An Overview of Automatic Bug Correction Using Genetic Algorithms and Evolutionary Techniques

David Strawn
CS 427
UNM
December 4, 2012

Introduction

Program maintenance and repair is one of the most time consuming and common jobs for software projects. Finding and repairing bugs in software is essential for the software to be stable, and correcting the bugs usually requires only small changes to the codebase. However, finding the bugs and seeing the correct solution is not always an obvious or easy task, even for the most trivial of software bugs. New research into genetic programming may provide a quick and efficient way to correct many bugs in common software. Genetic programming can allow the programmer to do little more than describe the bug or error in some way and the genetic bug fixing programming can then take over, finding optimal or near optimal solutions in fractions of the time that it would have taken a programmer to accomplish the same task. Genetic programming for automatic program repair is still a developing science, as is genetic programming in general and there are several approaches to solving this problem.

The two approaches that will be examined here involve Evolutionary Computing (EC), specifically Genetic Programming (GP) and Genetic Algorithms (GA). The first approach that will be examined uses an innovative change to the more standard fitness function in Genetic Programming making the fitness function change depending on two separate populations. In this first approach a co-evolutionary fitness function is used to evolve both the program and the test cases that are used to validate the program in an effort to find an optimal solution. This type of Genetic Programming imitates the natural biological evolution of predator and prey [1, 3]. This method allows for some very interesting types of Genetic Programming to drive the search for a valid repair to an invalid program, making both the testing and the program itself more robust [1].

The second method we will examine uses a more standard fitness function for the population of candidate programs to fix bugs in given software, but does so with out the need for a formal specification of the software or any other a priori knowledge. This allows this methodology to applied to many different types of bugs and programs, even when those running the test no little or nothing about the software they are fixing. Indeed it is possible for this approach to be used with an arbitrary number of programs and bugs, though certain types of bugs and issues are still currently beyond its ability to repair. Its feasibility at finding and repairing bugs is tested in several different contexts, which include C source code, Java byte code, and x86 assembly code [4, 5].

Co-evolution and Genetic Programming

The first type of automatic program repair that will be examined is described by Arcuri and Yao [1]. This approach relies heavily upon the co-evolution of the fitness function to evolve test cases and potential valid program simultaneously to find a valid solution to a given bug. Before this method is described in detail a brief overview of co-evolutionary Evolutionary Programming is presented.
Evolutionary Programming takes much of its methods and ideas for biological evolution. This is no different for co-evolutionary fitness functions. Co-evolutionary fitness functions attempt to pit two or more populations against each other mimicking the roles of predator and prey or host and parasite in a natural environment. This can be done by creating a fitness function that is shared between the two competing populations, or having by having two separate fitness functions that are each dependent on the state of the other population. Each population evolves to perform better against the other, hopefully driving each population towards an ideal state [3].

Several aspects of this approach are important to consider. First, each competing population must not interbreed with the other populations. In other words each population represents a distinct species. Additionally, in the implementation described by Rosin and Belew [3] special measures are taken to help ensure that each population does not specialize against the other but not the actual problem, leading to high fitness populations that are in fact merely local maxima in the search and quite far from a goal solution. This is done by keeping at least one individual for each new genome (A genome in this case is considered to be a unique trait in the population) around for successive generations. This ensures that genetic material is not lost forever after each round of evolution. There is also special consideration given to individuals who are robust against sections of the opposing population that other individuals of the same population are not, or more simply said if most individuals in population A beat two-thirds of opposing population B, but only one member of population A is able to defeat the other one-third of population B, that one is given preference in fitness since he has genetic material that most of the others lack, but is nonetheless extremely important. Both populations take turns playing host and parasite in order to evolve each population towards an ideal goal. Additionally each host is tested against a training set from the other population. This training set is generated in such a way to ensure that it is neither so easy that all of the host members defeat all the members of the set, nor so hard that all the members of the host population are defeated by the training set. This will prevent stagnation in either population [3].

