localization techniques rely on an oracle (typically a human expert) to define correct program performance. The presented approach removes this requirement, relying instead on a fitness function derived from the software specifications.

- The Coevolutionary Automated Software Correction (CASC) system. An automated software testing and correction system that advances the state of the art by:

  - Incorporating and automating the testing and verification tasks into the system.

  - Novelly extracting information from source code to construct a constraint system to better guide the search processes in the system.

  - Employs a polymorphic test case definition that allows for dynamic test case generation at run time, rather than the static test case set employed by competing state of the art approaches.

  - Novelly supports multi-objective optimization, mapping objectives to individual specifications for the software. In addition to the problem specific objectives, the system also provides a set of optional objectives that, if activated, are optimized after the problem specific objectives.

While touching on many areas of SBSE, the primary focus of the work summarized in this thesis is automated testing, correction, and verification of software. The other studies discussed are preliminary investigations, intended to explore the viability of the discussed approaches as well as formulate major problems to be solved in order to advance the approaches.

The work is an extension of the author's MS thesis [102]. The majority of the new discussions in this thesis are extended versions of published works [103, 104, 105, 106].

2. BACKGROUND

This section discusses the foundational concepts of the approaches presented in this dissertation. Works more specifically related to the presented approaches are discussed in the sections corresponding to the approaches.

2.1. SOFTWARE ENGINEERING FOUNDATION

Software engineering can be defined as “the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software” [91]. Software testing is a subset of the software engineering process that is focused on ensuring the correctness and completeness of a piece of software. There are many different types of formal software testing and each type has many different aspects. The FGFL and CASC systems perform both black box functional testing and structural testing [80]. Black box testing is a method of testing where the tester has no access to the internals of the program being tested, the only thing that can be seen is the input going in and output coming out. Functional testing is simply testing (at any level) for correct functionality of a program. Structural testing is when testing is performed with knowledge of the internal workings of a piece of software (also referred to as white box testing).

In the modern software development process, software testing is becoming increasingly important, particularly in financial terms. In 1978 Jones [52] estimated that catching an error in the system specification phase is approximately 50 times cheaper than it is to correct the error later in the system testing phase. In a 2002 news release [92] the National Institute of Standards and Technology (NIST) estimated that software errors cost the U.S. economy approximately $59.5 billion a year, which accounts for approximately 0.6 percent of the gross domestic product. In the referenced news release, the NIST states that “although all errors cannot be removed, more than a third of these costs, or an estimated $22.2 billion, could be eliminated by an improved testing infrastructure that enables earlier and more effective identification and removal of software defects”. From this, it is clear that efficient and effective software testing and correction methods need to be developed in order to keep up with advancements in software development methods.

2.1.1. Manual Software Testing. The general process of testing a piece of software manually involves planning out the testing strategy to use on the software, developing the actual test cases to use, executing the tests, gathering the results and analyzing and interpreting them, and repeating the process for any bugs that were identified. Manual software testing is often performed by a team of testers who have been involved (at least to some degree) in the development process.

Most software testing performed today is done by human testers. The process of testing a piece of software is quite time consuming. This is problematic because there are many modern tools available to developers which makes them able to produce code more quickly and efficiently, which means that testers are being asked to test more and more code in less time. Because of this, testing is becoming a bottleneck in the software production process.

2.1.2. Automated Software Testing. Search Based Software Engineering (SBSE) [40] is a relatively new area in software engineering. Works in SBSE rephrase software engineering problems as optimization problems and apply search-based optimization algorithms to them. SBSE approaches have been proposed for a wide variety of software engineering tasks; however, the majority of works in SBSE are on Search Based Software Testing (SBST) [114]. As such, there is a plethora of published work in this area (e.g., the CREST SBSE repository currently contains over 500 papers on SBST alone). High level discussion of SBST is provided in this section, with select works mentioned; for a more comprehensive listing/discussion of works in this area, the reader is directed to the CREST SBSE repository [114] and the published surveys on the area by McMinn [65, 66], Nie [74], and Dick [31].

Miller and Spooner were the first to employ SBST [70]. The method they presented involved setting the integers and conditional values in the program to arbitrary constants to drive the program down a pre-specified execution path, then various floating point inputs were provided as input to the program. Korel followed this idea up in [54] and [55] by actually executing the program being tested, whereas Miller and Spooner used symbolic execution. In Korel's implementation, if the execution follows the selected execution path at a branch point, then a zero is assigned to that branch point, otherwise a positive integer is assigned. So by minimizing the assignments, an input can be selected which follows the selected execution path.

A variety of optimization algorithms have since been used for SBST, with EAs being the most commonly used [37, 66]. Xanthakis et al. were the first to apply an EA to SBST [108] to perform structural testing of a program. Most SBST research is focused on structural testing. These works can be organized into the sub-categories of branch-distance based [70, 54, 55, 108, 67, 68], control based [78, 48], or a combination of both [93, 98, 99]. In general, a major issue for SBST structural testing is that the relationship between the components of a test case and the resulting program coverage is often difficult for these systems to detect, especially if the program has coverage elements that are rarely covered by test cases in the test case space.

Functional testing has received much less attention in SBST research. This is largely due to the difficulty of mapping specification to implementation in an automated manner. One approach of note (as it is discussed later in the context of another work) is that of Tracey et al. [93, 94]. Using formally structured pre- and post-conditions, Tracey rated test case performance by how close a test case was to satisfying the pre-condition while violating the post-condition. This rating was calculated using a table of scoring functions for the logical operators used in the pre- and post-conditions that indicated how close to being satisfied the a given relationship was.

2.2. EVOLUTIONARY COMPUTATION FOUNDATION

Evolutionary Computation (EC) is a type of computational intelligence which is inspired by the biological mechanisms of evolution. EC is a broad field which encompasses many different varieties of EAs. A typical EA creates and evolves a population of potential solutions for a given problem. The EA population initially samples the problem space in a random fashion, but in successive iterations it becomes more directed in its search for a solution. EAs are effective for solving Combinatorial Optimization (CO) problems. CO problems usually have large problem spaces and are classified as NP-Hard [22] (a class of problems which are believed to not be efficiently solvable in general). EAs, however, typically have the innate ability of being able to navigate large problem spaces well; this is why EAs so readily apply to CO problems. CO problems can rise from many fields, such as mathematics, artificial intelligence, and software engineering, which makes EC applicable to a large set of typically hard problems.

As mentioned earlier, the field of EC encompasses many different algorithms that all follow the same general evolutionary model, despite the fact that many of these algorithms were developed independently of each other. Historically, the three major algorithm families that made up EC were genetic algorithms, evolutionary programming and evolutionary strategies. More recently, GP has also joined the EC field. The CASC system uses two historical EA flavors: genetic algorithms and GP; which are combined using co-evolution.

2.2.1. Genetic Algorithms. Genetic algorithms are one of the first EC methods conceived and are still some of the more commonly used EA's today. The concept of the Genetic Algorithm (GA) was popularized by John Holland in the

1970's, particularly in his book published in 1975 [42], which was focused on his studies of adaptive behavior. The canonical GA is an EA whose individuals (i.e., the members of the evolving population) are represented as fixed length binary bit strings (other representations have since been used), which favor crossover over mutation as the principle variation operator.

Since the early 1990's a push has been made to combine many of the original algorithms in the EC field into one unified EA model. A typical cycle in the unified EA closely follows the standard evolutionary cycle: initial creation (typically random, but possibly seeded) followed by a cycle of reproduction and mutation, evaluation, and competition. This cycle continues until a predetermined termination condition is reached, such as a set number of generations have passed or a goal fitness is reached in the population. The structure for an EA is laid out by the evolutionary mechanisms it is based on, the only part that is application specific and must be decided by the implementer is the representation for an individual and the fitness function used to determine how well an individual is performing. If a non-standard representation is used, then appropriate customized variation operators need to be defined.

2.2.2. Genetic Programming. GP is a type of EA in which a tree representation is used for the the individuals in the evolving population. As data structures, trees have a wide range of application; however, the application which is most relevant to the CASC system is that of evolving computer programs. In this implementation, each individual in a GP population will contain the parse tree for the program which it ultimately represents. The first reported results of GP were published by Steven Smith in 1980 [90]. In 1985 Nichael Cramer [24] also published results yielded by using GP techniques. Since the early 1990's, John Koza has done a lot to popularize GP, particularly through his classic four-book series on GP [57, 58, 59, 60]. William Langdon and Ricardo Poli are two researchers who also have contributed significantly to the field of GP. In [62] Langdon presents many new and emerging GP techniques are along with the original foundations of GP. Poli has also done a lot of work on parallel distributed GP [79], which is focused on the evolution of programs which can readily be parallelized.

The methods used by GP are very similar to that of a typical EA except the evolutionary operators have been modified to interact with tree structures. Reproduction is performed using a crossover method in which subtrees are interchanged between parents to create the offspring. This makes the reproduction operator a very pivotal part of the evolutionary process because even exchanging a single subtree in a program parse tree can greatly change the outcome of the program the parse tree represents.

Mutation is performed by replacing a randomly selected subtree in an individual with a randomly created subtree. This implies that the mutation operator must be aware of the subtree functionality as to maintain the integrity of the program, e.g., a subtree with a binary arithmetic operator as the root, such as the addition operator, can only mutate to another binary operator subtree, such as subtraction, multiplication, or division. Another option is to make the mutation operator capable of removing or supplying operands in the event that the arity of the node changes.

To evaluate a GP individual representing a computer program, first the individual's parse tree is pretty-printed into its program form. The program is then compiled, if necessary, and executed. The fitness is then determined based on the output of the program.

2.2.3. Co-Evolution. Co-evolution is an extension of the standard EA model where the fitness of an individual is dependent on other individuals in its own or other populations. This relationship can be categorized as either cooperative or competitive. In nature the relationship can take on many forms, for example any predator-prey or non-symbiotic parasite-host relationship represents a competitive (although necessary) mutual dependence. An example of a more cooperative relationship would be nectar seeking insects performing pollination for the plants which supply the nectar.

The CASC system uses competitive co-evolution between a population of evolving programs and a population of evolving test cases. This competition is intended to create a type of evolutionary arms race between the two populations. An evolutionary arms race works much like an actual arms race except it occurs on a genetic level. As the individuals in one population improve in fitness, pressure is placed on the individuals in the other population to improve as well. This process will continue and, if given enough time to evolve, each population will ideally be driven to perform as well as possible. The concept of an evolutionary arms race is not a new one. Christopher Rosin [85, 86] performed extensive research on methods for competitive co-evolution, examining the parasite-host relationship which yields the evolutionary arms race.

The co-evolutionary method has a unique set of problems which can arise during the evolutionary process. These problems are inherently hard to detect, and as such much work has been done in finding ways to pre-emptively counter these problems. One possible problem that can occur in the co-evolutionary process is the phenomenon known as evolutionary cycling. Evolutionary cycling is basically the evolutionary version of rock-paper-scissors, i.e., the genetic configurations of the populations cycle back on themselves and do not advance past a certain point. This phenomenon is hard to detect because the cycle can involve hundreds of states. In [84] Rosin introduced the concept of an evolutionary history or hall of fame. The main purpose of the evolutionary hall of fame is to counter evolutionary cycling. The hall of fame works by storing the best individuals of every generation, then the individuals in following generations compete against individuals sampled from the other population(s) as well as from the hall of fame. So, to perform well an individual must outperform both the current generation's best individuals as well as the best ancestral individuals, which ideally will disallow cycling. This method is used in modern co-evolutionary systems to not only counter the possibility of evolutionary cycling, but to also speed up (and generally improve) the evolutionary process.

Another problem which can arise during the co-evolutionary process is evolutionary equilibrium. This is where the evolving populations come to a point where they are content with their performance against the other population(s). This contentness causes the evolution to fail in that the populations are no longer pushing each other to improve. John Cartlidge is a researcher who has put considerable work into addressing the potential problems in co-evolution [20], and this problem is one which he addressed. The solution Cartlidge presents is to temporarily remove some of the better individuals from one population causing the equilibrium to be lost, which in turn would prompt the populations to start evolution again.

The third significant potential problem that can arise during co-evolution is disengagement. In this case, one population evolves so much faster than the other that all individuals of the other are utterly defeated, making it impossible to differentiate between better and worse individuals without which there can be no evolution. To counter disengagement Cartlidge uses his “reduced virulence” method to inhibit the development of the excelling population, allowing the other population(s) to catch up in terms of performance.

Disengagement and cycling are the only two coevolutionary problems that have been observed in the CASC system. Steps have been taken to account for this, and are described in detail in Section 5.

3. FITNESS FUNCTION DESIGN

The design of an effective fitness function for a given problem is often difficult (even for experienced designers [47]). Evidence of this difficulty can be seen in publications like [83, 109, 33], which are by researchers who use EAs but have experienced difficulty in the design of an effective fitness function. The goal of this research is to both create a guide to assist non-expert practitioners in the design of high performance fitness functions, and the formalization of fitness function design to provide a foundation for rigorous investigation.

Many researchers informally develop application specific methods for fitness function design, but do not generalize these methods. Research into the evolution of fitness functions (e.g., [82, 25, 97]) has some relevance to fitness function design, though much of that research is also driven by application specific goals.

The goal of a fitness function is to guide the evolutionary process through the problem environment to an optimal solution. The effectiveness of the fitness function used by an EA is directly related to the effectiveness of the EA as a whole. The fitness function is the primary point in the EA where the problem specifications are enforced. For this reason, the problem specifications are an ideal location to start the fitness function design process. The presented guide starts by identifying the requirements that define a solution to the problem outlined in the specifications. Each requirement is classified using the provided taxonomy and then a fitness function component is generated (based on the classifications applied) that is responsible for enforcing the requirement in the fitness function. The fitness function components are finally combined into a fitness function for the problem, which can be either composite or multi-objective.

A number of approaches have been investigated in related research to attempt to classify fitness functions or determine their effectiveness for a particular problem. In [44] a survey of fitness function classification techniques for EAs using binary string representation is presented. Many of the classified techniques are focused on calculating GA-hardness, i.e., epistasis variance, fitness distance correlation, and bit wise epistasis. The survey points out where the presented methods have critical flaws, making them only applicable to specific problems. A similar survey for fitness functions used in evolutionary robotics provides another method for classification [73]. The classes presented, however, are specific to the types of fitness functions used in evolutionary robotics; the majority of fitness functions used in other types of EAs would all fall into a single class of the presented classifications.

3.1. FITNESS FUNCTION GENERATION

The proposed method for fitness function generation can be divided into a series of generalized steps. The first step is to identify the individual requirements indicated by the problem specifications, i.e., the problem requirements such that if a candidate solution fully satisfies each requirement, then it is a valid solution to the problem. The second step is to classify each requirement according to a taxonomy. Each classification implies information regarding the nature and design of an associated fitness function component. The last step is to use the classifications for each requirement to generate an appropriate fitness function component using the information from the requirement's classification. These fitness function components are then composited either into a single fitness function or implemented as separate dimensions in a multi-objective fitness function.

Through this discussion, the classic Traveling Salesman Problem (TSP) (see [23] for an introduction to the TSP) will be employed as running example to help illustrate the various classifications in the taxonomy. The specification for the TSP is: given an adjacency matrix A, find the shortest Hamiltonian circuit of the graph represented by A. Assume that non-adjacent nodes have an infinite length in the corresponding adjacency matrix entry. Non-infinite entries in A indicate both that the corresponding nodes are adjacent and the length of the path between the nodes.

3.1.1. Addressing Problem Specifications. The purpose of the fitness function is to guide the evolutionary process through the problem space, ultimately arriving at a valid solution to the problem. In order to generate an effective fitness function, the characteristics of a valid solution to the problem must be defined. If properly indicated, the problem specifications should contain this information in some form, which means that the first step is to identify the solution requirement(s) from the problem specifications.

In some cases, problem specific knowledge can be exploited to speed up convergence and/or generate solutions of a specific type. This exploitation can be included in the problem requirements and as such be assimilated into, and promoted by the fitness function. For example, assume that a program is being evolved and a specific set of statements s are *known* to be required in order to generate a high quality solution. In this case, the specification that a solution must contain s would be added to the problem specifications, and would generate additional requirement(s) for the fitness function to address. However, if it is only *thought* that these statements are required, then in the case this assumption is incorrect, having it as a requirement would result in a negative impact on the search. In such cases, the appropriate approach is to seed the initial population with this bias.

For some problems there may only be a single requirement of the solution, in which case the specifications can be used directly as the problem requirement when generating the fitness function. Otherwise, each identified requirement should address an atomic aspect of the solution, as doing so will facilitate the construction of a fitness function. The fitness function must address every aspect of a problem's defined solution in order to properly guide the evolutionary process. So, each solution requirement obtained from the problem specifications will ultimately yield a fitness function component. Once all requirements have been classified, the resulting fitness function components are combined into the fitness function for the problem. This ensures that every requirement set forth by the problem specifications is considered in the fitness function.

The TSP has two apparent requirements for a candidate solution (i.e., a path through the graph represented by A) to be a valid solution:

1. The path must be a Hamiltonian cicuit (i.e., it must visit all nodes without revisiting and it must return to the starting node).
2. The Hamiltonian circuit must be the shortest possible.

3.1.2. Classification Taxonomy.

After the problem requirements have been identified, the next step is to begin bridging the gap between written problem requirements and a fitness function. The method proposed addresses this task by defining a taxonomy, which classifies the problem requirements and in doing so provides information on the nature of the fitness function. The taxonomy is shown in Figure 3.1.

As was discussed in the previous section, the starting point is the problem specifications. Determining the solution requirements is the action going from *Problem Specifications* to *Problem Requirement*.

Next, an appropriate solution representation and EA configuration must be determined to solve the problem, which are both based on the problem specifications. There may be solution requirements that arise based on the algorithm selected. These requirements arise due solely to the selected EA configuration and as such cannot be expected to be included in the problem specifications, since for a given problem there may be a number of possible algorithms to use. As example of an *algorithm induced*

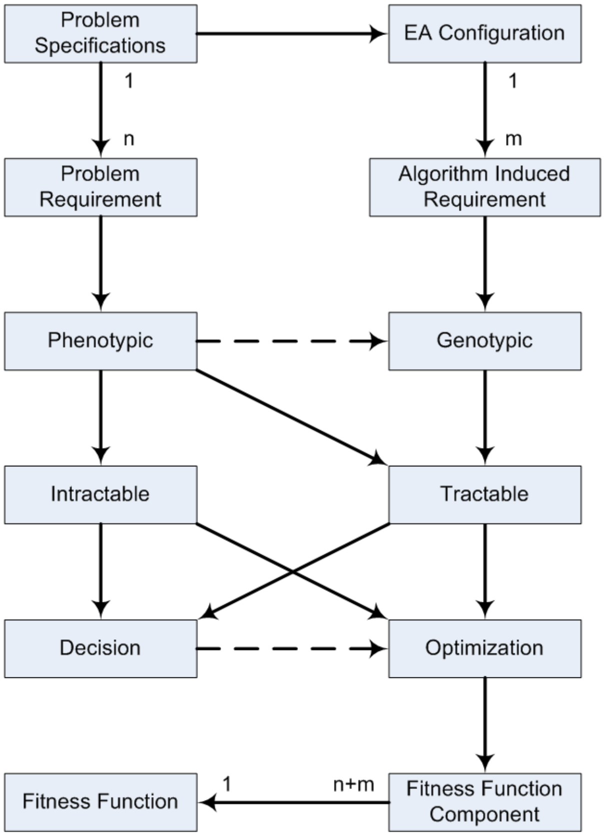


Figure 3.1: Requirement Classification Taxonomy

requirement, suppose that for a given problem we choose to employ Genetic Programming (GP). An issue that is typical to GP is tree bloat in the evolving candidate solutions [57]. So in response to this, suppose we want our fitness function to penalize bloating by promoting smaller tree structure. The fitness function component responsible for this requirement has nothing to do with the problem being solved, just the algorithm that was selected to solve the problem. This illustrates the need for *algorithm induced requirements*.

For the TSP example, assume that a standard EA is to be used and a candidate solution will be an ordered list of graph node identifiers which represents the solution's path through the graph. For example, if a candidate solution C=*[a, b, c, d, a]* then the path that C takes is node a, node b, node c, node d, then back to node a. Path lengths can be assessed using the A matrix with the candidate solution. To make this example simpler, assume that the EA operators will only generate valid paths through the graph (by using the A matrix), so the fitness function does not need to check path validity. There are no obvious *algorithm induced requirements* for this EA configuration and representation, so this example will only have problem based requirements.

3.1.2.1. Phenotypic and genotypic. The first level of classification addresses how the requirement will be assessed. A phenotypic requirement is based on some aspect of a candidate solution's expression in the problem environment, independent of the candidate's genetic representation. Conversely, a genotypic requirement is based on some aspect of a candidate solution's genetic structure.

Since the problem specifications are stated independent of a specific algorithm, it is impossible for a problem requirement to be genotypic. Similarly, algorithm induced requirements are based solely on the algorithm selected and are independent of the specific problem being addressed. As such, problem requirements are always phenotypic and algorithm requirements are always genotypic.

In many cases, it is advantageous to convert phenotypic requirements to genotypic, if possible. The reasoning behind this is that if desirable phenotypic behavior can be mapped to a genotypic configuration (i.e., desired candidate solution behavior can be mapped to a solution representation characteristic), then promoting this configuration will be much easier (as it will be known what genes are responsible for the desired behavior) and convergence will occur much quicker. In Figure 3.1 this conversion is shown as the dashed edge from *Phenotypic* to *Genotypic*.

Another way of looking at phenotypic to genotypic conversion is as a mapping of problem requirements to algorithmic requirements (due to the fact that these requirements are always phenotypic and genotypic, respectively). For example, consider the classic n-queens problem for which one requirement is that each placed queen must not be in the attack lines of the other $n-1$ queens (vertical, horizontal, and diagonal). A basic solution representation for this problem would be a two dimensional array of size n x n; however, the aforementioned problem requirement can be mapped to an algorithmic requirement to dramatically reduce the search space by adjusting the solution representation to an array containing a permutation of the numbers 1 through n, where element i represents what row in column i the queen is placed on. This conversion will speed up the convergence of the EA in question by significantly reducing the search space of the problem by removing horizontal and vertical attack lines from consideration, which will also make the assessment of a candidate solution easier.

In terms of the fitness function component, the genotypic and phenotypic classifications indicate whether or not input will be required by the component. Genotypic components will not require any input aside from the candidate solution itself, since they are based solely on the genetic structure of the candidate. Phenotypic components will require input from the problem environment, since the component is assessing an aspect of a candidate solution in the context of its problem environment. As such, phenotypic components will require the input of the entire, or a sampling of the, problem environment (which is decided in the second classification, discussed in the next section).

For the TSP example, both requirements from Section 3.1.1 are problem requirements, so they both fall under *phenotypic* classification. The first requirement has no obvious conversion to *genotypic* classification. However, the second requirement (i.e., the Hamiltonian circuit must be the shortest possible) could be converted to a *genotypic* classification by adjusting the structure of a candidate solution to also contain the path length to get to the indicated node in the solution array (and as such the requirement could be assessed based solely on the solution genotype). For example, a candidate solution *C=[(a,0); (b,3); (c,2); (d,1); (a,4)]* would have a total path length of 10. This conversion will not improve the rate of convergence of the fitness function, and is made for the sake of this example (as the conversion affects the remaining classifications).

The conversion of the second TSP requirement performs an *explicit* enrichment of the genotype by adding partial performance information to it. This style of enrichment will typically provide improved algorithm efficiency, by making component fitness calculation easier (though in this example the efficiency gain is negligible). The n-queens example discussed earlier, however, performs *implicit* genotype enrichment in which information from a problem requirement is used to enhance the algorithm with an improved solution representation. This style of enrichment can result in both efficiency and performance improvements, as the requirement is being mapped directly into the genotype of a candidate solution.

3.1.2.2. Tractable and intractable. The second classification is concerned with the practicality of assessing a given requirement. Essentially this classification determines if the resulting fitness function component will calculate the true fitness value or an approximation to the true fitness value. Suppose for a given requirement the problem domain is all real numbers; for this requirement it is impossible to calculate the true fitness value, so this requirement is intractable and an approximation must be made by using a sampling from the set of real numbers. On the other hand, suppose the problem domain for a requirement is a finite graph. This requirement would likely be classified as *Tractable*, since it is feasible to calculate the true fitness by operating on the problem graph. These two examples, though illustrative, do not represent all possible scenarios for this classification. Since the primary concern is practicality, a requirement may be classified as tractable in one case and yet in another the exact same requirement could be classified as intractable. This requirement is very much dependent on the resources available to a specific user.

In Figure 3.1 you can see that genotypic requirements can only be classified as tractable. The reasoning behind this restriction is that genotypic requirements are based on the genetic structure of a candidate solution, which implies that the calculation of the true fitness of such a requirement should be feasible as long as the candidate solution representation is practical.

This classification further fleshes out the details of the input to a fitness function component. Phenotypic fitness function components can calculate either the true or approximate fitness for the requirement. If the true fitness is calculated (i.e., the requirement is tractable), then the component will operate on the entire problem domain for the requirement, and thus will need to have access to it. If the component calculates an approximate fitness (i.e., the requirement is intractable), then it will operate on a sample set taken from the problem domain, which means a sampling method must also be decided upon for the fitness function component. In many cases a single sampling function is used for multiple fitness function components (to ensure that each component operates on the same sample set), so this decision may affect more than one component.

In the running TSP example, the first requirement was classified as *phenotypic* and as such can be either *tractable* or *intractable*. For this example, assume that resources are such that it is feasible to calculate the path cost for a full Hamiltonian circuit of the graph, if such a circuit is discovered. So since the problem space is manageable and navigation of it is feasible, it is possible to calculate the true fitness for the first requirement, thus the requirement is classified as *tractable*. The second TSP requirement was converted to *genotypic* classification so it will also be classified as *tractable*, based on the prior reasoning in this section.

Intractable classification has non-trivial implications for the fitness function component in question. Section 3.2 presents examples of *intractable* classification.

3.1.2.3. Decision and optimization. The third classification defines the basic nature of the requirement in question. If a requirement is either satisfied or it is not (with no intermediate satisfaction), then it is a decision requirement. If there are intermediate levels of satisfaction of the requirement, then it is an optimization requirement.

EAs require a gradient in the fitness function in order for the evolutionary process to be effective. If a fitness function is defined as a decision problem, then the evolutionary process degenerates into basically a random search. For this reason, any requirement that is classified as a decision requirement should be transformed into an optimization requirement in order to effectively guide the evolutionary process. For example, suppose that a requirement is that the output of a candidate solution is to be in sorted order. Clearly, the output is either sorted or it is not, so the requirement receives a decision requirement. So in order to transform the requirement to optimization, a method is needed for determining how close to being sorted the output is and then use that for the fitness function component.

Additionally, the more gradient that each fitness function has, the better; i.e., having a single *'partially satisfied'* intermediate level of satisfaction is not going to greatly benefit the evolutionary process. So, some requirements may be classified as optimization, but will still need to be refined in order to generate a more effective fitness function.

Per standard terminology, EAs attempt to maximize fitness values (i.e., more fit candidate solutions are better). So there may also need to be steps taken to modify an optimization fitness function component to adhere to this.

This classification (i.e., decision/optimization) indicates how the fitness function component is actually going to enforce the requirement that it is based on. As a result, this classification will often require more consideration than that of the other classifications. However, once this classification is made, the user should have a very good idea about how the fitness function component will work, making implementation much simpler.

The approach taken in assessing an optimization requirement should be carefully considered, as some methods (even with sufficient gradient) can still be ineffective at guiding the evolutionary mechanism to satisfy the underlying problem requirement. For example, if relationships exists between the genes in a solution, then an effective fitness function component should consider all such relationships when calculating the fitness. An example of this is shown in Section 3.2.2.1.

In the TSP example, the first requirement is stated as a decision problem, i.e., either the path is a Hamiltonian circuit or it is not. So that means that the next step is to decide on a way to convert this requirement into an optimization problem. One possible conversion is to reward for each unique node visited and to penalize for both revisiting a node and for not returning to the starting node (if necessary, a penalty could also be applied for illegal moves through the graph). So assuming that this conversion is acceptable, the requirement is classified as a decision problem with a conversion to an optimization problem.

The second TSP requirement is already an optimization problem, i.e., the shorter the path the better. However, the requirement is a minimization problem; so it must be converted to maximization. Negating the value is one method to perform this conversion, assume for the running example that this is what is decided upon.

3.1.3. Fitness Function Components. The last step is to combine all of the fitness function components into the fitness function for the problem. This step is largely dependent on the developer's desired format for the fitness function. One option for this step is to combine the fitness function components into a large single function in which the component fitness values are combined together into a single fitness value. This option may work well for some cases; however, combining the various component fitness values can sometimes be difficult. Weighting the component fitnesses can often be challenging and, even if a good weighting scheme is developed, it is still possible for components to conspire in order to increase their component fitness values to the detriment of overall performance. A second option is to use Multi-Objective EA (MOEA) [27] methods to calculate a fitness rank based on the pareto front generated by using each fitness function component as a new dimension. This allows for the optimization of the component fitness values without the potential trouble of component weighting and conspiracies.

In the TSP example, the first requirement was classified as a phenotypic, tractable, decision problem with optimization conversion. So from this, the component will take the problem space (i.e., the A matrix) as an argument (in addition to a candidate solution) and will calculate the true fitness using the method discussed. Suppose this method is implemented as a function called *CheckHamCirc* which takes an individual S and the adjacency matrix A as arguments.

The second requirement was classified as a (converted) genotypic, tractable, optimization problem. So the fitness function component for the second requirement will take only a candidate solution as an argument and will calculate the true fitness for the requirement by negating the path length of the candidate solution. Suppose this component is implemented as a function called *InverseLength* that takes a candidate solution as an argument.

The last step in this example is to generate the fitness function F for the problem using the fitness function components. One option is to combine the components into a single expression, like is shown in Equation 1.

$$F(S) = CheckHamCirc(S, A) + InverseLength(S) \tag{1}$$

If the component fitness values are normalized to fall in the same range (e.g., both return a value in [0,100]) then there will likely be no problem with using this fitness function. However, another option is to implement this fitness function as a two dimensional MOEA. In this case, the pareto front contains the current best solutions.

3.2. EXAMPLES

3.2.1. Finding the Inverse of a Function.

For this example suppose our problem is to find the inverse of a given function f non-symbolically. If the function g is the inverse of f, then if *f(x)=y* that means *g(y)=x*. A formal discussion of this problem can be found in [101]. From these specifications the sole requirement for this problem can be identified:

1. For a function g to be a solution, *f(g(x))* $= x$

Now an appropriate EA configuration must be decided upon. The evolutionary operators will need to be able to easily modify the elements in candidate solution functions, so a tree representation for the candidates would be useful; this means that GP would be a good decision for this problem. GP implementations often experience tree bloat during the evolutionary process, so it would be a good idea to promote smaller tree size in the fitness function, which means there will be an algorithm induced requirement:

2. Tree size should be minimized

3.2.1.1. Classify requirement one.

This first requirement was identified from the problem specifications, which means that it is phenotypic.

The domain for this problem is all real numbers (assuming no domain restrictions based on the particular function used for f). Clearly, it will be impossible to calculate a true fitness for this requirement, so it is intractable. This means that the component will require a sampling method of some sort to select values with which to test the candidate solutions. There are three implementation questions concerned with how this testing is conducted (these questions are typically common to problems requiring sampling methods):

- How many values is each candidate tested against?
- How often are the values reselected?
- Are all candidates tested against the same values?

Suppose for this problem it is decided that there will be 1000 values (decided upon arbitrarily for the purpose of this example) selected using a random sampling function prior to execution and each candidate will be tested against all selected values.

Since multiple samples give multiple intermediate fitness values for each candidate, a composition method will need to also be decided upon. Typical composition methods are summation and averaging; for this example suppose averaging is decided upon (i.e., the component fitness will be the average of all 1000 intermediate fitness values).

This requirement is stated as a decision problem, so it must be transformed to an optimization problem. One possible transformation is to use the fact that if a function g is an inverse of f, then $f(g(x)) = x$ and anything else yielded by this can be used to generate an error value (i.e., $error = \sqrt{f(g(x)) - x}$); suppose this method is decided upon for the fitness function component.

