University of Glasgow | School of Computing Science

Darwin: A Genetic Programming Framework for Cloning Web Services

George Kouzmov

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

Level 4 Project — 31 January 2014

Contents

Chapter 1

Introduction

In recent years web services have become a main method of communicating between applications and accessing needed information, creating similar to a distributive environment for many developers. Because of the standardized communication used by the web services they are expected to be used even more in the future providing even more commodities for both developers and users. The nature of web services is a black box system which means we are unfamiliar with its contents and we only know how to access it and what it returns.

That's where the idea came from, for copying a web services using genetic programming, which became a proof of concept project that reached very far. The genetic programming involves generating programs based on set of functions and then evaluating the quality (correctness) of the program via a fitness function. To prove that cloning of a web service can be achieved using genetic programming we used a symbolic regression problem where the genetic program needs to find a mathematical expression that yields the same result as a given one from a fitness function. But instead of using our own fitness function we are using a real web service hence generating a program that has the same functionality as the web service. Proving this meant that theoretically by increasing the complexity of our function set we can achieve copying of an even more complex web service.

1.1 Genetic Programming

Genetic programming is a branch in artificial intelligence inspired by biological evolution. It is a machine learning technique that optimizes a population of computer programs until they start performing the user specified functionality. Like the more widespread genetic algorithms (GA), genetic programming uses evolutionary algorithm based methodology to iterate through solutions until a correct one is found. In genetic programming a predefined size **population** is iterated through a user defined number of **generations**. The population consists of **individuals** which are programs represented as a tree structure. This tree structure is constructed from a predefined **primitive set** of functions and **terminals**. The terminals represent the bottom nodes of the tree, they can be both arguments and constants.[insert graph of an individual] A genetic program needs a **fitness function** it's a function through which each individual is evaluated and compared. This way the correctness of the program can be calculated. Usually there are two type of fitness function minimum and maximum where minimum is looking for individuals with fitness values closer to 0 (where 0 usually means that a solution is found) and maximum for the individuals with highest fitness result.

The way a genetic program works is it randomly generates individuals for the first generation evaluates them through the fitness function and after that two types of functions are applied - **crossover** and **mutation**. Crossover

is used to combine the genetic information of two individuals which means switching the nodes of a tree of one individual with another. With tree-based representation replacing a node means replacing the whole branch. When mutation is applied to an individual a single node is replaced with a new randomly generated one again changing a single node can mean changing the whole branch. After crossover and mutation are applied to the population a new generation that consists of the children of the old population, individuals from the old population and mutated individuals is created hence the iterations continue until the generations are completed. A result may not be found in the end of the generations.

One of the problems with genetic programming is searching the correct parameters to solve a specific problem. For each problem a specified number of populations, generations needs to be provided also the percentage of the population to which crossover and mutation are going to be applied. In more advanced cases the types of mutation, crossover and fitness function algorithms used need to be specified. Another complication is choosing the correct fitness function to evaluate the correctness of the individuals. If an incorrect fitness function is defined an individual with seemingly good fitness value may be far from correct. The primitive and terminal sets need to be chosen correctly as well. If too many primitives are chosen the generations might not be enough to find a solution to the problem because we are expanding the search boundaries. However if small sets are chosen we are limiting the search boundaries or the individuals won't have the big enough primitive set to satisfy the required functionality.

So using this methodology multiple problems can be solved. A common and a simple one is symbolic regression. In the case of symbolic regression an individual represents a mathematical expression and is looking to get the same results as another expression specified in the fitness function using the primitive set of operators available. In this project we are using this technique as a proof of concept for the capabilities of the framework.

1.2 Web Services

Web services are one of the main methods to provide utility computing. A definition for a web service is

> A software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format.

They represent an easy way for developers and clients to access and compute information. Usually through an API users are being able to communicate with a web service although sometimes they need to be accessed with raw URL or HTTP request. Unlike traditional client/server models, such as a Web server/Web page system, Web services do not provide the user with a GUI. Web services instead share business logic, data and processes through a programmatic interface across a network. The applications interface, not the users. Developers can then add the Web service to a GUI (such as a Web page or an executable program) to offer specific functionality to users.

Web services hold many benefits for developers. They represent components on the web that can be plugged in any project or software so it's functionality can be used. It improves communication between companies and users because information is distributed much faster. A good example of a web service is of Google who provide an API to their GPS maps which when given longitude and latitude return a map with the location of the user. This service has improved the user experience for many devices and because of it's good implementation has made the implementation of other similar services almost obsolete.