Co-evolution as described by Rosin and Belew [3] is extremely interesting and likely has many wide reaching applications across the field of Evolutionary Programming, although the full content of their text is beyond the scope of this document. The above description should be sufficient to instruct the reader on the basics of co-evolution in regards to the Evolutionary Programming described in the following sections.

Co-evolution and Automatic Program Repair

Using a modified version of co-evolutionary fitness as described above, Arcuri and Yao describe an interesting methodology for automatic software repair. Their method uses a formal specification of the software and a buggy version of said software, no further information is required. Programs and unit tests are evolved together in order to find a solution to the given buggy software [1].

Unlike the co-evolutionary approach of Rosin and Belew [3], the approach of Arcuri and Yao does not involve a shared fitness function directly, however the evolution of the two populations, candidate program solutions and unit tests, are co-dependent upon each other, and thus each fitness function is dependent on the fitness of the other population. Each population takes turns competing against the other, this drives the evolution of the candidate programs towards a goal program that is valid against the given specification and all unit test cases.

The fitness function for the candidate programs under this approach rewards programs that are less bloated, raise fewer exceptions, and pass more unit test cases. Evolution of these programs takes place through the usual operators of crossover and mutation, but there is also a small chance that the original buggy program will replace a given individual in the candidate program population. This is because the fitness function favors smaller programs, in order to minimize bloat, leading to the chance that the original program structure will be entirely lost. This is considered not ideal because it is assumed that the original program is somewhat close in structure to the correct program, and thus it is beneficial to preserve this genetic information about the program structure [1].

The fitness for the unit tests is calculated by how many bugs they find in the population of candidate correct programs. Unit test that find more bugs are considered more fit. This can be calculated because of
the formal specification that is initially given and is used as an Oracle. The unit tests are intensively evolved for 1024 generations against the most fit program, during which time no program is evolved [1].

To test this framework for automatic program repair, Arcuri and Yao tested against 8 buggy implementations of a Bubble sort implementation. Their framework was able to successfully find and correct 5 of the 8 bugs. This approach offers an interesting approach to the using co-evolution of program and test in order to find and repair bugs in software that might otherwise be difficult for humans to find and repair. While this was tested on non-trivial bugs, it was not tested on production level software and the viability of it is large scale programs with complex bugs has yet to be determined. Additionally this approach requires a priori knowledge of the correct solution which make it impossible to apply to unknown domains [1].

Automatic Program Repair with Positive and Negative Tests

Another approach to automatic software repair uses positive and negative test cases and does not require any formal specification to use as an Oracle. This approach encodes the faulty behavior of the program in one or more negative test cases that the program in its current state fails, but keeps a set of positive test cases that the program in its current state passes that describe the valid behavior of the program. In order for a program to be considered valid it must pass all of the positive and negative test cases [5, 4, 2]. This approach is very appealing because it does not require any a priori knowledge about the problem domain, or even the program in general. Only valid and invalid inputs need to be supplied to the Genetic Program in order for repair to be executed.

The first example of this type of Evolutionary Computing that will be examined uses C source code for off the shelf legacy software and attempts to evolve properly functioning solution programs. This has the advantage of testing this method immediately against real world problems, compared to the previously mentioned co-evolutionary approach. It assumes, that while the program in question is faulty, the statements required in order to make the program function properly are already present in the program in some other location or form. As such, it treats each statement in the source code as an atomic genome that can be added, deleted, or swapped with other statements in the source code. The approach also uses an advanced form of fault localization to heuristically determine which parts of the program are most likely to contain the semantically invalid code. This is done by weighting the paths of execution through code such that code that is exclusively executed on failed negative test cases is favored most for alteration, followed by code that is executed on positive or negative test cases, and finally code that is not executed on either test case is weighted least, as it is assumed to be less likely to be invalid in regard to the fault at hand. The fitness for each generation of programs intuitively favors programs that pass more test cases, and programs that do not compile at all are assigned a fitness value of 0. When and if a valid program is evolved, further evolution then takes place to remove unnecessary code in order to create a human readable patch that is reasonably small [5].

