Coevolutionary Automated Software Correction

Josh Wilkerson – jlw46@mst.edu
Daniel Tauritz – tauritzd@mst.edu

Proceedings of the 3rd Annual ISC Research Symposium
ISCRS 2009
April 22, 2009, Rolla, Missouri

Abstract—Testing and resulting error location and correction is a time consuming process. Test automation can help but only solves part of the problem. This paper describes a proof of concept for an approach which addresses in an integral and fully automated manner the complete cycle of testing, error location, and correction phases. The proposed system employs a coevolutionary approach where software artifacts and test cases are evolved in tandem. The test cases evolve to better find flaws in the software artifacts and the software artifacts evolve to better behave to specification when exposed to the test cases, thus causing an evolutionary arms race. Experiments are presented employing insertion sort as a simple test problem. The positive results obtained from these experiments demonstrate the potential of the presented Coevolutionary Automated Software Correction (CASC) system. Future work is outlined to address the task of determining an appropriate fitness function for a problem as well as to demonstrate the scalability of the CASC system.

I. INTRODUCTION

For a given software artifact, testing, locating the errors identified, and correcting those errors is a critical and time consuming process in software development. There has been much published on automating the testing phase in order to speed up this process, but that only addresses part of the problem. This paper introduces the Coevolutionary Automated Software Correction (CASC) system as a proof of concept for an approach which addresses in an integral and fully automated manner the complete cycle of testing, error location, and correction phases.

The approach taken with the CASC system views the problem of correcting a given software artifact as a search in the space of all software artifacts with as starting point the given software artifact and as goal point the desired corrected version. In theory, then, given an error measure which indicates how far a search point is from the goal point, the goal point can be reached by searching all points, for each point computing the error, and terminating when a point has been found with zero error. In practice, however, this is completely infeasible because computing the error over all possible test cases is typically infeasible by itself, let alone doing this for all search points until a goal point has been reached.

Evolutionary Algorithms (EAs) are a class of stochastic, population-based heuristic search algorithms inspired by biological evolution which on many real-world problems (which are typically ill-behaved in the sense of having non-concave, non-differentiable search spaces) can find high-quality solutions in a fraction of the time it would take for an exhaustive search. This makes them an ideal candidate for searching the space of software artifacts. Even EAs, though, can be defeated by an overwhelmingly large and ill-behaved search space, so while this paper focuses purely on proof of concept, the next step will be to determine the scalability of the proposed approach.

Because it is typically infeasible to compute the error over all possible inputs, a sufficiently extensive sampling of the test case space is often employed instead. Finding high-quality test cases is, however, a very hard problem in and of itself, so to perform a sampling biased towards high-quality test cases can be viewed as another search problem. EAs can again be readily applied to search this space.

The two searches in the spaces of software artifacts and test cases respectively are interdependent on each other. Namely, the quality (fitness in EA parlance) of a software artifact is now dependent on the test case sample, and vice versa, the fitness of a test case is dependent on the software artifacts it's applied to. The Competitive CoEvolutionary Algorithm (C-CoEA) is a type of EA specifically developed to solve this type of problem of competing, interdependent, evolving populations, also known as the parasite-host relationship in the CoEA literature [1], [2].

The test cases evolve to better find flaws in the software artifacts and the software artifacts evolve to better behave to specification when exposed to the test cases, thus causing an evolutionary arms race with as ultimate result a corrected version of the software artifact originally fed into the system. The CASC system exploits the reduced complexity of the fitness function relative to the software artifact to be corrected.

This paper is further organized as follows. Section II summarizes related work on applying computational intelligence techniques to software testing. This is followed by Section III which details our proposed CASC system approach, and then by Section IV which describes our experimental setup. The positive results obtained on our insertion sort test problem experiments are provided and discussed in Section V and demonstrate the potential of the proposed approach. Section VI summarizes our work and Section VII outlines our plan for future work to analyze the fitness function creation process and to demonstrate the scalability and real-world usability of the CASC system.

II. RELATED WORK

The CASC system is an application of Evolutionary Computing (EC) to software testing, a subfield of software engineering. The general type of testing that the CASC system performs is black box functional testing [3]. This is not the first time that EC has been applied to software testing. [4] presents a short survey of computational intelligence techniques, including EAs, applied
to software testing, specifically test case generation. The key to applying computational intelligence to test case generation is reducing the generation process to an optimization problem, which was first done by Miller and Spooner [5]. Their method 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 [6] and [7] 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 0 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. The Genetic Algorithm Data GEneration Tool (GADGET) [8], [9] employs a method somewhat similar to Korel’s, except that instead of trying to find a single input which will match an execution path, a set of inputs is sought which will maximize execution path coverage in the program. This is accomplished using an EA to evolve the inputs provided to the program. GADGET also supports random selection, gradient descent, and simulated annealing as alternative methods to generate test cases. The results presented in [9] show a small performance advantage for the EA. In [10] Pargas et al. present a test data generation method similar to GADGET, which employs an EA called TGen. It was shown to strongly outperform a random test data generator.

