Babel Pidgin: SBSE Can Grow and Graft Entirely New Functionality into a Real World System

Mark Harman, Yue Jia, and William B. Langdon

University College London, CREST centre, UK

Abstract. Adding new functionality to an existing, large, and perhaps poorly-understood system is a challenge, even for the most competent human programmer. We introduce a 'grow and graft' approach to Genetic Improvement (GI) that transplants new functionality into an existing system. We report on the trade offs between varying degrees of human guidance to the GI transplantation process. Using our approach, we successfully grew and transplanted a new 'Babel Fish' linguistic translation feature into the Pidgin instant messaging system, creating a genetically improved system we call 'Babel Pidgin'. This is the first time that SBSE has been used to evolve and transplant entirely novel functionality into an existing system. Our results indicate that our grow and graft approach requires surprisingly little human guidance.

1 Introduction and Backgroud

Despite much progress in software development environments, programming still includes many human activities that are dull, unproductive and tedious. In this paper we propose a new SBSE approach to software development: Grow and Graft, in which a new feature is grown (using genetic programming) and subsequently grafted into an existing system. This grow and graft development approach aims to reduce the amount of tedious effort required by human programmer in order to develop and add new functionality into an existing system.

Our approach is inspired by the recent trend in Search Based Software Engineering (SBSE) called 'genetic improvement' [2,8,10,11,14,15]. Genetic Improvement (GI) uses existing code as 'genetic material' that helps to automatically improve existing software systems. It has been used to repair broken functionality [10,14], and to achieve dramatic scale-ups for sets of small benchmarks [11,15], and also for a 50k LoC genome matching system [8], for graphic shaders [14] and for a CUDA stereo image processing system [7]. Related work on loop perforation has also reported dramatic speed-ups [13]. GI has also been used to port one system to a new version on a different platform [6].

Recently, it has been demonstrated [12] that GI can be used to transplant code [3] from one version of a system to another. In this previous transplantation work [12], code from several versions of MiniSAT (the 'donor') were transplanted into a specific version of MiniSAT using GI. The aim of this transplantation was to improve execution time for a specific task (Combinatorial Interaction Testing).

C. Le Goues and S. Yoo (Eds.): SSBSE 2014, LNCS 8636, pp. 247–252, 2014.

Our approach to software transplantation is to grow code for new functionality, rather than to improve existing non-functional properties of the system (such as execution time). We provide empirical evidence that grow and graft is, indeed, achievable on a large real-world system. Specially, we report on two grafting operations, carried out to insert new functionality into Pidgin, a 200kLoC C/C++ instant messaging system which has several million users worldwide.

In our first illustrative operation, we grafted a simple human-written 'countdown' code fragment. This graft augments Pidgin with the new feature that all the user's messages include the time remaining to the SSBSE 2014 challenge deadline. This example simply serves to illustrate the application of our grafting approach. We then report on a more challenging transplantation, in which we grew and grafted a new feature that augments Pidgin with a 'Babel Fish' that simultaneously translates the user's English language instant messages into Portuguese and Korean. This new functionality is sufficiently anachronistic that we can be relatively sure that no human has hitherto developed it. Nevertheless, it might be useful for improving communications between users who happen to be American, Brazilian and Korean; the general and co-program chairs of SSBSE 2014 and the chair of the SSBSE 2014 challenge track, for example.

1. Grow: First, we use Genetic Programming (GP) to grow, in isolation, fragments of code partly guided by 'suggestions' provided by the developer. The suggestions consist of the names of library functions the developer believes *may* be important and partial ordering constraints on when they should be called. The human may also provide a few necessary conditions for correctness that the programmer knows, from his or her intuition, ought to hold in any correct solution. Our approach thus *does* require a small contribution from the human to capture functionality (with tests) and to constrain the search space with high level humanly-intuitive suggestions; the rest is entirely automated. However the programmer is *not* required to choose any specific variable names, *nor* assignment statements *nor* expressions, *nor* to construct any specific calls, *nor* to determine where any of the statements should reside within the system to be improved.

