A Novel Co-evolutionary Approach to Automatic Software Bug Fixing

Andrea Arcuri and Xin Yao

Abstract— **Many tasks in Software Engineering are very expensive, and that has led the investigation to how to automate them. In particular, Software Testing can take up to half of the resources of the development of new software. Although there has been a lot of work on automating the testing phase, fixing a bug after its presence has been discovered is still a duty of the programmers. In this paper we propose an evolutionary approach to automate the task of fixing bugs. This novel evolutionary approach is based on Co-evolution, in which programs and test cases co-evolve, influencing each other with the aim of fixing the bugs of the programs. This competitive co-evolution is similar to what happens in nature for predators and prey. The user needs only to provide a buggy program and a formal specification of it. No other information is required. Hence, the approach may work for any implementable software. We show some preliminary experiments in which bugs in an implementation of a sorting algorithm are automatically fixed.**

I. Introduction

Software Testing is used to find the presence of bugs in computer programs [1]. If no bug is found, testing cannot guarantee that the software is bug-free. However, testing can be used to increase our confidence in the software reliability. Unfortunately, testing is expensive, time consuming and tedious. It is estimated that testing requires around 50% of the total cost of software development [2]. This cost is paid because software testing is very important. Releasing bug-ridden and non-functional software is indeed an easy way to lose customers. For example, in the USA alone it is estimated that every year around $20 billion could be saved if better testing was done before releasing new software [3].

At any rate, even if an optimal automated system for doing software testing existed, fixing the bugs would still be a duty of the programmers. Hence, there has been effort in developing *Automated Debugging* techniques (e.g., [4], [5] and [6]) to help the programmers to locate the bugs. However, to our best knowledge, only little work has been done on the the actual automation of repairing software (e.g., [7], [8] and [9]), and it is able to address only specific types of bugs. For example, in [8] only expressions and left-hand side of assignments might be repaired.

In this paper we present a novel system that, given as input a formal specification of a software and a buggy implementation of it, uses evolutionary techniques to try to fix it. There is no particular restriction on the type of bug that can be addressed. We name this task with the expression *Automatic Bug Fixing* (ABF). We gave a first idea of ABF in [10], and in this paper we show preliminarily experiments to confirm its validity and to see how its performance can be increased.

The system presented in this paper is based on our previous work on *Automatic Programming* (AP) [11]. The idea of that paper is to use Genetic Programming (GP) to evolve programs that satisfy a formal specification. The training set is composed by *Unit Tests* [12]. Because we use the formal specification for generating an *oracle*, we can yield as many unit tests as we want. However, we need to carefully choose a relatively small set of unit tests, because using all of them is infeasible (i.e., the computational cost of evaluating an evolutionary program would be too high). Hence, we use GP for evolving programs for passing the current set of unit tests, but at the same time we use *Search Based Software Testing* techniques [13] to yield new unit tests for finding bugs in the evolutionary programs. That generates a *co-evolution* [14], that hopefully will lead to an *arms race* that will bring to the evolution of a program that satisfies the given formal specification.

The difference of [11] with the framework that we present in this paper is that we are not trying to evolve a correct program from scratch. In fact, we start with a buggy version of it. Hence, we can use our AP system with one main difference: instead of starting with a random population of evolutionary programs, we seed the first generation with the buggy version that we want to fix. That means that all the individuals of the first generation are equal to the buggy program.

Because the programs implemented by software developers are usually close to being correct [15], we expect that ABF would be a much easier task than AP. That legitimates us to speculate that industrial applications might be possible in a not far future. However, we claim that ABF is more difficult than software testing, because for example one of its components is software testing itself. Therefore, if we consider the fact that, although the automation of software testing has been heavily investigated since the 1970s (e.g., [16]), it has not been solved yet, hence we expect that ABF will require at least the same amount of research effort.

The work in [17] on Mutation Testing has some similarities with our framework. Its goal is to find test cases that can recognise buggy *mutants* of the tested software. Mutants (generated with a precise set of rules) co-evolve with the test cases, and they are rewarded on how many tests they pass, whereas the test cases are rewarded on how many mutants they identify as semantically different from the original program. The similarity of our ABF work with [17]

Andrea Arcuri and Xin Yao are with the The Centre of Excellence for Research in Computational Intelligence and Applications (CERCIA), School of Computer Science, University of Birmingham, Edgbaston, Birmingham B15 2TT, UK; email: {A.Arcuri,X.Yao}@cs.bham.ac.uk

is hence evident. In fact, we can consider our evolutionary programs as mutants of the correct program. However, the goals of the two frameworks are different. Moreover, how the programs/mutants are generated and how the oracle, used for checking whether a test case is passed, is defined are different as well.

