On the Automation of Fixing Software Bugs

Andrea Arcuri
The Centre of Excellence for Research in Computational Intelligence and Applications (CERCIA),
The School of Computer Science, The University of Birmingham, Edgbaston, Birmingham B15 2TT, UK.
A.Arcuri@cs.bham.ac.uk

ABSTRACT

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. Techniques to help the software developers for locating bugs exist though, and they take name of Automated Debugging. However, to our best knowledge, there has been only little attempt in the past to completely automate the actual changing of the software for fixing the bugs. Therefore, in this paper we propose an evolutionary approach to automate the task of fixing bugs. The basic idea is to evolve the programs (e.g., by using Genetic Programming) with a fitness function that is based on how many unit tests they are able to pass. If a formal specification of the buggy software is given, more sophisticated fitness functions can be designed. Moreover, by using the formal specification as an oracle, we can generate as many unit tests as we want. Hence, a co-evolution between programs and unit tests might take place to give even better results. It is important to know that, to fix the bugs in a program with this novel approach, a user needs only to provide either a formal specification or a set of unit tests. No other information is required.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging;
I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search

General Terms

Algorithms, Experimentation, Reliability

Keywords

Automatic Bug Fixing, Automated Debugging, Repair, Co-evolution, Genetic Programming

1. INTRODUCTION

Software Testing is used to find the presence of bugs in computer programs [12]. 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 [5]. 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 [20]. Hence, automating the testing phase is one of most important problem in software engineering.

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., [23] and [15]) to help the programmers to localise the bugs.

Although promising results have been achieved, automated debugging is still an immature field. For example, to our best knowledge, most of the automated debugging techniques proposed so far have a non-trivial limitation. In fact, they rely on the presence of at least one test case that is passed by the buggy program under analysis. In other words, if the program fails on each generated test, these automated debugging techniques cannot be used. Moreover, even if a bug is correctly localised, it is still duty of the developer to change the code for fixing that bug, and that is not always a trivial task.

In this paper we propose a novel technique to address the ambitious goal of *Automatic Bug Fixing* (ABF). In other words, instead of just considering the localisation of the bugs, we also want to automatically repair the code for removing them. To our best knowledge, only little work has been done on the the actual automation of repairing software (e.g., [19], [18] and [22]). Moreover, the addressed types of bugs are very specific, whereas our approach is more general.

For being used, our approach requires at least one of the following:

- a set of unit tests. The bugs that we want to fix need to be revealed by at least one of these test cases. This is a trivial constraint, because if we want to fix a bug we already know that it exists, and the only way to know it is to have at least one failed test case.
- a formal specification of the buggy software under test. However, formal specifications are not widely employed in industry, and they might be difficult to write [13].

The idea of our approach is to modify the code of the buggy program with evolutionary techniques, like for example Genetic Programming (GP) [9]. In GP, programs are evolved to accomplish a particular task (a typical example is the evolution of a machine learning classifier). They are often represented as trees, 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 individ-

ICSE'08, May 10–18, 2008, Leipzig, Germany.
ACM 978-1-60558-079-1/08/05.

uals 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.

In the ABF problem, the goal is to pass all the provided unit tests. Therefore, not only we need that GP modifies the code to pass the failed unit tests, but these changes have to not compromise the former functionalities.

Unfortunately, even if all the test cases are passed, that does not necessarily mean that the bug is fixed. In fact, the system might evolve a partially patched program that is able to pass the former failed tests, but that might still have bugs. Therefore, the software developer needs to inspect the output of the ABF system. However, even if the system does not fix the bug, a partially patched program might give helpful information about the bugs.

Besides fixing the bugs, GP might change some parts of the code without changing its semantics. For example, a constant like 4 might be replaced by expressions like $2 + 2$ or $(0 * 3) + 100 - 96$. This type of changing might indeed complicate the human inspection of the source code after ABF has been applied. Therefore, a post-processing is required. One possibility is to use a new GP search, in which the fitness function is based on the structural difference from the original program, but with the constraint that all the unit tests need to be kept satisfied.