2. Graft: The working prototype grown by GP is a fragment of code that implements the desired functionality, but which does so entirely in isolation. The remaining challenge is to find a way to incorporate our GP-grown fragment into the larger real-world system. This is the task of the second, 'grafting', phase of our Grow and Graft approach. Grafting the donor fragment involves two activities: finding a viable host insertion location (or locations), and identifying the expressions that serve as parameters between the donor and the host.

This is the first time that either the SBSE or GP community has reported the successful evolution of entirely new functionality in a real world software system. Previous work has either concentrated on improving non-functional properties or repairing existing broken functionality rather than growing genuinely new functional behaviour. We believe that the ability to extend existing real world systems in this way may open many exciting possibilities: Future work can use Grow and Graft to invent software development approaches that blend a small amount of human intuition with a large amount of automated search.

2 Grafting an SSBSE Challenge Deadline Countdown

A small fragment of C code `count_down()` was written by hand, in complete isolation from Pidgin, and by a programmer unfamiliar with Pidgin. The code fragment takes the current time and returns a string containing the number of days to the SSBSE Challenge Deadline. In this case, the grow phase is trivial, since we need only a single line of code and this is supplied by the programmer. We can thus focus on and explain our fully automated grafting process that transplants [3] the new functionality into Pidgin.

We grafted the countdown into the Pidgin `Timestamp 2.10.9` plugin. We used this plugin as a template into which the code is grafted. In the pidgin plugin there is only one variable of type `time_t`, but there are five variables of type `char*`. We implemented a simple grafting tool that instantiates each possible value of the template, inserting it before each line of the existing `Timestamp 2.10.9` code. There are 14 possible insertion points and 7 possible value substations, so the search space is $7 \times 14 = 98$, of which 69 compiled and ran without error and 2 also passed all test cases. The grafting process took 13 seconds on a 2.66 GHz 2-core machine with 1Gb RAM.

In this case, grafting is an enumerable search problem. However, for larger systems, grafting may, itself, be an SBSE challenge. Grafting was made easier by the Pidgin plugin mechanism, which reduced the graftable search space. We needed only to graft code into the plugin, rather than the whole system. In general, any form of modularity could also be used to reduce the graft space.

3 Growing and Grafting Babel Fish into Pidgin to Create Babel Pidgin

We seek to grow a new functionality (which we christen 'Babel Fish') and then to transplant this into the Pidgin plugin `Text Replacement 2.10.9` using grafting, as we did in Section 2.

Growing the Babel Fish requires Genetic Programming (GP). Our GP system is strongly typed and evolves imperative language code statement sequences, the statements of which are either function calls or assignments. The GP system takes a grammar and a source template as input. The grammar specifies a list of data types and functions suggested by the developer as likely to be useful. For growing a Babel Fish we (the programmer in this example) provided the GP with the names of the `GoogleTranslate` API call for Portuguese and Korean together with the names of string processing library functions (`concat` and `strlen`). Of course, this is a significant help to the GP, which could hardly be expected to 'discover' that it should call `GoogleTranslate`, for example. Nevertheless, such suggestions clearly also denote the most *trivial* application of human intuition.

The GP system applies a single-point crossover operator with a probability of 0.8. After crossover, one of three mutation operators is applied (selected with uniform probability). The three mutation operators are variable replacement, statement replacement and statement swapping.

We use an aggressive elitism selection process that replicates the best individual and inserts it into the new population between 1 and 250 times with a Coupon Collector Distribution (with expected mean of 197 so, typically, 197 insertions). Our approach to elitism aims to ensure sufficient retention of promising code schema in the gene pool. The population size is 500 and the GP terminates when best fitness remains unchanged for 20 generations. All experiments were repeated 30 times to allow for inferential statistical comparison of results.

