En Garde: Winning Coding Duels Through Genetic Programming

Kiran Lakhotia
CREST, University College London

***Abstract*—In this paper we present a Genetic Programming system to solve coding duels on the Pex4Fun website[1]. Users create simple puzzle methods in a .NET supported programming language, and other users have to 'guess' the puzzle implementation through trial and error. We have replaced the human user who solves a puzzle (i.e. implements a program that matches the implementation of the puzzle) with a Genetic Programming system that tries to win such coding duels. During a proof of concept experiment we found that our system can indeed automatically generate code that matches the behaviour of a secret puzzle method. It takes on average 76.57 fitness evaluations to succeed.**

I. Introduction

In this paper we propose to use a Genetic Programming (GP) system to automatically generate C# code whose behaviour matches that of a secret implementation. In particular, we try to automatically solve *coding duels* on the Pex4Fun website [1]. Pex4Fun is a website provided by Microsoft Research. One of its goals is to provide a game like environment where users can learn or practice programming concepts. One feature of the website is the ability of users to create coding duels. First, a user (or teacher) creates a puzzle method and submits it to the Pex4Fun website. Then, other users can try and win the coding duel by providing an implementation of a method that matches the behaviour of the puzzle method.

Ultimately our goal is to use GP to evolve code that matches the behaviour of an existing software component by using testing. We view testing as a means of checking that our evolved implementation matches a subset of the behaviour of a component whose functionality we would like to replicate. An example application area where this might be of interest is in porting existing software to new platforms [2]. However, in this paper we only focus on using GP to try and win coding duels on the Pex4Fun website.

To help a user solve puzzles, Pex4Fun uses the Pex testing tool [3]. Pex is a dynamic symbolic execution tool for the .NET runtime. It systematically explores a program (or function), starting with a concrete execution. The path taken during concrete execution is also executed symbolically in order to obtain a *path condition*. A path condition describes constraints any program input must satisfy in order to traverse a particular program path. New inputs with which to execute the program are obtained by inverting one of the constraints in a path condition and asking a constraint solver to find a satisfying assignment for the variables denoting program inputs. A full description of Pex (and dynamic symbolic execution) is beyond the scope of this paper and the interested reader is referred to [3].

In the context of Pex4Fun, Pex is used to provide feedback to a user trying to win a coding duel about when their implementation of a method agrees and disagrees with the puzzle method. Consider a method implementation $u(x)$, provided by a user that accepts an integer x as input, and a secret puzzle method $p(x)$ that also takes an integer x as input. Pex will explore a meta program $m(x)$ whose semantics are $m(x) := assert(u(x) == p(x))$ [1].

After each attempt at solving a puzzle the user is given a list of input/output pairs for their program and the puzzle method. Figure I shows an example puzzle from the Pex4Fun website. Figure 2 shows two versions of a method submitted by a user for the example in Figure I, along with the 'feedback' provided by Pex4Fun.

We propose to replace a human user trying to solve puzzles with a GP system. The only information we provide to the GP is the number of inputs for which the puzzle and GP generated method produced the same output, and the number of inputs for which the outputs differed. Unlike a human, the GP does not know *what* the difference in output was, because, in our current system, we do not analyse and include this information in our fitness function.

II. Proposed Genetic Programming Approach

Genetic Programming [4] is a technique to automatically generate programs, based on the use of an Evolutionary Algorithm (EA) [5]. It follows the usual EA cycle of generating a population of candidate solutions, evaluating the solutions using a fitness function, and generating new solutions through the use of genetic operators such as crossover and mutation.

Candidate solutions are represented in a tree form. A program for example might be represented as an Abstract Syntax Tree (AST). The genetic operators of an EA then operate on this tree structure. Before explaining in detail how the GP and its operators work, we first describe the technology used to represent programs as trees.

A. Microsoft CodeDom

We use Microsoft's CodeDom[2] infrastructure to represent candidate C# programs. CodeDom is a namespace provided by Microsoft to enable dynamic source code generation and compilation. The namespace defines types, *e.g.* `CodeStatement`, `CodeExpression`, that can be used to represent the logical

[1] http://www.pexforfun.com/

[2] http://msdn.microsoft.com/en-us/library/650ax5cx.aspx

```
using System;
public class Program {
  // Can you fill the puzzle method to match the secret arithmetic operation?
  public static int Puzzle(int x) {
    return 0;
  }
}
```

Fig. 1. Coding duel challenge used for the proof of concept experiment. This code represents the template for the puzzle method a user (and our GP) has to solve.