The ABF task can be considered as a sub-field of Automatic Programming (AP), which consists of automatically generating code from a specification (and a set of unit tests can be consider as a special example of specifying a wanted behaviour). In fact, we can model ABF as an AP tool in which the code of the buggy program can be exploited for getting better results. Hence, ABF is a strictly easier task than AP.

Unfortunately, AP is a very difficult task [16], and most of the work done so far is based on rule-based transformations from formal specifications, which cannot be applied when the gap between high-level specifications and low-level implementations is too wide. On the other hand, recent work with GP [14] and our previous [2] (on top of whose framework we are building up our first ABF prototype) does not suffer from that problem, and they have given first promising results. Because ABF is easier than AP, we speculate that the ABF technique that we describe in this paper might have non-trivial applications in the coming years.

The paper is organised as follows. Section 2 describes a general framework for fixing bugs by using GP and a set of unit tests. Section 3 explains how to exploit the formal specification if it is given with the buggy program. Besides making possible the design of more descriptive fitness functions, a formal specification is an essential element to improve the performances by the use of *co-evolution*. Other possible Software Engineering tasks that might be addressed with this approach are described in section 4. Finally, section 5 concludes the paper.

2. AUTOMATIC BUG FIXING

In this paper we propose a framework for automatically fixing bugs. The software under analysis (SuA) and a set of unit tests T are the only things required as input (the exploitation of formal specifications will be explained in the next section).

We represent the SuA as a genetic program, and we use GP to evolve it by using T as the training set. The goal is to use GP to evolve the SuA until it is able to pass all the tests in T. In particular, a program is rewarded on how many unit tests it is able to correctly pass.

We want that, if the SuA has a bug, then there should be at least one unit test in T that is failed. Before applying ABF, a testing phase is hence done on the SuA for generating T. If no unit test is failed, the software developer might suppose that there is no bug, hence he would not use ABF. The key point here is that we need at least one failed test for applying ABF.

```
<(5,-2,3), 0> // 0 represents 'not triangle'
<(4,3,6) , 1> // 1 represents 'scalene'
<(9,9,16), 2> // 2 represents 'isosceles'
<(3,3,3) , 3> // 3 represents 'equilateral'

function classifyTriangle(a, b, c)
   return  a + b - c;
```

Figure 1: An example of a training set for the Triangle Classification problem [12] and an incorrect simple program that actually is able to pass all these test cases.

The use of a population is a key feature of GP, but instead of sampling random trees, we seed the first generation of evolutionary programs with copies of the SuA. In other words, in the first generation all the individual are equal to the SuA. We do that because we exploit the assumption that software developers do not write programs at random [7], hence we are assuming that the SuA is structurally near to the optimal solution.

At any rate, the scenario described in this paper is very different from the normal applications of GP:

- the training set T 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 overfitting the training set, it has to over-fit it. In fact, even if only one test in T is failed that means that the program still has bugs.
- in population based search algorithms there is the issue of maintaining diversity among the individuals to avoid premature convergence. Here the problem is more complicated, because diversity is still a precious feature, but we start from the assumption that the SuA is structurally close to the optimal solution.
- because the individuals of the first generation are all identical to the SuA, the role and importance of crossover and mutations in GP is different from the normal applications in which the first generation is randomly sampled.
- the Occam's Razor is a well known principle that is widely applied in machine learning. In a nutshell, we can briefly describe it as the preference of the simplest solution for a given problem. Hence, if two classifiers have the same performance on a training set, the simplest (i.e., the least complex, smallest, etc.) should be chosen. However, in the particular context of ABF that principle can be very harmful. In fact, there can be particular relations that are applied only to the test cases in T, and a very simple (i.e., small) incorrect program might pass all those unit tests, and hence given back by the ABF framework as the new patched version of the SuA. Figure 1 shows an example of this problem. Therefore, the structure and size of the SuA needs to be appropriately included in the fitness function.

The fitness function for the programs is based on how many unit tests they pass. However, using as fitness function only the number of passed unit tests might give not enough gradient for the GP

evolution. A simple extension would be to sum up the errors of the outputs from their expected results on all the unit tests in T. In other words, if the output of the computation on a unit test is a value K and the expected result is E, then using for example $abs(K-E)$ would give more gradient than a simple 0 if the test is passed and 1 otherwise.