3.2.1.2. Classify requirement two. The second requirement is algorithm induced, which means that it is genotypic and, therefore, tractable; so the true fitness will be calculated by the fitness function component.

The second requirement is stated as an optimization problem, so no transformation will be necessary. However, the requirement is a minimization problem and fitness functions are per definition maximized. One method of changing minimization to maximization is to negate the output value; assume this for this example.

3.2.1.3. Fitness function generation. The first requirement was classified as a phenotypic, intractable, decision problem with a transformation to optimization. From the classification process we determined that the candidate solutions will be tested against a static set of values selected before execution, say these values are stored in an array *D*. Also, the component fitness will be calculated by taking the average of all 1000 intermediate fitness values. With this information, the fitness function component can now be pieced together (where *g* is the candidate whose fitness is being calculated):

$$0.001 \cdot \sum_{x \in D} \sqrt{f(g(x) - x} \qquad (2)$$

The second requirement was classified as a genotypic, tractable, optimization problem (with a conversion from minimization to maximization). Through the classification process, the implementation of this fitness function component is straightforward (assume that the *TreeSize* function counts the number of nodes in the argument candidate solution):

$$-TreeSize(g) \qquad (3)$$

The final step is to decide how to combine the fitness components. Just like in the first example, this fitness function would likely work fine as a single equation (i.e., just sum the component fitness values) or a MOEA could be created using the components.

3.2.2. Correction of a Sorting Program. The presented guide was used to create the fitness and objective functions used by modern versions of the CASC system (described in Section 5). In the publication for the presented guide, a previous version of CASC was used as a real world case study for the guide. In this study the performance of a guide generated fitness function was compared to the fitness function used by Arcuri, the author of a similar automated software correction system (discussed in detail in Section 5.1.2).

Both CASC and Arcuri's system attempted to correct buggy versions of the bubble sort algorithm. The fitness functions used by these systems are based on the specifications of the software being corrected, and so the problem specification used were that of any sorting algorithm: the output of the algorithm should be the input in sorted order.

3.2.2.1. CASC system fitness function. From the specifications described in the previous section, two solution requirements can be identified:

1. The elements outputted must be in sorted order
2. The elements outputted must be a permutation of the elements inputted

Since we are using GP, tree bloat could be an issue; however, testing of the CASC operators has shown that the system does not suffer from bloating. So bloat does not need to be considered in the fitness function.

The first requirement is a problem requirement, which means that it will be phenotypic. Because the requirement is phenotypic, the next step is to determine if it is tractable or intractable. There is an infinite number of possible inputs to a sorting algorithm, so the problem is clearly intractable. This algorithm shows an example of a case where a requirement is intractable but the sampling method is built into the problem thus making an additional sampling method unnecessary. This is due to the fact that the candidate solution inputs are also being evolved in a second population. So when calculating the fitness for a candidate solution, the algorithm's selection operator will select the inputs from the other population. The component will have multiple intermediate fitness values, so a composition method will need to be selected; the CASC system uses the average of average value of the intermediate fitness values as the fitness, so that will be used in this example.

The first requirement is stated as a decision problem (i.e., the output is either sorted or it is not), so it must be transformed into an optimization problem. This transformation can be done in a number of ways, one of which is by first considering what it means for an element to be in sorted order among other elements: the element is greater than or equal to all elements before it and less than or equal to all elements after it.

A possible approach would be to generate a score for each output element x that is incremented by one if element x is less than or equal to element $x + 1$ and also if element x is greater than or equal to element $x - 1$. The scores for all elements would be summed and returned as the score for the input. However, using this approach only considers the relationship an element has with two of the other elements, disregarding all of the other relationships. This effectively limits the fitness function component to assess that a variable is in the correct position relative to its neighboring elements, which essentially means there is no penalty for how far out of place an element is relative to its correct position. More gradient in the fitness could be achieved (and as a result more evolutionary guidance provided) if all element relationships were assessed.

Instead, consider an approach that generates a score for each element x that is equal to the number of elements before x that are greater than or equal to x plus the number of elements after x that are less than or equal to x; the scores for all elements are then summed up and returned as the score for the output. In this approach all relationships between the elements are considered and each element is penalized based on how far out of position it is, which will provide greater guidance to the search. This scoring approach is implemented for this example as a function called *CountSorted*.

The second requirement is also a problem requirement, meaning that it is phenotypic. The fitness function component will need to operate on the same input values as the first requirement. As such, this requirement will receive the same classification as the first (i.e., intractable) and will not need a sampling method specified (it will use the same as the first).

The second requirement is stated as a decision problem, so it must also be transformed. A straightforward transformation is to just count the number of elements missing and use that value. The number of missing elements should be minimized, so a conversion to maximization must be performed. One way to convert this value is to use it to apply a penalty to the first component, e.g., if all elements are present, then there is no penalty, if half of the elements are present, then the fitness of the first component is halved, etc. Say a function is implemented to calculate the amount of penalty called *CalcPenalty*.

Using the classifications for the requirements, piecing together the resulting fitness function is straightforward. The requirements operate on the same set of inputs (determined by the algorithm selection method) and the second component acts as a penalty to the first component. The resulting fitness function is shown in Algorithm 1.

Algorithm 1 Guide Generated Fitness Function for CASC

```
$Score \leftarrow 0$
$Inputs \leftarrow SelectInputArrays()$
for $i \leftarrow 1$ to $SizeOf(Inputs)$ do
    $output \leftarrow execute(P, Inputs[i])$
    $newScore \leftarrow CountSorted(output)$
    $penalty \leftarrow CalcPenalty(inputs[i], output))$
    $Score \leftarrow Score + newScore \cdot penalty$
end for
$P.Fitness \leftarrow \frac{Score}{SizeOf(Inputs)}$
```

3.2.2.2. Results in comparison. The comparison was made against the original version of Arcuri's system [8, 11, 9]. In addition to some key design differences (described in Section 5.1.2), the system used two additional requirements that were not used in the CASC system. The full set of requirements considered in Arcuri's fitness function is as follows:

1. The elements outputted must be in sorted order
2. The elements outputted must be a permutation of the elements inputted
3. Program tree size should be minimized
4. Run time exceptions should be minimized

The added third requirement is a straight forward algorithm induced requirement, an example is shown in Section 3.2.1. The fourth requirement is an additional problem requirement that is made possible by the fact that Arcuri's system interpretively executes the candidate solution programs, whereas the CASC system compiles the programs and executes the resulting binary program. Using interpretive execution, Arcuri's system is able to count and respond to (i.e., step over) exceptions that arise during execution. Run time exceptions in the CASC system result in program termination and the candidate solution is assigned the minimum fitness value. The guide classifications of both fitness functions are shown in Table 3.1.

When comparing the systems, both were tested against the same buggy software and both used parameter configuration values that were as equivalent as possible. The software being corrected consists of eight buggy implementations of bubble sort detailed in [11].

Four system configurations were considered. Results are presented for Arcuri's system both using the automatically generated fitness function and additionally the same fitness function with an added short program penalization (under the rationale

Table 3.1: Classification of Both Fitness Functions Considered

Req.	Classification		
CASC Fitness Function			
1	Phenotypic	Intractable	Decision
2	Phenotypic	Intractable	Decision
Arcuri Fitness Function			
1	Phenotypic	Intractable	Decision
2	Phenotypic	Intractable	Decision
3	Genotypic	Tractable	Optimization
4	Phenotypic	Intractable	Optimization

that "it is very unlikely that a correct solution (i.e., an evolutionary program that satisfies the formal specification) might be much smaller (i.e., having many less nodes) than the buggy instance" [11]). Experiments were conducted with the CASC system using the guide generated fitness function as well as Arcuri's fitness function (except for the run time exception portion). The results of all experiments are summarized in Table 3.2.

Table 3.2: Experiment Success Rates Over 100 Runs

Bug ID	Arcuri	Arcuri w/ Penalty	CASC w/ Guide Fit.	CASC w/ Arcuri Fit.
1	64%	84%	98%	100%
2	74%	94%	92%	90%
3	83%	97%	96%	43%
4	68%	85%	53%	66%
5	68%	79%	0%	0%
6	0%	0%	6%	6%
7	0%	0%	0%	0%
8	0%	0%	0%	0%

The results presented in Table 3.2 provide evidence showing that the guide generated fitness function performs at least as well as the fitness function used in the state of the art system (and even performs better in some cases) for the first three bugs considered. For the fourth bug the guide fitness function is still competitive, though it does not perform as well as Arcuri's fitness function. In many of the CASC experiments using the Arcuri fitness function, a conspiracy of fitness function components occurred resulting in the solution program simply outputting the values $1, 2, 3, 4, ..$, regardless of input; Arcuri reports similar observations when using the fitness function with his system [9]. The guide generated fitness function, however, had no occurrences of a fitness function component conspiracy. The significance of these results is that they provide evidence that the guide can be used by a practitioner to generate a competitive fitness function (in terms of quality).

The performance of the two systems on the fifth bug is anomalous, as there is no obvious reason for the differences in performance. However, this difference may be due to the different operators employed by the two systems.

4. AUTOMATED FAULT LOCALIZATION

Fault localization is an essential step in software debugging and it is also the most expensive step in this process [96]. The high cost of fault localization is due to the fact that in many cases software errors are located manually employing software analysis tools and techniques. Therefore the task of fault localization would greatly benefit from automation. This serves as motivation for the development of automated tools and techniques that either assist in or autonomously accomplish fault localization.

This section presents the Fitness Guided Fault Localization (FGFL) system, which focuses on fault localization in software for which a fitness function can be derived and source code is available. This derivation can be from specifications (formal or informal), an oracle (e.g., the software developer), or some other source. In the FGFL system, the fitness function should both indicate when the software is performing correctly and quantify the degree of error when software operates incorrectly. The FGFL system novelly employs the fitness function to guide dynamic analysis of the software in question.

The FGFL system employs an ensemble of dynamic analysis techniques to perform fault localization. Currently, three such techniques have been implemented. The techniques currently implemented are: trace comparison, trend based line suspicion, and run-time fitness monitor.

Each FGFL technique can be activated or deactivated for a given run, which allows the user to omit a technique if it is expected to not be applicable for a particular program. The results for activated techniques are aggregated using a confidence based voting system in which each technique has a number of votes it can potentially apply to lines in the software suspected to contain the error.

A series of preliminary experimentation is presented using a prototype of the FGFL system in which it is tested against a variety of seeded software errors. Multiple technique configurations are also tested in an attempt to identify synergies between the techniques for various bug types. The results of these experiments demonstrate the potential of automated fitness guided fault localization and serve as a proof of concept for the FGFL system.

Of the three techniques currently implemented in the FGFL system, two are enhanced versions of established techniques; namely, execution slice comparison (or dicing) [6] and the Tarantula technique [51, 50, 49]. The FGFL versions of these techniques exploit the fitness function in order to strengthen the established techniques. The dicing technique achieves a higher degree of automation through the fitness function, which serves as an oracle that can indicate test cases that pass or fail at run time. The modification of the Tarantula technique to exploit the fitness function allows a higher degree of precision in the results by introducing a gradient to test case performance. The run-time fitness monitor is a completely novel technique, which tracks changes in fitness during the execution of the program.

4.1. RELATED WORK

Automatic fault localization is a very active research field with a vast amount of research literature. Due to space considerations, only the most pertinent research to the FGFL system is reviewed here. For a more in depth discussion of this field see [111, 107].

Static slicing [63] was proposed as a technique to assist in bug localization by isolating possible areas that can contain an error. Static slicing uses static code analysis techniques to determine the statements that may influence a variable at a given point in the program. Dynamic slicing [56, 5, 4] is similar to static, except that more information is used to determine which statements *actually* influence a variable reference, rather than the lines that *may*. An execution slice [3] is the set of lines that are executed for a given test case. The trace comparison technique implemented in the FGFL system is an enhanced version of execution slice comparison [6], which takes the set difference between the execution slices of a passed and a failed test case, termed the dice. The dice for the negative test case (i.e., the lines unique to the failed test case execution slice) are indicated as likely containing the error. This technique is described in more detail in Section 4.2.2. Also, a new FGFL technique is being considered, which combines dynamic slicing techniques with the fitness monitor; this is discussed further in Section 7.

Delta debugging [110, 112] is an approach that has some conceptual similarities to the trace comparison technique included in the FGFL system. In delta debugging, positive and negative test cases are sought after that minimize the difference between their execution traces. This is accomplished by studying cause-effect relationships by isolating portions of the test case input that are related to the observed error. Iterative delta debugging [13] applies the delta debugging technique to more complex errors that are masked by other errors in the software. Though similar conceptually, the implementation of the trace comparison and the assumptions made regarding the trace are different between the FGFL technique and delta debugging. Primarily, the assumption that minimizing the difference between positive and negative execution traces is beneficial is not held in the FGFL system, as an error can just as easily be the result of the code that was not executed as the code that was. The FGFL trace comparison technique takes a more conservative approach, using the largest difference between execution traces in order to avoid overlooking an error. This approach results in a sometimes increased search space versus delta debugging, which minimizes the search space at the cost of sometimes overlooking an error.

The Tarantula debugging tool [51, 50, 49] is a coverage-based technique that monitors, for each line, how many positive and negative test cases execute the line.

Lines are assigned suspicion levels based on the proportions of passed and failed test cases that executed the line in question. The trend-based line suspicion technique in the FGFL system is an enhanced version of this technique. In the Tarantula technique, the contribution a test case has to an executed line is binary, i.e., the test case was either passed or failed. With the addition of a fitness function, a gradient is achieved when assessing the test cases that executed a line; test cases that performed very poorly contribute more suspicion to the line than test cases that are near correct. Also, test cases that are passed reduce suspicion in the lines executed. The results are then adjusted using confidence values based on the minimum and maximum possible suspicion obtainable, which is similar to the H heuristic in Tarantula.

4.2. FITNESS GUIDED FAULT LOCALIZATION

The FGFL system currently operates on a subset of C++ programs (no focus has been placed on fault localization in object oriented software). During execution, the system takes the source code of the software to be corrected as input. Next, a copy of the source code is made that has been instrumented to produce technique-specific output that is used for analysis by the selected techniques. An automatically defined and instantiated class is added to the source code, which is responsible for managing the output details as well as various file streams used to output execution information. Calls to the member functions of this class are automatically placed throughout the source code as is appropriate for the activated techniques.

Figure 4.1 shows a high level view of the general operation of the FGFL system. Each block in this flow chart is discussed in detail in this section.

The FGFL system uses an EA to generate a set of test cases for the program in question. A polymorphic test case object is used by the system to define what a test case is for a given problem as well as various operations needed to evolve a population of the test case. This object is described in detail in Section 5.2.1.

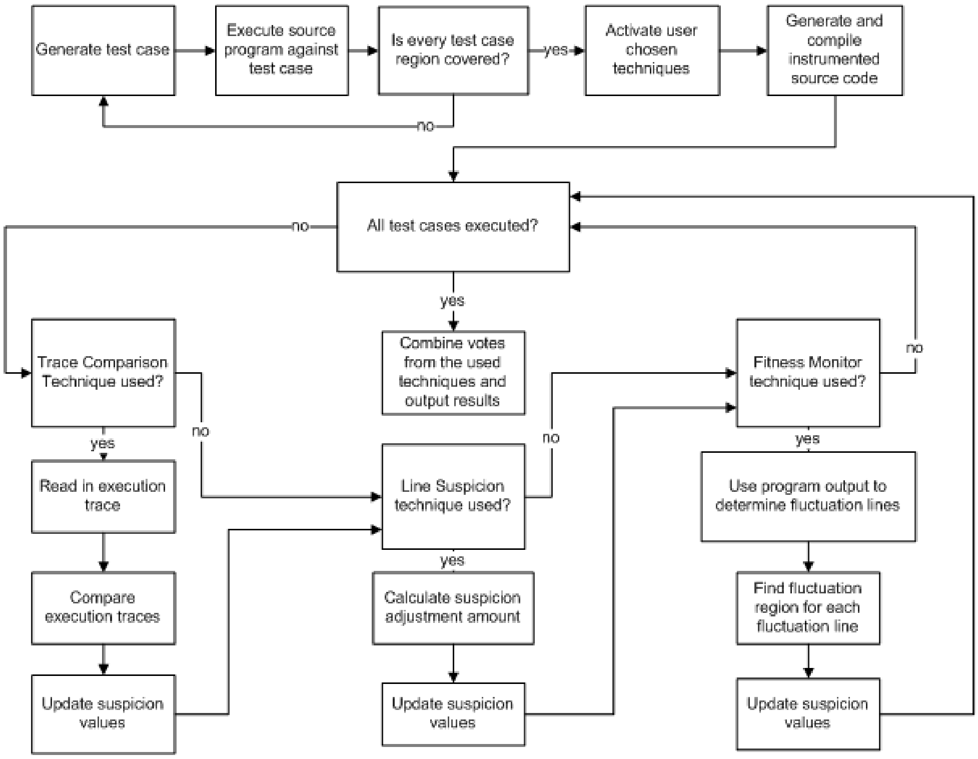

Generate test case
Execute source program against test case
Is every test case region covered?
yes
no
Activate user chosen techniques
Generate and compile instrumented source code
All test cases executed?
no
yes
Trace Comparison Technique used?
yes
no
Read in execution trace
Compare execution traces
Update suspicion values
Combine votes from the used techniques and output results
Line Suspicion technique used?
yes
no
Calculate suspicion adjustment amount
Update suspicion values
Fitness Monitor technique used?
yes
no
Use program output to determine fluctuation lines
Find fluctuation region for each fluctuation line
Update suspicion values

Figure 4.1: High Level Flow Chart of the FGFL System

Some of the techniques used by FGFL apply suspicion to lines based on the fitness value for a test case. Given this, there needs to be a balance between both positive and negative test cases as well as among the negative test cases. And so, the test case generation EA used in FGFL has two goals: first to create a specified number of positive test cases (i.e., test cases that do not demonstrate any error), second to create negative test cases (i.e., test cases that demonstrate the error) that have a uniform distribution of fitness values. The fitness used in the generation EA (called the generation fitness) is based on what is desired in the performance fitness values of the generated test cases. The first goal is accomplished by giving positive test cases an arbitrarily high generation fitness as long as the specified number of positive test cases has not been found. Once a sufficient number of positive test cases are created, additional positive test cases are given an arbitrarily low generation fitness. Negative test cases are assigned a generation fitness value equal to the minimum difference between the test case's performance fitness and that of all other test cases in the population. This essentially applies pressure between the negative test cases, pushing them apart from each other in terms of performance fitness. If the specified number of positive test cases or a uniform negative test case distribution is not created, then the user is prompted about whether or not to continue with the fault localization process.

Through testing all generated test cases, each selected technique generates a suspicion level for each line in the program being considered. The techniques are given an equal number of freely distributable votes, which are used to indicate where the error is according to the results obtained by the technique. These votes are applied to lines in the program based on the technique's confidence that the line contains an error. The confidence for a line is a function of the suspicion level for that line and both the maximum and minimum possible suspicion obtainable for the technique, which are described in the detailed technique descriptions later in this section. The number of votes allotted to each technique is equal to the Lines Of Code (LOC) count in the program being considered. Each technique can distribute its votes as it sees fit, with no restriction placed on the number of votes that must be cast (i.e., the techniques can use a fractional portion of the votes allowed). After each technique has distributed its votes across the program, the votes for each line from all techniques are averaged to obtain the final suspicion levels for each line of code. This aggregation scheme allows for a great deal of flexibility in the FGFL system and also allows the techniques to contribute to the result as much as is appropriate.

4.2.1. Running Example. Through this discussion, an incorrect version of bubble sort will be used as a running example. For the sake of clarity, the running example is a reduced version of the experiments detailed in Section 4.3. The formal specifications for this program are:

- $\forall$ 0 $\leq$ i $<$ SIZE: data[i] $\leq$ data[i+1]
- $\forall$ x $\in$ input: x $\in$ data

The program will take in a test case (an ordered list of $SIZE$ values) in the form of an *input* array. The *data* array will start as a copy of the *input* array (so the *input* array can be used to make sure values are not lost), be sorted in place, then will be output as the result of the program. The output (i.e., the resultant *data* array) of the program will be correct when it satisfies all program specifications. Pseudocode for the running example program is shown in Algorithm 2. To make this example more illustrative, assume that the pseudocode in Algorithm 2 is embedded in a longer program, say 15 lines, where all lines not shown can be assumed to be correct. In this figure, the error is shown on line 11, in which *data*[*i*] should be *temp*.

Algorithm 2 Pseudocode for Running Example Program

```
5:   $data \leftarrow input$
6:   for $i \leftarrow 0$ to $SIZE$ do
7:     for $j \leftarrow 0$ to $SIZE - 1$ do
8:       if $data[j] > data[j+1]$ then
9:         $temp \leftarrow data[j]$
10:        $data[j] \leftarrow data[j+1]$
11:        $data[j+1] \leftarrow data[j]$
         end if
       end for
     end for
```

The fitness function used in this running example will be the one generated using the fitness function design guide discussed in Section 3, shown in Figure 1 on page 29.

The test cases that will be used with this example are shown in Table 4.1, along with the output produced for each test case by the running example program and the resulting fitness for the run. This example will divide the fitness function range into four segments and will use one test case for each segment.

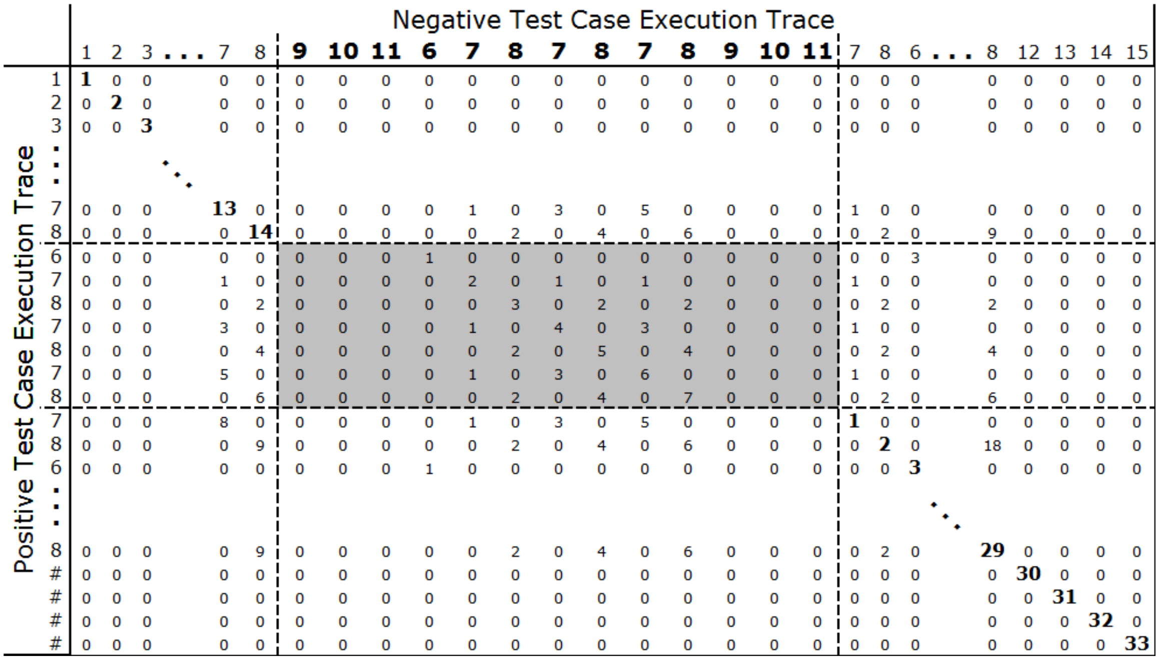
Table 4.1: Test Cases and Fitness Values for Running Example

ID	Input Array	Data Array	Fitness	Test Case Type
TC_1	[1, 2, 3, 4, 5]	[1, 2, 3, 4, 5]	1.00	positive
TC_2	[1, 2, 4, 3, 5]	[1, 2, 3, 3, 5]	0.80	negative
TC_3	[1, 2, 4, 5, 3]	[1, 2, 3, 3, 3]	0.60	negative
TC_4	[5, 2, 4, 1, 3]	[1, 1, 1, 1, 3]	0.40	negative
TC_5	[5, 3, 4, 2, 1]	[1, 1, 1, 1, 1]	0.20	negative

4.2.2. Trace Comparison Technique. The trace comparison technique in the FGFL system is conceptually based on the execution slice comparison technique presented by Agrawal et al. in [6], though there are a few notable variations in the FGFL version. The trace comparison technique compares the traces of each positive/negative test case pairing, attempting to find where in the execution the negative test case diverged from the positive. Rather than doing a strict set difference to find the negative execution dice, divergent execution paths in the negative traces are detected using a version of the dynamic programming solution to the Longest Common Substring (LCS) problem (where the strings being considered are the lists of line numbers executed when using the test cases) that has been modified to interpret the results differently.

Table 4.2 shows an example of the table generated by the modified LCS algorithm when comparing the traces for positive test case TC_1 and negative test case TC_3 in the running example. A value, x, in this table indicates that the last x lines in the traces at that point (i.e., the corresponding line in that row/column of the

Table 4.2: LCS Tabulation Used to Find the Divergent Path Between TC_1 and TC_3 in the Trace Comparison Technique



	Negative Test Case Execution Trace																											
Positive Test Case Execution Trace	1	2	3	...	7	8	**9**	**10**	**11**	**6**	**7**	**8**	**7**	**8**	**7**	**8**	**9**	**10**	**11**	7	8	6	...	8	12	13	14	15
1	**1**	0	0		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		0	0	0	0	0
2	0	**2**	0		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		0	0	0	0	0
3	0	0	**3**		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		0	0	0	0	0
⋮				⋱																								
7	0	0	0		**13**	0	0	0	0	0	1	0	3	0	5	0	0	0	0	1	0	0		0	0	0	0	0
8	0	0	0		0	**14**	0	0	0	0	0	2	0	4	0	6	0	0	0	0	2	0		9	0	0	0	0
6	0	0	0		0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	3		0	0	0	0	0
7	0	0	0		1	0	0	0	0	0	2	0	1	0	1	0	0	0	0	1	0	0		0	0	0	0	0
8	0	0	0		0	2	0	0	0	0	0	3	0	2	0	2	0	0	0	0	2	0		2	0	0	0	0
7	0	0	0		3	0	0	0	0	0	1	0	4	0	3	0	0	0	0	1	0	0		0	0	0	0	0
8	0	0	0		0	4	0	0	0	0	0	2	0	5	0	4	0	0	0	0	2	0		4	0	0	0	0
7	0	0	0		5	0	0	0	0	0	1	0	3	0	6	0	0	0	0	1	0	0		0	0	0	0	0
8	0	0	0		0	6	0	0	0	0	0	2	0	4	0	7	0	0	0	0	2	0		6	0	0	0	0
7	0	0	0		8	0	0	0	0	0	1	0	3	0	5	0	0	0	0	**1**	0	0		0	0	0	0	0
8	0	0	0		0	9	0	0	0	0	0	2	0	4	0	6	0	0	0	0	**2**	0		18	0	0	0	0
6	0	0	0		0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	**3**		0	0	0	0	0
⋮																							⋱					
8	0	0	0		0	9	0	0	0	0	0	2	0	4	0	6	0	0	0	0	2	0		**29**	0	0	0	0
#	0	0	0		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		0	**30**	0	0	0
#	0	0	0		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		0	0	**31**	0	0
#	0	0	0		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		0	0	0	**32**	0
#	0	0	0		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		0	0	0	0	**33**

traces) are matched. It is assumed that the executions will execute the same lines initially, which will be termed the path header, and will eventually sync back up after the divergent path to execute the same lines at the end, termed the path footer. The bolded values along the diagonal of Table 4.2 in the upper left hand segment represent the path header. The dashed edges at the end of the path header represent the start of the divergent path. The bolded values along the diagonal in the lower right hand segment represent the path footer. The dashed edges at the start of the path footer represent the end of the divergent path. The vertical dashed edges define the divergent path in the negative test case trace and the horizontal dashed edges show the lines that were executed in the positive test case until the two traces re-synced.

The table generated by the modified LCS algorithm contains a great deal of information regarding the execution traces. Methods to further exploit this information are being investigated and are discussed in Section 7.

The most notable weakness of Agrawal's technique (and as such the trace comparison technique in FGFL) is the assumption that any branching that occurs in the execution is relevant to the performance of the program. However, it is possible for a divergent path to exist between two positive test cases due to a benign (relative to program performance) branch in the execution. A benign branch that is distant (in terms of executed statements) from the actual error could affect the divergent path a great deal. If this problem is a possibility for a given program (it is not in the running example), a technique that can overcome this issue is to generate the execution dices between all positive test cases obtained; any lines that are present in one of the generated dices are marked as invalid boundaries for the divergent path. This technique can be further strengthened by generating additional positive test cases.

Suspicion is added evenly to line numbers found in the divergent paths of negative test cases. This technique is not useful in the event that there is no difference between positive and negative test cases, such as when the error is not a branch error.

Cases in which a strict set difference can be used to determine lines unique to the negative execution slice are still being investigated, as this would help to tighten the boundary for the bug location in some situations. Consider the running sorting example; in this example line 11 contains the error and as such the positive test case, TC_1, never executes lines 9-11. The divergent paths in the negative test case traces begin the first time that lines 9-11 are executed. Since these lines are in a branch statement that is contained in nested loops, both the branch statement (line 8) and the two loop statements (lines 6 and 7) are also included in the divergent path. Table 4.3 illustrates the differences between the lines that are included in the positive trace and the lines included in a divergent path. In the case of this error, it is safe to strictly remove any lines that are both in the positive trace and the divergent path from consideration, leaving only lines 9-11 as likely locations for the error. This results in a considerably tighter boundary on the potential bug location when compared to reporting all lines in the divergent path.

However, assume that the running example is adjusted to place the error on line 7 and line 11 is now correct. Say that the new error reduces the number of iterations of the inner loop. In this case the positive trace and the divergent paths will still include the lines indicated in Table 4.3, but if lines from the positive trace are removed from consideration in the divergent path, then the true bug location will not be reported. For this reason, the FGFL trace comparison technique does not remove lines in the positive trace from consideration in the divergent path.

Table 4.3: Comparison Between Traces for Running Example

Relevant Lines in Positive Trace	5	**6**	**7**	**8**			
Relevant Lines in Divergent Path		**6**	**7**	**8**	9	10	11

Votes are distributed in this technique evenly amongst the lines indicated by corresponding suspicion levels. There is not currently a mechanism implemented for this technique to apply only a portion of its votes. The binary nature of this technique makes the development of such a mechanism difficult, as a line was either executed by the negative test case or it was not. Possibilities for this addition are discussed in Section 7.

Table 4.4 shows the results of the trace comparison method on the running example.

4.2.3. Trend Based Line Suspicion Technique. The Trend Based Line Suspicion (TBLS) technique employs the fitness to determine the amount of suspicion to add to all lines in a given execution. Ideally, lines that are executed more frequently in low performance executions will tend to accumulate more suspicion than

Table 4.4: Vote Assignments for Trace Comparison Technique on Running Example

Line(s)	1-5	6	7	8	9	10	11	12-15
Votes	0	2.5	2.5	2.5	2.5	2.5	2.5	0

those executed by both positive and negative test cases. This technique is sensitive to both branch and loop related errors in software.

The TBLS technique is an enhanced version of the Tarantula technique presented by Jones et al. in [51, 50, 49]. As mentioned earlier, the TBLS technique exploits the fitness function in order to provide additional gradient the the Tarantula technique. In Tarantula, the H heuristic is used to calculate line suspicion based on the number of passed and failed test cases that execute the line. In the TBLS technique, the amount of suspicion applied to a line is a function of the fitness for the test case (i.e., how the program performed with the test case as input). The addition of the fitness function allows for a higher degree of precision in the application of suspicion to program lines.

