Evolving Source Code: Object Oriented Genetic Programming in .NET Core

John Speakman[1]

Abstract. Object Oriented Genetic Programming (OOGP) is a method of Genetic Programming (GP) which gives access to standard language libraries, iteration and object-oriented method calls. The implementation of OOGP in this paper shows the automatic generation of retrievable C# files, following standard C# coding conventions with potential access to the entire C# library, derived from a genetic sequence. This new implementation utilises .net Core Roslyn, using reflection, which allows for retrievable, runtime execution and unloading of dynamically generated C# files with scope control in a modern server environment. Experiments were performed on unit tests to validate the algorithms ability to solve simple programming tasks and generate functional, plain text code.

This is a new prototype designed to eventually act as the main Artificial Intelligence controller for a novel, behaviourally adaptive, Artificial-Life simulation. The design taken in the development of this algorithm stems from a requirement for a high potential variation in behaviour, processing efficiency in a server environment per iteration through generated code and low a minimal number of generations.

1 INTRODUCTION

The ability for an AI to generate, execute and evolve source code allows the potential search space of an AI to be as broad as that of a human programmer. With a broad search space comes the potential for highly dynamic, adaptive behaviour, through evolution, which may be applied directly as behaviour controllers for agents in games and Artificial Life environments.

This paper proposes a Template-Based Genetic Programming prototype, a novel solution to evolve, execute and output plain text code following standard C# coding conventions [1] with dynamic variables and scope handling. This also opens the plausibility of integrating automatic solutions to simple coding problems into future programming paradigms.

A group of genetically varied agents with the ability to reproduce against a fitness function (a quantified assessment of individual agents) will, given the right parameters, trend towards a solution which optimises their fitness score. If the agents are the body of a source code file and the fitness of agents is derived from unit tests (where the output from a method, when given a pre-defined input, is compared to a pre-defined, expected output), we can generate code which automatically solves simple coding problems. These automatically generated files can be used directly in other C# projects or re-used, through reflection, in the same project in which they were generated.

Koza [2]'s Genetic Programming introduced the first model for the use of a genetic sequence to construct a computer program. These early forms of GP used simple expression trees, though further exploration into Object Oriented program space indicated that an Object-Oriented approach can provide significant benefits in comparison with grammar-based systems [3]–[6]: improved performance, direct use of class libraries, iteration, object state, generation of reusable, callable classes, sub-classing existing classes.

To expand potential functionality, this algorithm also permits dynamic variable creation and re-use, using a push/pop Stack, roughly translating to the indentation level in source code, allowing more modular, in-method code. Alternative approaches have been taken for stack-based GP [7], [8], which demonstrated the benefits from the efficiency, simplicity and manipulation of modular architectures introduced using this approach, though had not been used for object oriented scope control or source code generation.

This project is built for ASP.NET Core 3.0, a modern, lightweight, cross-platform framework, compatible with multithreaded server environments, which fully support C#.

Some aspects of the architecture in this approach are taken from Template Based Evolution (TBE) [9], [10], an artificial life algorithm for rapid evolution of subsumption architectures, using a genetic algorithm. Of particular interest from this method is the use of evolving variables through a genome which execute into a template. This simulation varies from TBE, as it dynamically constructs new templates, using smaller templates, into source code from a genetic sequence at run time, where TBE builds into a pre-existing template.