At any rate, more sophisticated fitness functions could be employed. For example, considering each unit test as a different objective and then using multi-objective optimisation techniques [6] might be worthy.

The performance of our framework heavily depends on the quality of the training set T. In other words, ABF depends on the accuracy of the previous testing phase. Hence, poor results of ABF might depend on the testing rather than on ABF itself. Furthermore, for avoiding problems like for example the one described in figure 1, it would be better to have a relatively large T.

3. EXPLOITING CO-EVOLUTION

If a formal specification of the SuA is given, we can use it as an *oracle*. Hence, we can sample as many unit tests as we want, because we can use the oracle to automatically check the results. At any rate, the problem of choosing the appropriate test data still remains, but we can include any state-of-the-art testing system [11] in our framework.

We can sample a training set T, and then we apply the same GP system described previously. However, choosing a too large set is not feasible, otherwise the computational cost of the fitness function for GP would be too high. On the other hand, a small training set can easily lead us to problems as in figure 1. The solution is to try to have different test cases at each generation (or at each longer period of time) of GP, and because we have an oracle, we can do it. How to choose new test cases? Simply, we use state-of-the-art testing systems to sample new unit tests that make fail as many programs in the current GP population as possible. These new unit tests replace the ones in T.

This algorithm leads the system to a competitive *co-evolution*, in which the genetic programs are rewarded on how many tests they pass, and on the other hand the unit tests are rewarded on how many programs they make fail. This co-evolution is similar to what in nature happens between *predators* and *prey*, and hopefully an *arms race* will take place and bring to the evolution of a bug-less program. For example, faster prey escape the predators more easily, and hence they have higher probability of generating offspring. That 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.

The advantages of co-evolution are evident, in fact at each generation of GP it tries to optimise the best training set T. Unfortunately, producing an arms race in co-evolutionary algorithms is more difficult than it looks [8]. In fact, problems as *mediocre stable states* and *loss of gradient* might rise.

A mediocre stable state occurs when at each generation both the predators and prey seem to positively evolve, but actually they are following an infinite circular pattern without any real improvement. That happens because the co-evolution can lose memory of what happened in the past (i.e., the previous generations). Techniques based on *archives* have been proposed to handle this problem (e.g., [17]), in which at each generation some individuals are stored in a separated set (which often is implemented as a queue of finite size). The fitness of current generation individuals is also based with the interaction with those old individuals. It is important to note that these archives are conceptually similar to *regression test* [12] suites.

Moreover, if a formal specification is provided, more sophisticated fitness functions can be employed. For example, instead of calculating an euclidean distance between the output of a computation and its expected result, we can use a more descriptive heuristic that is based on how badly the formal specification is not satisfied. In particular, our heuristic is inspired by the Tracey's work on Black Box Testing [21].

For example, in a sorting algorithm, an euclidean distance would not give any particular reward when arrays have been partially sorted or when the constraint that the output array should be a permutation of the input one is satisfied. On the other hand, an heuristic based on the formal specification can recognise these positive cases, and it can evaluate them accordingly.

Our previous work on AP [2] used the same co-evolutionary framework for evolving programs to satisfy a formal specification. The difference is that, instead of exploiting a buggy program version that actually can be seen as solution that is close to the optimal one, it starts with a random population of programs. Hence, the task that we successfully solved in that work on some small programs is more difficult than ABF.

4. OTHER POSSIBLE APPLICATIONS

The framework that we have described in this paper could be applied, with some modifications, to solve other Software Engineering tasks (although at the moment we are not planning of investigating them), for example:

- when a formal specification is employed, 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.

- in the fitness function of the programs there could be included non-functional criteria. Instead of starting with a buggy program, a correct version would be provided. Our framework would be used to search for optimising these non-functional criteria without changing the semantics of the program. A valuable application would be for example the optimisation of code that should run on low-resource systems.

