Coevolving Programs and Unit Tests from their Specification

Andrea Arcuri and Xin Yao
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,X.Yao}@cs.bham.ac.uk

ABSTRACT

Writing a formal specification before implementing a program helps to find problems with the system requirements. The requirements might be for example incomplete and ambiguous. Fixing these types of errors is very difficult and expensive during the implementation phase of the software development cycle. Although writing a formal specification is usually easier than implementing the actual code, writing a specification requires time, and often it is preferred, instead, to use this time on the implementation.

In this paper we introduce for the first time a framework that might evolve any possible generic program from its specification. We use the Genetic Programming to evolve the programs, and at the same time we exploit the specifications to coevolve sets of unit tests. Programs are rewarded on how many tests they do not fail, whereas the unit tests are rewarded on how many programs they make fail. We present and analyse four different problems on which this novel technique is successfully applied.

Categories and Subject Descriptors

D.1.2 [**Software Engineering**]: Automatic Programming; D.2.5 [**Software Engineering**]: Testing and Debugging; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search

General Terms

Algorithms, Experimentation

Keywords

Automatic Programming, Coevolution, Software Testing, Genetic Programming, Formal Specification, Sorting

1. INTRODUCTION

Since the 1950s the goal of generating programs in an automatic way has been sought. A user would just define what he expects from the program (i.e., the requirements), and it should be automatically generated by the computer without the help of any programmers. This goal has opened a field of research that is called *Automatic Programming*. Unfortunately, this task is much harder than expected [11]. Transformation methods are what are usually employed to address this problem. The requirements need to be written in a formal specification, and sequences of transformations are used to transform these high-level constructs to low-level implementations. Unfortunately, this process can rarely be completely automated, because the gap between the high-level specification and the target implementation language might be too wide.

Software Testing is used to find the presence of bugs in computer programs [9]. If no bugs are 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. Automating the testing phase is an important goal that is greedily sought. Search based techniques have been applied to tackle this task with promising results [8].

Genetic Programming (GP) [6] is a paradigm to evolve programs. A genetic program can be often seen 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 than applied on the programs to generate new offspring. GP has been principally applied to solve real-world learning problems.

In this paper we present a novel framework for automatically generating programs from a formal specification. A population of candidate programs coevolves with a population of unit tests. GP is used to evolve the candidate programs, whereas search based software testing techniques are employed to evolve the unit tests. The fitness values of the candidate programs depend on how many tests they do not fail, whereas the unit tests are rewarded based on how many programs they make fail. The programs are pressured to evolve to fix the bugs that the current unit tests find in them, while the unit tests are pressured to find new bugs. A set of problems is used to validate the novel framework. Although coevolutionary algorithms are not novel, in this paper for the first time a formal specification is used to automatically create the fitness functions. This makes it possible to apply this framework to any problem that can be defined with a formal specification.

The paper is organised as follows: section 2 describes the characteristics of the problem. In section 3 there is explained how the coevolution is applied. A case study for validating the novel framework follows in section 4. A description of the related work can be found in section 5. Finally, section 6 concludes the paper.

2. EVOLUTION OF THE PROGRAMS

Given a specification of a program P, the goal is to evolve a program that satisfies it. To achieve this result, GP is employed. At each step of the evolution of GP, the fitness of each program is evaluated on a finite set T of unit tests that depends on the specification. The more unit tests a program is able to pass the higher the fitness value will be rewarded. This set T should be relatively small, otherwise the computational cost of the fitness function would be too high.

The set T is different from a normal training set. Let X be the set of all the possible input variables for P, and Y be the set of all the possible output. Given any input variable $x \in X$, we do not have the expected value $y^* = p^*(x)$, with p^* being the optimal program that we want to evolve and $y^* \in Y$ being the expected result for the input x. Hence, a unit test $t \in T$ instead of being seen as a pair $< x, y^* >$ (as a typical element of a training set would be), is a pair $< x, c >$, where c is a function $c(x, y) : X, Y \rightarrow \Re$. The function c gives in output a value of 0 if y is equal to y^*, otherwise a real positive value that expresses how different the two results are. A higher value means a bigger difference between the two results. Because the function c is the same for each $t \in T$, we can simplify the notation by considering only the input x for a unit test. In other words, $t \in X$ and $T \subseteq X$. A program $g \in G$, where G is the set of all the possible programs, is said to pass a test $t \in T$ iff $c(t, g(t)) = 0$.