User attempt 1 at solving puzzle:

```
using System;
public class Program {
  // Can you fill the puzzle method to match the secret arithmetic operation?
  public static int Puzzle(int x) {
    return x - 1;
  }
}
```

Pex found 1 difference between your puzzle method and the secret implementation.
Improve your code, so that it matches the other implementation, and 'Ask Pex!' again. You are not signed in. Sign In to rate duels and track your achievements. Help

	x	your result	secret implementation result	Output/Exception	Error Message
⊗	0	-1	0	Mismatch	Your puzzle method produced the wrong result.

User attempt 2 at solving puzzle:

```
using System;
public class Program {
  // Can you fill the puzzle method to match the secret arithmetic operation?
  public static int Puzzle(int x) {
    if(x <= 0) return 0;
    else return x;
  }
}
```

Pex found 2 differences between your puzzle method and the secret implementation.
Improve your code, so that it matches the other implementation, and 'Ask Pex!' again. You are not signed in. Sign In to rate duels and track your achievements. Help

	x	your result	secret implementation result	Output/Exception	Error Message
✓	0	0	0		
⊗	int.MinValue	0	int.MinValue	Mismatch	Your puzzle method produced the wrong result.
⊗	1	1	-1	Mismatch	Your puzzle method produced the wrong result.

Fig. 2. Example implementations for the puzzle method from Figure I along with the feedback provided by Pex.

structure of a program, independent of a specific programming language. A CodeDom representation can then be omitted as source code for different (.NET supported) languages such as C#, and compiled to an executable or system library (*i.e.*, `dll`).

To date, the CodeDom namespace does not include functionality for parsing existing source code into a CodeDom representation. For the purpose of this paper however it was necessary to be able to parse C# code into a CodeDom structure in order to provide an initial skeleton framework for the GP to work with. The structure of this skeleton framework depends on the information included with a puzzle. In some cases this merely represents a function signature (*e.g.*, similar to the example shown in Figure I). However some puzzle creators also include 'hints' in a puzzle. These could be in the form of comments, or, more importantly include a partial implementation of the puzzle method. If the GP did not have the facility to include such information in its program representations, then it would unnecessarily expand the search space for the GP, thus making it harder to find a solution.

In order to parse C# code into a CodeDom structure we adapted parts of the ILSpy[3] library. Specifically, we used `ICSharpCode.Decompiler.Ast.AstBuilder` to parse existing C# code into an AST, and then used the `ICSharpCode.NRefactory` library to convert the AST into a CodeDom representation.

B. Genetic Programming System

Our GP system is currently only able to generate conditional (*i.e.*, `if`) and assignment statements. An `if` statement takes the following form: $if(e1\ op\ e2)$, where $e1, e2$ are of type `CodeExpression` (*e.g.*, constants, variable references, binary operators, ...) and $op \in \{>, >=, <, <=, ==, !=, \&\&, ||\}$. An assignment statement takes the form $v = e1\ opA\ e2$, where v is a local variable and $e1, e2$ are of type `CodeExpression`. The arithmetic operator for assignment statements $opA \in \{+, -, *, /\}$, can denote *addition, subtraction, multiplication and division* operations. Note

[3] http://www.ilspy.net/

that an assignment can only be made to local variables, not method parameters. This restriction on assignments stems from a requirement for the coding duels imposed by the creators of Pex4Fun[4].

To enable modification of parameter values passed into the puzzle method, while satisfying the requirements of Pex4Fun, the GP creates a local variable for each formal parameter of a puzzle method, and assigns the local variable the value of the corresponding formal parameter. Currently this operation is only supported for value types (*e.g.*, `int`, `bool` etc.) and the GP cannot clone Array or other objects.

The starting point for the GP is the (C#) code fragment a user sees on the Pex4Fun website. This code fragment is then converted into a CodeDom representation and the GP creates a set of local variables, one for each formal parameter as described above. Next, a population of candidate solutions is generated by extending the starting CodeDom tree with randomly generated sub-trees. A sub-tree denotes statements to be added to the method body of the puzzle.