The TBLS technique uses an equation that is a function of the fitness for an execution and calculates a Suspicion Adjustment Amount (SAA). Currently, it is assumed that the fitness function is normalized to fall in $[0, 1]$, which implies that the fitness function needs to be bounded; however, a modification is possible to allow unbounded fitness functions and is discussed in Section 7. The range of this function should be centered about the midpoint in the fitness range and should output positive SAA for low fitness values and a negative SAA for high fitness values. The linear equation to achieve these characteristics in the SAA is:

$$SAA = -2 \cdot fitness + 1 \qquad (4)$$

In this function, the SAA is calculated by inverting the fitness and scaling it to fall between [-1,1]. Preliminary experimentation has shown that this linear equation performs well for the current experimental problem set. Non-linear versions of this equation may be investigated in the future if experimental results indicate a need for this.

Each line in the program has an associated suspicion level (zero initially). The program is executed for a set number of test cases; an even sampling of high/maximum fitness and low/minimum fitness runs is ideal, which is achieved by the segmented test set. For each execution, the SAA is calculated and added to the suspicion levels of the lines in the trace for the run. Ideally, lines that are executed more (or even solely) by the low/minimum fitness executions should have a higher suspicion level, whereas lines executed by either both low and high or just high fitness executions should have a low suspicion.

Algorithm 3 shows how confidence values (and ultimately, votes) are calculated from line suspicions. This algorithm is conceptually similar to the H heuristic used in the Tarantula technique. The S array is filled using the traces and SAA values generated by test case executions, after which Algorithm 3 is employed. After the application of this algorithm, the values in the C array indicate the number of votes that will be applied to the corresponding lines. Lines 1-5 make all suspicion values positive, if necessary. Lines 6-9 are where confidence levels are calculated. The calculation on line 7 determines how many votes will be applied to line i based on the proportion of suspicion the line contributes to the sum of all suspicion obtained. However, this calculation does not take into account the confidence of the results, i.e., at this point a small suspicion level (relative to the maximum possible suspicion) can receive a large number of line votes if it represents a large portion of the total suspicion that was obtained. This discrepancy is accounted for on line 8, which adjusts the vote amount obtained on line 7 based on how much suspicion line i obtained relative to

Algorithm 3 Algorithm for Determining Confidence Values

```
1: if Min(S) < 0 then
2:     for i = 1 to LOC do
3:         $S[i] = S[i] + |Min(S)|$
4:     end for
5: end if
6: for i = 1 to LOC do
7:     $C[i] = \frac{S[i] \cdot LOC}{Sum(S)}$
8:     $C[i] = C[i] \cdot \frac{S[i] - S_{min}}{S_{max} - S_{min}}$
9: end for
```

- C: array that ultimately contains confidence values for each line
- S: array that contains the calculated line suspicion values
- S_{max} and S_{min}: the maximum and minimum suspicion values possible, respectively
- $Min()$: returns the minimum value in the argument array
- $Sum()$: returns the sum of the values in the argument array

the total suspicion possible.

The minimum possible suspicion value, S_{min}, is attained when a line is in the execution trace of every test case that generates a negative SAA and in no test case traces that generate a positive SAA. Similarly, the maximum possible suspicion value, S_{max} is attained when a line is in the execution trace of every test case that generates a positive SAA and in no test case traces that generate a negative SAA. The FGFL system determines these values as it generates test cases.

Table 4.5 contains the unaltered suspicion values, raw votes (i.e., not taking confidence into consideration), and confidence-based votes for the running example. The raw votes in the table are the values of the C array after line 7 of Algorithm 3 and the confidence-based votes are these values after line 8.

Table 4.5: Vote Assignments for Trend Based Line Suspicion Technique on Running Example

Line(s)	1-5	6	7	8	9	10	11	12-15
Suspicion	-1	-1	-1	-1	0	0	0	-1
Raw Votes	0	0	0	0	5	5	5	0
Confidence Votes	0	0	0	0	3.46	3.46	3.46	0

4.2.4. Fitness Monitor Technique. The fitness monitor technique tracks fitness values during program execution. For each test case, the technique calculates the difference between the fitness before and after each line, which shows the effect that the line has on the fitness value. The lines that cause a change in fitness (termed fluctuation lines) and the surrounding lines are of interest in order to monitor them as units rather than individual lines, which is further explained later in this section.

When using the fitness monitor technique, the user supplies information on the variable(s) in the source program that are necessary for fitness calculation (using constructs provided by the FGFL system). In the running example, the *data* array would be indicated (the input array is already known, as it was provided by the system). Also, the user would indicate that *data* is an array of five integers and that the lines during which *data* should be monitored are 5 through 15 (this allows proper scoping and/or variable initialization, if needed). Using the provided FGFL constructs this process is simply accomplished in four statements. When creating the instrumented version of the source program, the FGFL system creates member functions in the generated class that are used to output the indicated variables to a file along with line numbers to indicate the last two lines that were executed. Calls to these functions are then placed throughout the source code. After execution, the file containing this information is analyzed and fluctuation lines are noted by the system.

For each fluctuation line, fluctuation regions are determined. A fluctuation region is defined as a set of lines during which the fitness is changing. Each region begins with a single fluctuation line, and additional lines are added to the region until the fitness becomes stable surrounding the region (i.e., adding additional lines would not affect the overall fitness change across the region). It is possible for fluctuation regions to overlap, which allows emphasis to form on lines that are commonly considered to contribute to changes in fitness. The purpose of the fluctuation regions is to attempt to monitor the overall effect of code segments, rather than single lines.

To illustrate the benefit of fluctuation regions, consider the program in the running example. In this program, lines 9-11 perform a swap operation. The fitness function for this example applies a penalty when values present in the input are not present in the output (i.e., the *data* array). Even in a correct version of the running example program, after line 10 is executed an input value will not be in the data array, which will cause a decrease in fitness. Without fluctuation regions this decrease in fitness would cause the fitness monitor technique to report that line 10 likely contains an error. However, with fluctuation regions, lines 9-11 are considered a region, which overall increases fitness (still assuming that a correct version of the program is being used) and as such will not be indicated to contain an error.

Lines contained in fluctuation regions that overall cause a decrease in fitness receive increased suspicion (incremented by one). Currently, regions that cause an increase in fitness do not affect line suspicion values; however, the possibility of these lines receiving a decrease in suspicion is being considered in future experimentation. After all test cases have been executed, the suspicion values are converted to raw votes and then to confidence based votes using Algorithm 3. S_{max} is defined as the total number of fluctuation lines found in all test cases. This value is achieved if the line is contained in every fluctuation region that causes a decrease in fitness. S_{min} is zero in this technique, which is achieved if a line is either never included in a fluctuation

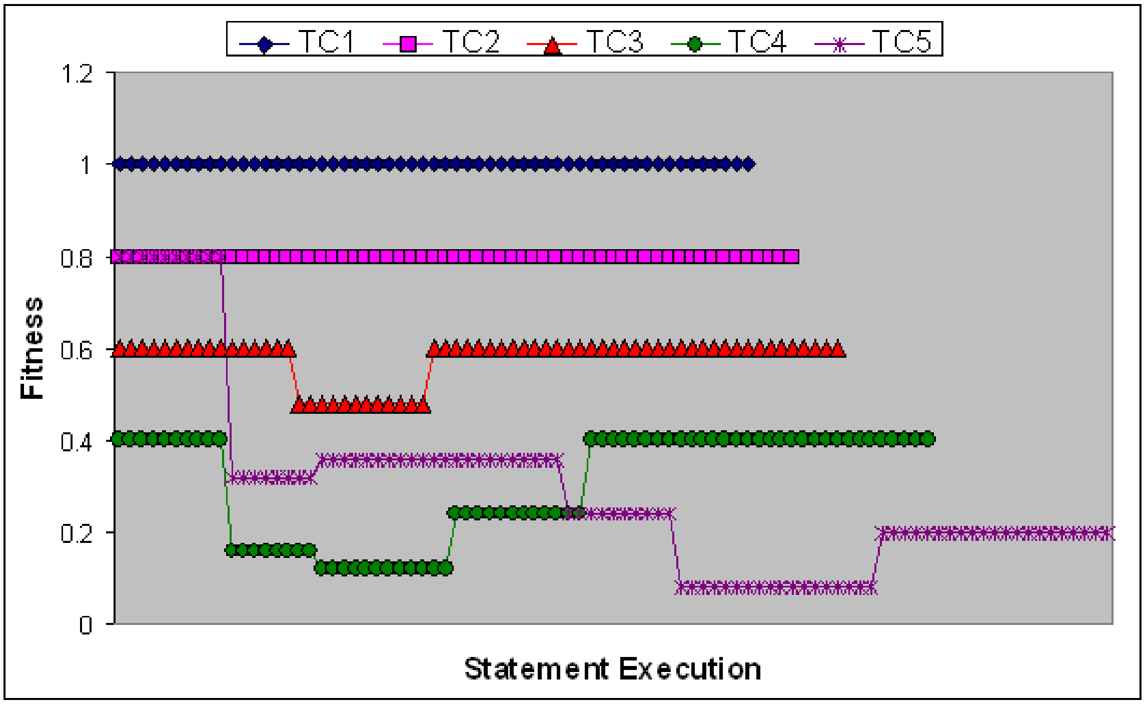


Figure 4.2: Fitness Plots for the Running Example

region or is only included in those that cause an increase in fitness.

Figure 4.2 shows the plots for fitness values during execution of the running example. For each test case, the only fluctuation line indicated is line 10. Each drop in fitness in Figure 4.2 is a point in the execution where line 10 was executed. The only exception is in TC_2. In this test case one value is out of place in the input array, which is reducing the fitness to 0.8; when line 10 executes, the last value is overwritten by the out of place value. At this point the array is in sorted order, but is being penalized for loosing a value from the input, resulting in a fitness of 0.8, and as such, no fluctuation in fitness is observed.

The fluctuation regions for TC_3, TC_4, and TC_5 are established around line 10, and include lines 9 and 11. Every time a fluctuation region has an overall reduction in fitness, the region includes these lines. Due to this, lines 9, 10, and 11 all have suspicion equal to S_{max} and receive all votes, as shown in Table 4.6.

Table 4.6: Vote Assignments for Run-Time Fitness Monitor Technique on Running Example

Line(s)	1-5	6	7	8	9	10	11	12-15
Votes	0	0	0	0	5	5	5	0

4.2.5. Result Combination.

After all test cases have been executed, the results are combined and presented. The overall votes for each line are calculated by taking the average votes applied to that line by each technique. Averaging was chosen over other aggregation techniques to counter false positives in the individual techniques that were not caught by the confidence based vote adjustments.

Table 4.7 shows the votes for all techniques and the overall votes applied to each line. Lines 9-11 received the most votes overall and as such would be reported as the most likely location for the error.

Table 4.7: Vote Assignments for Running Example

Line(s)	1-5	6	7	8	9	10	11	12-15
Trace Comp.	0	2.5	2.5	2.5	2.5	2.5	2.5	0
TBLS	0	0	0	0	3.46	3.46	3.46	0
Fitness Monitor	0	0	0	0	5	5	5	0
Overall	0	0.83	0.83	0.83	3.65	3.65	3.65	0

4.3. EXPERIMENTAL SETUP

The FGFL system has been tested on a statistical analysis program that reads in a set of values, performs a set of statistical calculations on the input values, sorts the input values, and then performs additional statistical calculations on the values that require the values to be sorted. Three versions of the program were used, each of which uses a different sorting technique: bubble sort, insertion sort, and selection sort. The programs are all between 46 and 50 lines long. The bugs for these programs were seeded in and are described in Table 4.8. These experiments are to function as both a proof of concept of the system as a whole and a test to expose strengths, weaknesses, and synergies within the system.

Table 4.8: Description of Buggy Programs used to Test the FGFL System

Algorithm	**ID**	**Bug Type**	**Bug Line(s)**
Bubble Sort	BBL_1	Incorrect Array Index Incorrect Array Index	26 27
Bubble Sort	BBL_2	'-' Used in Place of '+'	27
Insertion Sort	INS_1	Incorrect Index Variable	28
Insertion Sort	INS_2	Incorrect Loop Predicate	25
Selection Sort	SEL_1	Copy & Paste Error	27,28
Selection Sort	SEL_2	Incorrect Branch Predicate	26
Selection Sort	SEL_3	Incorrect Loop Predicate	25

For the presented experimentation a test case was defined as a set of seven values (this length was arbitrarily selected). 10 sets of randomly generated test cases were used to test each program. Each test case set consisted of 75 test cases, of which 15 were positive. The fitness range was divided up into four regions [0, 0.2), [0.2, 0.4), [0.4, 0.8), and [0.8, 1.0), with each region also consisting of 15 test cases.

The specifications for this program would contain assertions indicating that the first set of statistical results are correct, that the values were correctly sorted, and that the second set of statistical results are correct. As such, the fitness function for these programs would consist of components responsible for checking each set of assertions. For simplicity, the sorting portion of this fitness function has been focussed on, as that is the section of the program where all of the errors were seeded. The fitness function used in the running example (shown in Algorithm 1) was used in the presented experiments, with the modification to allow descending sorting as well as ascending under the rationale that difference between the two is just a matter of result interpretation and has no bearing on the correctness of the calculations.

4.4. RESULTS

The results obtained from the proof of concept experiments are summarized in Table 4.9. This table shows the average rank of the bug line(s) in the results for each possible FGFL configuration, where a line's rank is defined as the number of lines whose votes are greater than or equal to that of the line in question (if the bug spans multiple lines, then the minimum votes between the lines is used).

Table 4.9: Average Rank of Bug Line in Experiment Results

Prog.	Trace Comp.	TBLS	Fit. Mon.	Trace Comp. & TBLS	TBLS & Fit. Mon.	Trace Comp. & Fit. Mon.	All
BBL1	16.0	10.9	4.0	4.9	4.0	**3.0**	**3.0**
BBL2	16.0	4.9	2.0	4.9	**1.0**	2.0	**1.0**
INS1	17.0	45.9	**1.0**	22.4	**1.0**	**1.0**	1.0
INS2	17.0	45.0	**6.0**	20.9	12.1	5.3	10.6
SEL1	20.0	9.9	46.0	**4.5**	9.9	20.0	**4.5**
SEL2	**49.0**	**49.0**	**49.0**	**49.0**	**49.0**	**49.0**	**49.0**
SEL3	20.0	45.8	8.9	25.8	15.9	**8.3**	15.9

From these results it is apparent that the system has trouble identifying errors in control predicates (e.g., INS2, SEL2, and SEL3). This result, however, is not unexpected given the nature of the techniques in the FGFL prototype. Errors in control statements often require special consideration in fault localization techniques, and no such consideration was made in the FGFL prototype. None of the techniques were able to narrow down the location of the bug in SEL2. This error, in particular, caused a branch predicate to always evaluate to false. This result reveals the system's need for a technique focusing on statement reachability, which could implicate incorrect predicates as a result.

The single technique that seemed to perform the best was the run-time fitness monitor, with the exception of SEL1. In this program the error was in variable assignments which indirectly influence a drop in fitness; this indirect effect on the fitness caused the technique to report lines that were just a symptom of the true bug. In light of this result, a technique that combines the concept of the run-time fitness monitor with a dynamic slicing technique may be beneficial to the system; this is discussed further in Section 7.

In general, the addition of the trace comparison or run-time fitness monitor technique to another technique (including each other) appears to result in a lower average bug line rank for the technique. The addition of the TBLS technique to a technique, however, seemed to in many cases increase the average bug line rank for the technique. Inspection of the experiment result data indicates that this is largely due to benign branching in the statistical calculation portions of the program. This observation indicates that a similar mechanism as the one described in Section 4.2.2 (to account for benign branching) needs to be added to the TBLS technique.

On the non-control oriented bugs the system performed very well. With all techniques active, the system averaged a bug line rank of less than 5 (i.e., a 90% or more reduction in lines from the original source). In many cases the combination of all techniques outperformed the techniques operating independently, which indicates that the ensemble approach of the FGFL system is effective.

4.5. TARANTULA+

A second fitness guided fault localization system has been created, based on the trace comparison and TBLS techniques in the FGFL system. This system is called Tarantula+[1]. Tarantula+ removes many of the assumptions that Tarantula makes as well as improves and further automates the process originally proposed by Jones

Tarantula+ uses the results of trace comparison and TBLS techniques along with static analysis of the faulty program to discover lines that are most likely responsible for causing an error. The TBLS technique has been altered to allow test cases with more extreme fitness values alter line suspicion more dramatically. This is done by using Algorithm 4.

Algorithm 4 Suspicion Adjustment Amount Calculation used in Tarantula+

```
$SAA = -2 \cdot fitness + 1$
if $SAA < 0$ then
    $SAA = -(SAA^2)$
else
    $SAA = SAA^2$
end if
```

Additionally, the support was added to modified TBLS technique that allowed the user to indicate error branches in the program through the use of an error comment (the specific comment used is indicated to the Tarantula+ system via the system's configuration file). All lines in an identified error branch would automatically be

[1] The Tarantula+ system is the work of the author and his undergraduate mentee Alex Bertels. This section summarizes the preliminary studies performed on the system as presented in Alex's CS390 Undergraduate Research report and which are the basis for a conference paper in preparation.

given zero suspicion. This was done becuase positive test case execution traces would receive a negative SAA value, making them less suspicious than lines in error branches that were never executed, which could confuse the system results.

Tarantula+ has the capability of providing additional automated static analysis of the faulty program using the parse trees produced by the system's parser (the same parse used by the CASC system, described in Section 5.2.2.1). By using these trees to find various relationships between statements and code elements, additional suspicion can be applied to lines indirectly responsible for the incorrect outcome. For instance, the conditions within branch and loop statements that determine whether or not certain lines are executed indirectly affect the final outcome of the program. Using the trace comparison and TBLS techniques, branch and loop statements that contain the error can be ran by both positive and negative executions and would not receive as much suspicion as the lines within its scope. By allowing the loop or branch to contain as much suspicion as the most suspicious line that is in its scope, the statement can take responsibility for running lines that should not have been ran.

Another observation made by the system is that if an incorrect condition of an 'if' statement causes that condition to result in false when it should have been true, then the corresponding 'else if' and 'else' statements are given the option of executing lines within their scopes. This could result in the 'else if' or 'else' statements receiving suspicion for the 'if' or another 'else if' having wrong conditions. A solution to this problem was to take the sum of the suspicion to the corresponding 'if', 'else if', and 'else' statements and apply the sum to each of the statements.

The last observation currently made by the system addresses the idea that an incorrect assignment of a variable can affect any other statement that variable may be on. For each function, each variable will accumulate suspicion for each suspicious line that the variable appears in. These suspicion totals will be assigned to any line in which that variable is assigned a value or is incremented or decremented.

These three observations take advantage of the suspicion applied by the techniques and the relationships between statements to correctly distribute suspicion. This process is crucial to avoid misrepresenting the likeliness that a line contains the fault.

4.5.1. Preliminary Tarantula+ Results. Some preliminary experimentation has been performed using the Tarantula+. The results of these experiments are summarized in Table 4.10. In these experiments, the trace comparison technique was only used for analysis of single function programs or in programs where the divergent path was in one function. The current implementation of the technique does not provide meaningful results if the divergent path starts and ends in different functions. More work is currently being done to ensure that the technique only adds suspicion to those lines in the divergent path and not every line that falls between the start and end.

Two programs of the Siemens Suite, a widely-used set of programs for fault localization, were tested along with some additional programs. These programs are listed with a description of some of the errors tested, the techniques used, and how the error line placed in comparison to the other lines in the program. Programs such as *print_tokens2* and *replace*, which contain many branch and loop statements, allow for more variety in the execution traces. Having a unique execution trace is ideal for any fault localization system. The results for shorter programs like *remainder* and *triangleClassification* benefit less from the additional suspicion added by the analysis of the program and more from the direct results of techniques. Future work will include finding a balance between the analysis suspicion and technique suspicion based on the program length.

Table 4.10: Preliminary Tarantula+ Results

Program	**Error**	**Techniques Used**	**Error Rank** (Percentile)
print_tokens2	Condition: Incorrect Index	TBLS	97.70%
	Condition: Additional Condition	TBLS	95.70%
	Incorrect assignment	TBLS	96.90%
	Semicolon after if statement	TBLS	54.00%
replace	Condition: $>$ instead of $\leq$	TBLS	75.00%
	Incorrect assignment	TBLS	94.90%
tcas2	Condition: Missing Condition	TBLS + Trace Comparison	52.00%
		TBLS	0.00%
		Trace Comparison	52.00%
	Condition: || instead of &&	TBLS + Trace Comparison	94.80%
		TBLS	94.80%
		Trace Comparison	94.80%
	Incorrect Assignment	TBLS + Trace Comparison	98.30%
		TBLS	98.30%
		Trace Comparison	98.30%
remainder	Incorrect Assignment	TBLS + Trace Comparison	94.10%
	Incorrect Operator	TBLS+ Trace Comparison	30.00%
triangleClass.	Condition: && instead of ||	TBLS + Trace Comparison	47.00%
		TBLS	47.00%
		Trace Comparison	54.00%

5. COEVOLUTIONARY AUTOMATED SOFTWARE CORRECTION

For a given program, testing, locating the errors identified, and correcting those errors is a critical, yet expensive process. The National Institute of Standards and Technology has estimated that inadequate software testing tools and methods cost the U.S. economy alone between $22.2 and $59.5 billion a year [92]. Detecting and fixing errors is typically a difficult, time-consuming, and manual process. The number of software bugs typically exceeds the resources available to address them. In many cases, mature software projects are forced to ship with both known and unknown errors for lack of development resources to deal with every defect. Clearly, efficient and effective software testing and correction methods need to be developed [37]. A variety of challenges need to be overcome in order to achieve this, some of which are addressed in this article.

One of the major challenges is how to explore the large related spaces of test case and correction possibilities. The space of all possible programs is theoretically infinite (though limited in a practical sense by memory size). If a buggy program is viewed as a single point in this space, it is reasonable to assume that the correct version of the program will likely be near that point in terms of modifications, under the rationale that programmers do not create programs at random [29]. Even with this assumption, it is still impractical in non-trivial problems to exhaustively explore the space of all programs that are near the source program.

To explore these large search spaces, the efficient solution is Search Based Software Engineering (SBSE) [40], which is the application of artificial intelligence search techniques to solve problems in software engineering, including software testing and correction [39]. Most existing work in SBSE focuses on software testing [39], typically test case generation to maximize the coverage of possible scenarios. In general, the criteria to generate test cases are based on the use of metrics to ensure the coverage of the language meta-model, without considering explicitly the program specifications [30, 95, 113]. In other works, multi-objective optimization is used to minimize the number of test cases and maximize the coverage of the language meta-model [38, 61].

Automated software correction has received much less attention in the SBSE community [37]. The Coevolutionary Automated Software Correction (CASC) system [102, 103, 105, 106] performs automated program testing, correction, and verification through the use of EAs. CASC operates at the source code level (as opposed to lower level representations, such as assembly code), rating performance based on compilation and execution results. Programs and test cases are evolved in tandem, exploiting the competitive relationship naturally present between them. Similar to CASC, the works of Ackling [1], Arcuri [10], and Weimer [100] are also focused on automated software correction at the source code level through the use of EAs. These works differ from CASC in that they all rely on the provision of an a priori generated set of test cases for the program being corrected, whereas CASC relies on a test case definition to generate and evolve test cases dynamically at run-time. The systems presented by Ackling and Weimer support a very limited set of possible code modifications and, as such, are limited in possible applications. Arcuri's JAFF system performs code modifications in a manner similar to the CASC system, namely through the use of the concepts of Strongly Typed Genetic Programming [72] to define compatible modifications. Unfortunately, the constraints used in JAFF to define compatible modifications are not explicitly stated, making direct comparison with the system impossible, as these constraints directly define the problem space being navigated by the system.

This paper describes a revised and improved version of CASC, including the following major improvements:

- The CASC software correction process has been augmented with candidate solution identification and a verification cycle that performs focused testing on the candidate solution using a highly exploratory EA.
- The system exploits additional code element relationships to allow for automatic state space reduction, resulting in more intelligent program modification.
- The operators used for program modification during the correction process have been improved to focus on more promising search paths and better avoid invalid ones.
- A stagnation detection system has been added to determine if the correction search process has become stuck in a local optima and, if so, resets the search.

This paper is further organized as follows. Section 5.1 summarizes related work on evolutionary software correction. Section 5.2 details the design of CASC. The experiments run are detailed in Section 5.3; results and discussion of these experiments are presented in Section 5.3.5.

5.1. BACKGROUND AND RELATED WORK

5.1.1. Background. All optimization in the CASC system is driven by EAs. In general, an EA is a stochastic, population-based optimization algorithm, inspired by biological evolution [42]. A typical EA generation consists of reproduction (i.e., the creation of new individuals), evaluation (i.e., rating individual performance), and survival selection (i.e., removal of low performing individuals). These processes are described in detail for CASC in Section 5.2. The task of automated software testing, correction, and verification is quite complex, with many unique characteristics. Accordingly, CASC incorporates some advanced/specialized EA concepts to more effectively address various aspects of these tasks.

In terms of testing, a good test case is one that causes the program under test to perform incorrectly. Conversely, a good program is one that performs as specified for all test cases; however, in most cases it is infeasible to run a program against all test cases and so, in these cases, a program is considered good if it performs as specified for all sampled test cases. CASC exploits the inherent competitive relationship between a program and its associated test cases using a competitive COEvolutionary Algorithm (COEA) during the correction process. Competitive COEAs are a type of EA specifically developed to solve this type of problem of competing, interdependent, evolving populations, also known as the parasite-host relationship [41, 85, 86]. In CASC, the test cases evolve to better find flaws in the programs, while the programs evolve to better behave to specification when exposed to the test cases. The competition between the evolutionary cycles ideally results in an evolutionary arms race, promoting the escalation of the quality of individuals in the evolving populations. A typical competitive two-population COEA cycle is shown in Figure 5.1; the CASC COEA operates in a manner very similar to this.

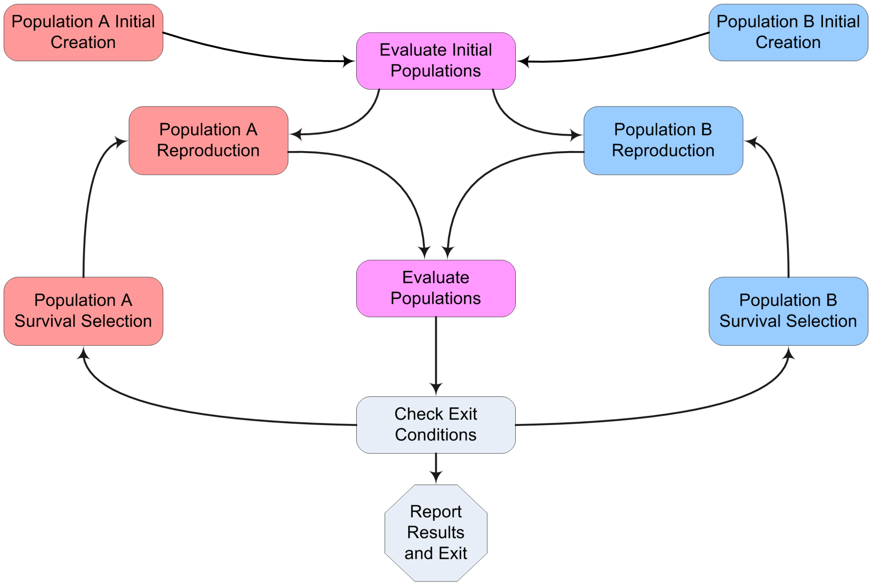

Population A Initial Creation
Evaluate Initial Populations
Population B Initial Creation
Population A Reproduction
Population B Reproduction
Population A Survival Selection
Evaluate Populations
Population B Survival Selection
Check Exit Conditions
Report Results and Exit

Figure 5.1: Typical 2-Population Coevolutionary Cycle

Adamopoulos et al. were one of the first to publish the concept of coevolving programs and test cases [2]. Their goal was to generate more effective program test cases. To achieve this goal a program was seeded with various errors resulting in a set of modified programs, termed mutants. Test cases were then developed whose goal was to identify the mutants. In order to generate more effective test cases, both the test cases and mutants were evolved according to their performance, creating a coevolutionary system.

The representation and specific evolutionary operators for the test cases are obviously problem specific, making the integration of these details into a generalized system a difficult design problem. In CASC, this problem is addressed through the use of the polymorphic design paradigm [14], as discussed in more detail in Section 5.2.1. Programs are represented as parse trees in CASC, and are evolved using the concepts of GP [59]. GP is a type of EA focused on the evolution of individuals being represented by trees (as opposed to the more typical array representation). Accordingly, there are evolutionary operators defined to specifically work with this representation. As program parse trees have a number of unique characteristics, a specialized representation language and set of GP operators have been defined for CASC, as detailed in Section 5.2.2.1 and Section 5.2.4.4, respectively.

CASC performs automated black-box functional testing of software. Initially, the testing is non-directed in terms of functionality. The system non-deterministically explores the functionality of the software, comparing actual output to expected output. If the expected output is not achieved, then faulty functionality has been identified. The input that created the erroneous output is then exploited (using an EA) to create other inputs that also demonstrate the error.

Structural testing has also been added to CASC; the benefits of which are investigated in the presented study. For every program that is detected as a candidate solution, a test case set is created that maximizes the decision coverage in the candidate solution. Just like in non-automated applications, the high coverage test case set is intended to improve the testing entity's ability to identify errors quickly and to improve confidence in the resulting verified solution.

CASC supports both Single- and Multi-Objective OPtimization (SOOP and MOOP, respectively), where the objectives being optimized are derived from the specifications for the software being tested and corrected. MOOP is a style of optimization developed to address the difficulties present in optimization problems with multiple, conflicting objectives. If SOOP is used for these problems, then objectives are often combined into a single objective using a weighted sum. Determining the optimal method of objective composition is a difficult task, often resulting in trade-offs between the objectives. MOOP simultaneously optimizes the objectives as independent functions. As such, MOOP is recommended in the CASC system for correction of programs with more than one specification. While MOOP was developed to address conflicting objectives, it is still beneficial to use in CASC even when objectives are not conflicting, as separate objective scores provide increased granularity for comparing individual performance.

CASC employs a NSGA-II [28] style MOOP system. NSGA-II organizes individuals into *fronts* based on dominance. Individual A dominates individual B, denoted $A \prec B$, if A performs as good as B on all objectives and outperforms B on at least one objective. The NSGA-II front creation algorithm is shown in Algorithm 5. In this algorithm, fronts are being calculated for population P, where S_a represents the set of individuals dominated by individual a, n_b represents the number of individuals that dominate individual b, and F_c is the set of individuals on front c. The individuals on front i are not dominated by any individuals on fronts i or greater.

The result of any search using MOOP is the non-dominated front at the conclusion of the search. This is typically viewed as a benefit of MOOP under the rationale that the user is given a variety of solutions to choose from after the search completes.

Algorithm 5 NSGA-II Front Generation Algorithm

```
for each $p \in P$ do
    for each $q \in P$ do
        if $p \prec q$ then
            $S_p = S_p \cup \{q\}$
        else if $q \prec p$ then
            $n_p = n_p + 1$
        end if
    end for
    if $n_p = 0$ then
        $F_1 = F_1 \cup \{p\}$
    end if
end for
$i = 1$
while $F_i \neq \emptyset$ do
    $H = \emptyset$
    for each $p \in F_i$ do
        for each $q \in S_p$ do
            $n_q = n_q - 1$
            if $n_q = 0$ then
                $H = H \cup \{q\}$
            end if
        end for
    end for
    $i = i + 1$
    $F_i = H$
end while
```

However, it can be reasonably expected that the user of a automated program correction system wants a single, corrected program as the result, rather than a set of programs. Previous versions of the CASC system simply reported this as the result of the run, putting the burden of identifying the solution to use from the resulting set onto the user. The presented addition of candidate solution identification and verification to CASC remedies this issue, with a single solution presented as the result of all runs that are successful.

The search space being navigated by the CASC system is infinite in theory, only being limited by memory size in practice. In general, the CASC system deals with this by limiting the search to paths that are more likely to yield a solution. This is accomplished by analyzing the code being corrected and limiting the modifications based on the information obtained. The revised version of the CASC system presented here improves on this technique by increasing the amount of information extracted from the code as well as introducing generic biases on modifications more likely to yield solution programs.

A common risk of all EA based approaches is the possibility that the search can be stuck in local optima in the search space. Previous versions of the CASC system exhibited this behavior in some cases. In response to this, the presented version detects when search stagnation has occurred, resetting the search when detected.