This yields very promising results when tested against real world software. In about 60% of time the algorithm was able to generate correct programs given the buggy versions of the programs and the test cases. Additionally a variety of bugs were found and corrected using this methodology, including stack overflows, segmentation faults, and infinite loops. Because the test cases are static and each test is distinct this method lent itself well to parallelization and many of the patches were created in less than a minute of real time execution[5].

An extension to the previous method was created that used assembly code rather than source code as a basis for correcting bugs in off the shelf software. This method offered similar results with the added robustness of possibly being applicable to any programming language that compiled to a assembly type code. It was tested with both x86 assembly and Java byte code, using C, Haskell, and Java as source languages [4].

One major weakness of the above method is the lack of sufficient positive test cases to encode the correct behavior of the program. While it is in many ways an advantage to not have to give a formal specification of a problem in order to perform automatic bug correction, it does require that all of the test cases encode all of the required behavior of the problem, with out this a program could be evolved to pass the given negative
test case, and all given positive test cases but change some core aspect of the program that was failed to be described in the test cases. This makes this one of the major concerns with this method [5, 4].

A updated and improved version of this method called GenProg was created to allow usage in a cloud computing environment, as well as having improved fix localization, methods for ensuring likely compilation of a generated repair candidate, and other improvements. While many of the core ideas behind this updated approach are the same, it was tested with much larger code projects, finding and repairing 55 out of 105 known bugs with a total of 5.1 Million lines of code and more than 10,000 test cases. Running these experiments on a cloud computing environment the researchers were able to give a monetary value to each bug their Genetic Programming was able to fix. Including the cost of the failed attempts to fix some bugs, it cost them a little less than $8 to fix the 55 bugs with an average repair time of 1.6 hours [2].

Further improvements to this method were made when combining it with human entered annotations that gave ‘hints’ to GenProg about which parts of the program are important. After these annotations were added to the program, GenProg was able to find and repair 71 of the 105 bugs (68%). This is a substantial improvement from the standard implementation of the program, but it requires some about of a priori knowledge about the problem. This approach is additionally less ideal since it requires human interaction in locating potential problem areas, often the hardest part of software repair. However the performance increases can not be overlooked and a hybrid approach may be ideal [2].

In general this method proved to very robust and cheap to implement. The later improvements to the method allowed it to scale well in parallel execution on cloud computing environments which are cheap to use. Without any a priori knowledge of the problem or program it is often able to find a valid correction solution in relatively little time, and performs even better when it is given small ‘hints’ by a human developer [2].

**Conclusion**

The correction of bugs in production grade software is essential to the viability and success of said software. To that effect large amounts of developers time is devoted to finding and correcting bugs in existing code, rather than creating and working on new features in the software project. These bugs are often small and simple, but nevertheless they can be extremely difficult for a human developer to find in a timely manner. Using methods of automatic software repair would alleviate the strain that this task often puts on projects, both monetarily and temporally, allowing them to focus on producing new code and features rather than fixing small, but important, bugs in existing code.

We examined two promising methods for automatic program repair with Evolutionary Computing. While the two methods differed significantly, they were both able to successfully repair faulty software with no human interaction. The first method used a very interesting method of co-evolution to evolve both candidate solutions and test cases so that a very robust solution could be found, however it was not tested on large scale real world software and how effective it would prove on said software remains to be seen. The second used positive and negative test cases to define valid and invalid behavior respectively, and was tested with source code and assembly on real world software. The latter method also lent itself well to parallel processing and was tested in a real world cloud computing environment. Further this latter approach has the advantage of not requiring any a priori knowledge about the problem domain, making it potentially applicable to many diverse types of bugs and programs.

While this field of Computer Science still promises much research into more advance repair techniques, it is evident that automatic program repair has many exciting possibilities. Some bugs in software proved to be out of the scope of these methods ability to repair, but many were found and repaired by these solutions, at a very modest cost. It may be soon that many bugs in software are found and repaired with out the aid of human developers by using such techniques, alleviating the much strain that such small errors can put on large software projects.
Bibliography