This paper gives to the Software Engineering community the important contribution of showing how the highly expensive task of bug fixing might be automated. At the same time, it brings to the Natural Computation community a novel context in which well known research fields such as co-evolution and GP are combined together. This novel combination yields many research questions that require to be investigated.

The paper is organised as follows. Section II presents the framework that we developed for automatically fixing bugs. A case study to validate our novel approach follows in section III. Finally, section IV concludes the paper.

II. A Framework for Fixing Bugs

For fixing bugs, we have developed a framework that is mainly based on four components:

1) Genetic Programming (GP).
2) Distance functions derived from formal specifications.
3) Search Based Software Testing.
4) Co-evolution.

A. Genetic Programming and Distance Functions

GP [18] is a paradigm for evolving programs. A genetic program is often represented as a tree, in which each node is a function whose inputs are the children of that node. A population of programs is held at each generation, where individuals are chosen to fill the next population accordingly to a problem specific fitness function. Crossover and mutation operators are then applied on the programs to generate new offspring. GP has been principally and successfully employed to solve real-world learning problems.

In our framework, we represent the input buggy program as a genetic program. Then, we use GP operators to evolve it with the aim of fixing its bugs. To note that any evolutionary technique for evolving programs could have been used (e.g., *Grammatical Evolution* [19]). Because we are assuming that the buggy program is structurally not too far from a global optimum, an Hill Climbing algorithm might be employed instead. However, we are sceptical about this latter option, because being structurally near to a global optimum does not necessarily mean being in its basin of attraction. In other words, a program tree that is close to a correct one might have a very bad fitness value (this is particularly true for bugs that are in areas of the code that are always executed regardless of the input, like for example all the statements before the first conditional branch in the execution flow). Moreover, although we are assuming that, given a set of operations for doing small modifications on program trees, a short sequence of operations exists for reaching a global optimum, there is no guarantee that the fitness landscape would be either smooth or without plateaus. An analysis of the fitness landscape and comparisons of different search algorithms will be done in future work.

The training set is composed by unit tests. A unit test can be viewed as a pair (i, r), where i is an input belonging to the input domain of the tested program, and where r is the expected result. To see whether an evolutionary program passes a unit test, the program will be executed with input i, and then the result will be compared with the expected r.

However, there are two main differences from the normal use of GP:

- the training set does not contain any noise.
- we are not looking for a program that on average performs well, but we want a program that always gives the expected results. Hence, a program does not need to worry about over-fitting the training set, it has to over-fit it. In fact, even if only one test in the training set is failed, that means that the specification is not satisfied.

For the fitness function of an evolutionary program, checking only whether the unit tests are passed might not give enough gradient to the search of GP. In the same way, a naive distance value between the actual result and the expected one is inappropriate in many cases. Hence, we use a distance function that is based on the formal specification. The distance function that we use is the same one employed in [20] for Black Box Testing. It heuristically measures the distance of the output of the computation from the expected result. A value of zero means that the unit test is passed, otherwise we get a value as high as far the program is from passing the test. Therefore, the fitness function of the evolutionary programs involves the minimisation of these distances over all the unit tests in the training set. In particular, the fitness function to minimise is:

$$f(g) = \frac{N(g)}{N(g)+1} + \frac{E(g)}{E(g)+1} + \sum_{t \in T_i} d_P(t, g(t)) \ , \qquad (1)$$

where the number of nodes $N(g)$ of the evolutionary program g is tried to be minimised (for contrasting bloat [21]) as well as the number of raised exceptions $E(g)$ (e.g., divisions by zero and accessing arrays out of their bounds). Note that, in case of exceptions, the run of g is not blocked, and a value of zero is returned by the node that yielded the error. The training set at the current generation is denoted by T_i, whereas d is the distance that we have previously described. Given the formal specification P, d takes as input the data of the unit test t and the output of the execution of g with t as input. Note that d does not calculate the distance between t and $g(t)$. In fact, it calculates the distance of $g(t)$ from the expected result. Table I shows some examples of this distance function applied for different types of predicates.