Testing of object oriented programs is currently a popular area of research. In [11], [12], [13] Wappler et al. discussed various approaches to evolutionary unit testing of object oriented software, such as the use of strongly-typed genetic programming methods in the evolutionary process or white box testing for the testing method. In [14] Tonella discusses the use of an EA to perform the unit testing of objects (e.g., classes). Tonella’s primary motivation was to test how an object performed given varying sequences of invocations of the object’s methods.

In [15] Wappler discusses many recent results which show Particle Swarm Optimization (PSO), a close relative of EC, outperforming both general and problem specific EAs as a test case generation technique for testing objects, both in terms of effectiveness and efficiency. The implied potential of employing PSO in the CASC system is discussed in Section VII.

Mantere introduced a software testing method similar to the one employed in the prototype CASC system [16], [17]. It uses a two-population coevolutionary system in which one population is a set of test cases for an application and the other is a set of various values for the parameters to be provided to the application being tested. He found that increasing the number of parameters being evolved correlated with better performance. This result supports the CASC system’s method of dynamically identifying evolvable elements as detailed in Section III.

The concept of coevolving programs and test cases is relatively new and as a result there are few publications on the topic. Adamopoulos was one of the first researchers to explore this application of coevolution in [18]. The overall goal of Adamopoulos’ research was to generate more effective program test cases. To achieve this goal an implementation 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 work of Arcuri et al. is much more similar to the CASC system. In [19] and [20] Arcuri introduces a coevolutionary system that evolves populations of programs and test cases. Like the CASC system, the goal of Arcuri’s system is to test and correct a buggy software artifact. The system uses a set of formal program specifications to generate unit tests for the software artifact in question. In its current state Arcuri’s system performs interpretive execution as opposed to actual compilation and execution of binary code.

III. CASC SYSTEM OVERVIEW AND DESIGN

The CASC system 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 software artifacts (currently C++ programs) and a population of test cases (i.e., program inputs). The fitness for each software artifact is determined by how well it performs against a set of test cases. Similarly, the fitness of a test case is determined by how well it performs against a set of software artifacts. Since each population is attempting to optimize these fitness values, an evolutionary arms race results. Figure 1 depicts the coevolutionary cycle used by the CASC system. The general flow of the system is as follows:

1) System Initialization. Prepare the system (i.e., initialize data structures, read in configuration settings, distribute relevant settings to system modules accordingly, etc.).
2) Population Initialization. The two populations (programs and test cases) are initialized/created.
3) Initial Evaluation. All of the individuals in the populations are evaluated and assigned fitness initial values.
4) Reproductive Phase. Parents are selected, crossover is performed creating new individuals, mutation is applied (if necessary) to a subset of the new individuals, and the individuals are entered into the general population.
5) Evaluation. Evaluate all individuals and assign appropriate fitness values.
6) Competition and Termination. Poorly performing individuals are selected (using modified tournament selection) and removed from their population.
7) Check Termination Conditions. If any termination conditions are satisfied, terminate the evolutionary cycle; otherwise go to the reproductive phase.

The first three steps are performed once only; the next four steps constitute the evolutionary cycle. Each population has its own separate evolutionary cycle; the cycles intersect only in the evaluation phase. The phases of the cycle are described in more detail in the following sections.

A. Program Population Initialization

The program population is based off a seed program (i.e., the program to correct) which is read in from a source file and transformed into an evolvable tree, whereas the test cases are generated randomly. Currently the CASC system supports C/C++ programs and the general section of code to correct must be enclosed by two specific comment statements, designating the code block as evolvable code to the system (this code section will be referred to as the evolvable section of the seed program).
The CASC parser is responsible for converting the seed program from written code into the parse tree representing the program’s source code. This parsing yields a lightweight parse tree representing the evolvable section of the seed program, i.e., the first individual of the program population (the parsing process is described in more detail in Section III-B). Initial population diversity is achieved by first making a clone of the seed individual and then performing a modified mutation phase on the clone; this phase will be referred to as the Initial Variation (IV) phase. The IV phase is very similar to the mutation phase (which will be discussed in detail in Section III-D). The principle difference between the two phases is that the amount that a given individual can change in the IV phase is determined randomly from a Gaussian distribution, whereas this amount is static in the mutation phase. On average, this will produce a large amount of moderately modified programs and a small amount of drastically modified programs.