5.1.2. Related Work. There are several studies that have recently focused on test case generation and bugs in software using different techniques. These techniques range from fully automatic to guided manual inspection. However, there are only a very few contributions focused on taking specifications into consideration when generating test cases and correcting software errors.

5.1.2.1. Test case generation. SBST is the most active research area of SBSE [39, 66]. Accordingly, there is an abundance of proposed approaches for test case generation, e.g., [17, 18, 19, 34, 45, 88]. For detailed discussions of the work in the SBST area, see the surveys published by McMinn [65, 66] and Nie [74].

EAs are the most popular search-based algorithms for test case generation [7]. Different criteria have been used to generate test cases, such as: the coverage of loops [30], the existence of internal states [113], and the presence of possible exceptions [95]. An important objective in addition to meta-model coverage is minimizing the number of test cases. The ideal scenario is to reduce the number of test cases without any loss of coverage. This is the main motivation for the few works to use multi-objective techniques to find the best compromise between these conflicting objectives [38, 61]. In general, the aim of branch coverage has been to find test cases which traverse a specific branch. [38] presents the first multi-objective approach to branch coverage which adds the conflicting objective of minimizing the consumption of dynamic memory. [61] employs different search based testing techniques to produce fewer test cases without loss of coverage. CASC differs from these works in its use of program specification based objectives and, as such, test cases are generated with the goal of demonstrating a specific bug in the program (i.e., the bug that is being corrected) rather than maximizing meta-model coverage. In addition, existing multi-objective works only address test case generation and not program correction.

5.1.2.2. Automated program repair. Current research in the area of automated program repair can be divided into three categories:

1. Correction of specific program errors [16, 75],
2. Program correction through modification of machine code [76, 87],
3. Program correction through modification of source code [106, 1, 10, 100]

Many of these works use EAs in the correction process [1, 10, 16, 76, 87, 100]. The search process in an EA is guided by a fitness function that rates an individual's performance on the problem in question. The fitness function in CASC is assumed to be derived from the program specifications; so, for a given program-test case pairing, a run is scored based on how close to specification the program ran. Similar requirements are made in work related to CASC, in which sets of test cases are required that are known to demonstrate the bug in question [1, 10, 100]; these sets are generated through the use of an oracle[2], which enforces the program specifications. The derivation of a fitness function from specification can be a difficult task, most obviously accomplished through formal specifications [104, 12]. This may seem unfortunate, given that formal software specifications are rarely used in industry [77].

[2] In software engineering, an oracle is a mechanism to determine whether a program passes or fails a specific test case [69], often in the form of a human expert's judgment.

However, as SBSE research progresses, the benefits of the automated testing, repair, and programming tools that exploit formal specifications will eventually outweigh the cost of developing formal program specifications.

Works in the first category are focused on the correction of specific software errors, detected by unique signatures for when the bug is occurring. The work of Bradbury et al. [16] is focused on the correction of concurrency errors through the use of EAs. In [75] Novark et al. show an approach for automated correction of memory allocation errors. The work of Sidiroglou et al. [89] describes a method for automated server patching when zero-day exploits are detected.

Works in the second category focus on evolving programs written in machine code [76, 87], under the rationale that machine code has fewer syntactic restrictions than higher level languages (and as such is generally more robust to naive modification) and that working at the machine code level increases the generality of the system (i.e., applicable to any source language that can be compiled to machine code). Orlov et al. evolve machine code compiled from Java programs [76]. Valid programs were produced by a crossover mechanism that performs a validation check on offspring produced, and repeats the crossover process if the offspring are invalid. Schulte, Forrest and Weimer present a similar approach, except that machine code is evolved at the instruction level (i.e., instruction operands are not modified) rather than at the code element level [87]. In this approach, code is never generated, only modified by adding, deleting and moving instructions. This implies that in order to fix an error, the instruction(s) needed for the repair must already be present in the source program to be corrected. A drawback of the approach of evolving programs at the machine code level is that when translating a language from a high level language to machine code, a great deal of context information that is relatively easy to extract at high level becomes difficult (or even impossible) to extract at the machine code level (e.g., type information, qualifiers, code intention). The loss of this information opens up a number of ultimately invalid search paths in the program space that can be trivially removed from consideration when working with a high level language.

The third category consists of work focused on automated repair through modifications made to the programs at the source code level. The three competing approaches in this category all rely on a set of test cases being provided that demonstrate the bug in question, whereas CASC has no such requirement.

Ackling et al. present a system prototype that, instead of evolving programs, evolves sets of modifications (i.e., patches) to the source program [1]; trivially allowing limitation of the number of modifications made in a patch. Promising results are presented for small programs, with a very limited set of modifications currently supported by the system. A limitation of the presented approach is that it is assumed that the error present can be corrected by replacing a code element with another of the same arity and precedence, as this is how modifications are defined in the system. This essentially locks the structure of program during evolution. For the supported set of modifications and the programs considered in the cited paper, this limitation is of little consequence, as the operator code elements represented are all of the same precedence and arity and the bugs considered can be corrected given these limitations. Future versions of this approach will need to account for this limitation in order to be a general purpose program correction system.

Weimer et al. present a system focused on correction of off the shelf legacy C programs [100]. A high level view of this system is shown in Figure 5.2. Program analysis methods are used to localize modifications to areas suspected to contain the error. The system makes modifications to source code at the statement level. This restriction, while leading to superior scalability, greatly limits the scope of errors correctable by the system, essentially making the system specialized to situations where the code statement(s) needed to correct an error can be assumed to be present in the source program.

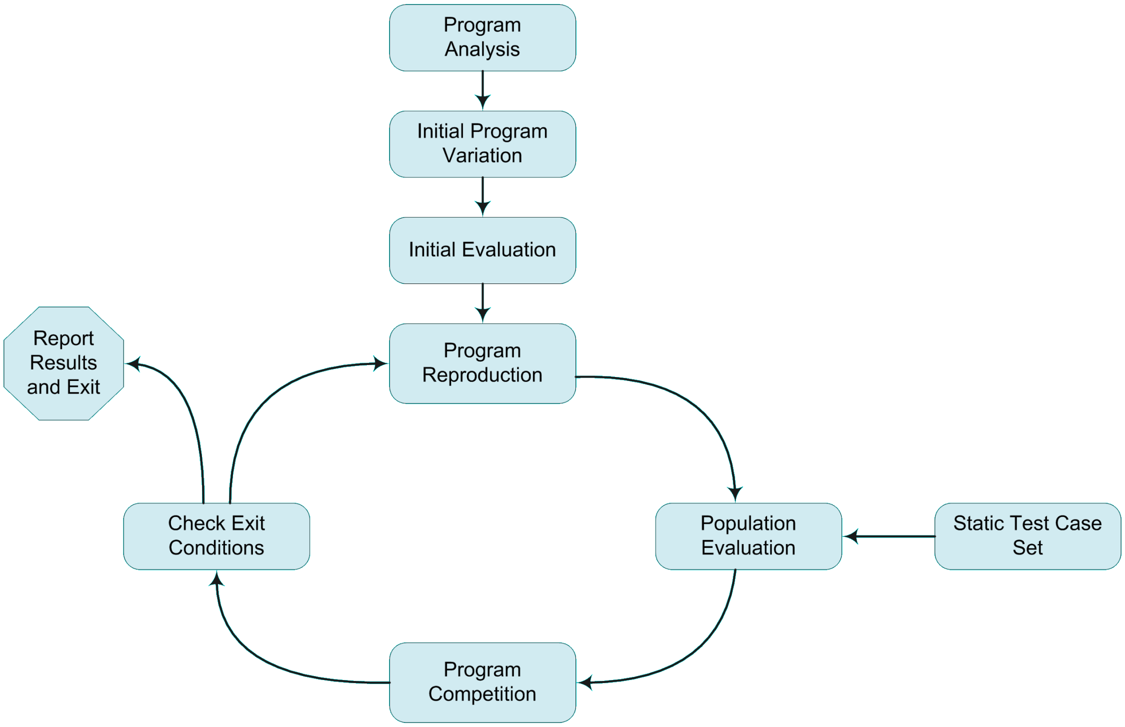


Figure 5.2: Weimer's Software Correction System

Arcuri's JAFF system [10] is the most closely related work to CASC. It focuses on the correction of Java source code and supports an explicitly defined subset of the language, with multiple hard coded aspects (e.g., specific variable names and numeric constants are added for each problem). Earlier versions of the system supported turn-based coevolution (i.e., repeatedly, the program population evolved for x generations, then the test case population was evolved for y generations) [8, 9, 11], while the most current version only performs program evolution [10]. A high level view of the current JAFF system is shown in Figure 5.3

One of the major distinguishing characteristics of Arcuri's system is its ability to automatically generate a fitness function for a problem given a set of formal specifications. This is accomplished using the scoring system proposed by Tracey et al. [93, 94]. This is a strong benefit as generating a fitness function for a problem often is a bottleneck, particularly for non-expert practitioners. However, composing

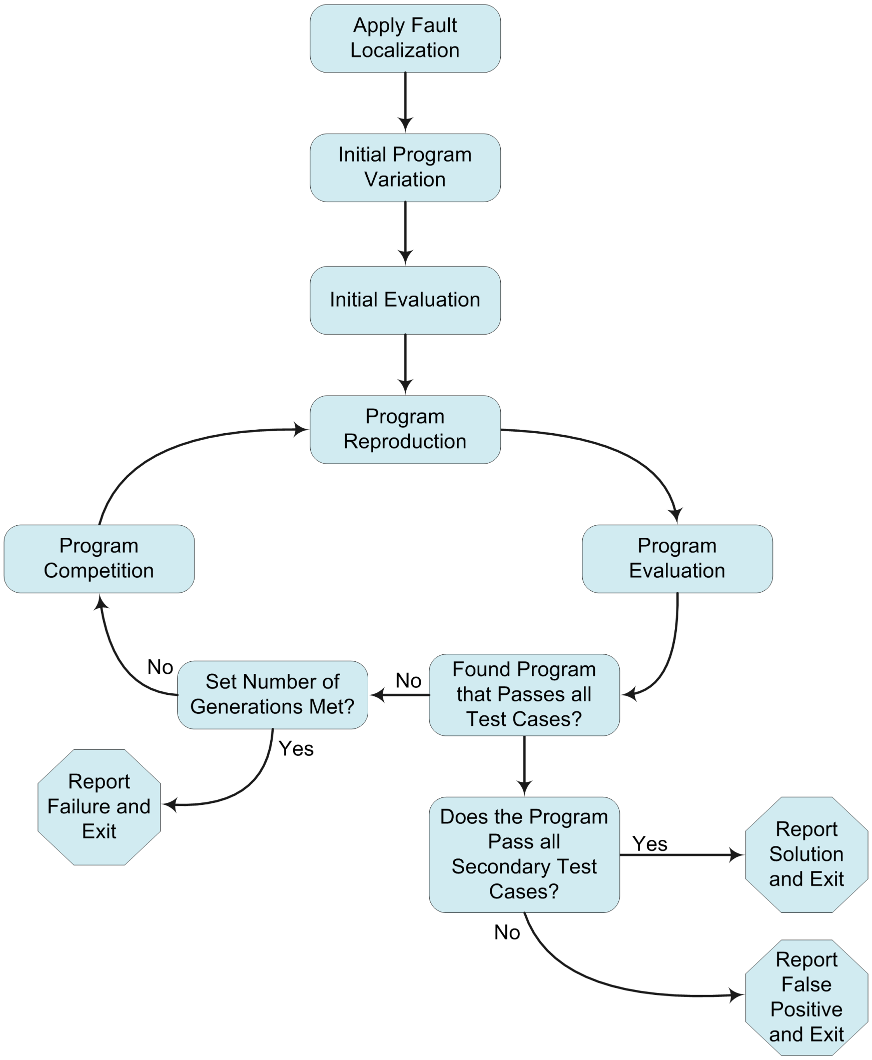

Apply Fault Localization
Initial Program Variation
Initial Evaluation
Program Reproduction
Program Competition
Program Evaluation
No
Set Number of Generations Met?
No
Found Program that Passes all Test Cases?
Yes
Report Failure and Exit
Does the Program Pass all Secondary Test Cases?
Yes
Report Solution and Exit
No
Report False Positive and Exit

Figure 5.3: Arcuri's JAFF System

the various objectives of a program into a single function is often a difficult process, resulting in trade offs between the objectives. These trade-offs are often decided upon by taking into account the specifics of the objectives. Having the fitness function generated automatically means that these decisions have been made ahead of time and hard coded into the system. Clearly, these hard-coded decisions cannot expected to be optimal for all problems. In contrast, CASC requires the fitness function to be provided to the system (described further in Section 5.2.1), which avoids the drawbacks of hard-coded decisions and offers greater user flexibility. Both the presented guide for fitness function design (Section 3) and the support of MOOP are intended to mitigate the burden of fitness function design on CASC users.

In [9] Arcuri presents a variety of applications for his system. In addition to automated program correction, Arcuri's system was applied to automatic program refinement and automatic improvement of execution time. When performing automatic program refinement[3] the system uses the problem specifications to create a fitness function (as described previously); however, in this case there is no source program to start from. The system's goal in this case is to evolve a program that satisfies the problem specifications provided. The problem of evolving a program from scratch is a very difficult one primarily due to the infinite problem space. For this reason, the problems that were used for experimentation were very basic problems (i.e., triangle classification, swap, order (a conditional swap), sorting, and median calculation). Even for these simple programs, Arcuri's system had difficulty consistently generating programs that satisfied the given problem specifications.

Arcuri's system was also applied to automatic improvement of program execution time. In this application the system's goal was to improve the execution time of a program while preserving the semantics of the program. In this application a program's fitness value is determined based on both semantic performance and non-functional performance (i.e., CPU cycles used during execution). The results presented demonstrate that the system was able to improve execution time in many of the problems considered.

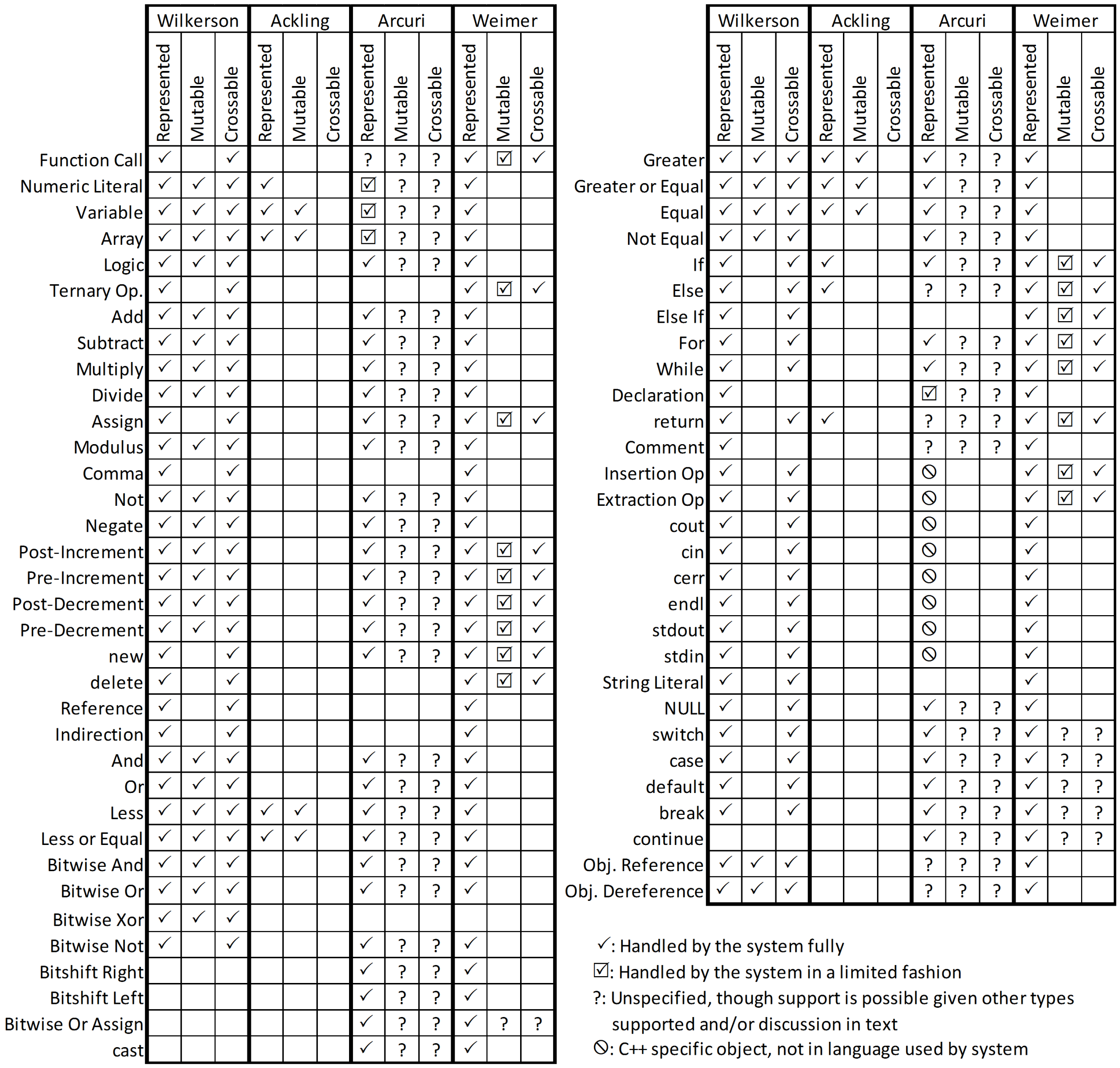
A majority of the related approaches discussed use some form of test case pass/fail fitness function, whereas CASC's fitness function is based on software specifications. The additional gradient and sensitivity provided through the use of specification based fitness is invaluable to the search process, even when there are not many edits needed to correct the problem. During search based software correction,

[3] A more intuitive name for this application would be automated program creation.

the program is being non-deterministically modified in search of performance improving modifications. The majority of these modifications can typically be expected to reduce the overall performance of the program. Pass/fail based fitness functions can be expected to struggle with this, as the degree of the failure is not reflected in this metric. For example, assume that functionality A contains an error in the source program. Now assume that a program is created that also introduces an error into functionality B. A test case that demonstrates both functionality A and B will not indicate the introduction of this error in a pass/fail based fitness function, only that the program failed. However, in a specification based fitness function, it can reasonably be assumed that all of the program functionality was represented in the specifications, and, accordingly, is accounted for in the fitness function. And so, when using a specification based fitness function, the program with an error in functionality B would be trivially rejected for introducing a decrease in performance; this can not be expected to always be the case when using a pass/fail based fitness function. The added gradient and sensitivity of a specification based fitness function not only allows for more precise guidance towards improving performance, but also more effectively ensures that the search does not get further away from a solution.

5.1.2.3. Comparison of related approaches. Investigation of the impact that non-trivial representation languages have on automated program correction systems is an important study for automated program correction. While a more comprehensive representation language theoretically allows more bugs to be handled by the system, the resulting search space increases proportionally with language size, which potentially makes the search impractical. CASC handles this increase in problem space by examining the source program and recording information about the code elements present and their relationships. This information is then exploited during program modification, essentially resulting in automatic state space reduction.

Figure 5.4 contains a summary of the representation languages used and code modifications supported by the approaches working at the source code level[4]. The problem space being navigated by these systems is a function of the size of the representation language used and the code modifications supported.



	Wilkerson			Ackling			Arcuri			Weimer		
	Represented	Mutable	Crossable	Represented	Mutable	Crossable	Represented	Mutable	Crossable	Represented	Mutable	Crossable
Function Call	✓		✓				?	?	?	✓	☑	✓
Numeric Literal	✓	✓	✓	✓			☑	?	?	✓		
Variable	✓	✓	✓	✓	✓		☑	?	?	✓		
Array	✓	✓	✓	✓	✓		☑	?	?	✓		
Logic	✓	✓	✓				✓	?	?	✓		
Ternary Op.	✓		✓							✓	☑	✓
Add	✓	✓	✓				✓	?	?	✓		
Subtract	✓	✓	✓				✓	?	?	✓		
Multiply	✓	✓	✓				✓	?	?	✓		
Divide	✓	✓	✓				✓	?	?	✓		
Assign	✓		✓				✓	?	?	✓	☑	✓
Modulus	✓	✓	✓				✓	?	?	✓		
Comma	✓		✓							✓		
Not	✓	✓	✓				✓	?	?	✓		
Negate	✓	✓	✓				✓	?	?	✓		
Post-Increment	✓	✓	✓				✓	?	?	✓	☑	✓
Pre-Increment	✓	✓	✓				✓	?	?	✓	☑	✓
Post-Decrement	✓	✓	✓				✓	?	?	✓	☑	✓
Pre-Decrement	✓	✓	✓				✓	?	?	✓	☑	✓
new	✓		✓				✓	?	?	✓	☑	✓
delete	✓		✓							✓	☑	✓
Reference	✓		✓							✓		
Indirection	✓		✓							✓		
And	✓	✓	✓				✓	?	?	✓		
Or	✓	✓	✓				✓	?	?	✓		
Less	✓	✓	✓	✓	✓		✓	?	?	✓		
Less or Equal	✓	✓	✓	✓	✓		✓	?	?	✓		
Bitwise And	✓	✓	✓				✓	?	?	✓		
Bitwise Or	✓	✓	✓				✓	?	?	✓		
Bitwise Xor	✓	✓	✓									
Bitwise Not	✓		✓				✓	?	?	✓		
Bitshift Right							✓	?	?	✓		
Bitshift Left							✓	?	?	✓		
Bitwise Or Assign							✓	?	?	✓	?	?
cast							✓	?	?	✓		

	Wilkerson			Ackling			Arcuri			Weimer		
	Represented	Mutable	Crossable	Represented	Mutable	Crossable	Represented	Mutable	Crossable	Represented	Mutable	Crossable
Greater	✓	✓	✓	✓	✓		✓	?	?	✓		
Greater or Equal	✓	✓	✓	✓	✓		✓	?	?	✓		
Equal	✓	✓	✓	✓	✓		✓	?	?	✓		
Not Equal	✓	✓	✓				✓	?	?	✓		
If	✓		✓	✓			✓	?	?	✓	☑	✓
Else	✓		✓	✓			?	?	?	✓	☑	✓
Else If	✓		✓							✓	☑	✓
For	✓		✓				✓	?	?	✓	☑	✓
While	✓		✓				✓	?	?	✓	☑	✓
Declaration	✓						☑	?	?	✓		
return	✓		✓	✓			?	?	?	✓	☑	✓
Comment	✓						?	?	?	✓		
Insertion Op	✓		✓				⊘			✓	☑	✓
Extraction Op	✓		✓				⊘			✓	☑	✓
cout	✓		✓				⊘			✓		
cin	✓		✓				⊘			✓		
cerr	✓		✓				⊘			✓		
endl	✓		✓				⊘			✓		
stdout	✓		✓				⊘			✓		
stdin	✓		✓				⊘			✓		
String Literal	✓		✓							✓		
NULL	✓		✓				✓	?	?	✓		
switch	✓		✓				✓	?	?	✓	?	?
case	✓		✓				✓	?	?	✓	?	?
default	✓		✓				✓	?	?	✓	?	?
break	✓		✓				✓	?	?	✓	?	?
continue							✓	?	?	✓	?	?
Obj. Reference	✓	✓	✓				?	?	?	✓		
Obj. Dereference	✓	✓	✓				?	?	?	✓		

✓: Handled by the system fully
☑: Handled by the system in a limited fashion
?: Unspecified, though support is possible given other types supported and/or discussion in text
⊘: C++ specific object, not in language used by system

Figure 5.4: Summary of Representation Languages and Supported Code Modifications for Systems Performing Correction at the Source Code Level

[4] This summary was compiled to the best knowledge of the authors; in some cases information had to be inferred from the published works for the considered approaches.

As can be seen, the approach presented by Ackling et al. is still in the early stages of development, currently supporting a small representation language and modification set. While promising results were presented on a limited problem set, the system in its current state lacks the representational richness necessary to be a general purpose software correction system, such as CASC. This precludes it from being compared using the problem set presented in this article, while giving it an unfair advantage on the published limited problem set when compared to a general purpose software correction system.

The approach presented by Weimer et al. appears to represent all code elements handled by the other existing approaches, based on the discussion in the published work [100]. In this approach, crossover is performed by exchanging full statements (i.e., full lines of code) between individuals. Accordingly, all code elements that serve as the root operation for a statement are indicated as crossable in the summary, with the exception of the few elements explicitly stated as not being modified in the cited work. Mutation is defined in this approach as the insertion, deletion, or swap of statements within a program. In the summary, code elements that serve as the root operation for a statement are indicated as supporting mutation in a limited fashion under the rationale that the system is operating at the statement level whereas the other approaches operate at the code element level. This approach has essentially been specialized for errors that can be corrected through statement duplication, moving, and/or deletion in the source program. As with all specialized approaches, it can be expected that this approach will excel when dealing with the problem it was designed for, but will struggle with other problems; in fact, this approach cannot correct programs that do not already have the statements needed to correct the bug(s) present. As such, comparing this specialized approach with the CASC system would not be appropriate, as CASC is a general fault correction system.

Arcuri's JAFF system supports a rich representation language. The primary limitation to the language used is that program specific content (e.g., variable names, numeric constants) must be added to the language manually for each program considered [10]. The JAFF system uses a constraint system to guide the code modification process; unfortunately, no details are given on the actual constraints used in the cited work. Because of this, the modifications supported for the represented code elements are left as unspecified in the summary. It is not expected that all of the listed unspecified modifications are supported, given the operators described in the cited work; however, they have not been stated as being disallowed, and so must be considered possible. Without this information, comparison with the JAFF system is difficult, as the size of the problem space cannot be determined. Regardless, the JAFF system is the only general fault correction system in the related approaches, and so a subset of the experiments conducted with JAFF have been reproduced using the CASC system, with comparison results reported in Section 5.3.5.

CASC has been augmented with a verification cycle to help ensure that solutions presented by the system are actual solutions. This process is initialized with a high coverage test case set, in order to both increase the efficiency with which false positive solutions are identified as well as improve confidence in resultant solutions. Related approaches mostly rely on the user of the system to identify and reject false positive solutions. The JAFF system employs two static test case sets that are provided to the system; one for the correction process and the other is used to verify candidate solutions. As stated previously, the effectiveness of static test case sets is dependent on the generating entity's ability to create a comprehensive test case set.

5.2. DESIGN

5.2.1. Approach Overview. As is shown in the high level CASC flow chart in Figure 5.5, CASC is organized into three conceptual modules: *System Initializa-*

tion, *Testing and Verification*, and *Testing and Correction*. The *System Initialization* module is passed through one time at the start of a run and is not reentered. This module performs setup tasks needed by the other modules.

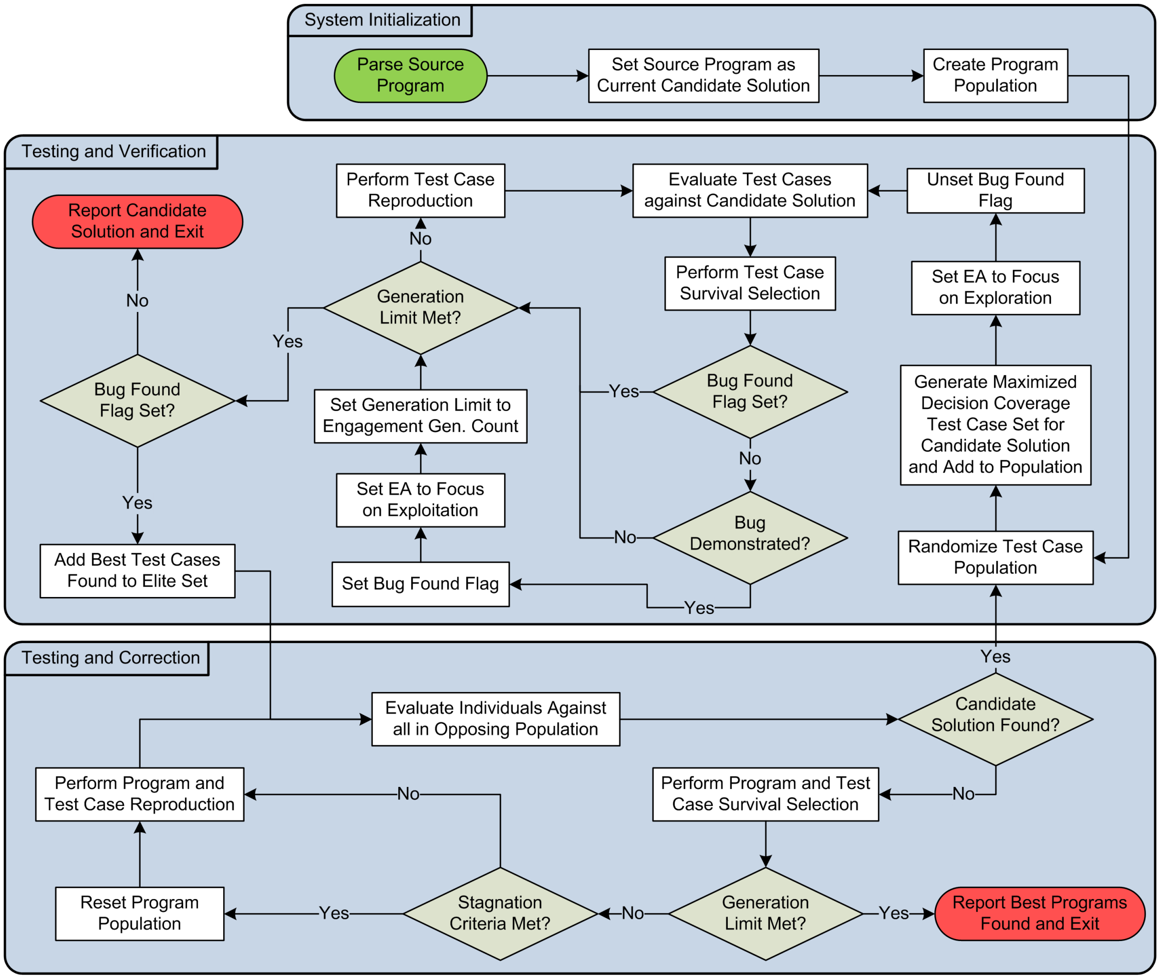

System Initialization
Parse Source Program
Set Source Program as Current Candidate Solution
Create Program Population
Testing and Verification
Report Candidate Solution and Exit
Perform Test Case Reproduction
Evaluate Test Cases against Candidate Solution
Unset Bug Found Flag
No
Generation Limit Met?
Perform Test Case Survival Selection
Set EA to Focus on Exploration
No
Yes
Bug Found Flag Set?
Set Generation Limit to Engagement Gen. Count
Yes
Bug Found Flag Set?
Generate Maximized Decision Coverage Test Case Set for Candidate Solution and Add to Population
No
Yes
Set EA to Focus on Exploitation
Bug Demonstrated?
No
Add Best Test Cases Found to Elite Set
Set Bug Found Flag
Randomize Test Case Population
Yes
Testing and Correction
Yes
Evaluate Individuals Against all in Opposing Population
Candidate Solution Found?
Perform Program and Test Case Reproduction
No
Perform Program and Test Case Survival Selection
No
Reset Program Population
Yes
Stagnation Criteria Met?
No
Generation Limit Met?
Yes
Report Best Programs Found and Exit

Figure 5.5: CASC Testing, Correction, and Verification Process

After initialization is complete, control is passed to the *Testing and Verification* module. This module attempts to identify test cases that demonstrate buggy behavior in the current candidate solution (in the first pass, this is the source program). If a bug is not demonstrated then the system exits; otherwise the system attempts to create additional test cases that demonstrate the bug and then passes control to the *Testing and Correction* module.

The *Testing and Correction* module is responsible for creating a program (based on the source program) that passes all test cases created by the system. If no such program is found, then the system exits; otherwise the created program is marked as the candidate solution and control is passed back to the *Testing and Verification* module.

CASC utilizes multiple EAs, which are described in this section. For each run, a set of EA strategy parameters used by the system are provided via a configuration file. Except when specifically noted, the set of EA strategy parameters provided in this file are used uniformly by the EAs in the system.