That function d works fine for expressions involving numbers (e.g., integer, float and double) and boolean values. For other types of expressions, like for example an equality comparison of pointers/objects, we need to define the semantics of the operator $-$. For example, it can return 1 if the two values are different and 0 otherwise. At any rate, this type

Predicate θ	Function $d_\theta(I, R)$
A	if a is TRUE then 0 else K
$A = B$	if $abs(a-b) = 0$ then 0 else $abs(a-b) + K$
$A \neq B$	if $abs(a-b) \neq 0$ then 0 else K
$A < B$	if $a - b < 0$ then 0 else $(a-b) + K$
$A \leq B$	if $a - b \leq 0$ then 0 else $(a-b) + K$
$A > B$	$d_{B<A}(I, R)$
$A \geq B$	$d_{B \leq A}(I, R)$
$\neg A$	Negation is moved inward and propagated over A
$A \wedge B$	$d_A(I,R) + d_B(I,R)$
$A \vee B$	$min(d_A(I,R), d_B(I,R))$
$A \Rightarrow B$	$min(d_{\neg A}(I,R), d_B(I,R))$
$A \Leftrightarrow B$	$min((d_A(I,R) + d_B(I,R)),$ $(d_{\neg A}(I,R) + d_{\neg B}(I,R)))$
A xor B	$min((d_A(I,R) + d_{\neg B}(I,R))),$ $(d_{\neg A}(I,R) + d_B(I,R)))$
$\forall x \in X : Q \bullet W$	if $X : Q$ is empty then 0 else $\sum_{v \in X:Q} d_W(I[v/x], R[v/x])$
$\exists x \in X : Q \bullet W$	if $X : Q$ is empty then K else $min(d_W(I[v/x], R[v/x]))$ where $v \in X : Q$

TABLE I

EXAMPLE OF HOW TO APPLY THE FUNCTION d ON SOME PREDICATES. K CAN BE ANY ARBITRARY POSITIVE CONSTANT VALUE. A AND B CAN BE ANY ARBITRARY EXPRESSION, WHEREAS a AND b ARE THE ACTUAL VALUES OF THESE EXPRESSIONS BASED ON THE VALUES IN THE INPUT SET I AND RESULT SET R. Q AND W CAN BE ANY ARBITRARY EXPRESSION.

of distance d is similar to the *branch distance* used in White Box Testing, and more details can be found in [13].

Unfortunately, the predicates involving $\forall$ and $\exists$ can be computationally expensive if the the set of values on which they depend on is too large. In our case study, these sets were always arrays of relatively small length, hence the computational cost was not so high. However, the case of expressions like $\forall x \in \Re \bullet W(x)$ cannot be handled in that way. In fact, although on a machine only a sub-set of $\Re$ can be represented (e.g., in many programming languages we have 32 bits representations for float values and 64 bits for double values), that sub-set would be still too large. We will address this problem in the future.

Other problem is that the same software specification can be written in different ways, and each of these versions will generate a different fitness function. Some of these fitness functions might have a smooth landscape, other unfortunately may have many local optima. Formal techniques for transforming a specification in an equivalent one that generates a better fitness function might be designed, and we will investigate this idea in future work.

B. Search Based Software Testing

Choosing an appropriate training set is not a trivial task. In fact, because from a formal specification it is possible to derive an oracle, we can generate as many unit tests as we want. We cannot use all the possible unit tests, because their number might be arbitrarily large, and the computational cost of the fitness function of the evolutionary programs linearly depends on the size of the training set. Hence, we need to use a relatively small training set, and choosing it is a problem that requires to be addressed.

Although sampling unit tests at random might work in some cases, we need to cover all the possible requirements of the software. Therefore, if the evolutionary programs do not satisfy these requirements (i.e., they have bugs), we should have at least one unit test in the training set that makes the evolutionary programs fail. To find this type of unit test, we use search based software testing techniques [13]. In other words, we *test* the evolutionary programs to try to find inputs for which they fail.

In the evolution of the unit tests, they are rewarded on how many evolutionary programs they find bugs in. The fitness function of the unit tests to maximise is hence:

$$f(t) = \sum_{g \in G_i} d_P(t, g(t)) \ ,$$

where G_i is the population of evolutionary programs at the generation i.

Note that, although we use search based software testing techniques for sampling the unit tests, any other automated software testing technique could be employed (e.g., *symbolic execution* [16]). Our choice was based on the successful performance that search based software testing systems have in literature. In particular, for the search algorithm we use Genetic Algorithms (GAs) [22].