2 AUTOMATIC CODE CONSTRUCTION & EXECUTION

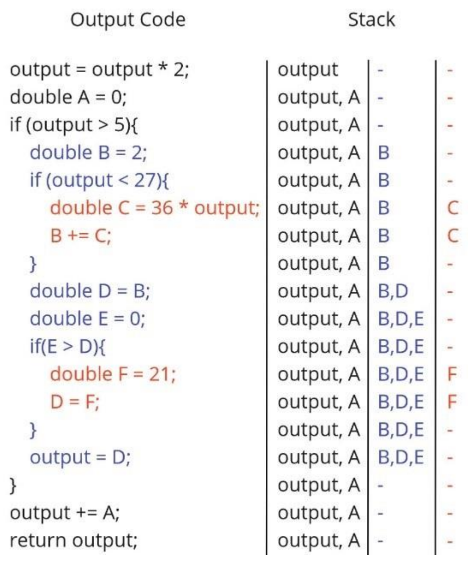
The approach to code generation taken in this paper uses pre-defined templates, which build single lines of code from a list of numeric values. Each line of code takes 3 inputs: the first input is used as a reference to a table of single line templates, which constitute the functionality and body of the code. The second and third inputs are used as values in those lines of code and may be used, non-exhaustively, as a value, variable, or function call.

Generation of reusable variables, scope and code indentation are handled by a push/pop stack: the code constructor accounts for lines which increment and decrement scope, where variables whose associated scope is pushed out of the current stack get removed from the available list of variables. If the genetic sequence terminates without resolving scope, scope is then

[1] Games Academy, Falmouth University, TR10 9FE, UK. Email: John.Andrew.Speakman@Falmouth.ac.uk

automatically resolved. This allows the use of more complex code structures, such as loops and if statements.

Code Excerpt 1 demonstrates the use of the push/pop stack to dictate which automatically generated variables (output, A, B, C etc.) are eligible for use by the next line of code. The colour and indentation depth indicate the depth within the stack which each line of source code applies to.



```
Output Code                    Stack

output = output * 2;           output    | -      | -
double A = 0;                  output, A | -      | -
if (output > 5){               output, A | -      | -
   double B = 2;               output, A | B      | -
   if (output < 27){           output, A | B      | -
      double C = 36 * output;  output, A | B      | C
      B += C;                  output, A | B      | C
   }                           output, A | B      | -
   double D = B;               output, A | B,D    | -
   double E = 0;               output, A | B,D,E  | -
   if(E > D){                  output, A | B,D,E  | -
      double F = 21;           output, A | B,D,E  | F
      D = F;                   output, A | B,D,E  | F
   }                           output, A | B,D,E  | -
   output = D;                 output, A | B,D,E  | -
}                              output, A | -      | -
output += A;                   output, A | -      | -
return output;                 output, A | -      | -
```

Excerpt 1. Scope controller stack displaying accessible variables per line of output code

This generated code is then wrapped with the applicable namespaces. At this point, the code may be returned as a syntactically correct .cs source code file or run through a compiler.

In order to execute this file in the same application it was generated, the code is compiled, at runtime. By using the C# .NET Compiler Platform “Roslyn”[11] code analysis package, an intermediate compilation object, “an immutable representation of a single invocation of the compiler”[12], may be generated from the source code. This compilation object is then emitted using reflection, which builds the object as a collectible[13], in-memory Dynamically Linked Library (DLL) directly into a memory stream, which allows the DLL to be unloaded directly, freeing memory and removing the need to store a physical DLL on the drive. Reflection is used to obtain the MethodInfo[14], a class which provides access to a methods metadata, used here to call the method, for any method in the generated assembly. This may then be stored in an array of delegates, so the method may be called without needing to implement reflection on any future call to the method, significantly reducing call time[15].

3 GENETIC ALGORITHM

Multiple variables per codon are necessary to handle automatically assigned variables properly: building a single line of code using this code constructor takes up to 3 arguments, giving a requirement for the genetic sequence to fit a 3 x N matrix.

The first value, used to determine the main template, selects against a weighting matrix for the likelihood of each template being relevant (for example, an addition call may be much higher frequency than a cosine call). This value has a lower mutation likelihood than the other two values in the codon, as its impact on mutation is significantly greater.