- our co-evolutionary approach could be helpful for Evolvable Software systems (e.g., [4]).

- instead of being used off-line, ABF could be embedded in the released software. If during its use a fault is found, that would automatically trigger the ABF framework to fix it. That would be very helpful for software that, although a patch can be provided, cannot be easily updated once released, like for example in hardware employed in space missions.

5. CONCLUSIONS AND FUTURE WORK

In this paper we have proposed a novel approach to automate the expensive software engineering task of fixing bugs. This approach is more ambitious than the previous literature on automated debugging, because we are not limited in only localising the bugs, in fact we also try to fix them in an automatic way. To our best knowledge, this is the first proposed general approach for automating that task.

Our system does not suffer of the limitation of requiring at least one test case that is passed, as in general it happens for previous automated debugging frameworks. In fact, in the extreme case in

which a bug makes fail the computation on each possible input, our framework can still be applied, although with likely worse performance.

We are building our prototype on top of our previous system for AP [2], and we are already getting promising results (to appear in [1]). In the future, we are planning to try to fix bugs in real-world Object-Oriented software, which also gives dire challenges from the point of view of testing [3]. In particular, we are planning to analyse how some software projects have been fixed by developers, and then applying our framework to see whether the same patches could be automatically evolved. The application to solve real-world problems would give strong evidence of the validity and feasibility of our novel approach.

Our framework is composed by several different components (e.g., GP engine, testing engine, fitness functions from formal specifications and co-evolution), that are combined together in a novel context. Hence, there is a lot of space for research on how to improve that combination and on how to exploit domain knowledge of the problem to achieve better performance.

Unfortunately, GP is computationally expensive. Although parallel implementations and runtime assembler compilation of the programs might speed up the system by several orders, the scalability issue still remains. In fact, the search space of GP is extremely huge, and it increases approximately exponentially with the size of the programs [10]. Therefore, we do not suggest to use our ABF framework on entire complex systems. On the other hand, it would be more feasible to apply it to fix units of computation (e.g., single functions) in a similar way as unit tests are used. Hence, the term *unit bug fixing* could be more appropriate.

At any rate, although GP might be computationally expensive, it has the great advantage that it might potentially fix any type of bug. In fact, it has indeed the potentiality of evolving a correct program even from scratch [2], although that mainly depends on the quality of the used fitness functions.

To reduce the computational cost of GP, it is important to narrow down the search space of the evolutionary operators. For doing that, one possibility could be to include state-of-the-art automated debugging techniques in our framework.

Although many research questions still need to be investigated and solved, we think that our framework for bug fixing might be already a 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 solution, but that are located in parts of the source code that are very complex for a human to analyse.

6. ACKNOWLEDGMENTS

The author is grateful to Xin Yao, Per Kristian Lehre and Ramón Sagarna for insightful discussions. This work is supported by an EPSRC grant (EP/D052785/1).

7. REFERENCES

[1] A. Arcuri and X. Yao. A novel co-evolutionary approach to automatic software bug fixing. To appear in the IEEE Congress on Evolutionary Computation (CEC), 2008.

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

[3] A. Arcuri and X. Yao. Search based software testing of object-oriented containers. *Information Sciences*, 2008. Accepted for publication.

[4] H. Barringer, D. Rydeheard, and D. Gabbay. A logical framework for monitoring and evolving software components. In *IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 273–282, 2007.

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

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

[7] R. A. DeMillo, R. J. Lipton, and F. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

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

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

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

[11] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, June 2004.

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

[13] G. Palshikar. Applying formal specifications to real-world software development. *IEEE Software*, 18(6):89–97, 2001.

[14] M. Reformat, C. Xinwei, and J. Miller. On the possibilities of (pseudo-) software cloning from external interactions. *Soft Computing*, 12(1):29–49, 2007.

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

[16] C. Rich and R. C. Waters. Automatic programming: myths and prospects. *Computer*, 21(8):40–51, 1988.

[17] C. D. Rosin and R. K. Belew. New methods for competitive coevolution. *Evolutionary Computation*, 5(1):1–29, 1997.

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

[19] 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.

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

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

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

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