This section is organized based on the major phases in the flow chart chart shown in Figure 5.5:

- System initialization is discussed in Section 5.2.2; with program parsing and program population initialization discussed in Sections 5.2.2.1 and 5.2.2.2, respectively.

- The *Testing and Verification* module is described in Section 5.2.3. Test case initialization is discussed in Section 5.2.3.1, coverage based test case set creation is discussed in Section 5.2.3.2, and the testing and verification EA is described in Section 5.2.3.3.

- The *Testing and Correction* module is described in Section 5.2.4. Program and test case evaluation is described in Section 5.2.4.1, with the optimization methods supported in CASC described in Section 5.2.4.2. The program reproduction operators used are described in Section 5.2.4.4. Search stagnation detection is discussed in Section 5.2.4.5.

The method of communication of problem specific information to an automated repair system is an important aspect of any such system; in order to be practical, the system must have the versatility to address a variety of problems. In the CASC system the test case object embodies all problem specific information needed by the system and supporting subsystems. This is accomplished through the use of the dynamic polymorphism design paradigm [14]. An Abstract Test Case (ATC) object is provided by the system, laying out guidelines for the expected functionality that a Problem Specific Test Case (PSTC) object will need. The goal of this design is to centralize all problem specific implementation in a single object, making the transition to new problems as smooth as possible. Key functionality included in the current ATC object is:

- Mutation, crossover, and randomization of test cases
- Creation of input needed by the program from the test case
- Creation of expected output for the test case
- Reading in and storing the output of an execution
- Scoring an execution (i.e., calculation of objective scores for the problem)

This functionality essentially makes the system able to serve as its own oracle, removing the need for an external oracle. Clearly, this object is limited to the application of a specific class of problems; namely those for which expected output can be generated for the test case. Through use of the polymorphic design of the system, additional ATC objects could be designed for other classes of problems.

The described ATC object may, at first, appear to impose steep requirements on the user of the CASC system; however, for any system to remove the need for a priori generated test case sets made through the use of an external oracle, similar requirements will need to be made. In other words, the system will need to be told what defines a test case for the problem and how to use and manipulate such a test case. Through the use of the described ATC object (or another similar object), automated software engineering systems (such as CASC) can clearly be expected to achieve a higher degree of automation and perform more comprehensive and intelligent search than those that rely on a priori generated test sets and external oracles.

The comprehensiveness of the testing performed by the CASC system is directly related to that of the PSTC being used. For example, if a bug is only demonstrated for test cases with duplicate genes, but the PSTC disallows the creation of duplicate genes in a test case, then the system will not be able to demonstrate the bug and, as such, be unable to correct it. If known beforehand, information regarding the nature of the error in the program could be exploited during the design of a PSTC by restricting the test cases that can be generated by the PSTC, making the implementation simpler. However, such restrictions introduce bias into the system and, as such, limit its effectiveness and so should be used with caution. Following the previous example, assume a PSTC is implemented that focuses exclusively on the generation and manipulation of test cases that contain duplicate genes. The CASC system would be expected to quickly identify the error in question and would likely correct it; however, if the error was masking a second error or, in the process of correcting the error, the system introduced a new error that was not reliant on duplicate genes, then the solution presented by the system would essentially be a false positive. Similarly, these dangers are also present when using a priori generated test case sets, since when these are used the system is completely reliant on the generating entity's ability to generate a test case set with the necessary degree of comprehension to correct the problem effectively.

For the remainder of this discussion, a running example will be used to demonstrate various aspects of CASC using the buggy bubble sort function shown in Figure 5.6. For the sake of the example, assume that this function is being focused on for correction and, as such, the specifications being used are specifically for the sorting function; namely that after execution the resulting *data* array is a permutation of input *data* array and that it is in sorted order. Also assume that the function shown is embedded in an otherwise complete program that in some way communicates the input and output *data* array. In this program the error is on line 6, which should be *data[j] = data[j+1]*.

```
  const int SIZE = 10;
  ...
  void sort(int data[])
  {
1   int i, j, temp;
2   for(i = 0; i < SIZE; ++i) {
3     for(j = 0; j < SIZE - 1; ++j) {
4       if(data[j] > data[j + 1]) {
5         temp = data[j];
6         data[j+1] = data[j];
7         data[j+1] = temp; } } }
  }
```

Figure 5.6: Buggy Bubble Sort Function

5.2.2. System Initialization Module. This module is primarily responsible for parsing the source program and creating the initial program population from the source program. After the first pass through, this module is never reentered during the run.

5.2.2.1. CASC parsing. The first major task in CASC is to parse the source program, resulting in a tree representation of the program that can then be analyzed and modified. A tree representation is used because it is a natural representation for the code elements, cleanly displaying relationships between them. This makes it relatively simple to perform modifications to, and generate the code represented by, the trees.

Special comment tags can be used in the source program to mark the start and end of the Evolvable Sections (ESs) of code (i.e., sections where a semantic error is suspected). If these tags are used, then the code that is not part of an ES is not modified during evolution. If no tags are indicated, then the bodies of all the routines in the program are considered to be ESs (support for file/global scope evolution is not yet supported by CASC). Clearly, the problem space for the program population can be dramatically reduced through the use of the ES tags. Because of this, it is highly recommended to first apply fault localization software/techniques to the buggy software artifact to identify the ESs (although preliminary scalability experiments showed CASC having a sub linear relationship with problem size [105]).

The CASC parser first converts the source code to *srcML* [21] using the srcML toolkit. srcML is an XML representation of source code containing both the code text and selective abstract syntax tree information. The srcML toolkit currently supports C, C++, and Java; hence these are the languages currently supported by CASC, though C++ is the language that has been focused on. The resulting XML document is processed using the *pugixml* [53] library, which creates object trees based on the innate tree structure of XML. ESs are identified using the XML node information and are converted to ES objects, which are added to a Program object. A CASC Program is essentially a set of one or more ES objects along with the information typically stored for individuals in an EA (e.g., fitness, objective scores, book-keeping data). Each ES object contains a forest of trees (representing the code for the ES, one tree per line of code) and a variable name registry for the ES (indicating valid variable names to use during code modification for the ES).

Name registries are created during the parsing process from global declarations, function parameter lists for the function containing the ES, and local declarations. Each name has an associated type (e.g., int, char, float), qualifier information, and modifier information stored for the name. Additionally, all name registries also share an object registry, containing a listing of known user-defined objects; essentially each registered user-defined object has a name registry associated with it, containing information on the public members of the object. The object registry allows for intelligent modification/use of user defined types during evolution. Member references are treated as atomic subtrees during code modification (i.e., the object, the member access operator, and the member being accessed), creating simple compatibility for modification between primitive variables and object instances.

Nodes in the ES trees are assigned a type indicating the nature of the node. Each node's type belongs to a node class, which is used during evolution to help maintain syntactic validity in generated programs. The node classes used by CASC for C++ programs are shown in Table 5.1. The *Misc* node class contains specific names (e.g., *cout*, *cin*, *NULL*), operators, and other code elements that should not be generated during code evolution.

The CASC parser monitors scope level during the parsing process; when ES trees are created, each root node is assigned the appropriate scope level. CASC uses scope to indicate lines affected by control statements.

For the running example, assume that lines 2-7 in the function shown in Figure 5.6 are indicated as an ES. The trees that would be generated for this program are shown in Figure 5.7 along with the associated name registry in Table 5.2.

5.2.2.2. Program population initialization. The program population is initialized by modifying copies of the source program employing the mutation and architecture alter operators. These operators are described in detail in Section 5.2.4.4; the primary difference in their application in this phase is that when doing mutation, the proportion of program nodes mutated is randomly selected from a Gaussian distribution.

Table 5.1: Currently Supported C++ Node Classes

Node Class	**Associated Node Types**
Function	Function Calls
Terminal	Numeric Literal, Variable, Array, Logic(*true*, *false*) Obj. Reference, Obj. Dereference
Ternary Operator	?:
Binary Operator	+, -, *, /, =, Modulus, Comma
Unary Operator	!, -, ++, - -, new, delete, &, *
Logical Binary Operator	&&, ||, <, >, <=, >=, ==, !=
Bitwise Operator	Bitwise And, Bitwise Or, Bitwise Xor, Bitwise Not
Branch	if, else, else if
Loop	for, while
Misc	Declaration, return, Comment, Insertion, Extraction, cout, cin, cerr, endl, stdout, stdin, String Literal, NULL, switch, case, default, break

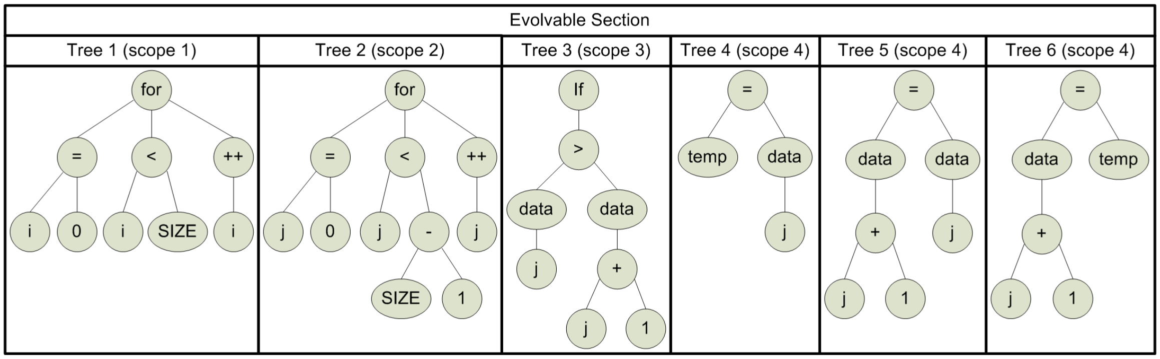


Figure 5.7: Parsing Result for Running Example

5.2.3. Testing and Verification Module. This module is responsible for searching for test cases that demonstrate errors in the current candidate solution. If a test case is discovered that demonstrates an error, then the test case is propagated throughout the test case population. This propagation ensures engagement between the test case and program population in the *Testing and Correction* module. If no test

Table 5.2: Name Registry Generated for Running Example

Name	Type	Details
SIZE	int	global, constant
data	int	array
i	int	
j	int	
temp	int	

cases are discovered that demonstrate an error, then the current candidate solution is reported as a result for the run and the system exits.

5.2.3.1. Test case population creation. The first task upon entering this module is to randomize the test case population. If entered from the *System Initialization* module, then this serves as the initial test case population creation. If entered from the *Testing and Correction* module, then a candidate program solution has been identified that passes all test cases in the current population. Thus, the test case population is re-randomized to initialize the verification process.

5.2.3.2. Covering test set creation. In addition to the initially random test case population, the system has the ability to generate and add a Covering test case Set (CS) for the candidate solution program to the test case population. The use of this functionality is designated by the user via the configuration options for the system. The goal of adding the CS to the test case population is to increase confidence in the results presented by the *Testing and Verification* module, as well as attempt early detection of false positive candidate solutions.

Coverage metrics are often used to indicate the extent to which a test case set tests a given program. The decision coverage metric was decided on as an initial exploration into the benefits of using a high coverage test set in the CASC system, as it is a relatively easy metric to calculate and maximize while still generating quality test case sets.

The CS is created by first instrumenting the candidate solution program to report decision outcomes to a file during execution. This is accomplished by first creating a fully parsed version of the candidate solution, i.e., parse trees are created for all routines in the program, rather than just for the ES(s). The program is then instrumented (via the parse trees) by identifying the decision outcomes, assigning each outcome a unique ID, and then modifying the trees to output the appropriate ID when the outcome is used. An EA style approach is used to perform CS creation. A population of test cases is initially created randomly. Each test case is executed using the instrumented program and the set of decisions covered is stored for the test case. After all test cases have been executed, the CS is then created using a greedy approach. Test cases with the most unique decision outcomes covered (not considering the outcomes already in the set) are added to the CS one at a time until no unique outcomes are covered by the remaining test cases. The next generation of test cases is created by replacing all test cases not in the CS from the last generation using the test case reproduction methods described in Section 5.2.3.3. This process continues until either a user specified number of generations have completed or all identified decision outcomes are covered by the CS.

5.2.3.3. Testing and verification. CASC uses an EA to perform testing and verification of candidate solution programs. This process begins by evaluating the test case population against the candidate solution program. This is done by executing the candidate solution program against each test case, scoring the results using the scoring function(s) included in the PSTC. CASC assumes that the program being corrected operates deterministically and, as such, uses a lookup table to store the score(s) for a given program-test case pairing to avoid unnecessary repeat executions. CASC fitness scoring is described in detail in Section 5.2.4.2.

After evaluation, if a bug has not already been demonstrated by the test case population, the system determines if a bug was demonstrated during the evaluation.

If a bug was demonstrated, then the system sets the flag indicating that a bug has been demonstrated and sets the number of generations remaining to be a user defined number of engagement generations; a recommended minimum for this parameter is the number of generations indicated for the *Testing and Correction* module. The engagement generations allow the demonstrated bug time to propagate in the test case population before entering the *Testing and Correction* module.

If the generation limit has not been met, the system goes on to perform survival selection. This process is conducted using an inverted tournament selection scheme. In this scheme a subset (of user defined size) of individuals are selected at random from the population, then the individual with the lowest fitness value is removed from the population.

Following survival selection, the system performs test case reproduction. Three reproduction operators are employed by the system: randomization, mutation, and crossover. These operators are problem specific and, as such, are part of the PSTC. The operators are applied based on a probability distribution. If a bug has not been identified yet, then a system defined distribution is used that is focused on exploration over exploitation (i.e., mutation and randomization is used much more frequently than crossover), under the rationale that if no bug has been identified, then there is no high performing genetic material identified to exploit. If a bug has been identified, then a user defined probability distribution is used, to allow fine tuning of the balance between exploitation and exploration, as desired.

CASC employs a modified fitness proportional selection with re-selection allowed. First a subset of individuals is selected randomly (similar to typical tournament selection). Next each individual is assigned a selection probability proportional to its fitness within the subset, with an arbitrarily chosen minimum probability of 1%. This method was chosen over rank based selection because it was found that in many experiments the selected subsets contained a significant portion of individuals with equal or near equal fitness values. It is very easy for individuals to have equal or near equal fitness but have quite different genotypic representations; this technique allows these differences a fair chance at propagation without arbitrary bias.

Once the generation limit is met, if a bug has not been demonstrated, then the candidate solution is reported as a solution and the system exits. If a bug was demonstrated, then the top performing test cases identified are made elite, guaranteeing their preservation for the rest of the run. This is similar to the Hall of Fame mechanism proposed by Rosin to prevent cycling in a coevolutionary search [86]. The main difference between the two approaches is that individuals placed in the Hall of Fame are removed from the population, making them unavailable for use during recombination; whereas in CASC, being made elite simply protects the individual from being selected during survival selection. If SOOP is being performed, then the set of non-elite test cases currently safe from survival selection (i.e., the top *survival-selection-tournament-size* minus 1 individuals) are made elite. If MOOP is being used, then the test cases in the non-dominated front are made elite. The test case population size is increased by the number of test cases made elite so as to maintain the number of evolving individuals across multiple verification cycles. To avoid population bloat, the number of test cases that can be made elite can never exceed 25% of the original population size. If the size of the new would-be elite set exceeds this, then just the oldest individuals are taken. Making these individuals elite ensures that a candidate solution program must pass these test cases as well as all others generated by the system in order to be presented as a solution.

5.2.4. Testing and Correction Module. If a bug is identified in the $Testing\ and\ Verification$ module, then the $Testing\ and\ Correction$ module is used to attempt to create a program that corrects the bug. While searching for a solution, the system will be non-deterministically creating modified programs that, in many cases, essentially introduce additional bugs into the program. The system's ability to recognize errors introduced into programs is determined by the accuracy of the specifications for the software (i.e., the definition of correct execution) and the degree of comprehension in the test cases used/generated by the system. To account for this possibility, CASC utilizes a two-population competitive coevolutionary cycle, which is basically two overlaid evolutionary cycles intersecting at the point of fitness evaluation. The two populations being evolved are a population of programs and a population of test cases (i.e., program inputs). The fitness of each program is estimated by its performance on a sampling of test cases. Similarly, the fitness of each test case is estimated by its performance on a sampling of programs. Since each population is attempting to optimize these fitness values, an evolutionary arms race results.

5.2.4.1. Evaluation and survival selection. Upon first entering the *Testing and Correction* module, the program population needs to be evaluated against the test case population created in the *Testing and Verification* module. If this is the first pass through the *Testing and Correction* module, then the program individuals need to be assigned initial fitness values. If the module is being entered after successfully identifying a false-positive candidate solution, then the program population needs to be exposed to the counter test case(s) that were identified.

The evaluation phase is the point where the two evolutionary cycles meet and the populations interact. In order to maximize population exposure, each individual is executed against all opponents in the competing population. Results from repeat program-test case pairings are retrieved from the lookup table. The remaining pairings are executed in parallel on a computing cluster employing MPI. CASC uses a master-slave parallel computing topology, where each slave computing node is responsible for handling all executions involving a subset of the program population (these sets are balanced during the competition and reproduction phases). The slave computing nodes run required executions concurrently in threads. Once all threads have completed and the results have been reported to the main node (and stored for future reference in the lookup table) the overall fitness for each individual is calculated.

Programs are compiled at creation during program reproduction. While the CASC system attempts to maintain syntactic validity during program modification, the system is currently only aware of a subset of the grammatical rules in the languages supported. To account for this, compilation results are monitored and if a program has a compiler error, then it is given an arbitrarily low fitness and marked as being invalid for execution. If a program experiences a run time error, then the program is assigned an arbitrarily low fitness for the pairing and the associated test case is assigned a high fitness for the pairing. Program time outs are monitored based on a user specified maximum allowable CPU time; if a program runs for longer than this limit, then the execution is terminated, the program is assigned an arbitrarily low fitness and is marked as invalid for future execution.

Both populations employ the survival selection method that is described in Section 5.2.3.3. Using this technique, individuals are removed from the population one at a time until the population size returns to the indicated population size. If a population is already at the indicated size (as may be the case upon first entry into the *Testing and Correction* module), then this process is simply skipped for that population.

5.2.4.2. Optimization methods. CASC supports both SOOP and MOOP. A run is scored by comparing actual program output with expected output for a given test case. In order for CASC to correctly guide the evolutionary process, the objective function(s) should be based on the specifications for the program being corrected. The same objective function(s) are used by both populations, where the optimization direction (i.e., minimize or maximize) for one population is reversed for the other (in SOOP it is assumed that the program population is maximizing the objective/fitness function). The objective functions are implemented as part of the PSTC.

In the CASC MOOP system, it is assumed that each objective is associated with an individual specification for the program. An individual's fitness is calculated as the percentage of its population that it dominates (which is determined based on the individual's objective scores). The number and optimization direction of the objective functions are user specified via the PSTC.

When using MOOP, intra-front genotypic diversity can be promoted by activating a fitness sharing [35] mechanism. Before fitness sharing is applied (and when fitness sharing is disabled), all individuals on a front are assigned the same fitness value (since they all dominate the same percentage of the population). For example, if the non-dominated front dominates 75% of the population and the second front dominates 50%, the non-dominated front will receive 0.75 fitness and the second front will receive 0.50 fitness. CASC's fitness sharing mechanism uses the unused fitness ranges between these fronts (i.e., [0.75, 1] and [0.50, 0.74]) to promote genetically unique individuals while still maintaing the calculated front structure.

First, a sharing factor is calculated between all individuals on a front using the following equation from Goldberg and Richardson [35]:

$$Sh(d) = \begin{cases} 1 - \left(\dfrac{d}{\sigma_{share}}\right)^{\alpha} & \text{if } d \leq \sigma_{share} \quad \text{(5a)} \\ 0 & \text{if } d > \sigma_{share} \quad \text{(5b)} \end{cases}$$

where in CASC, d is the genotypic distance between two individuals normalized by dividing by the number of genes in the largest individual in the current front. The value σ_{share} indicates the maximum distance two individuals can be apart and still share fitness. The value α is used to define the growth of the sharing factor (i.e., sub-linear, linear, or super-linear).

Genotypic distance between individuals is calculated by traversing the program trees in question, creating a list of the nodes in each program, and finding the Levenshtein (i.e., edit) distance [15] between the lists. Before the tree for each program line is traversed, a scope node is added to the list that indicates the scope for the nodes that follow it in the list (so that scope can be considered in the edit distance as well). Edit distance is then calculated between the node lists for two programs, the result of which is stored as the genotypic distance between those two individuals. Figure 5.8 gives a visual example of program distance calculation (assume that trees not shown in this figure are the same for both programs).

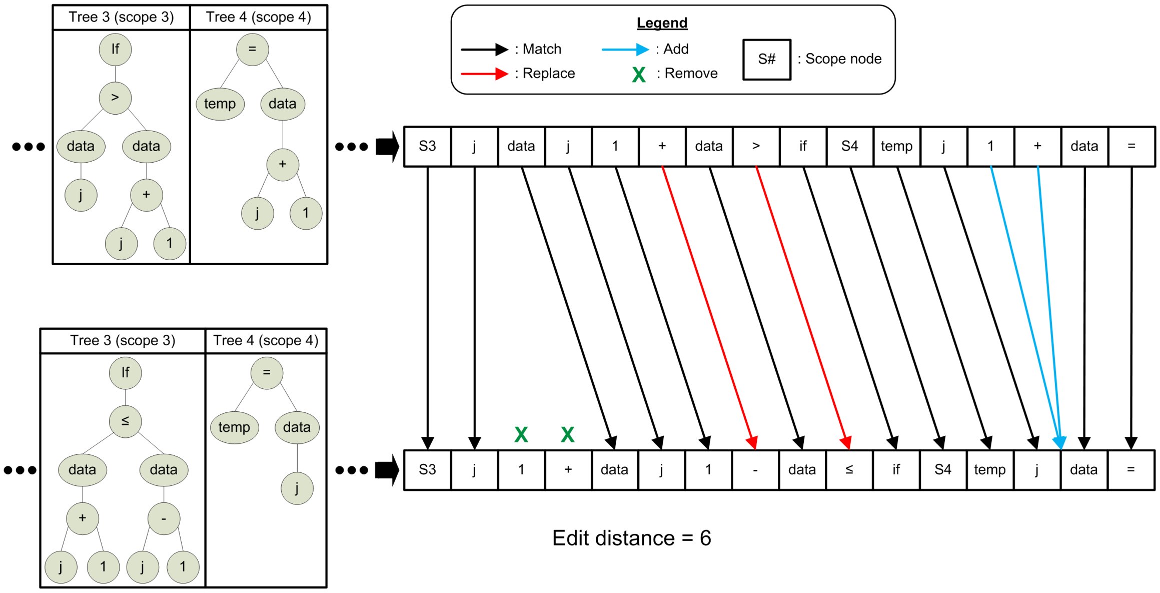


Figure 5.8: Example program distance calculation

The niche count for an individual x (nc_x) is the sum of the sharing factors between x and all of the individuals on the same front as x (including itself). The min and max niche counts (nc_{min} and nc_{max}) are then determined for all individuals on the front, and used to adjust the individual fitness values according to the following equation:

$$fitness_i = f_L + (f_H - f_L) \cdot \frac{nc_{max} - nc_i}{nc_{max} - nc_{min}} \qquad (6)$$

where f_L and f_H are the lower and upper bounds on the fitness range for the front, respectively. An example fitness sharing calculation is shown in Figure 5.9. This example applies the fitness sharing algorithm to a five program front, starting with a fitness of 0.5 and the next front having a fitness of 0.7. This gives the algorithm a range of [0.5, 0.69] to work with. Programs 4 and 5 are copies of each other and the other programs are unique. The edit distances between each program are shown in the symmetric matrix on the lower left. From this matrix, it can be seen that program 2 generally has a high edit distance to the other programs, relative to the others. Similarly, programs 4 and 5 have a relatively low edit distance. Using the calculations shown, these relationships are captured and shown in the resulting shared fitness values, where program 2 is given the highest fitness in the range and programs 4 and 5 remain at 0.5 fitness.

Overview:

Program Number	Original Fitness	Size	Niche Count	Shared Fitness
1	0.5	30	2.292	0.535
2	0.5	27	1.167	0.690
3	0.5	32	1.708	0.615
4	0.5	29	2.542	0.500
5	0.5	29	2.542	0.500

Values Used for Calculations and Their Sources:

$f_L = 0.5$	Original front fitness
$f_H = 0.69$	Min of next front fitness – 0.01
$\sigma_{share} = 0.15$	Constant
$\alpha = 1$	Constant
$nc_{min} = 1.167$	Calculated
$nc_{max} = 2.542$	Calculated

Calculations:

Edit Distances

	P1	P2	P3	P4	P5
P1	0	4	3	3	3
P2	4	0	5	5	5
P3	3	5	0	4	4
P4	3	5	4	0	0
P5	3	5	4	0	0

➡

Normalized Edit Disances (d)

	P1	P2	P3	P4	P5
P1	0.000	0.125	0.094	0.094	0.094
P2	0.125	0.000	0.156	0.156	0.156
P3	0.094	0.156	0.000	0.125	0.125
P4	0.094	0.156	0.125	0.000	0.000
P5	0.094	0.156	0.125	0.000	0.000

➡

Sharing factor (Sh(d))

	P1	P2	P3	P4	P5		**Niche Count**
P1: ∑	1.000	0.167	0.375	0.375	0.375	=	2.292
P2: ∑	0.167	1.000	0.000	0.000	0.000	=	1.167
P3: ∑	0.375	0.000	1.000	0.167	0.167	=	1.708
P4: ∑	0.375	0.000	0.167	1.000	1.000	=	2.542
P5: ∑	0.375	0.000	0.167	1.000	1.000	=	2.542

Figure 5.9: Example fitness sharing calculation for a front of program individuals

A general (i.e., non-problem-specific) objective can be activated in CASC that scores programs based on their edit distance from the source program. The rationale behind this objective is two-fold: first, it can reasonably be assumed that the source program is not far from correct in terms of edits needed, since programmers do not develop randomly; thus keeping the program population genetically close to the source program should help focus the search more effectively. Second, program bloat can become an issue in the program population; this objective will keep program individuals close to the same size as the source program. This objective is scored as: $score = \left\lfloor \frac{d_{src}}{w} \right\rfloor$, where d_{src} is the edit distance between the individual and the source program and w is the (user configurable) tier width allowed. The tier width defines the number of edits from the source program that are allowed before a penalty is incurred. For example, if w is set as 5, then a programcan have an edit distance of up to 5 from the source program, before being penalized by being assigned to the second tier.

The CASC system performs optimization at the global scope level (i.e., performance is rated on the program as a whole), while it performs modifications at a potentially deeper scope level (dependent on the ES(s) for the program). Obviously, the objectives being optimized must be able to detect incorrect behavior in the ES in order to both correct the bug and avoid introducing new bugs. Accordingly, the PSTC must be created to match the resolution of the specifications being used. For example, one of the programs used for CASC experimentation is a text replacement program that searches for instances of a provided pattern in text and replaces them with an indicated string. While this program was provided as an independent program, its functionality is common in a number of other programs. The testing and correction of this program could just as easily be the high resolution testing and correction of a program that has this text replacement functionality. If this were the case, the objectives and PSTC used for the correction of this program would not change, as they would need to be made to correct the text replacement program/functionality, not the larger program that the text replacement functionality is part of.

5.2.4.3. Multi-objective solution prioritization.

Generic objectives (such as the edits from source objective described in Section 5.2.4.2) improve the system's ability to guide its search processes. However, if MOOP is used, then the scores from these objectives can create a non-dominance relationship based purely on generic objective performance, disregarding the problem specific scores. For example, assume that the edits from source objective did not use the described tier system, but simply the number of edits from the source program. This would create the situation where an individual that performed perfect on all problem specific objectives and was one edit from the source would have a non-dominance relationship with the source program original (buggy) source program. This situation is the reason that the described tier system is used for this objective; however, it may not always be clear what value should be used for the tier-width, making this objective difficult to use in some cases.

Multi-Objective Solution Prioritization (MOSP) is an extension to NSGA-II that implements prioritization levels for objectives, allowing the system to focus on the optimization of high priority objectives (such as the problem specific objectives in CASC) before lower priority objectives (such as the edits from source objective)[5].

In order to implement MOSP, the ranking/sorting scheme needs to be modified to take into account different levels of optimization; this is achieved by modifying the dominance operator to compare only objectives at the required level. Rather than comparing all objectives for a problem at once, this operator compares only the objectives for dominance at the specified level, allowing the use of many objectives without destroying the dominance scheme established by the primary objectives. The

[5] The MOSP algorithm is the work of the author and his undergraduate mentee James Bridges. This section summarizes the algorithm which is the basis for James' CS448 Advanced Evolutionary Computing project report and a conference paper in preparation.

next addition to NSGA-II is a recursive call on each front produced by the algorithm. This effectively refines each front by evaluating it on subsequent prioritization level(s).

Using MOSP does not alter the dominance scheme imposed by NSGA-II, rather it further refines the results. Each front produced has the unique property that individuals along it will all dominate the same number of individuals; however, if it is deemed that an individual can dominate members on its own front by prioritization objectives, then effectively a new front is created. This new front is called a micro front; each micro front is positioned between the fronts created by the primary objectives. During construction, the micro fronts are guaranteed to never dominate the original front they were created from or individuals on higher micro-fronts for the current front and will always dominate the next lower primary objective front.

MOSP is built around the concept of prioritization levels. The zeroth prioritization level always contains the primary objectives; i.e., hard requirements of the problem to be solved. The other levels contain the soft requirements, each level can have 1 to n objectives. None of the prioritization levels can be null, each level must contain at least one objective. It can be derived that it does not matter how many objectives a level contains because the domination criteria is still the same, i.e., dominance criteria for an individual at its respective level. As discussed the use of these prioritization levels creates micro fronts. These micro fronts produce two useful properties. One property is that they create granularity among solutions in order to differentiate against the optional objectives. What this amounts to is an increased domination count which increases the chance for the individual to be selected as a parent to produce offspring. Likewise, if an individual is in a higher micro front than its respective primary objective front, then the individual has a higher chance of being selected as a member of the new population for the next iteration of the EA. The other property is the prevention of front pollution. Front pollution occurs when the objectives an individual is being ranked against detract from the main purpose of optimization (such as the example given initially in this section). The prioritization scheme prevents this type of behavior by ensuring primary objectives are satisfied first regardless of objective scores, only after the primary objective constraint has been satisfied will the optional objectives be applied.

Proof of concept experiments were conducted using the MOSP algorithm on the classic 0-1 Knapsack problem. These experiments had very strong results, indicating the general viability of the proposed approach. Initial studies of the incorporation of the MOSP algorithm into CASC have had promising results. Further investigation and study of the combination of these approaches is left as future work for the CASC system.

5.2.4.4. Program reproduction. New programs are produced by applying one of five GP operators: copy, reset, crossover, mutation, and architecture alteration. The operator to apply is selected based on a user configurable probability distribution. Then a new program is produced and added to the program population, and the process is repeated until a specified number of new programs have been produced.

CASC employs a modified fitness proportional selection with re-selection allowed. First a subset of k individuals is selected randomly (as in tournament selection). Next each individual is assigned a selection probability proportional to its fitness within the subset, with an arbitrarily chosen minimum probability of 1%. This method was chosen over rank based selection because it was found that in many experiments the selected subsets contained a significant portion of individuals with equal or near equal fitness values. It is very easy for program individuals to have equal or near equal fitness but have quite different genotypic representations; this technique allows these differences a fair chance at propagation without arbitrary bias.

The copy GP operator selects a program from the program population and places a duplicate copy into the program population. This operator's primary purpose is to promote the genetic material of successful individuals.

The reset GP operator from [10] makes a copy of the source program and puts it into the evolving population. The purpose of this operator is to make sure that the original genetic material does not get lost in the evolutionary process.

The crossover GP operator is the primary exploitation operator in the CASC system, using subtree exchange to recombine genetic material present in the program population. The node classification system is used to define compatibility between subtrees, as unrestricted subtree exchange between programs would have a high likelihood of resulting in syntactically invalid programs. While the compatibility system helps to ensure syntactic validity, it does not yet encompass all grammatical rules for C++, and as such can still result in syntactically invalid programs. Regardless, using the current compatibility system, the crossover operator yielded syntactically valid programs on average 88% of the time in the experiments presented in this paper.

