Grow and Serve: Growing Django Citation Services Using SBSE

Yue Jia, Mark Harman, William B. Langdon, and Alexandru Marginean

CREST, Department of Computer Science,
University College London, Malet Place, London, WC1E 6BT, UK

Abstract. Web services offer a quick and cost-effective way to augment existing software systems with new functionality. This makes them suitable for Genetic Improvement, a subarea of SBSE in which existing code is automatically improved using search based optimisation. We introduce a 'grow and serve' approach to Genetic Improvement (GI) that grows new functionality as a web service and apply it to the Django platform. Using our approach, we successfully grew and released a citation web service. We wish to demonstrate that GI can grow genuinely useful code in this way, so we deployed the SBSE-grown web service into widely-used publications repositories, such as the GP bibliography. In the first 24 hours of deployment alone, the service was used to provide GP bibliography citation data 369 times from 29 countries.

1 Introduction

Reusing bespoke features developed for a specific system on other systems requires a substantial amount programmers' effort. This effort can be reduced by implementing the features as web services, thereby using standard protocols to share data and to provide functionality to client applications. We argue that such service-based architecture, where available, provides a useful possible deployment mechanism for genetic improvement. We use a variant of the 'grow and graft' genetic improvement approach [5] to grow a new feature implemented in Python, which can then subsequently be served as a Django service module. We call this approach 'grow and serve'; it is 'grow and graft' genetic improvement *without the graft*.

Our approach is a form of genetic improvement [6, 9, 11, 15], which has been used for code migration [7], improving energy efficiency [3, 10, 13], memory/speed trade-offs [16], automated repair [1, 9], and performance improvement [8, 11, 12, 14, 15]. More specifically we use the 'grow and graft' approach [5], in which new functionality is grown in isolation and subsequently grafted into an existing system. However, in previous work, the grafting phase has specialised the previously grown code for the system into which it is to be grafted. Similar specialisation is also required in genetic improvement by program transplantation [2, 12]. However, in our approach, the grafting phase is not only highly general, it also becomes trivial; the extra functionality we grow simply becomes a web service module running on the Django framework.

This paper follows the result reporting style used in our previous work on 'grow and graft' genetic improvement, reporting on the guidance required for the grow stage as we did previously [5]. However, the primary claim of the present paper is that we have been able to grow *useful* functionality that was not previously available. For this reason we chose to grow an application that we believe may be useful to some readers. Our agenda is to migrate genetic improvement research from demonstration examples, grown in the laboratory and of primarily scientific interest, to real -world usable code. Our starting point for this outward genetic improvement spread is our own academic community, since we believe we might hope to understand some of their requirements.

Specifically, we use our tools to provide a citation reporting service. The reader can use this to augment existing webpages with citation information, served by Django, genetically improved by the incorporation of the citation service module we grew. We make this available, so that the reader can investigate the code produced by genetic improvement, but also so that he or she can, for example, automatically augment any publication listing websites with citation information.

Citation numbers can provide helpful information in publication repositories. However, this information is missing from many publication repositories (such as the GP bibliography, the mutation and the SBSE repositories). It is not straightforward for the provider of the bibliography to provide citation information, for example from Google Scholar, without considerable effort or supporting technology. We believe that this makes this an interesting and worthwhile candidate for our genetic improvement approach. To demonstrate the usefulness of this new functionality, we have used it to augment the GP bibliography website (and others) with this functionality, an instance of 'GP for GP'; using GP to envolve an improvement to the website concerned with GP. We recorded the first 24 hours of deployment usage, finding that the service was used to provide GP bibliography citation data 369 times from 29 countries.

2 Approach Used to Grow and Serve a Citation Service

Django is an open source web application framework with a stateless service with which we provide a simple URL link to query the number of citations for a publication. The URL takes the title of the publication and returns the number of citations it has attracted. We use the Google Scholar website to source citation information, but our approach could equally be applied to other citation data providers, such as Microsoft Academic Search or Research Gate.