B. Parsing in the CASC System

EAs are typically very computationally intensive. As with most systems which employ EAs, the computational complexity of the CASC system is dominated by its evolutionary aspect. For this reason every effort was made to reduce the computational complexity of the evolutionary processes used. A large focus was put on minimizing the overall complexity of the program Abstract Syntax Trees (ASTs, essentially a parse tree) that are used by the CASC system. The ideal structure for the program AST is lightweight and easy to traverse and manipulate.

The CASC system employs a parser generated by the ANTLR parser generator [21] as a front end to read in and parse the seed program. ANTLR works by taking in a file containing the specifications for a language grammar. Based on the grammar rules in the file, a parser is produced. The output parser can be written in a number of common programming languages. The CASC system itself is written in C++, for simplicity’s sake, that is the language that was used for the parser (i.e., a parser which parses C++ and is written in C++). The ANTLR library of objects and functions is used as a backbone for the code generated from the grammar file. The output produced by the parser (that is useful to the CASC system) is an AST. The form that the AST takes (i.e., how the parse tree(s) are actually constructed) completely depends on how the grammar is defined in the original grammar file; so for a given language there could be many possible AST representations that are all correct.

The ANTLR system is used initially to create the parsing tools necessary to parse a given language. These tools (and any output from them) are derived from the various base objects in the ANTLR libraries; so any AST yielded from parsing is reliant on the ANTLR libraries. Also, the ANTLR data structures used to hold the AST’s are large complex objects derived from multiple base classes in the ANTLR libraries; these objects contain data and functionality that is largely not useful to the CASC system. For these reasons the actual AST’s produced by ANTLR are not used directly during evolution, instead trimmed down versions of the AST’s are created by analyzing the ANTLR output AST’s. The trimmed down trees are very lightweight and are not reliant on the ANTLR system. These reduced versions of the AST’s are then passed on to the CASC system to be used in evolution (step 6 in Figure 2). Of course, this means that each new grammar that is used by the system needs to have an associated AST translator either developed or provided. This may seem like a drawback of the system; however, the alternative of using the raw AST produced from the parser would most likely require that the grammar file itself be modified to make the AST match what is expected in the CASC system, which in many cases is expected to be more time consuming that writing an appropriate AST translator.

After the parsing tools have been created (step 1 in Figure 2), a library is created that serves as the front end parsing for the CASC system. In a typical run, a source code file will first be provided to the system (step 2). Next, the source code file is preprocessed, during which the code sections to be evolved are picked out and provided to the (ANTLR) parsing tools (step 3), which produces the ANTLR AST (step 4). The AST produced by the parsing is then translated into a light-weight AST (step 5). Lastly, this AST is provided to the CASC coevolutionary system (step 6), which then takes over.

C. Test Case Population Initialization

The input for most programs can be expressed as one or more lists of values of some type. This is exactly what a general test case is defined as in the CASC system. The values for a test case
are randomly generated and assigned to an individual. There is an aspect to the test case generation that is problem specific and must be specified for each new program. Along with the type of values, there may be problem specific limitations on the actual values themselves.

D. Genetic Operations on the Program Population

New programs are produced by applying one of three genetic operators to the programs in the current program population. The genetic operators employed are reproduction, crossover, and mutation. The operator to apply is selected (using probabilities provided in the CASC configuration file), 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.

The reproduction genetic operator first selects a program from the program population (allowing future re-selection) based on fitness (i.e., tournament selection is employed). Next the selected program is copied to a new individual, which is put into the program population unaltered. This operator is also known as the copy operator; its primary purpose is to promote the genetic material of successful individuals in the population by cloning them.

The crossover genetic operator selects two programs based on fitness, with future re-selection allowed. Next, compatible subtrees are selected in the two selected program trees (i.e., a crossover point is selected). Finally, two new programs are produced by swapping the selected subtrees in the two programs. The new programs are then added to the program population. The primary purpose of this operator is to explore the problem space using the genetic material present in the program population.