#	Category	Description of fitness component
1	Essential	Compiles using `gcc`
2	Essential	Must not crash
3	Essential	No system warnings appear
4	Essential	Correct output
5	Necessary	Portuguese-trans gets correct string
6	Necessary	Korean-trans gets correct string
7	Necessary	`concat` gets correct string
8	Necessary	Output contains Portuguese-trans
9	Necessary	Output contains Korean-trans
10	Necessary	Output is different from input
11	Inclusion	Call to Portuguese-trans
12	Inclusion	Call to Korean-trans
13	Inclusion	Call to get text buffer
14	Inclusion	Call to set text buffer
15	Inclusion	Call to buffer start
16	Inclusion	Call to buffer end
17	Inclusion	Call to `strlen`
18	Inclusion	Call to `concat`
19	Ordering	buffer start before get text
20	Ordering	buffer end before get text
21	Ordering	Portuguese-trans after get text
22	Ordering	Korean-trans after get text
23	Ordering	Portuguese-trans before set text
24	Ordering	Korean-trans before set text

Fig. 1. The 24 Fitness Components Used

We experimented with 8 different fitness functions, composed of subsets of 24 equally-weighted fitness components drawn from those defined in Figure 1 on the lefthand side of this page. We do this in order to understand the trade off between human (programmer) effort and automated (GP) effort. In an ideal world, the GP would do all the work. However, since traditional GP has tended to grow only simple and small functions rather than whole programs [5,9] this may be unrealistic. It may also be unnecessary; the programmer need only offer a few simple and (to a human) *naturally intuitive* suggestions of criteria she or he expects to be important in any correct solution. Naturally, we would prefer that the human would be required to provide the least information possible to guide the GP, since we wish to place the least possible software development burden on human shoulders.

In order to experiment with this human-machine tradeoff, we categorised our fitness functions into four distinct categories: (E)ssential, (N)ecessary, (I)nclusion and (O)rdering. Essential fitness captures the implicit test oracle [4] required by any implementation of any program and therefore requires no human guidance. Necessary fitness consists of properties that the programmer knows will be necessary in any correct solution for the problem in hand. The Inclusion category lists the names of library functions that the programmer believes may prove useful. Finally, the Ordering category is a mechanism through which the programmer specifies partial ordering constraints on the calls made during execution.

Even if the programer were to be asked to provide all of this information, then the (human) effort required would be relatively low. Certainly, human effort would be lower than that required were the human asked to *write* the program extension from scratch, *and* to work out how it should interface to the existing system, *and* to determine where it should be located.

3.1 Results from Growing and Grafting Babel Fishes

Fitness name	Components used (see Fig. 1)	successful growths (p val. compares to E)	mean fitness evaluations ($N = 30$)
E	1..4	0 (p=N/A)	10,500
EI	1..4,11..18	0 (p=N/A)	10,833
EO	1..4,19..24	0 (p=N/A)	11,333
EN	1..4,5..10	2 (p=0.306)	15,483
EIO	1..4,11..24	1 (p=0.500)	13,366
ENI	1..4,5..10,11..18	8 (p=0.030)	16,266
ENO	1..4,5..10,19..24	7 (p=0.044)	14,516
ENIO	1..24	9 **(p=0.020)**	15,800

Fig. 2. Results for Babel Fish Growth

Figure 2 (on the left) presents the results of our experiments on 8 different fitness functions. In order to more rubustly analyse the results we use a nonparametric two-tailed binomial test to compare the number of successful transplants that we achieved using the essential fitness, E, with each and all of those we achieved using fitnesses EI to ENIO in Figure 2. We use the Hochberg correction in order to account for the fact that we are performing five different inferential statistical tests. With an α level of 0.05, this corrected statistical test indicates that the result for ENIO is significantly different to that for E (with a Vargha-Delaney $\hat{A}_{12}$ effect size 0.64). The p values for ENO and ENI are also smaller than 0.05, but are not considered significant after the Hochberg correction has been applied.