The GP uses two genetic operators for reproduction: a crossover and a mutation operator. The crossover works at the statement level; once two CodeDom trees are selected for reproduction, all (top level) statements (of the puzzle method) from each tree are converted to a list. The operator then proceeds with a standard one point crossover operation, swapping statements between the two lists. Since the two lists of statements can be of different length, the crossover point is chosen to lie within the range of the shorter of the two lists. Note that the last statement in a list cannot be chosen as a crossover point to ensure the GP does not accidentally remove the last `return` statement of a method. The current implementation of the crossover operator means only top level statements are swapped between trees; for example, the operator cannot swap the *true* or *false* blocks of an `if` statement, only the entire `if` statement.

The mutation operator implements three operations: adding a randomly generated statement, removing a statement and mutating an expression. The *add* operation simply generates a random assignment or `if` statement. The *remove* operation visits every statement in a CodeDom tree (including nested statements) and randomly picks a statement to remove from the tree. The *mutate expression* operation visits all `CodeExpression` nodes in a CodeDom tree and picks a random expression to mutate. Valid mutations are changing a binary operator to one of $\{>, >=, <, <=, ==, != , \&\&, ||\}$, changing a variable reference (either to another variable, or, replacing it with a constant value), or mutating an existing constant value by replacing it with a newly generated random value.

III. Proof of Concept Experiment

In order to investigate if the proposed GP system is in principal able to solve coding duels on the Pex4Fun website, we carried out a proof of concept experiment. We selected an existing coding duel from the Pex4Fun website (shown in Figure I[5]). Note that we do not know who created and submitted this coding duel to Pex4Fun.

TABLE I. The GP configuration used for the proof of concept experiment.

Population size	10
Maximum generations	200
Parent selection strategy	Roulette wheel selection pressure: 2.0 number of parents to select: 5
Mutation probability	0.6
Probability of generating an assignment statement	0.7
Probability of assigning `null` to objects	0.8
Probability of using constant value in expression	0.2
Min/max ranges for random constant values	[-10, 10]
Probability of generating new statement	0.25
Probability of negating the value of a variable	0.4

We then configured the GP to use the settings shown in Table I. The choice of parameters with which to run the GP was arbitrary. However, we performed some preliminary experiments that indicated they represent a good configuration out of all the possible settings we tried.

When creating the initial population, the GP generates random statements that are appended to the puzzle method body. The *Probability of generating new statement* parameter controls when the GP stops adding statements. The purpose of this parameter is to control the size of the generated CodeDom trees.

If a random statement is created, the GP favours assignment statements (70%) over conditional statements. When an assignment statement is created, the GP chooses to create a new expression on the right hand side of the assignment 60% of the times, and negates the value of a variable 40% of the time. The parameter *Probability of using constant value in expression* controls whether to reference a variable in an expression, or, whether to create a constant value. In case the GP uses a constant value, it generates random values in the range $[-10, 10]$.

We use an elitist steady-state GP; in every generation existing members of a population are replaced by fitter offspring. The size of the population remains constant and is limited to 10 candidate solutions. The GP is allowed to run for a maximum of 200 generations. If the GP is not able to solve the coding puzzle within this limit, the GP terminates. We monitor the Pex4Fun website to check whether the coding puzzle has been solved or not. If the GP finds a solution, then the website displays a winners medal.

To select parents for reproduction we use a roulette wheel strategy. This selection operator selects 5 candidate solutions for reproduction. From these we generate 5 offspring via repeated application of the crossover operator. Every offspring has a 60% chance of being mutated. During mutation, the GP adds a new random statement with 40% probability, removes a randomly selected statement with 20% probability and mutates a randomly selected expression with 40% probability.

In order to interact with the Pex4Fun website we use Selenium[6]. Initially, Selenium is used to load the website with

[4]See the 'Coding Duels Requirements' section in the documentation for Pex4Fun

[5]Perma-link: http://www.pexforfun.com/default.aspx?language=CSharp&sample=ChallengeArithmetic1

[6]http://seleniumhq.org/

TABLE II. RESULTS OF THE PROOF OF CONCEPT EXPERIMENT.