Django turns features implemented in Python into web applications or web services. Therefore, we grow the citation service as Python code using the 'grow' phase of our 'grow and graft' approach [5]. The 'graft' phase becomes trivial; we simply copy the automatically-grown Python code into an existing Django template running on an Apache server. Our GP system implements a strongly-typed GP that takes a grammar file and a test harness as input and outputs a program that passes all the tests specified in the test harness.

The grammar file specifies a set of data types and potential APIs suggested by the developer as likely to be useful to GP. Retrieving citation information is clearly not straightforward, and we do not expect the GP to discover this for itself. Rather, we provide GP with several different types of APIs that it may find useful. These can be divided into four categories: handling HTTP requests, parsing HTML trees, string manipulation, and list manipulation. All of these functionaries are Python built-in functions or are supported by widely-used packages (such as the `lxml` and `requests` models).

Our approach is therefore to give the GP phase 'hints', in the form of pre-existing code that may be useful. As we have argued previously [5], we believe that these hints would be trivial for the human programmer to provide, but almost impossible for GP to discover by itself. As such, the provision of such hints represents an ideal trade-off between human and machine-based effort.

The GP system was designed to evolve imperative programs, formed by a sequential list of assignments and functional calls [5]. We adapted the system to evolve Python code by converting the object-oriented APIs into an imperative form. For example, function $foo.bar(arg)$ is turned into function $foo\_bar(foo, arg)$.

We manually created these functions, to expedite experimentation. However, the process we followed was entirely algorithmic and therefore could have been fully automated. These functions are provided in the test harness file. The test harness also includes the functional tests and fitness computation components for evaluating the evolved code segments. At each generation, GP evolves a population of code segments and inserts them into the test harness. The test harness is executed and evaluated for fitness.

Fitness functions: We experimented with 8 different fitness functions, composed of a set of 22 equally-weighted fitness components. The default (starting) fitness value is set zero, which denotes a completely useless candidate solution. The fitness value is subsequently incremented, based on the candidate solution's ability to satisfy each of the different fitness components.

The first set of components are the 'essential' fitness requirements; that the new code must pass the test cases that capture correct functionality. We designed five black-box functional tests to cover the different possible forms of input and feedback from the source of citation data, as shown in Table 1. We increase the fitness value by 1 if the execution of a test completes without raising an exception. The fitness value is further increased by 1 if the evolved function also returns the expected output. This gives us 10 essential fitness components; 2 for each of the 5 tests, shown in Table 1.

Our grow and graft genetic improvement research agenda starts with a fundamental assumption: For many programming tasks, it will prove to be easier for the programmer to specify a few criteria for successful solutions, than it will be for the programmer to *generate* the solutions from scratch. This notion that 'checking is easier than generating' goes the very heart of the motivation for SBSE itself [4]. We think of these criteria for successful solutions as 'hints' provided by the programmer to the GP.

1	Characristic	Input	Expected Output
1	Full Title	'Higher Order Mutation Testing'	return 'Cited by 102'
2	Key Words	'Babel Pidgin'	return 'Cited by 5'
3	1 Citation	'Genetic Improvement for Adaptive Software Engineering'	return 'Cited by 1'
4	0 Citation	'Achievements, open problems and challenges for search based software testing'	return 'No Citation'
5	Bad Title	'sdfsdsdf sdoi jsdlkfjsdljlksdlkadslkfsad-jlsdfkljsdflksd'	return 'No Citation'

Table 1. The 5 Functional Black Box Test Cased Used for Essential Fitness

We designed three sets of assistant fitness functions to provide these hints (See Table 1). These functions are classified according to our assessment of human effort required to provide them. The set of 'Inclusion' fitness functions specify the names of functions that might be included in a successful solution. We do not give the GP any information about the parameters to pass nor the expected results with these fitness components. The programmer simply has to identify a set of candidate functions which may (or may not) prove to be useful in a candidate solution. We believe that this requires very little human effort, since most programmers will be readily able to call to mind a set of such possible candidate functions for any given programming task. In the case of the problem in hand, the functions we make available with the 'Inclusion' fitness are simply the data structure manipulation functions likely to be useful in any solution.