The two other values in the codon are numeric and assume the frequency in code to favour low integers, especially 2 and 3, though with a lower probability of calling 1. The formula used in this model is:

$$y = (1000 / (1+x/10))-9$$

The algorithm proposed in this paper assumes that the varying depth of scope has a similar effect to positional depth in grammar trees. Applying crossover and mutation at greater depth in grammar trees shows to have a higher likelihood, per mutation, of producing beneficial effects on fitness with a reduced likelihood of detrimental effect [16]. Replicating this, mutation may be applied proportionally to the push/pop stack depth. Crossover may be applied on a similar basis, increasing likelihood of crossover proportional to depth.

Due to the use of a genetic sequence, most standard genetic algorithm functions may be applied: mutation, injection, removal, etc.. Chromosomal block structures may also be implemented, using functions within a class as a chromosome with independently generated code, which can call other functions, even those within the same automatically generated file. While not yet tested, the introduction of these mechanisms is expected to, on average, significantly improve the diversity and fitness of agents over multiple generations.

As this system was intended to support an Artificial Life simulation, with an implicit fitness function, the algorithm can breed individuals on demand, rather than requiring distinct generational batch breeding (though batch breeding can still be applied). This would allow agents to breed based on their current state, independent of other agents or timeframes, where breeding becomes bound to the agent’s ability to survive and breed naturally within their environment.

4 IMPLEMENTATION

This project is currently early in development, being at the first stage capable of producing measurable results. As this is an early prototype, many of the proposed systems have not yet been implemented and the full potential of a complete solution is yet to be explored.

The tested solution runs on a .NET Core, multithreaded environment, with the intention of optimising the number of simultaneous calls to dynamically generated code in an asynchronous environment. This implementation was built to complete simple unit tests, where input(s) were automatically passed to the function, and the resultant outputs were compared against a pre-defined value. The fitness function assessed the number of unit tests which matched this value and, where there was no perfect match, the difference between outputs and that value, generating an associated score with an emphasis on perfect matches.

For these tests, only simple random mutation was implemented, constructing each new generation by duplicating the best agent from the previous generation with random mutation.

As a prototype, the number of defined templates available to the simulation is very low, currently only accessing simple maths and mathematical comparisons. Similarly, a weighting matrix against the relevancy of templates is also still not yet implemented. The direct effect will have severely increased the likelihood of detrimental mutation and resulted in a generally lower fitness per generation. The fitness function may also be extended to reward through fitness score, reduced length of code and execution time, increasing performance over time.

Even with this simple implementation, we can achieve successful completion of simple unit tests, showing identifiable improvement per generation.

For the following simulations, all agents worked with a pre-set number of lines of code, though potential improvements are expected from injection and removal of code in later simulations. Results from simple tests, with simple problems, indicated a low number of lines tends to solve unit tests in a lower number of generations. Simple unit tests, for example, attempting to divide or multiply by 2, were often completed within the first generation and high complexity tests are yet to be applied.

The following results, shown in Figure 1, show 5 simulations, each with 100 agents over 20 generations, attempting to generate the value 1457 when given an input of 100. Agent fitness above 0.8 is within 0.01 of the correct output.

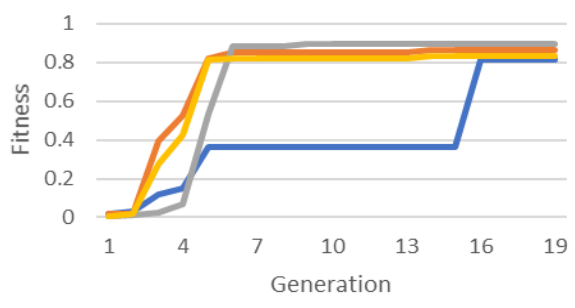


Figure 1. Graph of Fitness / generation for 5 simulations

Code Excerpt 2 displays the generated code output from the highest scoring agent from one of the simulations, with fitness >0.8. The sections in Grey represent the main body of the evolved code. The section in green is an automatically generated end of file scope termination. Sections in blue are a wrapper, with namespaces, to generate a syntactically correct .cs file. To verify the code's validity, this code was exported into a separate C# project where it compiled and executed successfully.