The first compatibility check performed is intended to help preserve intention in the code. This is done by matching the expression roots for the trees containing the selected subtrees. Expression roots in the CASC system are the root node for the line, an *Array* node, or a *Function Call* node. The expression root is determined by starting at the root of the selected subtree and traversing up (toward the root) until one of these nodes is found. Matching the expression roots for the exchanged subtrees makes it so that, for example, an addition subtree that serves as an index for an array is only swapped with other indexing subtrees. Similarly, an assignment subtree that is part of a for statement will only be swapped with other assignment statements that are being used the same way.

The second compatibility check used during crossover is focused on achieving syntactic validity in the resulting programs. For the selected subtree, the classification of the root node for the subtree (i.e., node class and type) is used to determine the classifications that the root node for another subtree must have in order to be compatible. Table 5.3 shows the classification compatibilities currently used by CASC.

Table 5.3: Node Compatibilities for Program Crossover

Root Node Class	**Root Node Type**	**Compatible Classification(s)**
Binary Op. Terminal	Binary Op: =	Binary Op: =
	Binary Op: all except = Terminal: All	Binary Op: all except = Terminal Unary Op: Negate Unary Op: (Pre)Increment Unary Op: (Pre)Decrement
Unary Op	Negate (Pre)Increment (Pre)Decrement	Binary Op: all except = Terminal Unary Op: Negate Unary Op: (Pre)Increment Unary Op: (Pre)Decrement
	Not	Unary Op: Not Logic Op: All
Logic Op	All Types	Unary Op: Not Logic Op: All
Bitwise Op	Biwise Not	Bitwise Not
	Bitwise And Bitwise Or Bitwise Xor	Bitwise And Bitwise Or Bitwise Xor
Special	Coment Declaration	None
All Other	All Other	Nodes with same type

After the two parents are selected for crossover, a subtree s_1 is selected at random in the first parent. In the second parent, all subtrees that pass both compatibility checks with s_1 are identified. If no compatible subtrees are found in the second parent, then a new subtree is selected in the first parent and the process is repeated. The second subtree is then selected at random from the compatible set and the subtrees are exchanged, creating two new individuals.

Figure 5.10 illustrates a possible crossover operation for the running example. In this example, the j node outlined in red is selected as the subtree to exchange in parent 1. The nodes outlined in red and blue in the second parent represent compatible subtrees in the second parent, all of the other nodes do not pass one or both of the compatibility checks. For example, the *temp* node in tree 4 is in the Terminal node class, making it compatible with j; however, *temp*'s expression root type is Assignment whereas j's is Array (the *data* node), making it incompatible. The + node outlined in red in parent 2 is the node that is selected to be exchanged with parent 1, resulting in the two child programs shown.

The mutation GP operator is responsible for exploring the program search space by bringing in new genetic material to the population. The mutant program starts as a copy of the selected parent program. Next all of the mutable nodes in the mutant are identified. Mutable nodes are defined as those that have a reasonable mutation that can be performed on them. The mutable nodes of the CASC system and their possible mutations are summarized in Table 5.4.

Subtrees are generated as needed by the various mutations. For non-context specific nodes (i.e., Binary Ops, Unary Ops, Logic Ops), generated node types are simply randomly selected as is described in Table 5.4. For context specific nodes (i.e., non-Number Terminals), the name registry is used to randomly select an appropriate/valid name to use for the node. Local and global names that are available in the ES are stored as such in the name registry. During name selection, if there are both local and global names available, then there is 25% bias put towards selecting a local name over a global one in order to promote more eploration using local names than global ones. Also, the name registry indicates which names are constant, which is used to avoid generating code that attempts to modify a constant variable.

If a Number is to be generated, then its value is randomly selected from a Gaussian distribution centered around zero, favoring numbers in the range $[-5, 5]$,

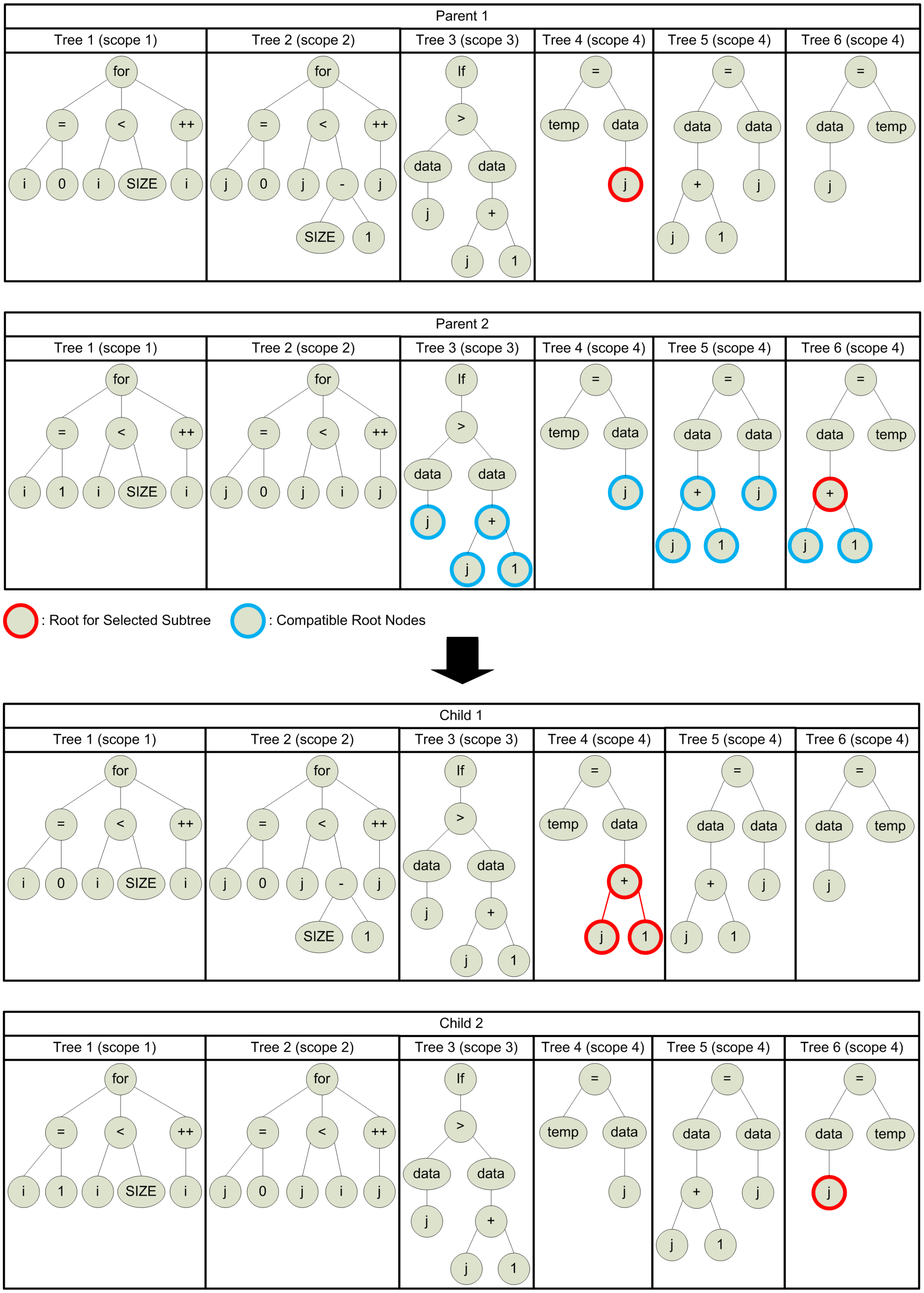


Figure 5.10: Example Program Crossover

as it is typical programming practice to make larger numbers constant variables in most cases. Negative values are allowed for these values, thus Number nodes are not mutated by making them the child of a Negate node (as other Terminals can be).

Additionally, the variable type information associated with the names in the name registry is used to restrict the amount of type mixing that is introduced by generated code. After the node to be mutated is selected, the subtree containing the node (bounded by the expression root, as defined in the discussion for the crossover GP operator) is iterated over. During this process, the variable types present in the subtree are gathered; then during code generation, only names with the same or compatible types (e.g., the various integer types are all compatible) are used. If this results in no available names for a particular mutation, then the variable type restriction is removed. This restriction is used with a 75% probability, promoting exploration using types already present in the containing expression.

Figure 5.11 shows an example of a possible mutation for the running example. The nodes highlighted in blue in the parent have been selected for mutation. The $<$ node is mutated by being replaced with a Not node, making the $<$ node a child of the new node. The + node is mutated by being overwritten with its left-hand-side operand. The resulting mutant would be a solution to this problem.

The architecture altering GP operator is used to make drastic changes to the architecture of a program. The alterations currently implemented are the insertion of a randomly generated assignment statement, the insertion of a randomly generated flow control block (i.e., loop or branch), and the deletion of lines of code. The alteration to perform is selected randomly (with equal probabilities). If a flow control block is generated, then the controlling statement is randomly generated and the position and lines affected by the flow control block are randomly selected. The purpose of this operator is to allow for drastic changes to the program that may be necessary in order to find a solution.

Table 5.4: Possible Node Mutations

Node Class	**Node Type**	**Possible Mutation(s)**
Unary Op	(Pre)Increment (Pre)Decrement	Randomly change to one of the other increase/decrease unary ops
	Not Negate	Overwrite operator with operand
Terminal	Logic	Flip logic
	Variable Array	Replace with randomly generated binary op (with node as operand)
		Replace with randomly generated terminal
		Replace with 'Negate' (with node as operand)
	Number	Replace with randomly generated binary op (with node as operand)
		Adjust the value by +/- 1
		Replace with randomly generated terminal
Binary Op	All (excluding Assign and Comma)	Replace with randomly selected Binary Op (excluding Assign and Comma)
		Replace with operand node (randomly selected)
		Replace with 'Negate' (with node as operand)
Logic Op	All	Replace with randomly selected Logic Op (excluding Assign and Comma)
		Replace with operand node (randomly selected)
		Replace with 'Not' (with node as operand)
BitwiseOp	Bitwise And Bitwise Or Bitwise Xor	Randomly change to different binary bitwise operator

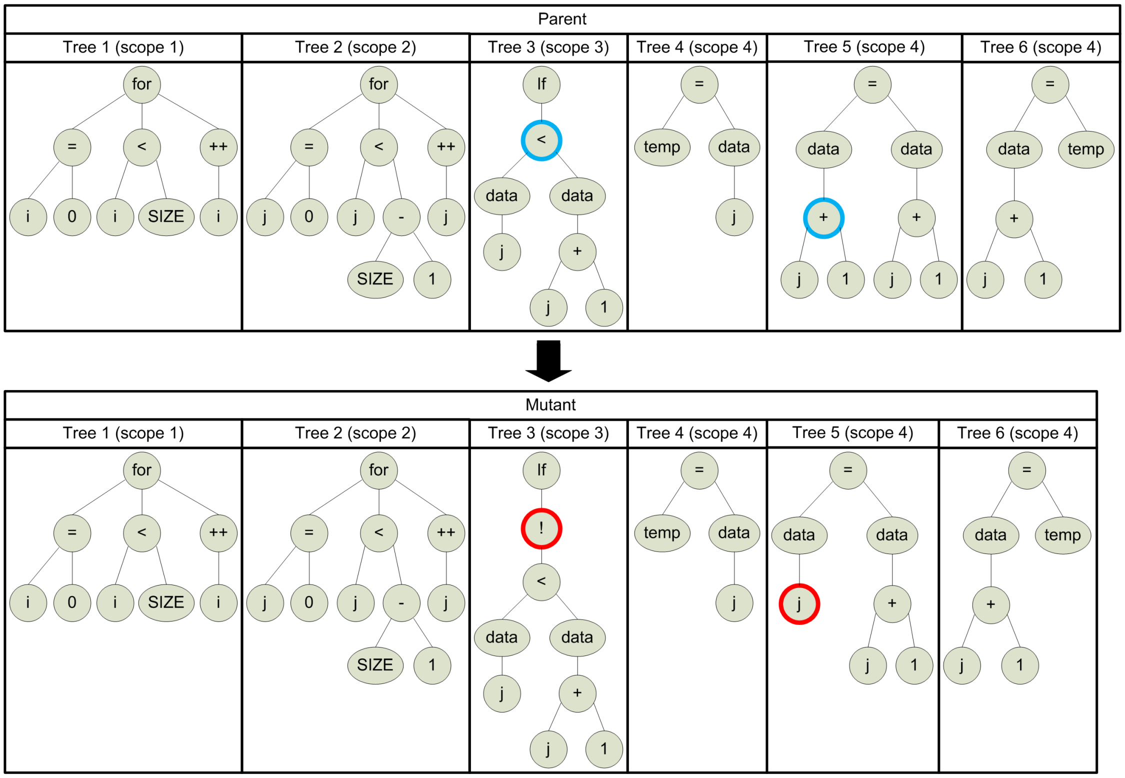


Figure 5.11: Example 2 Node Program Mutation

These operators are randomly applied based on supplied probabilities. After the specified number of new programs have been generated, the new programs are compiled and evaluated as necessary, which is described in detail in Section 5.2.4.1. At this point, the number of successes for the crossover and mutation operators are recorded, where a success is when the program produced outperforms the source program. These statistics are used to implement adaptive parameter control with a frequency specified by the user. For example, say there were 200 mutations performed since the last parameter update and 120 of these were successes (60%) and that there were 150 crossovers performed, 75 of which were successful (50%). In this case the system would increase the likelihood of performing mutation and decrease the likelihood of crossover, both by a single user specified amount. The minimum probability that each operator can have is 10%, which allows the operator to recover in the future, if appropriate. This allows the system to respond to the current state the program population is in.

5.2.4.5. Stagnation detection. At the end of each generation, the performance of the program population is analyzed to check for stagnation. Essentially, if the top performing program(s) are not surpassed for an extended period of time, then the search has likely become stuck in a local optima and, as such, has stagnated.

If MOOP is being used, then stagnation is determined based on the size and stability of the non-denominated front. If the same individuals remain on the non-dominated front for an extended period of time, then the search is at the very least struggling, if not stuck in a local optima. The check for non-dominated front stability allows for the removal of individuals due to survival selection as well as the addition of new individuals to the front. Additionally, the size of the non-dominated front is checked, as a typical characteristic of convergence is the clustering of individuals at an optima. If it is determined that the non-dominated front has been stable for a set number of generations and 95% of the population is on the non-dominated front, then the system assumes that stagnation has occurred.

If SOOP is being used, then stagnation is determined based on the number of consecutive generations that the top program individual has been the best program seen. At the end of each generation, the unique ID number of the best program in the population is compared to that of the previous generation. If the same ID is detected for a set number of generations, then the system assumes that stagnation has occurred.

When stagnation is detected, the system restarts the program search process. All programs in the current population are replaced by the source program, which are subsequently modified in the reproduction phase, starting off the new search. The test case population is maintained through this process, namely keeping all test cases from the previous search. This is done to help guide the new search more effectively, keeping it away from any false positive solutions that may have been identified in previous searches.

5.3. EXPERIMENTAL SETUP

The experiments conducted were intended to not only demonstrate the system's current abilities, but also to ascertain its limits through exposure to increasingly difficult problems. There are a number of general aspects that define problem difficulty for the CASC system. Two of these aspects were focused on in these experiments: ES size and number of bugs present.

The size of the ES(s) in the program defines the number of trees present in the program, as such this directly impacts the size of the search space. CASC currently delegates ES identification to an external fault localization utility (i.e., the FGFL system, Tarantula, or Tarantula+), and so has little control on this aspect of problem difficulty. In the presented experiments, the ESs were placed as conservatively as possible (i.e., to contain the entire routine(s) containing the error(s)) in order to make as few assumptions as possible regarding the external fault localization utility's abilities. System performance overall can be expected to improve significantly as ES size decreases through fault localization.

The number of errors present also affects the difficulty of a given problem; especially if there is a relationship between the errors (e.g., one error masking another). As the number of errors increases, so does the amount of work needed to find a solution. Currently the system relies on program performance to propagate any partial solutions that are discovered. And so, if correcting an error does not result in improved program performance, then the correction is not guaranteed propagation or even survival. Clearly, the issue of identification and propagation of partial solutions needs to be addressed, and is discussed more in Section 7. For the presented experiments, increasing numbers of bugs were seeded into the programs being corrected, with both bugs that were independently identifiable and that had relationships with each other.

The programs used in the presented study were from the Siemens test suite and a subset of those used by Arcuri in his latest JAFF publication.

The Siemens test suite [43] programs were downloaded from the Software-artifact Infrastructure Repository [32]. The Siemens test suite consists of seven programs (with varying goals), each of which have multiple bug versions with seeded errors. These programs, originally written in K&R C, were updated to C++, but were otherwise kept semantically equivalent to the original. The results presented here are focused on CASC's performance on the Siemens programs *printtokens*2 and *replace*. The *printtokens*2 program is 570 lines of code, consisting of 19 routines. The *replace* program is 564 lines of code, consisting of 21 routines. The *printtokens*2 bug versions used were v4, v6, and v7. The *replace* bug versions used were v11, v13, and v33. These bug versions were selected because the bugs are simple, reasonable errors in routines that are at a deep scope level within the program. The ESs were set to encompass the entire routine containing the bug, to demonstrate how the system performs when fault localization is only able to indicate the routine that contains the error

The programs used from Arcuri's work were *remainder* and *triangleClassification*. These were selected because they were some of the larger programs considered by Arcuri (in terms of tree nodes) and they are relatively straightforward programs, used in a wide variety of software engineering research. Unfortunately, the details of the specific bugs used in the Arcuri's study were not given, making a true comparison between the systems impossible. Bugs were seeded into these programs for the presented study. The details of the bugs used are given in Figure 5.12.

Given the three bugs used for each program, say *bugA*, *bugB*, and *bugC*, three versions of each program were made, $p1$, $p2$, and $p3$. Program $p1$ contains

remainder

```
1:  int main(int argc, char* argv[]) {
2:   int num, denom;
3:   int R = 0, Cy = 0, Ny = 0;
4:   num = atoi(argv[1]);
5:   denom = atoi(argv[2]);
6:   R = num;
7:   if(num!=0) {
8:    if(denom!=0) {
9:     if(num>0) {
10:     if(denom>0) {
11:      while((num-Ny)>=denom){
12:       Ny = Ny + denom;
13:       R = num - Ny;
14:       Cy = Cy + 1; } }
15:     else {
16:      while((num+Ny)>=abs(denom)){
17:       Ny = Ny + denom;
18:       R = num + Ny;
19:       Cy = Cy - 1; } } }
20:    else {
21:     if(denom>0) {
22:      while(abs(num+Ny)>=denom){
23:       Ny = Ny + denom;
24:       R = num + Ny;
25:       Cy = Cy - 1; } }
26:     else {
27:      while((num-Ny)<=denom){
28:       Ny = Ny + denom;
29:       R = num - Ny;
30:       Cy = Cy + 1;}}}}}
31:  cout << Cy << ” R ” << R << endl;
32:  return 0; }
```

remainder **Bugs**

v1 Line 13, replace Ny with Cy
v2 Line 22, replace >= with <=
v3 Line 6, replace num with denom

triangleClassification

```
int main (int argc, char* argv[]) {
 int a, b, c;
 a = atoi(argv[1]);
 b = atoi(argv[2]);
 c = atoi(argv[3]);
 if(a<= 0 || b<= 0 || c<=0)
  cout << ”Invalid” << endl;
 else if(a==b && b==c)
  cout << ”Equilateral” << endl;
 else if(a==b || b==c || a==c)
  cout << ”Isosceles” << endl;
 else if(a!=b && b!=c && a!=c)
  cout << ”Scalene” << endl;
 return 0; }
```

triangleClassification **Bugs**

v1 Line 6, replace second || with &&
v2 Line 8, replace c with a
v3 Line 10, replace second a with b

Figure 5.12: Details on *remainder* and *triangleClassification* Bugs

bugA, program *p*2 contains *bugA* and *bugB*, and program *p*3 contains *bugA*, *bugB*, and *bugC*. The sizes of the resulting ESs used for these programs is summarized in Table 5.5 (on page 108). The program IDs established in this table will be used in the following discussions.

Table 5.6 (on page 109) summarizes the configuration values used for the experiments conducted in this study, unless specifically noted otherwise in the discussion. These configuration values were selected with minimal forethought, as parameter tuning is typically a difficult, time consuming process that, if required, would reduce the practicality of the system. The *Normal Operation* probability distribution for test case operators is used for all EAs in CASC except during the verification process, in which the *Exploration Focused* probability distribution provided automatically by the system is used.

Table 5.5: Summary of Programs and Bugs used in the Study

ID	**Program**	**Bugs**	**LOC for ES(s)**	**# Nodes in ES(s)**
R1	*replace*	v11	53	144
R2		v11, v13	84	213
R3		v11, v13, v33	113	294
P1	*printtokens2*	v4	85	193
P2		v4, v6	208	233
P3		v4, v6, v7	246	307
M1	*remainder*	v1	59	154
M2		v1, v2,	59	154
M3		v1, v2, v3	59	154
T1	*triangleClassification*	v1	16	85
T2		v1, v2	16	85
T3		v1, v2, v3	16	85

In the following sections, the PSTC objects used for each program are described. As the CASC system relies on expected output to detect malfunction, most

Table 5.6: Configuration Details for Experiments

Generations		**MOOP**	
Verification	1500	Fit. Share. Alpha	1
Engagement	200	Fit. Share. Sigma	0.15
Correction	150		
CS Generation	150		

Program Population		**Test Case Population**	
Population Size	200	Population Size	100
Children per Gen.	200	Children per Gen.	100
Tourn. Size	20	Tourn. Size	10
Survival Tourn. Size	40	Survival Tourn. Size	20
Copy Prob.	2.5%	*Exploration Focused:*	
Reset Prob.	5.0%	Mutate Prob.	35%
Base Crossover Prob.	20.0%	Crossover Prob.	10%
Base Mutate Prob.	70.0%	Randomize Prob.	55%
Arch. Alter Prob.	2.5%	*Normal Operation:*	
Adap. Param. Freq.	10 Gen.	Mutate Prob.	45%
Adap. Param. Reward	2.5%	Crossover Prob.	45%
Tier Width for Edits from Source Objective	5 per ES	Randomize Prob.	10%
Allowed CPU Time	1 sec.		

of the presented PSTCs were designed using a back-to-front design scheme , with the *remainder* PSTC being the only exception (the reasoning behind this is discussed in Section 5.3.3). In the back-to-front design scheme, the expected output is generated for the test case, then the input that would yield that output is created. Following this design scheme, the core data for each test case is the expected output (rather than the more typical input). Accordingly, this is what is typically modified during test case reproduction.

Additionally, the back-to-front design scheme makes the most sense in practical application of the CASC system. If the input was the core data used for each test case, then the generation of expected output would involve, in many cases, essentially encoding the program that is being corrected into the test case. Clearly, this would be an unrealistic requirement.

5.3.1. Test Case Details: *printtokens2*. *printtokens*2 is a lexical analysis program. The token types supported are Number, Identifier, Keyword, Character, Comment, String, and Special Symbol; with a restriction on the length of comments. A test case is randomly created by first randomly selecting a token type, and then randomly generating an appropriate lexeme for the token.

Recombination of *printtokens*2 test cases is achieved using uniform crossover. Tokens and their associated lexemes are uniformly selected to form a child. The length of the child test case is chosen randomly from the inclusive selection between the lengths of the two test cases. If the child's length is greater than the shorter parent, then there is a bias towards the test case with longer length. A *printtokens*2 test case can be mutated through a change of the lexemes or token types that the test case contains, selected with even probability. When a token type of a test case is mutated, a new token type is chosen at random and the lexeme associated with it is regenerated in a similar manner to that of the initial randomization of the test case. The mutation of a lexeme of a token is more involved; depending on token type, the lexeme is modified accordingly to preserve validity for the token type. Number token lexemes are selected randomly from the range [0,100]. Keyword and Special Symbol token lexemes are selected randomly from the keywords or special characters specific to *printtokens*2. Mutation of the identifier is performed through addition, modification, or complete removal of characters in the lexeme. An important feature to note is that the randomization and mutation phases generate only valid lexemes that are accepted by the correct program.

During evaluation, input to *printtoken*2 is created by concatenating the lexemes of the test case into a space separated list. Expected output for a test case is generated based on the known token type and lexemes of the test case in question. When comparing the expected output with the actual output of a *printtokens*2 program, the objectives are:

1. The token types match
2. The lexemes match
3. The order of the token types match
4. The order of the lexemes match

Objective 1 is calculated as the percentage of token types missing in the output. The other objectives are calculated as the edit distance between lexemes (objective 2) or order of token types and lexemes (objectives 3 and 4, respectively). When using MOOP, objective 1 is maximized and the others are minimized (from the perspective of the program population). When using SOOP, the minimization objective scores are negated, and then all of the scores are summed, effectively converting the problem into a maximization problem.

A sample *printtokens*2 test case is shown in Figure 5.13.

5.3.2. Test Case Details: *replace*.

replace is a pattern matching and replacement program. A *replace* test case consists of three main parts:

1. A pattern to search for
2. A replacement string to substitute in for words that match the pattern
3. Text to search for pattern matches in

Patterns consist of single characters, wild-card characters, conditional character lists, conditional character ranges, and escape characters (tab and end-line characters). All pattern elements can also have the closure qualifier, indicating that at that point in the pattern a matching word[6] can have zero or more instances of that pattern

[6] Note that in this discussion, “words” in the input text are considered to be strings of characters separated by either space characters or end lines

Test Case:
(Character, *a*), (Keyword, *xor*), (Identifier, *bcd*), (Comment), (String, *“text”*), (Endline), (Keyword, *lambda*), (Number, *42*)

Input: (surrounding quotes added)

```
“a xor bcd ; “text”
lamda 42”
```

Expected Output:

```
identifier,“a”.
keyword,“xor”.
identifier,“bcd”.
keyword,“lambda”.
numeric,42.
```

Figure 5.13: Example printtokens2 Test Case

element. The conditional character pattern elements indicate a set of characters that are acceptable at that point in the program. These elements can have the negation qualifier to indicate that all characters except those in the set can be used at that point. Patterns (as a whole) can also have *bol* and *eol* qualifiers, which indicate that to match the pattern a word must be either at the beginning or end of a line in the text, respectively. The replacement string consists of either characters or *ditto*’s (represented by the & character). A *ditto* indicates that when replacing a matched word we insert the matched word itself in the replacement string at that point. For example, if the replacement string “->&<-” was used and the word “*abc*” was matched, the output would be “->*abc*<-”. The text to be searched for pattern instances (i.e., the target text) is composed of words generated specifically to be either instances or non-instances of the pattern. The expected output is generated by exploiting the fact that all words in the target text are known to be instances or non-instances.

Uniform crossover is used for all three components of the test case. When performing mutation, one of the three components is selected (with equal probability) and then one mutation operation is performed on that component (the operations have equal probability of selection, as well). The pattern can be mutated by swapping locations of pattern elements, deleting a pattern element, randomly altering the text for a pattern element, adding a random pattern element, altering the qualifier(s) for a pattern element, and altering the *bol* or *eol* qualifiers for the pattern. The replacement string can be mutated by swapping two characters in the string, removing a character, randomly changing a character, and adding a character to the string. The target text can be mutated by swapping two pattern instance locations, removing a pattern instance (adding a non-instance), swapping the location of a pattern instance and a non-instance, generating new text for a pattern instance, and adding a pattern instance (removing a non-instance).

Expected output for a test case is generated by replacing the known pattern instances in the input text with the appropriate replacement text string. When comparing the expected output with the actual output of a *replace* program, the objectives are:

1. Non-pattern-instances in the input text are not altered

2. Pattern-instances in the input text are replaced with the replacement string

After a program individual runs, the words in the output are read in and stored as a list. Then the expected output for the test case is generated and stored in the same manner. The list of expected words is then iterated and the edit distance between the expected output and the actual output is summed, where the sum for pattern-instances and non-instances is kept separate. After all words have been iterated, the resulting sums are reported as scores for the appropriate objective function. Accordingly, the program population attempts to minimize these scores, while the test case population is maximizing them. If SOOP is used, then these scores are summed and returned as the fitness for the run. The resulting value is then negated for program individuals (since CASC assumes the program population is always maximizing fitness when performing single-objective optimization).

An example of a *replace* test case is shown in Figure 5.14.

Pattern:
(Character, “*a*”), (Wildcard), (Character List, “*bcd*”), (Character Range, “*1-9*”, Closure), (Character, “*!*”)

Replacement Text: “`-->&<--`” (where the & character indicates a *ditto*)

Target Text:
(“`qw$gh^A`”), (“`zxXcv`”), (“`a|b1234!`”, instance), (“`+bvAsDw`”), (Endline),
(“`aad!`”, instance), (“`yu#aB`”), (“`a\(d3!`”, instance), (Endline),
(“`Tbc>?`”)

Expected Output:
```
qw$gh^A  zxXcv  -->a|b1234!<--  +bvAsDw
-->aad!<--  yu#aB  -->a(d3!<--
Tbc>?
```

Figure 5.14: Example replace Test Case

5.3.3. Test Case Details: *remainder*. The *remainder* program takes as input two integers (a numerator and a denominator), and outputs the quotient and remainder from dividing the input values. There are five basic cases to consider in the *remainder* program: the four possible permutations of signs for the numerator and denominator and the case when either of the inputs are zero.

As mentioned earlier, the *remainder* PSTC does not use the back-to-front design scheme. Quotient calculation for integer division is a primitive operation in C++, and so that was used to determine the expected quotient for a given numerator and denominator. This result was then used to calculate the remainder for the division using naive calculation[7] (i.e., remainder = numerator - (denominator * quotient).

remainder test cases are mutated by simply randomly selecting a new sign and a new value for the numerator or denominator. Recombination of *remainder* test cases uses uniform crossover between the parent test case's numerator and denominator values.

The objectives used for the *remainder* program are:

1. The correct quotient should be specified by the program
2. The correct remainder should be specified by the program

Both objectives are scored by taking the absolute value of the expected value minus the output value. Accordingly, these objectives are both minimization objectives, from the perspective of the program population. When using SOOP, the objective scores are summed and negated.

An example *remainder* test case is shown in Figure 5.15.

Input: (Numerator, "42"), (Denominator, "-13")

Expected Output: (Quotient, "-3"), (Remainder, "3")

Figure 5.15: Example remainder Test Case

5.3.4. Test Case Details: *triangleClassification*.

The *triangleClassification* program takes as input 3 integers indicating the lengths of a sides of a triangle and outputs the type of triangle indicated (equilateral, isosceles, or scalene), using the tightest classification possible. If any of the side lengths are less than or equal to zero

[7] This approach was used instead of basic modulus because the *remainder* program operates slightly different than C++ modulus in some of the aforementioned cases.

then the program indicates that invalid input has been provided. The core data item for *triangleClassification* test cases is the triangle type expected, including invalid. The type is selected first, then appropriate side lengths are randomly generated.

Mutation of $triangleClassificaion$ test cases is performed by selecting a new type, and then randomly changing as few of the sides as possible to represent the new type. When performing recombination of *triangleClassification* test cases, the type of the child test case is randomly selected from within the set of types covered by the parent test cases. That is to say, if the parent test cases are the same type, then the child will be that type. If the one parent is of type equilateral and the other is of type scalene, then the child can be any type except invalid (as equilateral triangles are also isosceles). If either of the test cases are invalid, then the child can be any type.

There is only a single objective for the *triangleClassification* program: the correct classification should be given. Program individuals attempt to maximize this objective, and it is scored using Table 5.7, where the vertical axis is the expected type and the horizontal axis is the indicated type. Given the way that the *triangle-Classification* program is written, it is possible that a program could be created that gives no output; when this is the case, the indicated type is NA in the table. This table was designed to capture how close to correct a classification is by taking into account how much more incorrect it could be. For example, if a test case represents an equilateral triangle, but is classified as scalene, then this result is scored 0.25. The rationale behind this is that at least the program indicated that it was a triangle, rather than indicating invalid or simply providing no output at all.