As we expected ENIO, which provides the most guidance to the GP, performs the best: almost a third of its runs result in a successful Babel Fish transplant into Pidgin. This result is statistically significantly better than the results obtained using the essential fitness E alone, which provides insufficient guidance. The results for other fitness choices are also encouraging. They indicate that the grow and graft approach can augment existing real world systems with new functionality, guided by only very modest (and easily obtained) human intuition. Our results provide evidence that the most powerful form of guidance comes from assertions that capture necessary conditions for correctness. Simply adding these simple and intuitive necessity constraints (N) to the essential fitness components (to give EN) leads to successful transplants. However, of all the fitness components with which we experimented, we speculate that defining such necessary constraints would tend to require the most programmer knowledge and effort. It is therefore encouraging that simply using Inclusion and Ordering constraints (EIO fitness in Figure 2) is sufficient to guide the search to a successful transplant. We believe that this result is exciting because it provides an existential proof that a real world systems can be augmented with new functionality with the most meagre of human guidance.

Figure 2 reports results for growth. The graft phase is entirely automated. There are 23 possible insertion points and 2 possible value substitutions, giving a graft space of 46. We use the Babel Fish whose growth was guided by EIO fitness to illustrate graft performance over all our evolved Babel Fishes. Since it was grown with least human effort, it is encouraging that, like all our Babel Fishes, it has at least one successful graft point. Of the 46 graft attempts, 2 failed to compile, 3 crashed, 24 executed without crashing, but failed the functionality test, while 17 were grafted entirely successfully (and thus equally good). The grafting tool enumerated all 46 solutions in 24s. Computationally and conceptually, grafting is surprisingly easy and effective.

4 Conclusions

We have demonstrated that Genetic Improvement can be used to grow code features in isolation, largely oblivious of the system into which they are subsequently to be grafted. Surprisingly little human guidance and domain knowledge is required. Future work will investigate further the minimal human guidance requirement for Grow and Graft Genetic Improvement (GGGI).

Acknowledgement. This work is part supported by the DAASE [1] and GISMO projects [2].

References

1. Harman, M., Burke, E., Clark, J.A., Yao, X.: Dynamic adaptive search based software engineering (keynote paper). In: ESEM, pp. 1–8 (2012)
2. 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)
3. Harman, M., Langdon, W.B., Weimer, W.: Genetic programming for reverse engineering (keynote paper). In: WCRE (2013)
4. Harman, M., McMinn, P., Shahbaz, M., Yoo, S.: A comprehensive survey of trends in oracles for software testing. Tech. Rep. Research Memoranda CS-13-01, Department of Computer Science, University of Sheffield (2013)
5. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge (1992)
6. Langdon, W.B., Harman, M.: Evolving a CUDA kernel from an nVidia template. In: IEEE World Congress on Computational Intelligence, pp. 1–8. IEEE (2010)
7. Langdon, W.B., Harman, M.: Genetically improved CUDA C++ software. In: EuroGP (to appear, 2014)
8. Langdon, W.B., Harman, M.: Optimising existing software with genetic programming. IEEE Transactions on Evolutionary Computation (TEVC) (to appear, 2014)
9. Langdon, W.B., Poli, R.: Foundations of Genetic Programming. Springer (2002)
10. Le Goues, C., Forrest, S., Weimer, W.: Current challenges in automatic software repair. Software Quality Journal 21(3), 421–443 (2013)
11. Orlov, M., Sipper, M.: Flight of the FINCH through the java wilderness. IEEE Transactions Evolutionary Computation 15(2), 166–182 (2011)
12. Petke, J., Harman, M., Langdon, W.B., Weimer, W.: Using genetic improvement & code transplants to specialise a C++ program to a problem class. In: EuroGP (to appear, 2014)
13. Sidiroglou-Douskos, S., Misailovic, S., Hoffmann, H., Rinard, M.C.: Managing performance vs. accuracy trade-offs with loop perforation. In: FSE, pp. 124–134 (2011)
14. Sitthi-amorn, P., Modly, N., Weimer, W., Lawrence, J.: Genetic programming for shader simplification. ACM Transactions on Graphics 30(6), 152:1–152:11 (2011)
15. White, D.R., Arcuri, A., Clark, J.A.: Evolutionary improvement of programs. IEEE Transactions on Evolutionary Computation (TEVC) 15(4), 515–538 (2011)