A problem with web services are that if they go offline or they have a bug, they affect the whole network and every user using them. Every non-open sourced framework is essentially a black box and the way it uses it's parameters is unknown. However these problems are minor to the overall benefit of the web services. Because of the way the internet works the amount of web services has essentially created a distributed system with multiple components. Developers can integrate complex systems easier than before and can easily plug in additional functionality as long as the right web service is there

1.3 Problem

The purpose of the project is to develop a framework that can clone a web service. Theoretically it is possible to complete this through genetic programming. Initially the project was separated into three steps - evaluate individuals over a network, clone and existing web service and improve a web service. The project aims to clone an external web services and provide a GUI to generate the configuration files needed for the framework to work.

Theoretically when using an external web service we know the parameters we are giving to it, we have an idea of the way it works and we know the results. Combining this knowledge meant it was possible to use an external web service as a fitness function to our genetic program and a way to specify the primitive set used. After part two of the project was completed it was proved that it is possible to clone the functionality of a web service. Because it is a proof of concept project a simple web service was used in order to verify the expected result. However the project proves that with the right choice of primitive set it is possible to clone the functionality of even more complex web services.

Chapter 2

Context

2.1 Background

2.2 Requirements

2.3 Organization

2.3.1 Meetings and Minutes

2.3.2 Agile Development

2.3.3 User Stories

2.3.4 Sprints

Chapter 3

Design

When designing the framework the idea behind it was to provide a simple way to control the key parameters in a genetic program. Since the beginning of the project we were aware of the scope of the project and the complexity it can reach so another key design feature was modularity. Component based software engineering practices were used throughout the project, helping to organize the multiple technologies used in the project.

3.1 Multiple Technologies

Among the key issues in the development was combining all the technologies used in the project. A genetic programming framework, REST based web service, an XML parser, a GUI, code inspection and generation libraries; each was needed in order to create a framework that can clone web services. For each component that was needed there were multiple options available so the best ones needed to be chosen which meant trying out pack of frameworks and libraries.

3.1.1 RESTful Web Services

RESTful web services were needed for creating a service that can evaluate individuals over a network. Simplicity was important for the framework we are choosing. Three of the frameworks encountered were WebPy and CherryPy, Django all of them lightweight and commonly used. Other options for a REST framework were Flask and Bottle. However CherryPy and Django are the only frameworks that handle POST and GET request correctly, without any problems. In order to fix this issues with WebPy, Flask or Bottle additional libraries would of been needed. It was desired to avoid more dependencies because they would just make the project more complex. That meant the choice was between two. The more simple choice was CherryPy but that wasn't the only reason why the project was chosen. Another of David White's projects was using a cloud of RaspberryPis which meant this potentially can be used as a test for the Darwin framework. Because CherryPy provided support for RaspberryPy's it was the obvious choice.

3.1.2 Genetic Programming Framework

Genetic programming frameworks didn't have as many choices as RESTful web services however there was the option of implementing one. Pyevolve, pystep, pygene and DEAP were my only choices for genetic programming frameworks in python. None of them was minimalistic and simple enough, however implementing my own framework seemed as a much more complicated task that can slow down the project and it's final result. So a decision was made, that one of the frameworks was going to be used for the project which meant spending time learning more about the features they provide and how to use them correctly. The most common and supported one was the DEAP framework. It was a newer one but it had really good documentation, tutorials and examples making it the correct choice. However as a future development option for the project remains creating a genetic programming framework related to the project.

3.1.3 Code Inspection and Generation

Code inspection and generation was needed in order to achieve simplicity for the user. The idea was with fewer parameters to do a lot which meant a lot of code generation and code inspection. For inspecting code the best solution was inspect.py library which provided a way to extract source from function calls. It is a library that comes with python package has great documentation, examples and tutorials which proved only positive for the speed of development since not a lot of time was spend learning the inspect library.

Among one of the problems with using DEAP was it didn't support conversion between indiviudals and python source code or AST (Abstract Syntax Tree). Representing an individual as raw code or an AST meant that the evaluating web service can be simplified even more. This was an important part of the project however there are a lot of choices but the most correct one was the AST library which provides code generation while representing code as a syntax tree. This feature was very compatible with the way individuals are represented in genetic programming and it was more robust and safe way of doing code generation rather than using raw string. This started the idea of an open source contribution to the DEAP framework for generating python source and AST for a DEAP individuals.