5.3.5. General Experimentation Results. Each program was run 30 times using MOOP and SOOP, totaling in 720 runs. Figure 5.16 shows the percentage of runs that yielded a solution for each program using both MOOP and SOOP. Figure 5.17 shows the success rate for the runs ordered by the number of bugs present

Table 5.7: Scoring Table Used for triangleClassification

	Equ.	Isc.	Scl.	Inv.	NA
Equ.	1.00	0.75	0.25	0.00	-1.00
Isc.	0.75	1.00	0.50	0.00	-1.00
Scl.	0.25	0.50	1.00	0.00	-1.00
Inv.	0.00	0.00	0.00	1.00	-1.00

in the program. With a single bug present, the system performed quite well on all programs, achieving an average 86.25% success rate on these runs. With two bugs the average success rate falls, to 74.58%. With three bugs present, the system really struggled, achieving only 26.88% average success rate. This highlights the system's need for the addition of partial solution identification techniques.

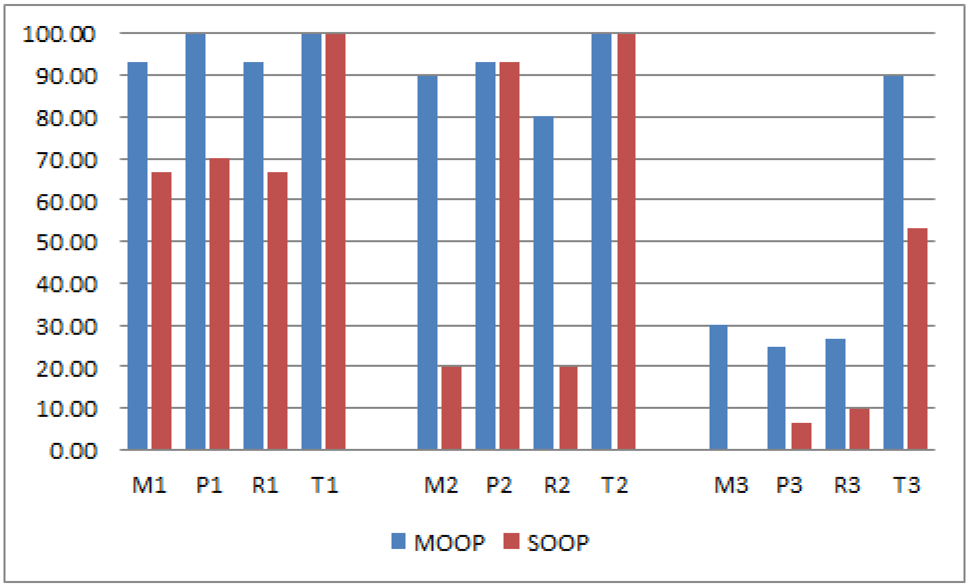


Figure 5.16: Percentage of Runs Yielding a Solution in General Experiments Ordered by Program

MOOP outperformed SOOP in the majority of the programs considered, and performed at least as good for all programs. This was the expected result, as even when the problem objectives are not conflicting, MOOP can still be expected to perform at least as good as SOOP. If the three bug runs are omitted, MOOP achieved an average success rate of 93.75%, whereas SOOP only achieved 67.08%. Both optimization methods struggled when there were three bugs present in the program, though MOOP still performed better than SOOP on average in these cases.

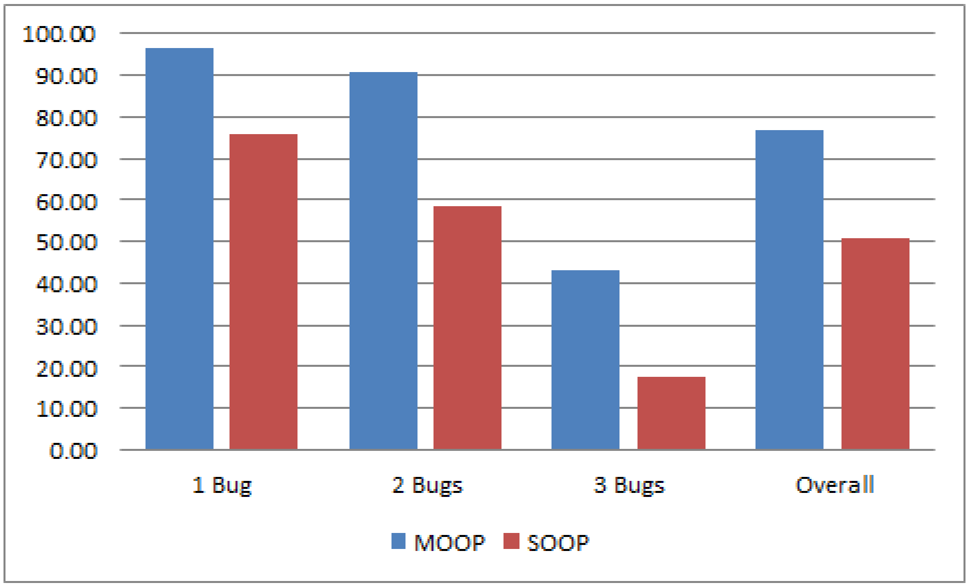


Figure 5.17: Percentage of Runs Yielding a Solution in General Experiments Ordered by Number of Bugs Present

Solutions presented by the system were subjected to manual testing to determine if the solution was a true solution. This testing used a hand-crafted set of test cases, designed specifically to demonstrate functionality implemented in the ES(s) for the bugs (not just the functionality affected by the bug(s)). The output of the solutions for these tests was compared that of the original unmodified program and if no differences were shown, then the solution was a true solution. Figure 5.18 shows the percentage of solutions presented by the system that are true solutions ordered by the program used for the runs. Figure 5.19 shows this percentage as well, except ordered by the number of bugs present in the program used for the runs.

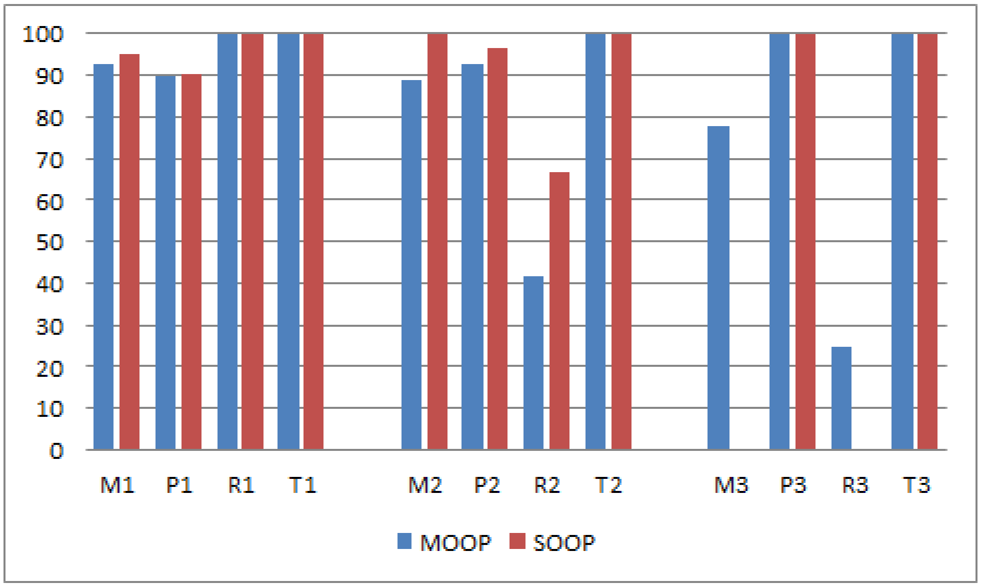


Figure 5.18: Percentage of Solutions Yielded in General Experiments that are True Solutions Ordered by Program

Programs with more a complicated test case space will naturally be more prone to the presentation of false solutions, as these programs often have more border cases and/or rarely traversed execution paths. For example, the test case space for the *triangleClassification* program is defined by a finite set of relationships between the input values. The test case space for the *replace* program, however, is dramatically more complex, defined by pattern elements, pattern element order, pattern modifiers present, replacement text used, etc. And so, with all other aspects equal, *replace* will have a higher likelihood of producing a false solution than *triangleClassification*, since the test case space is so much larger for *replace*. This is reflected in Figure 5.18, which shows that solutions presented for *triangleClassification* were always true solutions and the number of true solutions found for replace decreased as the number of code elements increased.

The percentage of solutions generated that are true solutions is indicative of the CASC verification system's performance. Like the system's correction module, the verification module performed very well with up to two bugs present in the program, achieving an average true solution rate of 96.04% with one bug and 85.81% with two bugs. The performance of the verification system cannot be effectively assessed when three bugs were present, since few solutions were presented overall by the system.

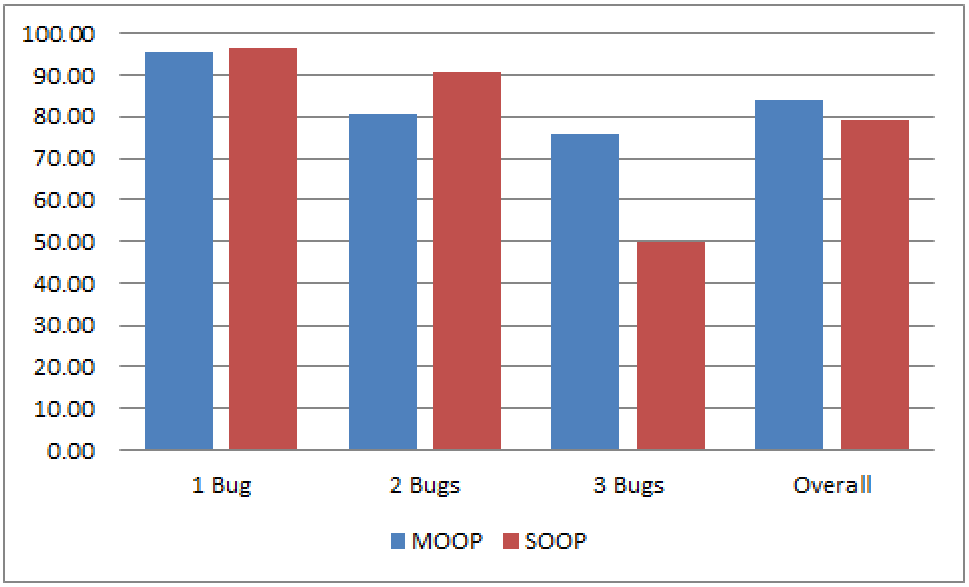


Figure 5.19: Percentage of Solutions Yielded in General Experiments that are True Solutions Ordered by Number of Bugs Present

Figure 5.20 shows the average number of verification cycles used in the runs that yielded a solution. The values shown indicate the number of times the system entered the verification EA in the $TestingandVerification$ module. All runs that yield a solution result in at least one verification cycle; values greater than one for this statistic indicate how often false positive solutions were identified and rejected by the system. Where Figure 5.18 and Figure 5.19 indicate how often the verification system did not catch a false solution, Figure 5.20 indicates how often these solutions were caught by the system and then system went on to find another solution.

In general, the *remainder* and *triangleClassification* programs used more verification cycles than the other programs for all configurations. This is due to the rate at which the test case population is able to converge on a genotypic structure that generates an error in the prevailing members of the program population. Both the *remainder* and *triangleClassification* test case spaces are much smaller than that of *printtokens2* and *replace*, making convergence much easier. With rapid convergence possible, the test case population will quickly cluster around identified optima, then when a program is identified that corrects the error being focused on, it will pass all of the test cases that were exploiting the error. However, while correcting the original error, additional errors could be created and not identified, due to the focused nature of the test case population. And so, when the candidate solution is identified and the system goes into the *Testing and Verification* module the error introduced during correction is identified trivially, since the test case population is no longer focused on a specific genotypic configuration.

Figure 5.21 displays box plots for the number of evaluations performed when the solution for a run was found. The boxes shown are bounded by the first and third quartiles of this value, the cross is at the median, and the endpoints of the lines on the top and bottom of the boxes are at the max and min value, respectively. An evaluation is defined as a single program-test case pairing. This gives an overview of the effort used to generate a solution for the problems considered. For the one and two bug problems, most solutions were presented in less than five million evaluations. The *remainder* runs generally took more evaluations to generate a solution than the others. This is due to both the number of verification cycles used in these runs and the arithmetic nature of the program (making it very sensitive to arbitrary modification).

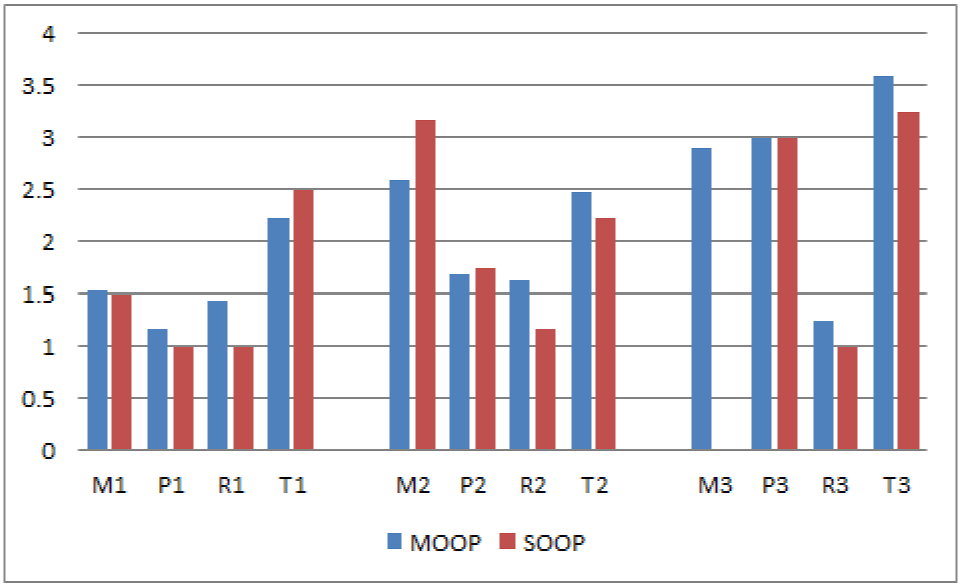


Figure 5.20: Average Number of Verification Cycles Used in Successful Runs

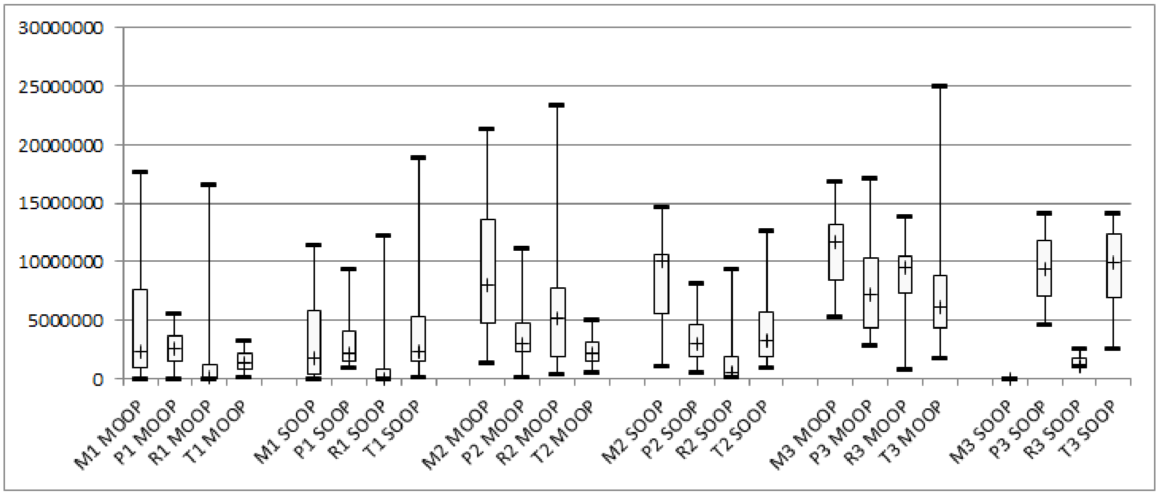


Figure 5.21: Box Plot for the Number of Evaluations Used to Generate a Solution

Figure 5.22 shows box plots for the CPU time used in all of the experimental runs, in seconds. The majority of runs finished in less than 20000 seconds (i.e., approximately five and a half hours). A large amount of time during runs is spent waiting for non-terminating programs to be killed, which is currently done at the second resolution. These times could be reduced through higher resolution program timing and the addition of more search guidance to the system.

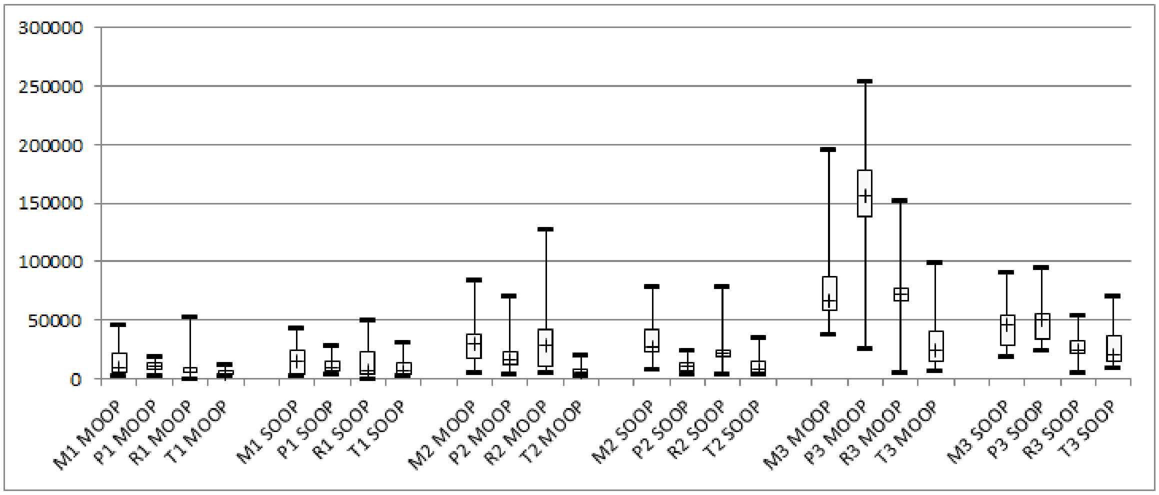


Figure 5.22: Box Plot for the CPU Time Used for the Experimental Runs in Seconds

5.4. SCALABILITY EXPERIMENTATION SETUP

The full search space of possible programs is infinite; however, when correcting software it can be assumed that the incorrect program is not far from the correct program [29]. With this assumption, the search space for CASC can be loosely approximated as:

$$S = (mN)^k \tag{7}$$

where N is the number of nodes (i.e., atomic code elements) in the incorrect program, m is the number of modifications that can be performed on a node, and k is the minimum number of modifications needed to reach a solution [9].

Reduction of k would have the most affect on the search space; unfortunately, the k value cannot be controlled. However, for 50% of bugs the value of k is 10 or less; for 95% of bugs k is 50 or less [81, 26]. Reduction of the m value would also result in search space reduction; however, it would also reduce the scope of CASC's effective application, since reducing m reduces the number of tools CASC has at its disposal.

The N value, however, can be reduced by limiting the portion of the source program that CASC considers. In previous versions of CASC, the section of code considered by the system was indicated using special guard statements recognized by the CASC parser, which were set manually. N can be reduced by more aggressively setting the guards at the risk of leaving some, or even all, of the incorrect code elements out. Fault localization techniques can be used to increase the confidence of guard placements.

A preliminary scalability study for CASC was presented in 2011 [105]. A buggy version of bubble sort presented by Arcuri [11] was used as the base program in this study (specifically bug 4 from the cited publication). At the time, CASC supported only SOOP. The fitness function used is shown in Algorithm 1 (where the *numMissing* function returns the number of elements in *input* that are missing in *output*). Ten versions of this program were created in which each version has an additional line included between the guard statements marking the buggy code, e.g., *OneLine* has just the buggy line of code included between the guards, $TwoLines$ has the same line as *OneLine* plus one more non-buggy line, $ThreeLines$ has the same lines as $TwoLines$ plus one more non-buggy line, etc.. The effect that adding a line has on the N value varies depending on the line that was added. Table 5.8 shows the N values for the generated source programs.

For each version of the source program, 50 experimental runs were executed. The parameters used for the experiments are summarized in Table 5.9.

Table 5.8: Number of Nodes (**N**) in Source Programs

OneLine	4	*TwoLines*	11	*ThreeLines*	15
FourLines	23	*FiveLines*	37	*SixLines*	49
SevenLines	58	*EightLines*	64	*NineLines*	67

Table 5.9: Parameters for Scalability Experiments

Num. Generations	400
Program Population Parameters	
Pop. Size	50
Programs Created Per Gen.	50
Selection Tourn. Size	5
Survival Tourn. Size	10
Copy Probability	2.5%
Reset Probability	5%
Architecture Alter Probability	2.5%
Crossover Probability	45%
Mutation Probability	45%
Off-by-One Mutation Bias	10%
Nodes Altered in Mutation	1
Copy/Mutation Update Freq.	10 Gen.
Copy/Mutation Update Reward	2.5%
Test Case Population Parameters	
Pop. Size	50
Seeded Test Cases	5
Test Cases Created Per Gen.	25
Selection Tourn. Size	10
Survival Tourn. Size	10
Mutation Rate	10%
Values Changed in Mutation	1

5.4.1. Previous Scalability Results.

This section presents the results from the scalability study for CASC presented in 2011 [105]. Ideally, the CASC system should be able to correct the buggy program as quickly as possible on every run. Accordingly, the scalability study is focused on the success rate of the system and the birth generation of solutions in successful runs.

The results for the experiments are summarized in Table 5.10. A linear trend line can be generated for the success rate (shown in this table) with an associated R^2 value of 0.9076; based on this is evidence it can be hypothesized that there exists a linear correlation between success rate and problem size. A sub-linear correlation seems to also be possible; as the problem size increases, the success rate is reduced by decreasing amounts (with the exception of the *SixLines* experiments, which performed unexpectedly well). This possibility is confirmed by the generation of a logarithmic trend line, which produces nearly the exact same R^2 value (0.9075).

Table 5.10: Results Summary for 2011 Scalibility Study

	Success Rate	**Average Birth Gen. of Solution**
OneLine	100%	15
TwoLines	98%	26
ThreeLines	88%	36
FourLines	78%	123
FiveLines	68%	135
SixLines	76%	126
SevenLines	64%	174
EightLines	62%	118
NineLines	60%	190

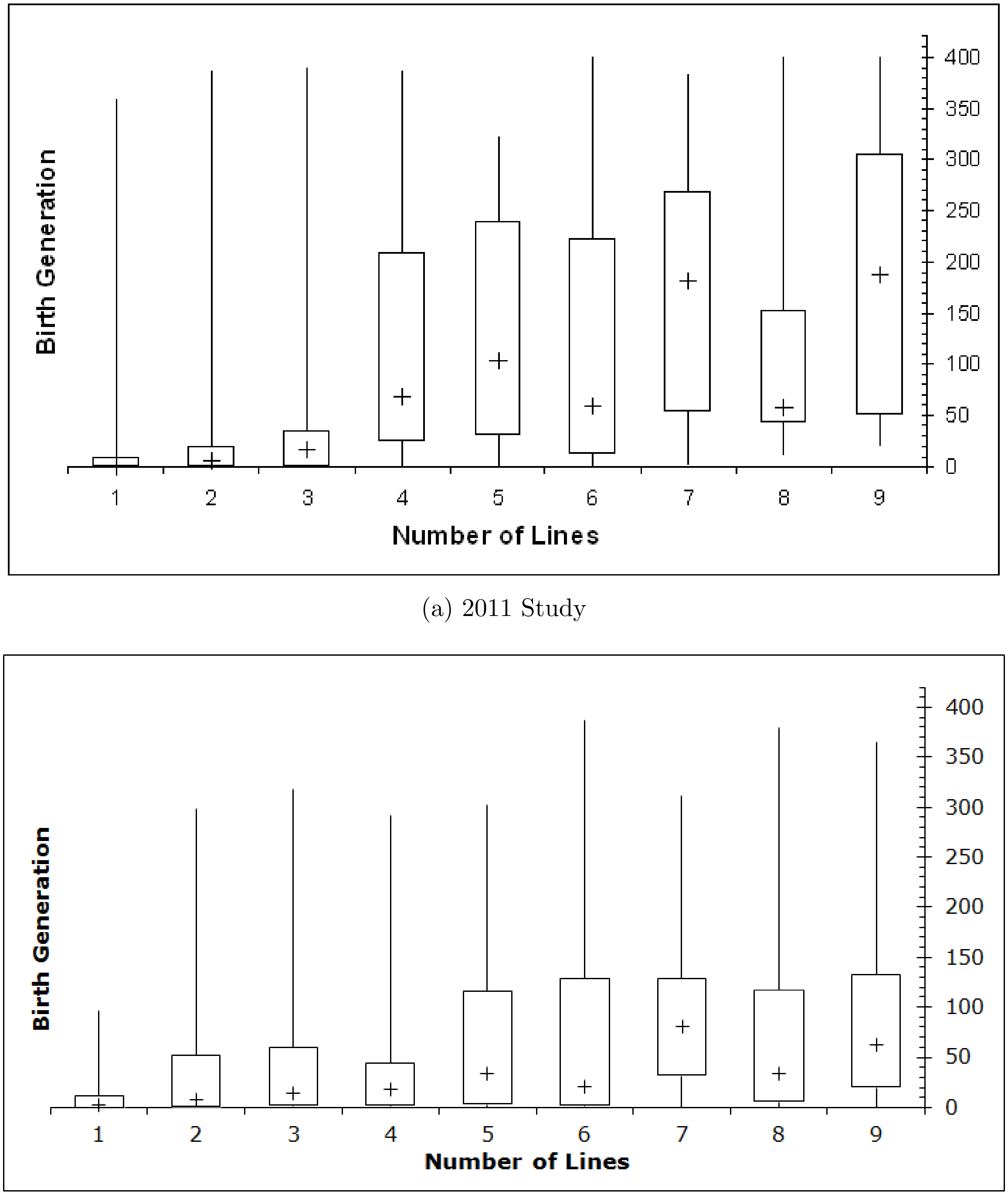
Figure 5.23 gives an overview of the distribution of birth generations for valid solutions (based on the minimum and maximum generations observed and first, second, and third quartiles) for the scalability studies, with that of the original study shown in Figure 5.23a. As discussed previously, as the number of code elements increases, the number of successful experiments decreases. This impacts the statistical confidence for the higher node count experiments; however, each experiment contributed at least 30 experiments to Figure 5.23a.

In the version of the CASC system used in the previous study, individuals were ranked first by fitness, then by node count. This allowed for the promotion of equally fit individuals with fewer nodes over those with more nodes. Using this approach, it was possible for a solution to be found early on in the search, then replaced in the last generation by a new solution that has one less node. This situation is believed to be the cause of the high maximum generations reported in Figure 5.23a.

For the first three programs, 75% of the solutions were found in less than 50 generations. With the addition of the fourth line, 75% of the solutions were found in less than 210 generations, which is a significant increase given that the fourth line only adds eight nodes to the source program. The fourth line is a branch statement whose *then* clause contains the other three lines included in the program. It is likely that the observed increase in convergence rate is due to the relationship that the fourth line has with the other three lines. Modifications made to line four can determine whether or not the other three lines are executed; as such modifications made to the other three lines may not impact the performance of the program at all.

The experiments for programs $FourLines$ through $NineLines$ roughly perform along a steadily increasing trend, with the exception of the $EightLines$ experiments. There is no clear reason for the distribution of these results to be so notably different from the expected values (relative to the performance of the other experiments).

The average birth generation of valid solutions is indicative of the convergence rate of the CASC system. The average birth generation of valid solutions for these experiments is shown in Table 5.10. A plot of these values for both scalability studies is shown in Figure 5.24, with that of the original study shown in Figure 5.24a. Also shown in this figure are generated linear and logarithmic trend lines for these values, whose R^2 values are 0.8186 and 0.848, respectively. This result indicates that the estimated convergence rate of the previous version of the CASC system as problem size increases is at worst linear, and very possibly sub-linear.

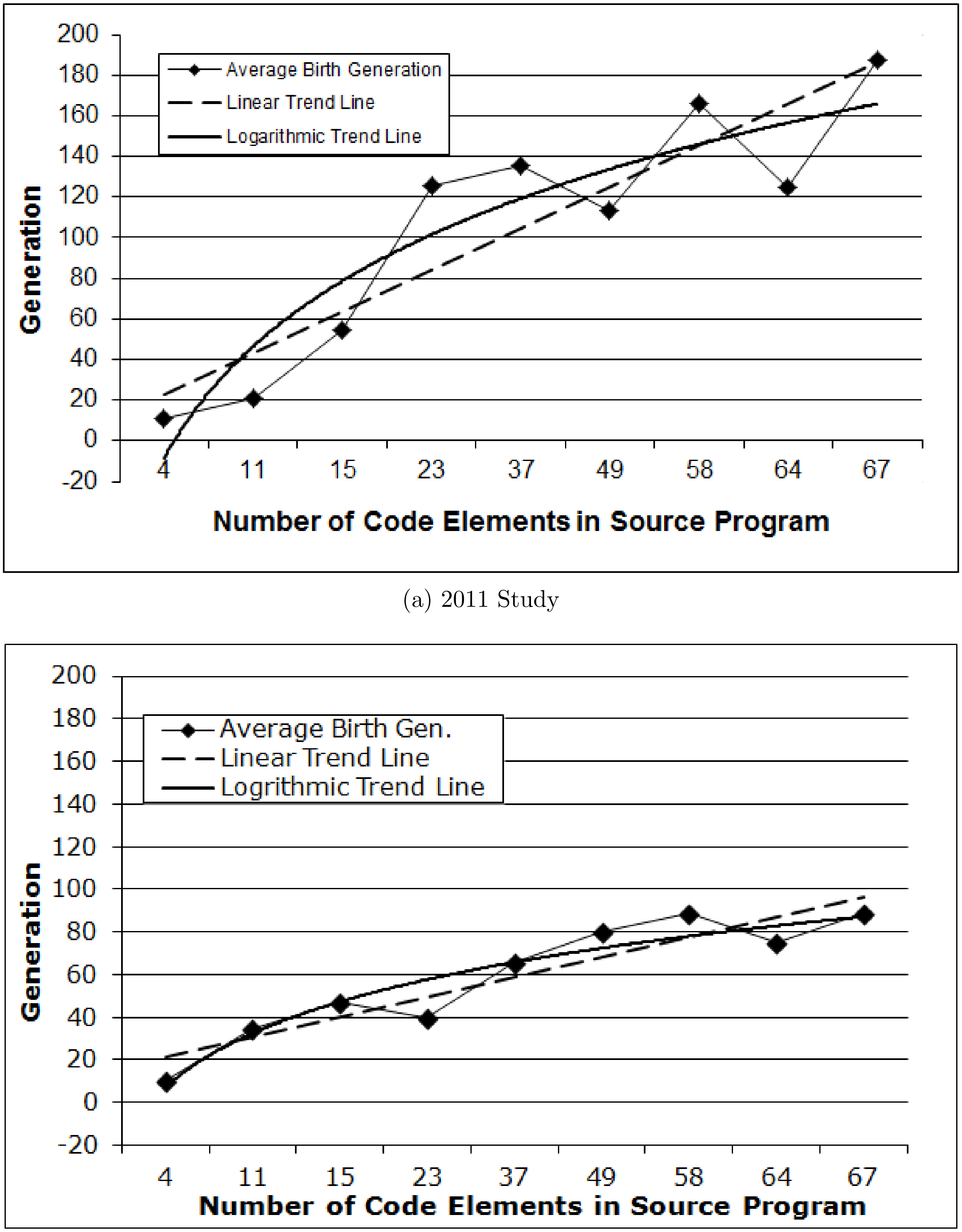


(a) 2011 Study

(b) 2012 Study

Figure 5.23: Box Plots of Solution Birth Generation for Scalability Studies

5.4.2. New Scalability Experimentation Results. The 2011 scalability study was redone for the version of the CASC system described in this dissertation.