The mutation operator first selects a program based on fitness, with future re-selection allowed. Next, mutation is applied by selecting a subset of compatible nodes to be altered randomly. It is assumed that the original software artifact is close to the correct implementation, so nodes that would greatly impact the structure of the program (specifically loops, branches, and assignment statements) are not considered for mutation. The effect of this operator can range from the alteration of a single node to the creation and/or destruction of entire subtrees in the program. After the specified number of nodes have been mutated, the mutant program is then added to the program population. The purpose of the mutation operator is to generate new genetic material for the evolutionary process to work with.

After the specified number of new program have been generated, the new programs are compiled and evaluated as necessary, which is described in detail in Section III-F. Redundant evaluations are not performed (i.e., programs that are cloned are determined to be the same as other programs in the current population).

At this point, the number of successes for the crossover and mutation operators is recorded, where a success is when the program produced outperforms the source program. These statistics are used to implement adaptive parameter control in the CASC system. 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. This allows the system to respond to the current state the program population is in.

E. Test Case Reproduction

At the high level, test case reproduction follows the model of typical EA crossover. Parents are first selected (using the tournament selection method described previously), crossover is performed, and the offspring is created. This process can easily be extended for more complicated test cases.

For each offspring the system randomly determines if mutation will be performed (the chance of mutation occurring is specified in the CASC configuration file, separate from the mutative chance in programs). For each test case that is to be mutated, values are randomly picked from the test case and modified according to the test case specific implementation.

F. Fitness Evaluation

The evaluation phase is the point where the two evolutionary cycles meet and the populations interact. It is assumed that for the evolving programs, each set of inputs will map to one and only one output (i.e., the programs being evolved are deterministic). A lookup table is employed to speed up the evaluation process in the event of a repeat program/test-case pairing.

Both populations follow the same general algorithm for performing fitness evaluation. For each individual in the evaluating population the following steps are performed. First a set of (unique) opponents to test the individual against is selected from the opponent population. For each opponent selected, the hash table is checked to see if the program-test case pairing has been done before; if it has, then the result is retrieved and the opponent is removed from the opponent set.

Next, for each opponent still in the opponent set, program execution occurs. After all executions have been completed, the outputs are analyzed according to the implementation specific fitness function and a fitness is applied for each trial. The pairings and results are then hashed. Lastly, the fitness for the evaluating individual is determined as the average fitness across all trials.

Prior to evaluation, programs are compiled as necessary. Compile time errors are checked for in the vent that a mutation resulted in a syntactically invalid program, if a program does not compile then it is assigned an arbitrarily low fitness and is not evaluated further. Run time errors and program time outs are also monitored throughout this process. If either occur, the offending program is assigned an arbitrarily low fitness.

Fitness evaluation is the most time consuming phase in the CASC system. It is possible to use distributed fitness evaluation to help speed up this process. In the experimentation detailed here a cluster using master-slave topology was used to reduce run-times.

G. Competition

Competition is performed using the same algorithm for both populations. The only factor considered when deciding who to remove is fitness, i.e., the new individuals in the populations are just as likely to be removed as the older individuals. For each population a reverse tournament selection is performed to determine what individuals to remove (i.e., select the individual with the lowest fitness instead of the highest).
H. Termination Criterion

The CASC system terminates if a specified fitness value is either met or exceeded by a program individual (then the run is considered a success) or if a specified number of generations is completed without finding the goal fitness. The best program found is presented as the result.

IV. EXPERIMENTAL SETUP

The development of an effective fitness function is a major obstacle for this field of research. Most of the current CASC experimentation has been focused on generating data for determining if there is an equivalency between fitness functions and, if so, what conditions must hold for such an equivalency to exist. The discovery of such a relationship would imply that if a fitness function could be developed that meant a certain set of conditions then that function would be guaranteed to eventually converge on a solution.

Also, there have been two major recent updates to the CASC system. The most influential of the two is the CASC system was updated to employ GP techniques, allowing it to control and alter nearly every aspect of the evolving programs. The other recent update is the addition of adaptive parameter control to the system (see Section III-D for details). All of the current experimentation has the secondary goal of exploring the effect that these changes have had on the system.

The current experiments are being run on a buggy implementation of insertion sort. The bugs seeded into are either basic logic errors and off-by-one errors. These errors were chosen for their simple yet common nature in development. Two buggy implementations were used for the experiments discussed below; one had only a single off-by-one error in it (relatively easy for the system correct, theoretically) and the other had the same off-by-one error as well as a logic error and an incorrect statement (making the program much more difficult to correct).

A. Fitness Functions

Three fitness functions are being used in the current experiments. In all three cases, a program can either sort data in ascending or descending order and receive a perfect fitness. The first fitness function used considers all the values ahead of and behind each value in its calculation, the second fitness function just considers the values ahead of each value, and the third fitness function just considers each value's neighbors.