1	Inclusion	Call to request get
2	Inclusion	Call to generate html tree
3	Inclusion	Call to search html tree
4	Inclusion	Call to filter list
5	Inclusion	Call to concat
6	Ordering	concat before send
7	Ordering	generate html after input
8	Ordering	search html after generated
9	Ordering	filter list after search
10	Necessary	concat gives correct link
11	Necessary	Correct call to Google Scholar
12	Necessary	result contains citation data

Fig. 1. The 12 Fitness Component Hints

The 'Order' fitness components denote a slightly more sophisticated requirement of human effort. They capture constraints on the ordering in which included calls are performed. Clearly, this requires more thought on the part of the programmer. However, we believe that even novice programmers are aware of simple ordering constraints such as 'concatenating partial results together before passing onto the output'. It may even be possible for non-programmers to provide this kind of hint.

Finally, the most sophisticated fitness components are the 'necessity conditions', which denote pre-and post-conditions on the states of computation. These can be thought of as intermediate white box assertion checks that complement the essential black box test cases. Providing such assertions requires more effort from the programmer, but it may help to guide the GP to solutions faster. Perhaps, more importantly, these assertions may provide a useful interface between human understanding and search-based automation. The assertions can be used to constrain solutions such that they satisfy the programmer's conception of expected behaviour at key checkpoints in the computation.

Without such assertions, a perfectly valid solution may be evolved that passes all black box tests, yet remains incomprehensible to the human programmer. Ultimately, such evolved 'source' code may become 'the new object code', removing this concern for all programmers who are content to trust the backend object code that emerges from the 'compilation' process [6]. However, in the intermediate period during which we seek uptake of ideas like genetic improvement, such checkpoints may provide a useful human-machine interface.

Experiment: We used the default crossover, mutation, and elitism operators from the original babel pidgin system [5] with values 0.5, 1.0 and 0.05 respectively and population size 200. The GP terminates when best fitness remains unchanged for 30 generations. We run our experiments on an iMac running OSX 10.10. To speed up the evaluation, we cached the Google Scholar webpage test case results for faster fitness evaluation. This cached call is replaced by an external Google Scholar enquiry link in the final version evolved. All experiments were repeated 30 times to allow for inferential statistical comparison of results.

Results: The results are shown in Table 2. Fitness components are labelled as follows: E: Essential, I: Inclusion, O: Ordering and N: Necessity. In Table 2 we list the eight choices of fitness components in increasing order of sophistication, loosely denoting the programmer effort required to provide the hints to the GP. We analyse the results using a nonparametric two-tailed binomial test to compare success achieved using the essential fitness, E, with each and all of those we achieved using more sophisticated fitness. We use the Hochberg correction in order to account for the fact that we are performing seven different inferential statistical tests.

Fitness Used	Successful runs in 30 trials	Time in seconds	f=Fitness evaluations
E	0 (p=N/A)	204	6,306
EI	0 (p=N/A)	281	7,400
EO	0 (p=N/A)	379	10,226
EN	0 (p=N/A)	348	9,686
EIO	1 (p=0.500)	425	10,806
ENI	0 (p=N/A)	438	11,133
ENO	9 **(p=0.020)**	443	11,633
ENIO	16 **(p=0.002)**	499	12,700

Fig. 2. Results for Growing Django service

With an α level of 0.05, the widely-used threshold for statistical significance, this corrected statistical test indicates that the result for ENIO and for ENO is significantly different to that for E (with a Vargha-Delaney $\hat{A}_{12}$ effect sizes 0.76 and 0.65 respectively). Overall, the results indicate the importance of ordering constraints, and the power of providing the necessity constraints to capture simple pre- and post-conditions.