C. Co-evolution

As we explained in more detail in [11], having a fixed training set for the entire evolution of GP is not an appropriate choice. We would get better results if, at each generation of GP, we also evolve the unit tests to improve their ability of finding bugs. That generates a *competitive co-evolution* [14], in which the two species (evolutionary programs and unit tests) influence each other. Figure 1 shows the relations of the fitness function between programs and unit tests.

This type of co-evolution is similar to what in nature happens between *predators* and *prey*. For example, faster prey escape predators more easily, and hence they have higher probability of generating offspring. This influences the predators, because they need to evolve as well to get faster if they want to feed and survive. In our context, the evolutionary programs can be considered as prey, whereas the unit tests are predators. Hopefully, this co-evolution will lead to an *arms race* that will lead to the evolution of a program that satisfies the given formal specification.

Unfortunately, producing an arms race in co-evolutionary algorithms is more difficult than it looks [23]. In fact, problems such as *mediocre stable states* and *loss of gradient* might arise.

D. Particularities of Automatic Bug Fixing

Instead of sampling random trees, we seed the first generation of evolutionary programs with copies of the buggy

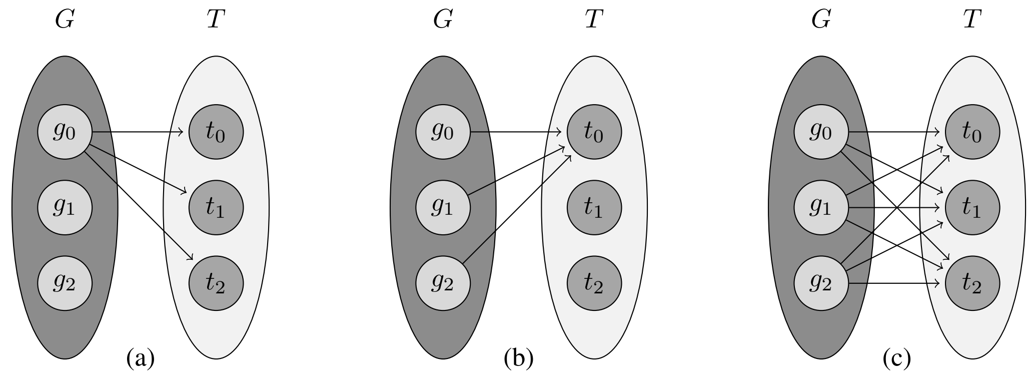

Fig. 1. G is the population of programs, whereas T is the population of unit tests. For simplicity, sets of cardinality 3 are displayed. In picture (a) it is shown on which unit tests the fitness of the first program g_0 is calculated. On the other hand, the picture (b) shows on which programs the fitness for the first unit test t_0 is calculated. Note that the arc between the first program and the first unit test is used in both fitness calculations. Finally, picture (c) presents all the possible $|G| \cdot |T|$ connections.

program. This is very different from the usual application of GP, and it might have some important consequences.

Because software developers do not create programs at random [15], the buggy program would be structurally near to the optimal solution. However, its fitness value (based on eq.1) might be extremely poor. This happens because a single bug might completely change the output of a program. The problem is that GP is driven by the fitness values and not by the structural distance from the optimal solution (that is unknown). Therefore, in the case of that type of bug, it is very likely that during the first generations all the genetic material of the buggy program would be lost because replaced by smaller and simpler programs. These smaller programs will be far away from the optimum, but because they are smaller and likely having a higher fitness, they will be preferred by the GP evolution. Eventually, these smaller programs might evolve up to the optimal solution [11], but they would do it without exploiting any useful information given by the buggy program.

We designed two simple methods to address the problem of the loss of genetic material of the buggy program. The first consists of a new GP reproduction operator. In addition to the usual crossover and mutations, a small probability exists that an individual will be replaced by the original buggy program. This helps to recover the original genetic material in the case it gets lost during the search.