How to automatically derive the function c will be explained later in this section. It is important to note that the function c is not required to be able to compute the expected result y^* for an input x. Being able to state whether a particular result k is correct for an input x (i.e., $k = y^*$) is enough. That can be done even without knowing the value of y^*. However, we introduce here an example to clarify this concept. Assume that P is a sorting algorithm that takes as input an array of integer and sorts it. Given as input an arbitrary array $x =< 4, 3, 2, 1 >$, if the output of g is $y =< 1, 4, 2, 3 >$, we will not need to know $p^*(x)$ to conclude that $g(x) \neq p^*(x)$, because y is not sorted. We can conclude that an array is not sorted by looking at the specification of the sorting algorithm. In this particular case, we have a 4 before a 2, which is enough to conclude that the array is not sorted. Although a specification can state whether an array is sorted or not, it cannot say how to sort it.

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

- the training set T can be automatically generated with any cardinality. There is no need of any external entity that, for a given set of x, says which are the relative y^*.
- usually, because there is available only a limited number of pairs $< x, y^* >$, all of them are used for the training. On the other hand, in our case we have the problem of choosing a subset T, because using the entire X is generally not feasible.
- 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. E.g., $\forall x \in X \bullet g(x) = p^*(x)$. 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 specification is not satisfied.

2.1 Fitness Function for the Programs

At each step/generation i of the coevolution, the current population of genetic programs G_i is executed on the test set T_i. The fitness of each $g \in G_i$ is based on its ability to pass the unit tests in T_i. It is a minimisation problem. If a g passes all the tests, it will have $f(g) = 0$. Otherwise, we will have a score that represents how badly the program breaks the specification. In other words, if the predicate of the specification is true, we will get zero, otherwise we will get a score as higher as the predicate is far from being evaluated as true. Formally:

$$f(g) = \sum_{t \in T_i} d(P, g(t)) \ . \qquad (1)$$

The distance function d calculates how far the result $R = g(t)$ is from satisfying the post-condition P of the specification. Such a distance is the same one employed in [12] for Black Box Testing. The difference is only on what it is applied to. In that work the pre-condition of the function is conjuncted with the negated post-condition. The distance is applied on that predicate. A distance value 0 means that a fault that breaks the specification is present for the input t. On the other hand, in this paper we use this distance directly on the post-condition. A distance value 0 means that the post-condition is true, so the program is correct for that particular input t. In other words, to calculate $d(P, g(t))$ the program g is executed with input t and then the result R is compared against the post-condition P to see whether the result is correct or not.

A problem with GP is *bloat*, i.e., the increasing of the size of the programs with redundant or not useful code. Parsimony controls might be used to limit this problem (e.g., including in the fitness function a penalty term based on the size of the program). However, in our particular case, parsimony can be deceptive. In fact, although during the evolution of the programs the optimum p^* can be evolved, even if elitism is used in the GP evolution, p^* can be lost. For example, if a g_z is not optimal, smaller than p^* and able to pass all the tests in the current population of tests, then g_z will be preferred to p^* although it is not optimal. It is noteworthy that such a behaviour would not happen in a usual coevolutionary algorithm, in which the fitness value of an individual is solely based on interactions with the opposite populations. The use of archives might help to handle this problem.

3. COEVOLUTION

Choosing a good training set T is not easy. An ideal set T^* would be one that, if a program $g \in G$ passes all the unit tests $t \in T^*$, then it is guaranteed that it will pass all the tests in X. That means that if g completely fits the data in the training set, then it is guaranteed that g is correct. A trivial set that satisfies such a constraint is $T = X$. However, the set should be as small as possible, otherwise the computational cost of the fitness function of g would be too high.

The problem is that finding such an optimal $T^* \neq X$ is impossible, because in the set G of all possible programs there is always at least one program that fits all the data in this hypothetical T^*, and can have, for example, a special "if" statement based on the value of a $t \in X \setminus T^*$ that makes the program fail on that test t. Although the use of a limited set of primitives (e.g., without "if") might not yield this problem, this is not true in the general case. At any rate, even if an optimal T^* existed, there is no guarantee that the GP will be able to evolve a program that fits all the data in T^*. However, a good strategy for choosing a training set T is needed, because the performance of the GP depends on it. In the following, we describe the use of *coevolution* to find T.