Encouragingly, the results also suggest that, perhaps, these more sophisticated pre-and post-conditions are not always required in order to find successful solutions. Since the approach can be repeated multiple times, and the programmer can and use fitness to reject inadequate solutions, we need only be successful on one occasion within reasonable time, after repeated executions. Since a successful solution is found using EIO fitness after only 425 seconds, we have tentative evidence that useful functionality can be grown in isolation and deployed as a service using relatively modest programmer hints.

3 Deployment, Conclusions and Future Work

We deployed the service on the Microsoft Azure cloud, incorporating it into the GP[1] and Mutation testing repositories[2]. We also made the citation counting service available as text-returning[3] and image-returning [4] services for others to use. We believe that 'grow and serve' may prove to be widely applicable: The approach will be applicable to any software framework, such as Django, into which behaviour-describing modules can be deployed.

References

1. Arcuri, A., Yao, X.: A novel co-evolutionary approach to automatic software bug fixing. In: CEC. pp. 162–168 (2008)
2. Barr, E.T., Harman, M., Jia, Y., Marginean, A., Petke, J.: Automated software transplantation. In: ISSTA (2015), to appear
3. Bruce, B.R., Petke, J., Harman, M.: Reducing energy consumption using genetic improvement. In: GECCO (2015), to appear.
4. Harman, M., Jones, B.F.: Search based software engineering. IST 43(14), 833–839 (2001)
5. Harman, M., Langdon, W.B., Jia, Y.: Babel pidgin: SBSE can grow and graft entirely new functionality into a real world system. In: SSBSE (2014)
6. Harman, M., Langdon, W.B., Jia, Y., White, D.R., Arcuri, A., Clark, J.A.: The gismoe challenge: Constructing the pareto program surface using genetic programming to find better programs (keynote paper). In: ASE. pp. 1–14 (2012)
7. Langdon, W., Harman, M.: Evolving a CUDA kernel from an nVidia template. In: CEC. pp. 1–8 (July 2010)
8. Langdon, W.B., Harman, M.: Optimising existing software with genetic programming. TEC 19(1), 118–135 (Feb 2015)
9. Le Goues, C., Nguyen, T., Forrest, S., Weimer, W.: Genprog: A generic method for automatic software repair. TSE 38(1), 54–72 (Jan 2012)
10. Li, D., Tran, A.H., Halfond, W.G.J.: Making web applications more energy efficient for OLED smartphones. In: ICSE. pp. 527–538 (2014)
11. Orlov, M., Sipper, M.: Flight of the FINCH through the Java wilderness. TEC 15(2), 166–182 (April 2011)
12. Petke, J., Harman, M., Langdon, W., Weimer, W.: Using genetic improvement and code transplants to specialise a C++ program to a problem class. In: EuroGP. pp. 137–149 (2014)
13. Schulte, E., Dorn, J., Harding, S., Forrest, S., Weimer, W.: Post-compiler software optimization for reducing energy. In: ASPLOS. pp. 639–652 (2014)
14. Sitthi-amorn, P., Modly, N., Weimer, W., Lawrence, J.: Genetic programming for shader simplification. ACM TOG 30(6), 152:1–152:11 (2011)
15. White, D.R., Arcuri, A., Clark, J.A.: Evolutionary improvement of programs. TEC 15(4), 515–538 (Aug 2011)
16. Wu, F., Weimer, W., Harman, M., Jia, Y., Krinke, J.: Deep parameter optimisation. In: GECCO (2015), to appear.

[1] `http://www.cs.bham.ac.uk/~wbl/biblio/`
[2] `crestweb.cs.ucl.ac.uk/resources/mutation_testing_repository/index.php`
[3] `yuejia.cloudapp.net/gpcitation/publication-title/`
[4] `yuejia.cloudapp.net/gpcitation/img/publication-title/`