Another simple technique is to penalise short programs. The idea is that the number of nodes of the optimal solution should not be too different from the one of the buggy program. So, given a constant δ, when we calculate the fitness value of an evolutionary program g, if $N(g) < \delta N(buggy)$, then we will replace the penalty term $\frac{N(g)}{N(g)+1}$ in eq.1 with the constant 1. Note that, although 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, the opposite is not true. In fact, due to possible redundant and unused pieces of junk code, a correct program can be arbitrarily big. Due to the problem of *bloat* [24], this latter situation is indeed very likely.

```
for(i=0 to A.length-1)
   for(j=0 to A.length-2)
      if(A[j] > A[j+1])
        {
           tmp    = A[j];
           A[j]   = A[j+1];
           A[j+1] = tmp;
        }
```

Fig. 2. Pseudo-code of a bubble-sort algorithm.

Often, in the GP literature crossover is preferred over mutation operators. In fact, it is not rare that only crossover is employed. However, in the problem that we analyse in this paper crossover is likely not very useful, because the starting solution is already close to a global optimum, and that can be reached with a relatively short sequence of mutations. Hence, we give more emphasis (i.e., an higher probability) to mutations.

III. Case Study

For validating our novel framework, we did experiments on an instance of a sorting algorithm. In particular, we use a bubble-sort implementation [25]. Figure 2 shows a pseudo-code for that algorithm.

A possible formal specification (in first order logic) is:

Function: $void\ sort(array\ A)$

Pre-condition : $True$

Post-condition: $\forall i \in [0, A'.length - 2] \bullet A'[i] \leq A'[i+1] \wedge$
$isPermutation(A', A)$

where the state of the array A after the function is executed is represented by A'.

The distance function that is automatically generated from the formal specification and thus used in the fitness functions is:

$$d_P(A,A') = \omega(\omega(s_0(A')) + \omega(s_1(A,A')))\ ,$$

where, given $z = A'.l - 2$, we have:

$$s_0(A') = \sum_{i=0}^{i=z} \begin{cases} 0 & \text{if } A'[i] \le A'[i+1]\ , \\ A'[i] - A'[i+1] & \text{otherwise,} \end{cases}$$

$$s_1(A,A') = \begin{cases} 0 & \text{if } \mathit{isPermutation}(A',A) \text{ is true,} \\ 1 & \text{otherwise,} \end{cases}$$

$$\omega(x) = \frac{x}{x+1}\ .$$

This choice of analysing a sorting algorithm was done because sorting algorithms are well known, they have a specification that is not directly mappable to an implementation and, finally, there has already been work on evolving sorting algorithms with GP (e.g., [26], [27] and [11]).

It might be arguable the fact that we try to fix bugs in a sorting algorithm when there has been already successful work about the more difficult task of evolving sorting algorithms from scratch. However, our approach is different because the framework we developed is *general* and might be used for any feasible software (of course, evolving programs for solving for example the *halting problem* [25] is not feasible). In fact, it makes no previous assumption on the software that will be tried to be fixed. The user only needs to provide as input a formal specification and a buggy implementation of it. Moreover, the set of used GP primitives (described in more detail later) defines a rich language that does not have any bias regarding the evolution of a sorting algorithm.

A. Set of Primitives

The basic components of a genetic program are called *primitives* [18]. Often, the choice of the primitives is optimised for the particular task that the GP needs to solve. However, in this work we present a general set of primitives that defines a rich Turing equivalent language (although as not as rich as languages like *C* and *Java*).

We use Strongly Typed Genetic Programming [28] to enforce the syntactical correctness of the programs. The node constraints are: void, integer value, integer variable and array variable.

We use 34 basic primitives. Unfortunately, for reasons of space we can only briefly describe them. However, names are consistent with their semantics.

- 10 integer constants with value between 0 and 9.
- 4 commands: `skip`,`loop`, `if` and `seq`.
- 1 terminal `index` that represents the current iteration of the closest loop.
- 5 arithmetic expressions: `add`, `sub`, `mul`, `div` and `mod`.
- 6 boolean expressions: `bigger`, `bigger_or_equal`, `equal`, `and`, `or` and `neg`.
- 3 variables: `result`, `array_tmp_0` and `integer_tmp_0`.
- 2 operators for reading and writing integer variables: `read_memory` and `write_memory`.
- 3 operators for manipulating arrays: `length`, `array_value` and `new_array`.

Then, the input variables of the software and any constant that appears in the formal specification will be automatically added to the set of primitives. In the particular case of the considered sorting algorithm, we just add the variable `array_input_0` that represents the input array.