Because the elements t of a training set can be automatically generated, we can use a different training set T_i at each different generation i of the GP. To simplify the following discussions, we assume that GP maintains a population with only one program g_i during generation i. A unit test on which g_i fails gives more information to assist the evolution of g_i than a passed unit test. In fact, if a test t is failed, the program g is evolved until it passes it (or until the GP has been stopped). In the case in which all the unit tests are passed, g_i is not able to evolve any further. If $T \neq T^*$, the program is not necessarily correct and there is no more guidance to its evolution. Hence, the idea is to let the test set T_i *coevolve* with the program g_i, with the aim being to give to g at each generation a set of tests that it is not able to pass yet. Let N_i be the set of all the possible unit tests at generation i for which g_i passes, and F_i be the set of tests for which it fails. We have $N_i \subseteq X$, $F_i \subseteq X$ and $N_i \cup F_i = X$. A strategy S is used to choose the next T_{i+1} from T_i and g_i. A reasonable strategy would be one that generates new unit tests that the current g_i will fail. Given:

$$v(i,t) = \begin{cases} 0 & \text{if } t \in N_{i-1} \ , \\ 1 & \text{if } t \in F_{i-1} \ , \end{cases} \tag{2}$$

we can formulate the choice of the strategy as an optimisation problem, in which the fitness to maximise is:

$$f(S) = \sum_{i=0}^{i=H} \sum_{t \in U_i} v(i,t) \ , \tag{3}$$

where H is the number of the allowed generations for the GP, and $U_i \subseteq T_i$ is the largest subset of T_i in which all the elements are unique. For the sake of clarity, the cardinality of the training sets T_i does not change during the search, and there can be redundant unit tests in a T_i. We employ the *coevolution* of programs and their unit tests to solve this optimisation problem. However, it is important to remember that the goal is to evolve a program g^*, not to find the best sequence of T_i. Finding this sequence is a secondary task that is addressed because it will help to solve the principal problem.

Even if the best strategy S^* is used, none of the T_i will be the optimal T^* (that usually does not exist). Furthermore, not only using S^* does not imply that g^* will be evolved and not only it is true that g^* might be evolved with a sub-optimal strategy, but it might even happen that S^* forbids such an evolution. An example will help to clarify this statement: consider a boolean function that takes as input one integer and says whether it is positive or not. The program g^t returns always true, whereas g^f returns always false. For simplicity we consider a training set of size 2. If all elements in T_i are positive, the GP might evolve g^t for g_i, and that completely fits the data. At the next generation, the set T_{i+1} in which all the elements are negative makes g_i fails all the tests, hence it optimises function (3). Then, the GP might evolve g^f for g_{i+1}, and that, again, completely fits the data. The system can go on in this way, switching the training set from all positive to all negative, with the GP that always evolves either g^t or g^f. This strategy is optimal regarding function (3), but it is obviously not a good strategy. A strategy that at each generation would generate T_i with both positive and negative values would give much better results. The choice of a strategy depends on the particular *solution concept* [2] that we want to optimise. However, usually we do not know which are the underline objectives of a problem. Moreover, although a g_i can pass all the unit tests in T_i, usually there is no guarantee that any g_j with $j > i$ will be able to do the same. Hence, function (3) captures some desired properties that we want from a strategy, but it is not enough. For example, keeping in T_i some previous passed tests might help to prevent the devolution of the programs g_j.

In literature, this problem of continue changing of the fitness landscape and memory loss is named the *Red Queen Effect* [10]. Archive methods have been proposed to handle it, with a lot of effort spent on designing algorithms that guarantee a monotonic improvement of the results [5]. Another problem in coevolutionary algorithms is for example the *loss of gradient*.

The simplest strategies are generating each T_i at random or generating only one set at random and using it for all the training. Although these two techniques might sound naive, they are often employed in literature.

When a coevolution is employed, for each $t \in T_i$ is calculated a fitness value. A fitness function is used to reward unit tests that the programs in the current population G_i have difficulties in passing. In contrast to function (1), this is a maximisation problem. A value of 0 means that all the programs in G_i pass the test t. The fitness function is:

$$f(t) = \sum_{g \in G_i} d(P, g(t)) \ . \tag{4}$$

It is important to outline that the distances $d(P, g(t))$ are the same used for function (1). Hence, they are calculated only once, and then they are used for both the fitness functions.