(a) 2011 Study

(b) 2012 Study

Figure 5.24: Trend Lines Generated for Average Solution Birth Generation for Successful Experiments in Scalability Studies

Since the original study, CASC has had a number of high impact changes made, in terms of the search space being navigated. Most notably (in the context of this study), the system now represents a much larger portion of the C++ language and supports many more possible code modifications. With these enhancements the system can handle a much wider variety of programs and bugs, but at the cost of a much larger search space.

The CASC system used in the 2011 study supported 32 node types for program tree construction. In the 2011 version, variable names were discovered by the system when they were used in code (as opposed to the declarations being specifically sought out). The code modifications possible were similar to what is used in the current version, but were less guided.

The version of the CASC system described in this dissertation supports 59 node types for program tree construction. Even though the programs used in the study are the same (i.e., contain the same base code elements), code elements not already present in the source code can be introduced during code modification. And so, with more types that can be generated (as described in Section 5.2.4.4), this represents a significant increase in search space from the version used in the 2011 study. Additionally, the CASC system now identifies all names available in the ES through complete source code parsing. Each variable name identified is essentially another node type available for use in code generation; as such, each name identified results in an increase in search space. These increases in search space are contrasted by more guidance added to the search. The major sources of search guidance are the identification and exploitation of apparent code intention, variable type and qualifier sensitivity, and finer granularity in performance assessment through MOOP.

The goal of reproducing this study is to provide a grounds for comparison between the version of the CASC system used in the original study and the current version. This comparison provides insight into how well the current version of the

CASC system is handling the increase in problem space by using the previous study as a benchmark.

The same parameter values used in the original study (shown in Table 5.9 on page 125) were used when appropriate; otherwise, the parameter values used in the general experiments were used (shown in Table 5.6 on page 109). MOOP was used in the new study, since it was shown to be generally at least as good as SOOP, and better in many cases.

Table 5.11 summarizes the results from the new study. When compared to the previous results summary in Table 5.10 on page 126, it is clear from the observed success rate that the system was able to much more consistently find solutions, despite the increased problem space.

Table 5.11: Results Summary for 2012 Scalibility Study

	Success Rate	**Average Birth Gen. of Solution**
OneLine	100%	10
TwoLines	98%	35
ThreeLines	98%	47
FourLines	94%	40
FiveLines	92%	66
SixLines	88%	80
SevenLines	94%	89
EightLines	92%	75
NineLines	100%	89

Figure 5.23b provides a summary of the birth generations for solution programs found in successful runs of the new experiments. The generation count used to create this figure only includes generations in the *Testing and Correction* module, since that is the location where the program search space is being navigated in the CASC system.

In these plots, the cross symbols indicate the median of the recorded birth generations and the bottom and top of the boxes are the first and third quartiles. In general, the boxes for the new study are smaller and lower in the plot. Smaller boxes indicate more consistency in the data, while boxes that are lower indicates that solutions are found earlier in the run.

As discussed in the previous section, the addition of the fourth line introduced a great deal of interdependence between the lines being evolved; resulting in a significantly more complex search space. This increase in complexity is reflected in the solution birth generations for the 2011 study. The plots for the 2012 study, however, do not indicate that the system has any issue dealing with this increase in search space complexity. This can likely be attributed to the addition of context sensitivity to the code modifications supported. With this addition, the system is able to identify apparent intent in the code, and perform modifications accordingly. In the context of this study, this results in increased sensitivity to the fact that the fourth line is a branch statement, and that the condition for the branch should be modified with that in mind.

Figure 5.24b shows a plot of the average birth generation for solutions in this study. In the same manner as the 2011 study, trend lines have been added to this plot to approximate the system's rate of convergence on a solution. Again, it is clear that system performed much more consistently in 2012 study, as the line plot of average solution birth generation is much smoother than in the 2011 study. Both linear and logarithmic trend lines are also shown in this plot. The R^2 value for the linear trend line is 0.8671 and was 0.9055 for the logarithmic trend line. The higher R^2 values (relative to the 2011 study) indicate a tighter fit for the trend lines to the data. This indicates that the current CASC system holds more tightly to the hypothesis that the convergence rate for the system is at worst linear, and very possibly sub-linear with program size.

6. CONCLUSION

This dissertation discusses research into fitness function design, fitness guided fault localization, and automated software testing, correction, and verification.

The presented guide for fitness function design is targeted at non-expert practitioners, but also formalizes fitness function design and thus establishes a foundation for the rigorous investigation of this critical component of EAs. A requirement classification taxonomy for fitness function design is presented which shows in a series of steps how, in a structured manner, to transform problem specifications into a set of fitness function components and ultimately their composition into a fitness function. A series of examples illustrates the applicability of the guide to a variety of problem types. The results demonstrate the guide's ability to generate fitness functions that are competitive with expertly designed fitness functions.

A prototype of the FGFL system is presented, which consists of an ensemble of automated fault localization techniques that exploit a fitness function for the faulty software in question. Experimentation was presented that served as proof of concept for both fitness guided fault localization in general and the FGFL system in particular. The experimentation was conducted on seven programs with different seeded bugs. The results indicate that all techniques currently in the FGFL system have trouble dealing with control based faults. However, on other fault types the system performed quite well. Individually, the novel run-time fitness monitor technique performed the best out of the three techniques, resulting in 82% or more of the lines being removed from suspicion from the source program (with control faults omitted). With all techniques active, the system yielded a 90% or more reduction in lines from the source. The final conclusion is that employing a fitness function has the ability to improve the existing state of the art in automated fault localization.

The results presented here merit investigating all other state of the art automated fault localization techniques for potential enhancement employing a fitness function.

The primary focus of the research summarized in this dissertation was on automated software correction. The current version of the CASC system is described in detail. Major features of this system include: support for both single- and multi-objective optimization, automatic detection and utilization of code element relationships during correction, a polymorphic test case definition that allows for run-time test case generation and evolution, a testing and verification cycle, and support for structural testing to increase solution confidence. Experimentation is presented that demonstrates the CASC system's ability to handle problems of increasing difficulty, both in terms of problem size and the number of bugs present. On one and two bug problems (of varying sizes), the system performed well in all regards. The introduction of a third bug and the resulting increase in problem size resulted in much lower performance, clearly indicating current limits of the system. Additionally, the published CASC scalability study from 2011 was reproduced with the current system in order to investigate how well the system handled the increase in search space that came with the recent additions to the system, using the original study as a benchmark. The new study showed that the system was able to outperform the version used in the original study, despite the increase in search space.

7. FUTURE WORK

7.1. FITNESS FUNCTION DESIGN

The presented research on fitness function design has uncovered a number of avenues for further investigation to improve the provided guide.

- The next logical step is to supplement the guide with methods for determining the quality of the generated fitness functions. This would allow the guide to assess fitness functions in a quantitative manner in addition to the current qualitative approach. These methods could guide users to develop quality fitness functions using methods like fitness landscape characterization [71], fitness function approximation [46], adaptive fitness function design [64], etc.

- There is still an element of fitness function design experience that is helpful in the design process even while using the guide. This could be removed by performing focused investigation and formalization of these areas.

- Fitness functions are not used solely by EAs. A step that would widen the applicability of the outlined guide would be to generalize the design process to black box search algorithms. This would likely necessitate some reworking of current guide elements, but would overall likely be beneficial to a much wider group of researchers.

- It seems possible that generalized coding templates could be used to generate a suggested fitness function given a few problem specific details and the classifications applied. This would definitely increase the usefulness of the guide for practitioners, though it will likely be difficult to decide on a set of optimal implementations for common fitness function design needs. However, if such a set of implementations could be decided upon, it would then be possible to encode the whole process into a user guided tool that would generate a fitness function in a fully automated manner, again, dramatically increasing the usefulness of the guide.

7.2. THE FGFL SYSTEM

The FGFL system is in a state of active development. The future steps for the system are:

- Benchmarking: The next major step for the FGFL system in general is to begin testing on commonly used program test suites in order to do direct comparisons with state of the art fault localization methods. The Siemens Test Suite is the first set of programs that will be focused on, after which additional programs from the Software-artifact Infrastructure Repository will be used for further testing.
- Trace Comparison Technique: The values in the region of the LCS table that indicate the divergent path could be used to determine substrings within the divergent path that are executed in the positive test case as well. The lines in these substrings should generate less suspicion since they were executed by both positive and negative test cases. This approach would allow a gradient to be used when applying votes for this technique as well as help reduce the assumptions made regarding the behavior of the program in question.
- Trend-Based Line Suspicion Technique: Currently, this technique assumes that the fitness function is bounded. An extension to allow unbounded fitness functions could use approximated boundaries based on observed performance during testing. After all test cases have been executed, a function based on the maximum and minimum observed fitnesses could generate approximated boundaries.

Using these boundaries, the observed fitness values could be normalized to fall in the range of [0,1], after which the technique could generate suspicion and votes as normal.

Also, the experimental results presented indicate a need for a mechanism to reduce the effect that benign branching can have on the TBLS technique.

- Dynamic Slicing: Through the experimental results it was shown that the performance of a technique can be dramatically effected by errors that indirectly influence the fitness value. A possible new technique for the FGFL system could use dynamic slicing techniques to backtrack to statements that influence highly suspicious lines. This technique could be applied automatically when a notable discrepancy between technique results is detected, like, for example, the results for the fitness monitor on program SEL1 in Section 4.4.

7.3. THE CASC SYSTEM

The CASC system has a number of very promising potential avenues of research:

- The system currently spends a lot of time waiting for non-terminating programs to be killed. Accordingly, the system would benefit from a higher resolution timing mechanism, to kill long-running programs earlier.
- The CASC system would likely benefit from rigorous parameter sensitivity testing and tuning. Discovery of commonly effective parameter configurations would both improve CASCs results as well as make it a more accessible system.
- Currently the CASC system uses uniform parameter settings for many of the EAs used by the system. An investigation into the possible benefits of decoupling these EA parameter values should be conducted.

- The code relationships detected by the CASC system greatly improve its ability to perform intelligent correction. The current set of relationships detected are basic programming knowledge for human developers; it is anticipated that the addition of more advanced concepts to the CASC system will improve its performance even further.

- Currently the CASC system only handles evolving programs that use objects from the outside, i.e., from routines that are not part of these objects. Extending the system to be able to perform correction in routines that are in objects will greatly increase the application of the system.

- While the system is able to create syntactically valid programs on average 88% of the time, the incorporation of a more comprehensive definition of the C++ grammar can be expected to be of great value to the system. An ambitious method to accomplish this would be to build in support for the use of a grammer file in the system. This file would contain grammatical rules for the source language of the program being corrected. This would both ensure syntactic validity during code modification as well as provide a basis for the dynamic definition of code modification techniques.

- A more targeted testing technique would be a strong improvement to the CASC system; particularly when dealing with programs containing multiple bugs. If specific errors in the output could be detected, then the system could focus on creation of test cases demonstrating that error, allowing for focused error correction. If achieved, this would allow the system to correct errors one at a time, rather than attempting to correct them simultaneously.

- The addition of linkage learning [36] concepts to the program evolution process in CASC would allow the system to more efficiently promote partial solutions in the program genotypes. This improvement can be expected to dramatically speed up the system in general.

- There are a number of objectives that can be generally applied to the CASC system. The presented MOSP algorithm would make the incorporation of these objectives into the CASC system very straightforward, and as such would make the system generally more flexible.

BIBLIOGRAPHY

[1] T. Ackling, B. Alexander, and I. Grunert. Evolving Patches for Software Repair. In *Proceedings of GECCO 2011 - the Genetic and Evolutionary Computation Conference*, pages 1427–1434, New York, NY, USA, 2011. ACM.

[2] K. Adamopoulos, M. Harman, and R. M. Hierons. How to Overcome the Equivalent Mutant Problem and Achieve Tailored Selective Mutation Using Co-evolution. In *Proceedings of GECCO 2004 - the Genetic and Evolutionary Computation Conference*, pages 1338–1349, 2004.

[3] H. Agrawal. *Toward Automatic Debugging of Computer Programs*. PhD thesis, Purdue University, 1991.

[4] H. Agrawal, R. A. Demillo, and E. H. Spafford. Debugging with Dynamic Slicing and Backtracking. *Software Practice and Experience*, 23(6):589–616, June 1993.

[5] H. Agrawal and J. R. Horgan. Dynamic Program Slicing. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 246–256, New York, NY, USA, 1990. ACM.

[6] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong. Fault Localization Using Execution Slices and Dataflow Sets. In *Proceedings of the 6th IEEE International Symposium on Software Reliability Engineering*, pages 143–151, 1995.

[7] S. Ali, L. Briand, H. Hemmati, and R. Panesar-Walawege. A Systematic Review of the Application and Empirical Investigation of Search-Based Test Case Generation. *IEEE Transactions on Software Engineering*, 36(6):742–762, 2010.

[8] A. Arcuri. On the Automation of Fixing Software Bugs. In *ICSE Companion '08: Companion of the 30th International Conference on Software Engineering*, pages 1003–1006, New York, NY, USA, 2008. ACM.

[9] A. Arcuri. *Automatic Software Generation and Improvement through Search Based Techniques*. PhD thesis, University of Birmingham, 2009.

[10] A. Arcuri. Evolutionary Repair of Faulty Software. *Applied Soft Computing*, 11(4):3494 – 3514, 2011.

[11] A. Arcuri and X. Yao. A Novel Co-Evolutionary Approach to Automatic Software Bug Fixing. In *IEEE Congress on Evolutionary Computation, 2008*, pages 162–168, June 2008.

[12] A. Arcuri and X. Yao. Co-Evolutionary Automatic Programming for Software Development. *Information Sciences*, (In Press), 2010.

[13] C. Artho. Iterative Delta Debugging. In *Proceedings of the 4th International Haifa Verification Conference on Hardware and Software: Verification and Testing*, pages 99–113, Berlin, Heidelberg, 2009. Springer-Verlag.

[14] J. J. Barton and L. R. Nackman. *Scientific and Engineering C++: An Introduction with Advanced Techniques and Examples.* Addison-Wesley Professional, 1994.

[15] P. E. Black. "Levenshtein Distance", in Dictionanry of Algorithms and Data Structures. Accessed June 2011. http://xlinux.nist.gov/dads/HTML/Levenshtein.html.

[16] J. S. Bradbury and K. Jalbert. Automatic Repair of Concurrency Bugs. In *Proceedings of the International Symposium on Search Based Software Engineering - Fast Abstacts*, Sept. 2010.

[17] R. C. Bryce and C. J. Colbourn. One-Test-at-a-Time Heuristic Search for Interaction Test Suites. In *Proceedings of GECCO 2007 - the Genetic and Evolutionary Computation Conference*, pages 1082–1089. ACM, 2007.

[18] O. Bühler and J. Wegener. Evolutionary Functional Testing of an Automated Parking System. In *Proceedings of the 9th International Conference on Information Systems Analysis and Synthesis*, Orlando, Florida, 2003.

[19] O. Bühler and J. Wegener. Evolutionary Functional Testing. *Computers & Operations Research*, 35(10):3144–3160, 2008.

[20] J. P. Cartlidge. *Rules of Engagement: Competitive Coevolutionary Dynamics in Computational Systems.* PhD thesis, University of Leeds, 2004.

[21] M. L. Collard. Addressing Source Code Using srcML. In *IEEE International Workshop on Program Comprehension Working Session Textual Views of Source Code to Support Comprehension*, pages 3–5. IEEE, 2005.

[22] W. Cook, W. Cunningham, W. Pulleybank, and A. Schrijver. *Combinatorial Optimization.* John Wiley and Sons, 1997.

[23] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms.* The MIT Press, 2001.

[24] N. L. Cramer. A Representation for the Adaptive Generation of Simple Sequential Programs. In J. John, editor, *Proceedings of an International Conference on Genetic Algorithms and the Applications.* Carnegie Mellon University, 1985.

[25] K. Dahal, S. Remde, P. Cowling, and N. Colledge. Improving Metaheuristic Performance by Evolving a Variable Fitness Function. In *8th European Conference on Evolutionary Computation in Combinatorial Optimization*, pages 170–181, 2008.

[26] V. Dallmeier and T. Zimmerman. Extraction of Bug Localization Benchmarks from History. In *IEEE International Conference on Automated Software Engineering (ASE)*, pages 433–436, 2007.

[27] K. Deb. *Multi-Objective Optimization using Evolutionary Algorithms.* John Wiley and Sons, 2001.

[28] K. Deb. *Multi-Objective Optimization using Evolutionary Algorithms.* Wiley-Interscience Series in Systems and Optimization. John Wiley & Sons, Chichester, 2001.

[29] R. DeMillo, R. Lipton, and F. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer*, 11(4):34–71, 1978.

[30] E. Diaz, R. Blanco, and J. Tuya. Tabu Search for Automated Loop Coverage in Software Testing. In *Proceedings of the International Conference on Knowledge Engineering and Decision Support*, pages 229–234, 2006.

[31] S. Dick and A. Kandel. *Computational Intelligence in Software Quality Assurance.* World Scientific, 2005.

[32] H. Do, S. G. Elbaum, and G. Rothermel. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[33] L. Doitsidis and N. Tsourveloudis. An Empirical Study for Fitness Function Selection in Fuzzy Logic Controllers for Mobile Robot Navigation. In *Annual Conference on IEEE Industrial Electronics*, pages 3868–3873, Nov. 2006.

[34] S. Ghazi and M. Ahmed. Pair-Wise Test Coverage Using Genetic Algorithms. In *Proceedings of The 2003 Congress on Evolutionary Computation*, volume 2, pages 1420–1424, 2003.

[35] D. E. Goldberg and J. Richardson. Genetic Algorithms with Sharing for Multimodal Function Optimization. In *Proceedings of the International Conference on Genetic Algorithms and their application*, pages 41–49, Hillsdale, NJ, USA, 1987.

[36] B. W. Goldman and D. R. Tauritz. Linkage Tree Genetic Algorithms: Variants and Analysis. In *Proceedings of GECCO 2012 - the Genetic and Evolutionary Computation Conference*, New York, NY, USA, 2012. ACM.

[37] M. Harman. The Current State and Future of SBSE. In *2007 Future of Software Engineering*, FOSE '07, pages 342–357, Washington, DC, USA, 2007. IEEE Computer Society.

[38] M. Harman, S. G. Kim, K. Lakhotia, P. McMinn, and S. Yoo. Optimizing for the Number of Tests Generated in Search Based Test Data Generation with an Application to the Oracle Cost Problem. In *Proceedings of the International*

Conference on Software Testing, Verification and Validation Workshops, pages 182–191, 2010.

[39] M. Harman, S. A. Mansouri, and Y. Zhang. Search Based Software Engineering: A Comprehensive Analysis and Review of Trends Techniques and Applications. Technical Report TR-09-03, Department of Computer Science, King's College London, Apr. 2009.

[40] M. Harman, U. Ph, and B. F. Jones. Search-Based Software Engineering. *Information and Software Technology*, 43(14):833–839, 2001.

[41] W. D. Hillis. Co-Evolving Parasites Improve Simulated Evolution as an Optimization Procedure. *Physica D*, 42(1-3):228–234, 1990.

[42] J. Holland. *Adaptation in Natural and Artificial Systems.* MIT Press, Cambridge MA, 1992, 1st Edition: 1975, University of Michigan Press, Ann Arbor.

[43] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the Effectiveness of Dataflow- and Control-Flow-Based Test Adequacy Criteria. In *Proceedings of the International Conference on Software Engineering*, pages 191–200, May 1994.

[44] T. Jansen. On the Classification of Fitness Functions. Technical report, University of Dortmund, 1999.

[45] Y. Jia and M. Harman. Constructing Subtle Faults Using Higher Order Mutation Testing. In *Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 249–258, Sept. 2008.

[46] Y. Jin. A Comprehensive Survey of Fitness Approximation in Evolutionary Computation. *Soft Computing*, 9(1):3–12, 2005.

[47] Y. Jin and J. Branke. Evolutionary Optimization in Uncertain Environments - A Survey. *IEEE Transactions on Evolutionary Computation*, 9(3):303–317, 2005.

[48] B. Jones, H.-H. Sthamer, and D. Eyres. Automatic Structural Testing Using Genetic Algorithms. *Software Engineering Journal*, 11(5):299–306, Sept. 1996.

[49] J. A. Jones. *Semi-Automatic Fault Localization.* PhD thesis, Georgia Institute of Technology, 2008.

[50] J. A. Jones and M. J. Harrold. Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 273–282, New York, NY, USA, 2005. ACM.

[51] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of Test Information to Assist Fault Localization. In *Proceedings of the 24th International Conference on Software Engineering*, pages 467–477, New York, NY, USA, 2002. ACM.

[52] T. C. Jones. Measuring Programming Quality and Productivity. *IBM Systems Journal*, 17(1):39–63, 1978.

[53] A. Kapoulkine. pugixml Procssing Library. http://www.pugixml.org/, Accessed June 2011.

[54] B. Korel. Automated Software Test Data Generation. *IEEE Transactions on Software Engineering*, 16(8):209–215, Aug. 1990.

[55] B. Korel. Automated Test Data Generation for Programs with Procedures. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 209–215, 1996.

[56] B. Korel and J. Laski. Dynamic Program Slicing. *Information Processing Letters*, 29:155–163, Oct. 1988.

[57] J. R. Koza. *Genetic Programming: On the Programming of Computers by the Means of Natural Selection*. MIT Press, Cambridge MA, 1992.

[58] J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge MA, 1994.

[59] J. R. Koza. *Genetic Programming III: Darwinian Invention and Problem Solving*. Morgan Kaufmann, 1999.

[60] J. R. Koza. *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Kluwer Academic Publishers, 2003.

[61] K. Lakhotia, M. Harman, and P. McMinn. A Multi-Objective Approach to Search-Based Test Data Generation. In *Proceedings of GECCO 2007 - the Genetic and Evolutionary Computation Conference*, pages 1098–1105, New York, NY, USA, 2007. ACM.

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

[63] J. R. Lyle and M. Weiser. Automatic Bug Location by Program Slicing. In *Proceedings of the 2nd International Conference on Computers and Applications*, pages 877–883, 1987.

[64] M. Majig and M. Fukishima. Adaptive Fitness Function for Evolutionary Algorithm and Its Applications. In *International Conference on Informatics Education and Research for Knowledge-Circulating Society*, pages 119–124, 2008.

[65] P. McMinn. Search-Based Software Test Data Generation: A Survey. *Software Testing, Verification, and Reliability*, 14(2):105–156, 2004.

[66] P. McMinn. Search-Based Software Testing: Past, Present, and Future (keynote paper). In *Proceedings of the 4th International Workshop on Search Based Software Testing*, pages 153–163. IEEE Computer Society, 2011.

[67] C. Michael and G. McGraw. Automated Software Test Data Generation for Complex Programs. In *Proceedings of the 13th IEEE International Conference on Software Engineering*, pages 136–146, 1998.

[68] C. Michael, G. McGraw, and M. Schatz. Generating Software Test Data by Evolution. *IEEE Transactions on Software Engineering*, 27(12):1085–1110, Dec. 2001.

[69] E. Miller and W. E. Howden. *Software Testing and Validation Techniques.* IEEE Computer Society, Long Beach, CA, 1978.

[70] W. Miller and D. Spooner. Automatic Generation of Floating Point Test Data. *IEEE Transactions on Software Engineering*, SE-2(3):223–226, Sept. 1976.

[71] M. Mitchell, S. Forrest, and J. Holland. The Royal Road for Genetic Algorithms: Fitness Landscapes and GA Performance. In *Proceedings of the First European Conference on Artificial Life*, pages 245–254, 1991.

[72] D. J. Montana. Strongly Typed Genetic Programming. *Evolutionary Computation*, 3(2):199–230, 1994.

[73] A. Nelson, G. Barlow, and L. Doitsidis. Fitness Functions in evolutionary robotics: a survey and analysis. *Robotics and Autonomous Systems*, 57(4):345–370, 2009.

[74] C. Nie and H. Leung. A Survey of Combinatorial Testing. *ACM Computing Surveys*, 43(2):11:1–11:29, Feb. 2011.

[75] G. Novark, E. D. Berger, and B. G. Zorn. Exterminator: Automatically Correcting Memory Errors with High Probability. *Communications of the ACM*, 51(12):87–95, Dec. 2008.

[76] M. Orlov and M. Sipper. Flight of the FINCH through the Java Wilderness. *IEEE Transactions on Evolutionary Computation*, 15(2):166–182, 2010.

[77] G. Palshikar. Applying Formal Specifications to Real-World Software Development. *IEEE Software*, 18(6):89–97, 2001.

[78] R. P. Pargas, M. J. Harrold, and R. Peck. Test-Data Generation Using Genetic Algorithms. *Software Testing, Verification & Reliability*, 9:263–282, 1999.

[79] R. Poli. *Parallel Distributed Genetic Programming, Invited Chapter in D. Corne, M. Dorigo and F. Glover (Eds), New Ideas in Optimisation*, chapter 27, pages 403–431. McGraw-Hill, 1999.

[80] R. Pressman. *Software Engineering: A Practitioner's Approach.* McGraw-Hill, 6th edition, 2005.

[81] R. Purushothaman and D. Perry. Toward Understanding the Rhetoric of Small Source Code Changes. *IEEE Transactions on Software Engineering*, 31(6):511–526, 2005.

[82] S. Remde, P. Cowling, K. Dahal, and N. Colledge. Evolution of Fitness Functions to Improve Heuristic Performance. *Learning and Intelligent Optimization: Second International Conference*, pages 206–219, 2008.

[83] A. Rodrigues, P. de Mattos Neto, and T. Ferreira. A Prime Step in the Time Series Forecasting with Hybrid Methods: The Fitness Function Choice. In *International Joint Conference on Neural Networks*, pages 2703–2710, 2009.

[84] C. D. Rosin. *Coevolutionary Search Among Adversaries.* PhD thesis, University of California: San Diego, 1997.

[85] C. D. Rosin and R. K. Belew. Methods for Competitive Co-evolution: Finding Opponents Worth Beating. In L. Eshelman, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 373–380, San Francisco, CA, 1995. Morgan Kaufmann.

[86] C. D. Rosin and R. K. Belew. New Methods for Competitive Coevolution. *Evolutionary Computation*, 5(1):1–29, 1997.

[87] E. Schulte, S. Forrest, and W. Weimer. Automated Program Repair Through the Evolution of Assembly Code. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 313–316, New York, NY, USA, 2010. ACM.

[88] T. Shiba, T. Tsuchiya, and T. Kikuno. Using Artificial Life Techniques to Generate Test Cases for Combinatorial Testing. In *Proceedings of the 28th Annual International Computer Software and Applications Conference - Volume 01*, COMPSAC '04, pages 72–77, Washington, DC, USA, 2004. IEEE Computer Society.

[89] S. Sidiroglou and A. Keromytis. Countering Network Worms Through Automatic Patch Generation. *IEEE Security & Privacy*, 3(6):41–49, Nov. 2005.

[90] S. Smith. *A Learning System Based on Genetic Adaptive Algorithms.* PhD thesis, University of Pittsburgh, 1980.

[91] U. S. A. Standards Coordinating Committee of the IEEE Computer Society. *IEEE Standard Glossary of Software Engineering Terminology.* IEEE Computer Society, Dec. 1990 (Reaffirmed in 2002).

[92] G. Tassey. The Economic Impacts of Inadequate Infrastructure for Software Testing. Technical report, NIST, May 2002.

[93] N. Tracey. *A Search-Based Automated Test Data Generation Framework for Safety-Critical Software.* PhD thesis, University of York, 2000.

[94] N. Tracey, J. Clark, and K. Mander. Automated Program Flaw Finding Using Simulated Annealing. In *Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '98, pages 73–81, New York, NY, USA, 1998. ACM.

[95] N. Tracey, J. Clark, and J. McDermid. Automated Test-Data Generation for Exception Conditions. *Software - Practice and Experience*, 30(1):61–79, 2000.

[96] I. Vessey. Expertise in Debugging Computer Programs: An Analysis of the Content of Verbal Protocols. *IEEE Transactions on Systems, Man and Cybernetics*, 16(5):621–637, Sept. 1986.

[97] C. Voudouris. Guided Local Search. Technical report, European Journal of Operational Research, 1995.

[98] J. Wegener, A. Baresel, and H. Sthamer. Evolutionary Test Environment for Automatic Structural Testing. *Information and Software Technology*, 43(14):841–854, 2001.

[99] J. Wegener, K. Buhr, and H. Pohlheim. Automatic Test Data Generation For Structural Testing Of Embedded Software Systems By Evolutionary Testing. In *Proceedings of GECCO 2002 - the Genetic and Evolutionary Computation Conference*, pages 1233–1240, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc.

[100] W. Weimer, S. Forrest, C. Le Goues, and T. Nguyen. Automatic Program Repair with Evolutionary Computation. *Communications of the ACM*, 53(5):109–116, May 2010.

[101] E. Weisstein. Inverse function. From Mathworld - A Wolfram Web Resource. Accessed August 2010. http://mathworld.wolfram.com/InverseFunction.html.

[102] J. L. Wilkerson. Co-Evolutionary Automated Software Correction: A Proof of Concept. Master's thesis, Missouri University of Science and Technology, 2008.

[103] J. L. Wilkerson and D. R. Tauritz. Coevolutionary Automated Software Correction. In *Proceedings of GECCO 2010 - the Genetic and Evolutionary Computation Conference*, pages 1391–1392, 2010.

[104] J. L. Wilkerson and D. R. Tauritz. A Guide for Fitness Function Design. In *Proceedings of GECCO 2011 - the Genetic and Evolutionary Computation Conference*, pages 123–124, New York, NY, USA, 2011. ACM.

[105] J. L. Wilkerson and D. R. Tauritz. Scalability of the Coevolutionary Automated Software Correction System. In *Proceedings of GECCO 2011 - the Genetic and Evolutionary Computation Conference*, pages 243–244, New York, NY, USA, 2011. ACM.

[106] J. L. Wilkerson, D. R. Tauritz, and J. Bridges. Multi-Objective Coevolutionary Automated Software Correction. In *Proceedings of GECCO 2012 - the Genetic and Evolutionary Computation Conference*, New York, NY, USA, 2012. ACM.

[107] W. E. Wong and V. Debroy. A Survey of Software Fault Localization. Technical Report UTDCS-45-09, University of Texas at Dallas, Nov. 2009.

[108] S. Xanthakis, C. Ellis, C. Skourlas, A. L. Gall, S. Katsikas, and K. Karapoulios. Application of Genetic Algorithms to Software Testing. In *Proceedings of the 5th International Conference on Software Engineering and its Applications*, pages 625–636, 1992.

[109] C. Yalcin. Evolving Aggregation Behavior for Robot Swarms: Evolving Aggregation Behavior for Robot Swarms: A Cost Analysis for Distinct Fitness Functions. In *International Symposium on Computer and Informational Sciences*, pages 1–4, 2008.

[110] A. Zeller. Isolating Cause-Effect Chains from Computer Programs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 1–10, New York, NY, USA, 2002. ACM.

[111] A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging.* Morgan Kaufman, 2005.

[112] A. Zeller and R. Hildebrandt. Simplifying and Isolating Failure-Inducing Input. *IEEE Transactions on Software Engineering*, 28(2):183–200, Feb. 2002.

[113] Y. Zhan and J. A. Clark. The State Problem for Test Generation in Simulink. In *Proceedings of GECCO 2006 - the Genetic and Evolutionary Computation Conference*, pages 1941–1948, New York, NY, USA, 2006. ACM.

[114] Y. Zhang, M. Harman, and A. Mansouri. The SBSE Repository: A Repository and Analysis of Authors and Research Articles on Search Based Software Engineering. 2012. CREST Centre UCL. Accessed June 2012. `http://crestweb.cs.ucl.ac.uk/resources/sbse_repository/`.

VITA

Joshua Lee Wilkerson was born in Springfield, Missouri. He graduated with honors from Strafford High School in the spring of 2001 and enrolled as an undergraduate at the University of Missouri - Rolla, now Missouri University of Science and Technology (S&T), later that fall. He graduated cum laude in summer of 2005 with a BS in computer science. He was enrolled in the S&T computer science graduate program in fall of 2005. He received his master's degree in computer science in December of 2008. He receive his Ph.D. in computer science from S&T in August of 2012. He then went on to work at the Naval Air Warfare Center, Weapons Division at China Lake, California; a part of the Naval Air Systems Command organization in the United States Department of the Navy.