B. Analysed Bugs

Figure 3 shows a correct implementation of a sorting algorithm. In our experiments, we considered 8 different bugs. In other words, we started with the correct implementation given in figure 3, and from that we generated 8 different versions. In each of them, we manually inserted a different bug. Although their choice is fairly arbitrary, we tried to consider different types of bugs, that result in different kinds of wrong/missing sub-tree structures:

1) the `1` in node (9) is replaced by `2`.
2) `bigger` in node (11) is replaced by `equal`.
3) the sub-tree $(19, 20, 21)$ is replaced by `index`.
4) the sub-tree $(43, 44, 45)$ is replaced by `index`.
5) the nodes $(0, 1, 2, 3)$ are removed. The node (4) becomes the new root.
6) the sub-tree $(24, 25, 26, 27, 28, 29)$ is replaced by `skip`.
7) the if statement in (10) is removed, and node (22) becomes the new child of (4).
8) the entire tree is replaced by a `skip` command.

C. Experiments

For the GP engine, we use ECJ [29]. All the other components of the framework are written in Java.

Our framework is composed by complex components as the GP engine, the software testing engine and the co-evolutionary algorithm that links these two engines. Therefore, there are many parameters that the framework requires to set, and that need to be tuned. In general, we set the parameters with values that are common in literature. However, we did experiments with different probability values of crossover and mutations, and the one reported are the best we found. Due to space limitations, we can only briefly describe the parameters that we think are the most important.

The population of GP is composed by 1000 individuals. A tournament selection with size of 3 is employed. With probability 0.09, an individual remains unchanged between two generations, whereas with probability 0.01 it is replaced by the original buggy program. Crossover is done with probability 0.1, whereas mutation is done with probability 0.8. When a mutation event occurs, one out of six different mutation operators in ECJ is uniformly chosen (i.e., we do not privilege any particular of them). Elitism rate is set to 1 individual for generation.

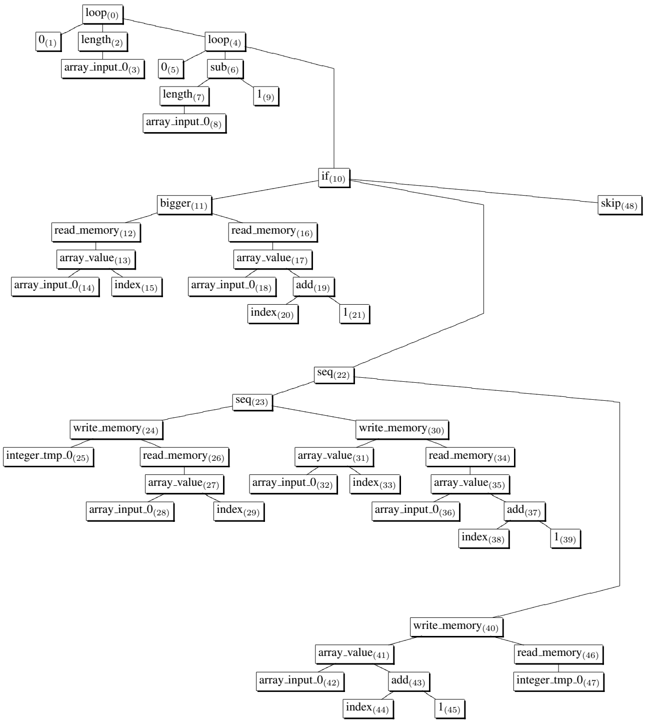

Fig. 3. GP implementation of a bubble-sort. It consists of 49 nodes, with a max depth of 10. Each node is numerated from (0) to (48). Node (0) is the root of the tree. All the `index` refers to the loop in node (4) and none to the loop in node (0). The `skip` command in node (48) represents the empty *else* branch of the `if` node in (10).

As we expected, in our trials we found the best performance when crossover had a low probability. However, when we tried to remove it (i.e., by setting its probability to 0), we got worse results. Hence, somehow crossover is still useful.

The population of unit tests is composed by 32 individuals, that are evolved with a simple GA. The actual representation of an individual depends on the type of input that the function under test takes. In our case study, that means that the individuals are arrays of integer type. For simplicity, in our framework the employed GA is specialised for handling arrays. However, for handling other types of inputs, we can include in our framework any state-of-the-art software testing tool.

A single point crossover is employed with probability 0.75. However, the parents can have different lengths. In such cases, the length of the new offspring will be the average of the lengths of the parents. Mutation of an array position is done with probability $1/l$, where l is length of the array in the unit test. A mutation adds a discretised gaussian noise (mean 0 and variance 1) to the mutated value. Elitism rate is set to 1 as in GP.