3.1.4 XML Files

XML files are commonly used for configuring systems. Since the system is working with many parameters starting the code and defining each and every one took a lot of time and made the code look messy. XML file was a simple solution to a simple problem. There are many libraries in python used for parsing XML files - expat,minidom,BeautifulSoup,lxml ect. In the case of choosing and XML parser the chosen one was the most simple to understand and implement and lxml gave me that impression considering the good documentation it had.

3.1.5 Graphical User Interface

Graphical user interface was the next step into project simplification. It's role is to generate XML files for the user with the ease of controlling all the parameters needed. For creating GUI in python there are several good choices Tkinter,PyQT,wxPython. In my past I've had experience with Tkinter and I wasn't excited about working with it again. It looked much more complex compared to the other two. In sites such as StackOverflow the community was supporting wxPython. An additional benefit to using wxPython was the fact that there were a lot of good tools for generating layout. However my previous experience with GUI frameworks like Swing and it's similarities with wxPython, gave me the advantage of understanding wxPython quickly and writing a decent GUI in no time. In the end I didn't use any of the layout generating tools like wxGlade, wxDesigner or DialogBlocks because learning to use them was going to cost almost the same time as learning wxPython.

3.2 Architecture

A top down approach was applied for identifying the major components in the system. Client's requirements were the starting point for identifying those components. They were clear and simple enough to create a high level view of the framework , with which to explain the framework's basic behaviour and how it's going to work see figure 3.1.

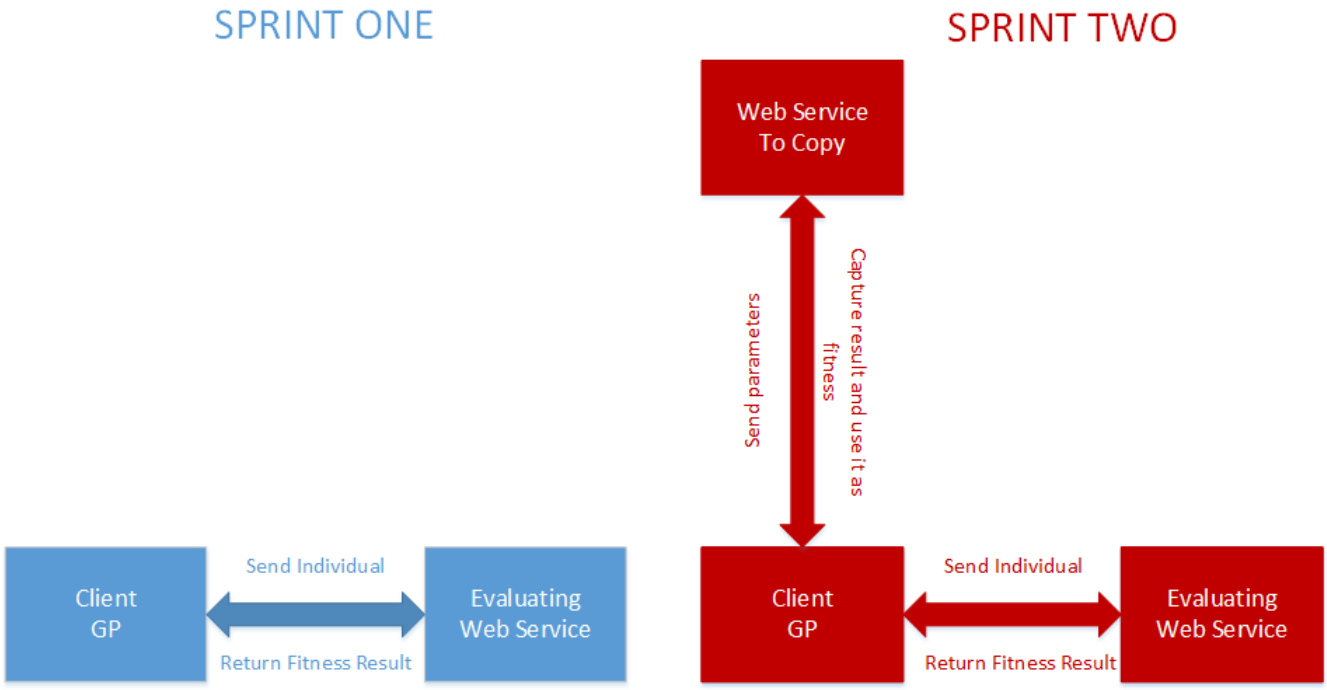

Figure 3.1: Top level view of sprint one and two

By displaying the way the high level components communicate it was later possible to identify lower level ones.