Number of trials	30
Number of successes	30
Number of failures	0
Average number of fitness evaluations	76.57
Average number of compilation failures	17.6

```
// ------------------------------------------
//  <autogenerated>
//      This code was generated by a tool.
//      Mono Runtime Version: 4.0.30319.1
//      Changes to this file may cause incorrect
//      behavior and will be lost if the code is
//      regenerated.
//  </autogenerated>
// ------------------------------------------
public class Program {
    public static int Puzzle(int x) {
        int local_x = x;
        local_x = (-1 * local_x);
        return local_x;
    }
}
```

Fig. 3. Example implementation generated by our GP system that matches the behaviour of the puzzle we tried to solve.

the perma-link for a puzzle and grab the source code for the puzzle outline from the website. This string is then converted into a CodeDom representation using our extensions of the ILSpy library.

During every fitness evaluation, the GP serializes the CodeDom representation of the candidate solution to C# code and attempts to compile the generated source code. If the compilation fails, we assign a penalty score (*i.e.* a fitness value of 0) to the candidate solution. Otherwise we use Selenium to update the code field on the Pex4Fun website and start the Pex exploration (by simulating a user clicking the 'Ask Pex' button). This will return a results table like the ones shown in Figure 2. For each test input the table displays the output produced by the GP solution and the secret puzzle, and if the outputs are equal or differ.

The fitness function used during the proof of concept study is simply the ratio of matching input/output values for a GP generated solution and the secret puzzle implementation.

Table II shows the results of the proof of concept experiment. We ran the GP 30 times, using different random number seeds, on the selected coding duel. The GP was able to win the coding duel in all of the 30 trials. Figure 3 shows one of the solutions generated by the GP. On average it took 76.57 fitness evaluations (*i.e.* about 13 generations) to find the solution. During the search, an average of 17.6 programs generated by the GP did not compile (and thus received a fitness score of 0).

IV. CONCLUSIONS AND FUTURE WORK

In this paper we proposed to use Genetic Programming to automatically generated code that matches the behaviour of a secret implementation. The general principle is to try and replicate the code of a black-box component by simply observing input/output relationships. We used the Pex4Fun website as a means to try out our approach. The website enables users to create and participate in *coding duels*.

A coding duel consists of a user creating a simple method (puzzle) and other users having to 'guess' the semantics of the puzzle method. In order to help a user solve a puzzle they can use the testing tool Pex to generate inputs to both, their 'guess' and the puzzle. Pex will then show the differences in input/output pairs between the two implementations.

We carried out a proof of concept experiment of our proposed Genetic Programming system on one of the coding duels that exist on the Pex4Fun website. Our system was able to generate code that matches the behaviour of the selected puzzle in all of the trials we performed.

The next step is to try our approach on more puzzles and see whether we can win more coding duels. We expect to have to tune the settings shown in Table I in order to find a good general performance of our GP. We are also working on adding the capability to generate loops to our GP, in case this feature is required by some of the puzzles.

We also plan to experiment with different fitness functions. In particular we are working on including the *difference* in output between a puzzle and GP solution in the fitness computation. In addition we are considering to generate very large programs to start with. We expect larger trees to generate more input/output pairs, and thus provide more information about the difference in behaviour of a puzzle method and candidate solution. These measures are designed to increase the amount of information available to the GP, and we anticipate that this will have a positive impact on the effectiveness of our system.

ACKNOWLEDGEMENT

We would like to thank Prof. Mark Harman and Dr. William Langdon from CREST, University College London for their discussions and feedback on the proposed idea. Kiran Lakhotia is funded through the EU project FITTEST (ICT-2009.1.2 no 257574).

REFERENCES

[1] N. Tillmann, J. de Halleux, T. Xie, and J. Bishop, "Pex4fun: Teaching and learning computer science via social gaming," in *CSEE&T*, 2012, pp. 90–91.

[2] W. Langdon and M. Harman, "Evolving a cuda kernel from an nvidia template," in *Evolutionary Computation (CEC), 2010 IEEE Congress on*, july 2010, pp. 1 –8.

[3] N. Tillmann and J. De Halleux, "Pex: white box test generation for .net," in *Proceedings of the 2nd international conference on Tests and proofs*, ser. TAP'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 134–153. [Online]. Available: http://dl.acm.org/citation.cfm?id=1792786.1792798

[4] W. B. Langdon, *A Field Guide to Genetic Programing*. Lulu Press, 2008.

[5] J. H. Holland, *Adaption in Natural and Artificial Systems*. University of Michigan Press, 1975.