The co-evolution is done for 50 generations. For improving the performance, at each generation the best unit test is added to a special archive called *Hall of Fame* [30]. The fitness of an evolutionary program is also based on that archive. The unit tests in the archive do not evolve. Every 5 generations of the co-evolution, the unit tests are intensively evolved for 1024 generations against the best program in the current population. In other words, for calculating the fitness of a unit test only the best program is executed. During that intensive evolution of the unit tests, no program is evolved.

To note that the number of individuals and the number of generations are relatively low compared to the literature on GP. The reason is that, although we are trying to fix a non-trivial program, it is very far from real-world software, in which functions can be long hundreds of lines of code. Therefore, if we need a large amount of resources for solving a relatively simple problem, that will not be a good sign of the applicability of our research prototype to more complex software.

In our experiments on a 3.0 GHz machine, a typical run of the framework for fixing a bug takes between two and four minutes. However, using faster languages as C and compiling the evolutionary programs in native code, instead of executing them by interpretation, would dramatically speed up the framework [31]. However, given the same setting, the results would be the same. The reason for using Java was that it is a very easy and powerful language to use, that made possible the implementation of the framework in a short period of time.

Table II shows the results of the experiments on fixing the 8 different bugs we previously described. We also did experiments on the penalisation of shorter programs using $\delta = 1$. The results in table II shows that the penalisation increases the performance. Fisher's Exact Tests at a 0.05 level of significance confirm the improvement of the results, but not for the bug number 5 (and of course not for 6, 7 and 8).

Simple bugs as 1, 2, 3 and 4 were correctly fixed most of the time. We did fix a complex bug as 5, but our framework was not able to fix the bugs number 6 and 7. It might be argued that we could fix them by simply increasing the number of individuals and/or allowing more generations. However, doing that would be in contrast with what we claimed before. Hence, more research on how to improve the algorithms given the same amount of resources is required.

The bug number 8 actually means that the entire program should be evolved from scratch, and we did not manage to do it. However, the scope of this paper is about fixing bugs, and not on evolving complete software.

IV. Conclusions

In this paper we have proposed a novel approach for repairing software in an automatic way (i.e., ABF). In contrast to the little literature on the subject, in our system there is no particular restriction on the type of bug that can be fixed (e.g., wrong binary operation or missing of a block of statements).

The framework is based on co-evolution, GP and search based software testing.

To validate the framework, we carried out experiments on buggy versions of a bubble-sort algorithm. Although we successfully fixed most of the bugs, some were too difficult and we did not solve them. This leads us to investigate novel ways for improving the performance of our framework.

Bug Id	Base Algorithm Performance	Shorter Programs Penalisation Performance	Fisher's Exact Test p-value
1	64	84	0.001
2	74	94	0.000
3	83	97	0.001
4	68	85	0.003
5	68	79	0.054
6	0	0	1.000
7	0	0	1.000
8	0	0	1.000

TABLE II

NUMBER OF TIMES, ON A TOTAL OF 100 TRIALS, IN WHICH THE BUGS WERE CORRECTLY FIXED. P-VALUES OF FISHER'S EXACT TESTS TO COMPARE WHETHER THE PENALISATION IMPROVES THE PERFORMANCE ARE ALSO REPORTED.

Moreover, different types of software should be investigated as well.

Evolving correct programs from scratch is currently a very hard task [11]. However, we think that our framework for bug fixing might be a very useful tool for the software developers. The reason is that a bug that is difficult to be fixed by a human might be, on the other hand, very easy for our framework. That would be the case of bugs that consist of only small differences from the correct code, but which are located in parts of the source code that are very complex for a human to analyse. However, to really scale up to real-world software, it will be compulsory to design techniques for narrowing down the search space of the evolutionary operators. One possibility could be the exploitation of automated debugging techniques.

Another problem that is equivalent to ABF is *Automatic Implementation of New Requirements*. Given a specification $S1$ and a correct program $P1$, when we add new requirements to the specification (and we get $S2$), there is the problem of changing the program to satisfy the new specification. Hence, we can consider $P1$ as a buggy implementation of $S2$, and that is similar to the problem of ABF.