Once the fitness function is calculated for each test in T_i, a Genetic Algorithm [4] is employed to evolve the next test set T_{i+1}.

Programs	Random	Coevolution
Sorting	75	62
MaxValue	11	88
MaxOccurrence	61	52
AllEqual	54	74

Table 1: Number of evolved programs on a total of 100 **that are correct.**

4. CASE STUDY

To validate our approach, we considered four different functions that take as input an array: the sorting of the array (*Sorting*), the search of the highest value (*MaxValue*), the search of the highest occurrence (*MaxOccurrence*) and testing whether all the elements are identical (*AllEqual*). Due to space limitations we are not able to provide here their formal specifications.

For evolving the programs, the open source library ECJ [7] has been employed. The population size of the programs is 1024, whereas the population size of the test cases is 32. The allowed generations are 100. Each 5 generations, the test cases are separately evolved for 1024 generations using only the current best program for the fitness function. In this special case, no program evolves. For each problem, 100 experiments were carried out. Table 1 shows how many time a correct program was found. Again, due to space limitations we cannot provide full details of the experiments.

On *Sorting* and *MaxOccurrence*, coevolution gives worse results than a random choice of the test cases. This is due to the fact that we used only a simple coevolutionary algorithm, which performance can be improved for example by using archives and speciation. However, we get significant improvements on both *MaxValue* and *AllEqual*.

5. RELATED WORK

To the best knowledge of the authors, no previous work on coevolving programs with test cases using a formal specification exists. However, in literature there is some work that somehow is related to the problem addressed in this paper.

In 1990, Hillis investigated how to evolve a sorting network [3]. It modelled the task as an optimisation problem, in which the goal is to find a correct sorting network that does as few comparisons of the elements as possible. It used evolutionary techniques to search the space of the sorting networks, where the fitness was based on a finite set of tests (i.e., sequences of elements to sort): the more tests a network was able to correctly pass, the higher fitness it got. For the first time, Hillis investigated the idea of coevolving such tests with the networks. His experiments showed that, when the coevolution was used, shorter networks were found.

Evolving a sorting algorithm has been already attempted in the past (e.g.,[1]). However, none of those attempts deal with the issue of creating the best training set. Furthermore, in those cases the sorting problem is used only as an example in which different features and problematics of GP could be studied. Their goal is not to build up a system able to automatically generate different types of software.

6. CONCLUSION

Evolving programs from their specifications is a charming field that this paper addresses for the first time. This goal is here achieved by a coevolution of genetic programs and unit tests. Although the work presented in this paper is in a preliminary state, it gives to the Natural Computation and Software Engineering communities the important contribution of showing that this approach is feasible.

At any rate, a lot of research questions are still unanswered and need to be investigated in the future. The main question is whether this approach might scale to real-world software. Although our experiments on four non-trivial problems were successful, at the current state of our research we are not able to state whether the fitness functions that are automatically generated from the specifications can give enough gradient for evolving real-world software.

7. ACKNOWLEDGEMENTS

This work is supported by an EPSRC grant (EP/D052785/-1). The authors wish to thank Damien Jade Duff for the useful discussions.

8. REFERENCES

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

[2] S. G. Ficici. *Solution Concepts in Coevolutionary Algorithms*. PhD thesis, Brandeis University, 2004.

[3] W. D. Hillis. Co-evolving parasites improve simulated evolution as an optimization procedure. *Physica D*, 42(1-3):228–234, 1990.

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

[5] E. D. Jong and J. Pollack. Ideal evaluation from coevolution. *Evolutionary Computation*, 12(2):159–192, 2004.

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

[7] S. Luke. *Issues in Scaling Genetic Programming: Breeding Strategies, Tree Generation, and Code Bloat*. PhD thesis, University of Maryland, 2000.

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

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

[10] J. Paredis. Coevolving cellular automata: Be aware of the red queen. In T. Bäck, editor, *Proceedings of the Seventh International Conference on Genetic Algorithms (ICGA97)*, San Francisco, CA, 1997. Morgan Kaufmann.

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

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