B. CASC Configurations

Each experiment was run with the same configuration values, shown in Table I. The majority of these values were set using parameter tuning performed across multiple previous experiments.

V. RESULTS AND DISCUSSION

The preliminary results obtained so far have been both promising and enlightening. It was anticipated that implementing GP on the system would cause the problem space to explode in size, making it much more difficult for the system to find a solution. However, the results have been to the contrary. Against the program with a single bug in it, the system has yielded approximately a 98% success rate. Additionally, in the cases where the system found a solution, it was discovered typically within the first 20 generations.

<table>
<thead>
<tr>
<th>Configurations Common to Both Populations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max. Number of Generations</td>
</tr>
<tr>
<td>Goal Fitness</td>
</tr>
<tr>
<td>Number of Opponents During Evaluation</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Program Population Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Population Size</td>
</tr>
<tr>
<td>Number of Children Per Generation</td>
</tr>
<tr>
<td>Parent Selection Tournament Size</td>
</tr>
<tr>
<td>Competition Tournament Size</td>
</tr>
<tr>
<td>Chance of Using Reproduction Genetic Op.</td>
</tr>
<tr>
<td>Base Chance of Using Crossover Genetic Op.</td>
</tr>
<tr>
<td>Base Chance of Using Mutation Genetic Op.</td>
</tr>
<tr>
<td>Frequency of Parameter Update</td>
</tr>
<tr>
<td>Reward Used in Parameter Update</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Test Case Population Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Population Size</td>
</tr>
<tr>
<td>Number of Children Per Generation</td>
</tr>
<tr>
<td>Parent Selection Tournament Size</td>
</tr>
<tr>
<td>Competition Tournament Size</td>
</tr>
<tr>
<td>Number of Values in Test Case</td>
</tr>
<tr>
<td>Range of Values</td>
</tr>
</tbody>
</table>

The system had more trouble finding a solution for the program with three bugs in it, observing only approximately a 1% success rate. The issue here is that correcting one of the bugs may not result in an observable increase in the fitness, making it hard for the system to know when it is doing something right. This basically implies that the system's method of testing is too coarse and that either the fitness function needs to be more sensitive or the test cases need to be more targeted; this is discussed in more detail in Section VII.

```c
for(i=0; i<size; i=i+1) {
    for(j=i; ((data[j]>data[j-1]) && (j>0)); j=j-1) {
        temp=data[j];
        data[j]=data[j-1];
        data[j-1]=temp;
    }
}
```

Fig. 3. Insertion Sort Solution Yielded by CASC System

The programs yielded by successful experiments are somewhat varied. The majority of the successes result in an expected, typical insertion sort implementation, shown in Figure 3. There are also a number of successful experiments which resulted in unexpected solutions. For example, consider the solution shown in Figure 4. In this solution the inner for loop has been set up to operate like an if statement that, if it is executed, performs a swap and resets the loop control variable for the outer for loop. Basically, this program is performing bubble sort. The majority of the other atypical solutions also are implementations of bubble sort, although not all of them are exactly like the solution shown in Figure 4. This result was unexpected and is very exciting because it shows that
the system was able to take the tools that it had available and find an unexpected solution to the problem presented to it.

for (i=0; i<size; i=i+1) {
    for (j=i; ((data[j]<data[j-1])&& (j>0)); i=0) {
        temp=data[j];
        data[j]=data[j-1];
        data[j-1]=temp;
    }
}

Fig. 4. A typical Insertion Sort Solution Yielded by CASC System

In terms of the fitness function investigation, there really needs to be more data to make any solid statements. However, based on the results obtained so far there may be an equivalence relationship between the fitness functions being used. There have been identical solutions yielded by all three fitness functions. So even though the functions in question are significantly different, they were all able to guide the system to the same solution, which implies that they may indeed be equivalent in some way.

The adaptive parameter control seems to be performing quite well in most respects. When the program population is not showing signs of convergence, the mutation operator seems to dominate, which makes logical sense since the system is searching the problem space for new genetic material to enhance the current population. When the program population is showing signs of convergence, then the crossover operator begins to dominate, which also makes sense since the system is closing in on a solution and is exploring options in the current genetic material in the program population.

VI. CONCLUSION

coming soon...

VII. FUTURE WORK

coming soon...

ACKNOWLEDGMENT

This work is supported by the Missouri S&T Intelligent Systems Center.

REFERENCES