V. ACKNOWLEDGEMENTS

The authors are grateful to Josef Baker, David Robert White, Per Kristian Lehre and Ramón Sagarna for insightful discussions. This work is supported by an EPSRC grant (EP/D052785/1).

REFERENCES

[1] G. Myers, *The Art of Software Testing*. New York: Wiley, 1979.

[2] B. Beizer, *Software Testing Techniques*. New York: Van Nostrand Rheinhold, 1990.

[3] G. Tassey, "The economic impacts of inadequate infrastructure for software testing, final report," *National Institute of Standards and Technology*, 2002.

[4] A. Zeller, "Automated debugging: Are we close?" *IEEE Computer*, pp. 26–31, November 2001.

[5] M. Auguston, C. Jeffery, and S. Underwood, "A framework for automatic debugging," in *IEEE International Conference on Automated Software Engineering (ASE)*, 2002, pp. 217–222.

[6] M. Renieris and S. Reiss, "Fault localization with nearest neighbor queries," in *IEEE International Conference on Automated Software Engineering (ASE)*, 2003, pp. 30–39.

[7] M. Stumptner and F. Wotawa, "Model-based program debugging and repair," in *Proceedings of the International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, 1996.

[8] S. Staber, B. Jobstmann, and R. Bloem, "Finding and fixing faults," in *13th Conference on Correct Hardware Design and Verification Methods (CHARME)*, 2005, pp. 35–49.

[9] W. Weimer, "Patches as better bug reports," in *Proceedings of the 5th international conference on Generative programming and component engineering*, 2006, pp. 181–190.

[10] A. Arcuri, "On the automation of fixing software bugs," to appear in the Doctoral Symposium of the IEEE International Conference on Software Engineering (ICSE), 2008.

[11] A. Arcuri and X. Yao, "Coevolving programs and unit tests from their specification," in *IEEE International Conference on Automated Software Engineering (ASE)*, 2007, pp. 397–400.

[12] IEEE-Standards-Board, "Ieee standard for software unit testing: An american national standard, ansi/ieee std 1008-1987," *IEEE Standards: Software Engineering, Volume Two: Process Standards*, 1999.

[13] P. McMinn, "Search-based software test data generation: A survey," *Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, June 2004.

[14] W. D. Hillis, "Co-evolving parasites improve simulated evolution as an optimization procedure," *Physica D*, vol. 42, no. 1-3, pp. 228–234, 1990.

[15] R. A. DeMillo, R. J. Lipton, and F. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[16] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, pp. 385–394, 1976.

[17] K. Adamopoulos, M. Harman, and R. M. Hierons, "How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution," in *Genetic and Evolutionary Computation Conference (GECCO)*, 2004, pp. 1338–1349.

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

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

[20] N. J. Tracey, "A search-based automated test data generation framework for safety-critical software," Ph.D. dissertation, University of York, 2000.

[21] S. Luke and L. Panait, "A comparison of bloat control methods for genetic programming," *Evolutionary Computation*, vol. 14, no. 3, pp. 309–344, 2006.

[22] J. H. Holland, *Adaptation in Natural and Artificial Systems, second edition*. Cambridge: MIT Press, 1992.

[23] S. G. Ficici and J. B. Pollack, "Challenges in coevolutionary learning: Arms-race dynamics, open-endedness, and mediocre stable states," in *Artificial Life VI*, 1998, pp. 238–247.

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

[25] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms.*, 2nd ed. MIT Press and McGraw-Hill, 2001.

[26] K. E. Kinnear, Jr., "Generality and difficulty in genetic programming: Evolving a sort," in *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, 1993, pp. 287–294.

[27] A. Agapitos and S. M. Lucas, "Evolving modular recursive sorting algorithms," in *Proceedings of the European Conference on Genetic Programming (EuroGP)*, 2007, pp. 301–310.

[28] D. J. Montana, "Strongly typed genetic programming," *Evolutionary Computation*, vol. 3, no. 2, pp. 199–230, 1995.

[29] S. Luke, "Issues in scaling genetic programming: Breeding strategies, tree generation, and code bloat," Ph.D. dissertation, University of Maryland, 2000.

[30] C. D. Rosin and R. K. Belew, "New methods for competitive coevolution," *Evolutionary Computation*, vol. 5, no. 1, pp. 1–29, 1997.

[31] P. Nordin and W. Banzhaf, "Evolving turing-complete programs for a register machine with self-modifying code," in *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, 1995, pp. 318–325.