3.2.1 Overall Design

After the high level components were designed, the user stories helped create a more detailed view of them. At that point of time the project focused on sprint one which meant focusing on evaluating individuals over a

network. User stories helped identify how user was going to approach the framework hence it was able to create components with known input and return parameters. The result was a high level view of first sprint with all of it's components identified see figure 3.2

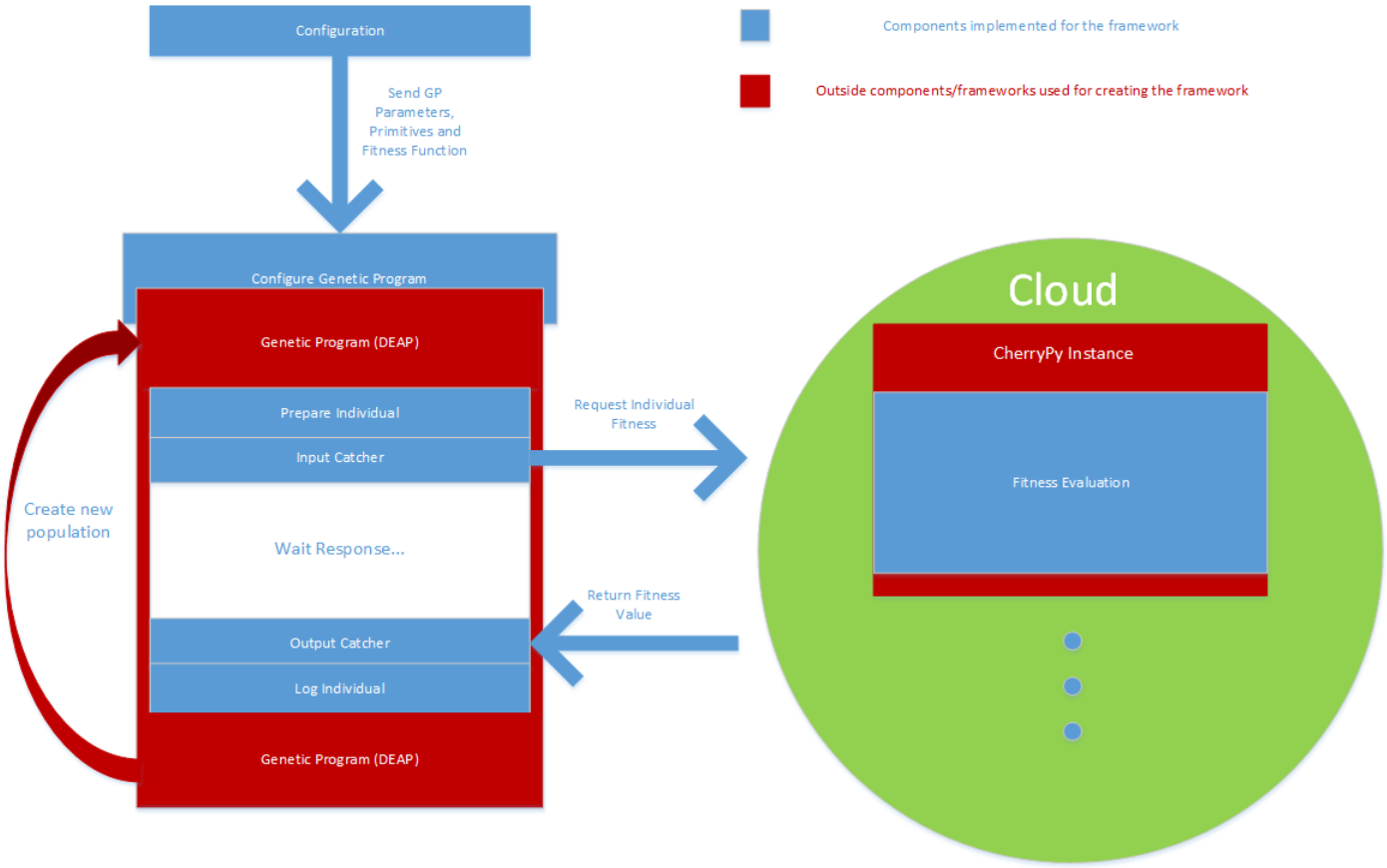

Figure 3.2: High level view of sprint one and the identified components used

Even though python2.7 unlike python3.3 is a semi object language for better use of software engineering practices and better ordering of the project, objects were used to represent the different components. The diverse capabilities of python and large set of libraries helped design a complex framework with very simple architecture. Later on more specifics for the client design will be explained. As you can see in figure 3.2 Configuration, Configure Genetic Program and Genetic Program represent the three main components of the client and CherryPy with an evaluation component represents the evaluating web service. This design helped us insert the technologies we are using at the correct places. For example Input Catcher and Output Catcher were later on substituted by a single component that using the requests library handles the requests and logs the traffic to the evaluator. These steps from top to down helped us clearly identified the classes that are going to be in the components.

3.2.2 Client

Before explaining how the client works, DEAP framework needs to be explained. It is configured through set of method calls to a *toolbox* this *toolbox* is prior configured by a *primitive set* where all the primitives and terminals that the genetic program is going to use are defined. The toolbox give easy API for accessing and controlling the individuals, generations and population as well as methods that invoke mutation, crossover and evaluation over the population.

Knowing how the DEAP framework works, it was later obvious that the same parameters that were needed for DEAP's configuration were going to be needed for Darwin's as well. Through the *Configuration* class shown in figure 3.2 we are able to set all the primitive functions and the fitness function needed for the program. Later on the client decided that there should be a preset number of primitive sets and the user will choose through the GUI which to enable. However in the case of an advance user if additional primitives need to be added the class can simply be extended and add them as normal methods. No other configurations regarding the primitives need to be done since Darwin is handling everything else. [insert class diagram]