This simulation was set to output, per agent, every generation, a C# file with 15 lines of code in the function body. The output displayed above only utilised 7 of these lines, indicating the liability to create junk code and emphasizing the importance of genetic removal and dynamic genetic sequence lengths, as seen frequently in GP [17].

While subject to substantial change with further development, some simple, preliminary performance tests have been performed[2]. To generate, build and execute a MethodInfo class from a file with a single line of code in the function body took, on average, 16.5ms and accessing this class from a delegate took on average 7e-4ms. While solving a simple unit test, the application took 25 seconds to generate, build, execute, breed and display 10 generations of agents, with 100 agents per generation. No significant change to performance was noticed when varying the number of lines of code in the function body between 5 and 30 lines, per agent. All tests test resolved and executed correctly, including a larger experiment generating 10,000 agents, each building 300 lines of code.

```
using System;
using System.IO;
namespace RoslynCore
{
    public static class AutoCode
    {
        public static double FunctionA(double output)
        {
            output += Math.Cos(20);
            output = output;
            if (output > 5)
            {
                output += 18;
                double A = output;
                A = output * 67;
                output += 3;
                output = 12 * output;
                if (output < 27) {
                    output = output * 6;
                    if (output !=0)
                        output = output / 6;
                    output += 51;
                    output = Math.Pow(output, 16);
                    output = A;
                }
            }
            return output;
        }
    }
}
```

Excerpt 2. Example output code,
output when input is 100: 1456.897

5 CONCLUSIONS

The prototype implemented in this paper successfully generated, executed and returned syntactically correct C# files with potential access to the entire available C# library. These tests successfully implemented dynamic scope and variable control, with the ability to automatically generate new variables and restrict their application to within their local scope.

Through simple best-agent mutation over multiple generations, where each generation produces, compiles and executes a new group of C# files, this algorithm successfully completed multiple simple unit tests and returned the solution as a file automatically.

All generated files executed without runtime or compilation issues, both within the live server environment in which they were constructed and, using the output source code, independently in other C# environments. Efficiency of execution when calling generated files is also promising, as they can be called using delegates.

[2] Testing on localhost IIS Express 10, i7-7700HQ

6 FUTURE WORK

The genetic algorithm and breeding functions for this system are still in progress with the anticipation of greatly improved performance per generation. Following this, a more robust benchmark showing the full extent of the capability and impact of automatic, plain text source code generation is to be carried out. Further research is also required to statistically determine the distribution of common lines of source code, in order to produce an optimal template selection weighting matrix.

This algorithm was designed with the intention of eventually acting as the behavioural controller for a server-based Artificial life simulation. This is intended for use by multiple simultaneous, geographically distributed users in a co-creative, modifiable virtual environment, bringing an emphasis on reducing the runtime processing requirements while maximising the quality of behavioural output on a server framework.

The client-side application for this model is intended for mobile and mixed reality devices, with an initial benchmark for the HoloLens. Continuing work in this direction will breed virtual agents in a virtual environment, using an implicit fitness function dictated by natural selection, rather than an explicit unit test. These agents will need to adapt to indirect human interaction, where users will modify the geometry and interactable objects within the virtual environment, directly impacting the survivability and implicit fitness function of agents. This introduces the need for further optimisations between software efficiency, speed of adaption and adaptive potential in development of the evolutionary algorithm.

When dealing with behaviour controllers for human interaction with this algorithm, a pre-defined genetic sequence which constructs a common behaviour may be implemented. For example, an initial BOID [18] template could be recreated using a manually entered genetic sequence, removing the need for an early, low functionality, high failure rate species. This initial functionality may then be mutated and expanded through adaptive evolution, where dynamic code generation permits absolute modification of behaviour from that point forward.

Alternative development outside of Artificial Life could see this algorithm being used as an alternative solution to common GP problems, particularly where the output is intended for human interpretation. It may also be used to approximate solutions or solve simple programming tasks in everyday programming, forming the basis of a form of pair programming between a human and an AI with integration into an IDE, where the human acts to guide the AI by outlining the required functionality.