In figure 3.2 you can see that for the configuration of the genetic program we need other parameters which are not specified in the *Configuration* class. When setting up the framework they can be set manually or configured by an XML file that carries all the information needed. There are only two parameters that are mandatory and the framework won't work without them - the address of the evaluating web service and the target service that needs to be cloned. Any other parameter is automatically set up with a standard value for solving the symbolic regression problem. The class that configures the framework acts like a data model since all the parameters needed through any other step of the framework is instantiated there. This data model is later used by the *Populator*.

The *Populator* class is basically a wrapper around the DEAP framework. What the class do is it creates the first population then sends each individual over the network to the evaluating web service. After each individual's fitness value is updated it repeats the process until the number of generations are reached. Besides executing the actual genetic program the *(*Populator) logs the traffic between the services and handles the output of the genetic program - either displaying the best individual or generating a web service working with the best individual.

3.2.3 Evaluator

The final big component is the Evaluator [reference to class diagram]. It is a CherryPy service that can be ran on any machine running python and has only one method - evaluate. It takes an individual, test arguments in the form of a dictionary and fitness results evaluates the individual comapres with the fitness function and return the result. Initially the indiviual was in a DEAP represented form however that meant that in order to evaluate it the same DEAP configuration needed to be set up on the evaluating side as on the client one. This was unpractical,slower, it was introducing many dependencies and much harder to set up. The Evaluator needs to be easy to set up, easy to modify because it is expected to be ran on multiple machines including RaspberryPis. Because of that the individuals was later represented as source code which meant it was able to run on any machine that runs python. This way simplicity was achieved for this component by moving most of the overhead work to the client.

3.3 GUI Design

-why use gui

3.3.1 Design Principles

3.3.2 Purpose of the GUI

3.3.3 Iterations and Final Design

Chapter 4

Implementation

4.1 Technologies

4.1.1 DEAP

4.1.2 CherryPy

4.1.3 Inspect

4.1.4 Requests

4.1.5 wxPython

4.2 Implementation

4.2.1 DEAP Set Up

4.2.2 Generation of Individuals

4.2.3 Evaluation Over a Network

4.2.4 Code Generation

4.2.5 GUI

Chapter 5

Evaluation

5.1 Unit Testing

5.2 System Testing

5.3 UI and Usability Testing

Chapter 6

Problems Encountered and Future Development

6.1 Problems

6.1.1 Multiple Technologies Problem

6.1.2 Metaprogramming

6.2 Future Development

6.2.1 Expand Framework

6.2.2 Improving Web Services

6.2.3 Unit Tests as Fitness Functions

6.2.4 Co-Evolution

Chapter 7

Conclusion

7.1 Contribution

Chapter 8

References