REFERENCES

[1] BillWagner, "C# Coding Conventions - C# Programming Guide." [Online]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/inside-a-program/coding-conventions. [Accessed: 19-Feb-2019].

[2] J. R. Koza, *Genetic programming: on the programming of computers by means of natural selection*. Cambridge, Mass: MIT Press, 1992.

[3] A. Agapitos and S. M. Lucas, "Evolving a Statistics Class Using Object Oriented Evolutionary Programming," in *EuroGP*, 2007.

[4] W. S. Bruce, "Automatic Generation of Object-oriented Programs Using Genetic Programming," in *Proceedings of the 1st Annual Conference on Genetic Programming*, Cambridge, MA, USA, 1996, pp. 267–272.

[5] S. Ventura, C. Romero, A. Zafra, J. A. Delgado, and C. Hervás, "JCLEC: a Java framework for evolutionary computation," *Soft Comput.*, vol. 12, no. 4, pp. 381–392, Oct. 2007.

[6] R. Abbott, "Object-oriented genetic programming, an initial implementation," in *in International Conference on Machine Learning: Models, Technologies and Applications, 2003*, 2003, pp. 26–30.

[7] T. Perkis, "Stack-Based Genetic Programming," in *International Conference on Evolutionary Computation*, 1994.

[8] L. Spector, J. Klein, and M. Keijzer, "The Push3 execution stack and the evolution of control," in *Proceedings of the 2005 conference on Genetic and evolutionary computation - GECCO '05*, Washington DC, USA, 2005, p. 1689.

[9] C. J. Headleand and W. J. Teahan, "Template Based Evolution," in *Proceedings of the 15th Annual Conference Companion on Genetic and Evolutionary Computation*, New York, NY, USA, 2013, pp. 1383–1390.

[10] Bangor University, Wales, UK, C. Headleand, L. Cenydd, and W. Teahan, "Berry Eaters: Learning Color Concepts with Template Based Evolution," in *Artificial Life 14: Proceedings of the Fourteenth International Conference on the Synthesis and Simulation of Living Systems*, 2014, pp. 473–480.

[11]. "NET Core - Cross-Platform Code Generation with Roslyn and .NET Core." [Online]. Available: https://msdn.microsoft.com/en-us/magazine/mt808499.aspx. [Accessed: 21-Jan-2019].

[12] "Compilation.cs." [Online]. Available: http://source.roslyn.codeplex.com/#Microsoft.CodeAnalysis/Compilation/Compilation.cs,ec43f5a2c70b26f1. [Accessed: 04-Apr-2019].

[13] "Collectible assemblies in .NET Core 3.0 | StrathWeb. A free flowing web tech monologue." .

[14] rpetrusha, "MethodInfo Class (System.Reflection)." [Online]. Available: https://docs.microsoft.com/en-us/dotnet/api/system.reflection.methodinfo. [Accessed: 21-Feb-2019].

[15] rpetrusha, "Emitting Dynamic Methods and Assemblies." [Online]. Available: https://docs.microsoft.com/en-us/dotnet/framework/reflection-and-codedom/emitting-dynamic-methods-and-assemblies. [Accessed: 27-Feb-2019].

[16] T. Castle and C. G. Johnson, "Positional Effect of Crossover and Mutation in Grammatical Evolution," in *Genetic Programming*, 2010, pp. 26–37.

[17] C. Ryan, J. Collins, and M. O. Neill, "Grammatical evolution: Evolving programs for an arbitrary language," in *Genetic Programming*, vol. 1391, W. Banzhaf, R. Poli, M. Schoenauer, and T. C. Fogarty, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 83–96.

[18] C. W. Reynolds, "Flocks, Herds and Schools: A Distributed Behavioral Model," in *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, New York, NY, USA, 1987, pp. 25–34.