Automated Transplantation of Call Graph and Layout Features into Kate

Alexandru Marginean, Earl T. Barr, Mark Harman, and Yue Jia

UCL, Department of Computer Science, CREST Centre

Abstract. We report the automated transplantation of two features currently missing from Kate: call graph generation and automatic layout for C programs, which have been requested by users on the Kate development forum. Our approach uses a lightweight annotation system with Search Based techniques augmented by static analysis for automated transplantation. The results are promising: on average, our tool requires 101 minutes of standard desktop machine time to transplant the call graph feature, and 31 minutes to transplant the layout feature. We repeated each experiment 20 times and validated the resulting transplants using unit, regression and acceptance test suites. In 34 of 40 experiments conducted our search-based autotransplantation tool, μScalpel, was able to successfully transplant the new functionality, passing all tests.

1 Introduction

We recently introduced a search based technique for automated software transplantation [2,7]. Guided by dependence analysis and testing, our approach uses a variant of genetic programming to identify and extract useful functionality from a donor program, and transplant it into a (possibly unrelated) host program. We implemented our approach as a tool called μScalpel, which is publicly available [1].

In this challenge paper, we illustrate the way in which realistic, scalable, and useful real-world transplantation can be achieved using μScalpel. We apply our tool to the SSBSE 2015 Challenge program `Kate`[1], a popular text editor based on KDE. Its rich feature set and available plugins make it a popular, lightweight IDE for C developers. We perform two automated transplantations using μScalpel. In the first one, we transplant call graph drawing ability from the GNU utility program `cflow`, to augment `Kate` with the ability to construct and display call graphs.

This is a useful feature for a lightweight IDE, like `Kate`, and would clearly be nontrivial to implement from scratch. Using our search based autotransplantation, μScalpel, the developer merely needs to identify the entry point of the source code in the donor program (`cflow` in this case) and the tool will do the rest; extracting the relevant code, matching names spaces between host and donor and executing regressions, unit and acceptance tests. Like much previous work on genetic programming [12], our approach relies critically on the availability of high quality test suites. We do not directly address this issue in the present paper, but believe

[1] `http://kate-editor.org`

that existing achievements in Search Based [5] and other [4] test data generation techniques will help us to ensure that this reliance is reasonable and practical.

Our second transplantation incorporates a pretty printer for C, which `Kate` only partially supports and which its users have requested. At the time of writing, we deployed a new version of `Kate` that incorporates these features. We hope to be able to report on the uptake of this ‘genetically improved’ `Kate` at the conference.

Our work is closely related to recent achievements in genetic improvement, which have been able to dramatically speed up real world systems [9,14,15], port between languages [8], balance memory consumption and execution time [16], reduced energy consumption [3,13] and fix bugs [10]. Most closely related to our approach is work on auto-specialisation using transplantation [11] and grow and graft genetic improvement [6]. Whereas auto-specialisation transplants from different versions of the same system (or closely related systems) and grow and graft transplants newly grown simple features, μSCALPEL transplants large-scale features (and subsystems) from one or more donors into an unrelated host.

2 The μSCALPEL Transplantation Framework

We presented a framework for transplanting a feature between two unrelated systems and a tool to implement our approach [1] in our recent ISSTA paper [2], so we provide merely a summary here to make the paper self-contained. Given a host H program that is lacking a feature of interest, a donor program D, that implemented the feature, and a lightweight annotation system, our tool, μSCALPEL, attempts to autotransplant the feature from D into H. The feature of interest in the donor is called the Organ. From the entry of the donor, there are one or more path that reaches the Organ Entry point, called veins.

Our approach uses Genetic Programming (GP), augmented by static analysis, for extracting, configuring, and transplanting the organ into the host environment. GP explores combinations of statements on the vein, and in the host–donor variable mappings, that will enable the organ to execute in the host environment, guided by testing. The first stage of our approach uses context insensitive slicing on the call graph of the donor program to construct a map, with the key being the variables available in the vein, and the values being the variables in scope at the implantation point in the host [2]. GP is used to transplant the feature in the host system, having the host–donor map, and the code base represented by these context insensitive slices. The search space has two dimensions: the variable mapping and the statements available to form the transplant itself. The tool that implements out approach, μSCALPEL, is publicly available [2].

3 Applying Autotransplantation to Kate

We chose two popular, real world systems, as the donor programs: GNU `cflow`[2], and GNU `Indent`[3]. The former generates call graphs for C programs, a feature that is missing from `Kate`; the latter pretty prints C source files, with far fewer restrictions than `Kate`’s existing built-in indentation functionality. `Kate`’s existing

[2] http://gnu.org/software/cflow [3] http://gnu.org/software/indent

indentation feature fails to wrap a line that is too long, for example. It simply adds space or tabs in a programming language independent manner, whereas GNU's `Indent` exploits language awareness to provide far better formatting functionality.

μScalpel requires user to provide an implantation point in H, and the entry point of the feature in D. We chose one of the `Kate` plugins as the implantation point. We start from the time date plugin template[4], and annotate the entry point in `Kate`, in the function '`void TimeDatePluginView:: slotInsertTimeDate()`' of the plugin. This function is called every time the user selects the menu element corresponding to the current plugin. We chose this point as the implantation point for allowing the user to chose whenever wants to generate the call graphs. The annotation added in the host is: '`void TimeDatePluginView:: slotInsertTimeDate(){ /* IP */`' .

For the `cflow` donor, the desired functionality is to transplant the tree form of the call graphs. Thus, we label the function '`tree_output()`' as the organ entry in the donor system. The organ generates the call graph of a C program and displays it in the tree format (option '`-T`' from `cflow`). The annotation is: '`void tree_output(){ /*OE*/`'. For the `Indent` donor, the desired functionality is to enable `Kate` to completely format a C source file. We want to format the current opened document in the `Kate`'s main page. Thus, we label the function '`indent_single_file()`' as the organ entry point. This function reformats the source file, according to the settings of GNU `Indent`. The annotation is: '`exit_values_ty indent_single_file(BOOLEAN using_stdin){/*OE*/`'. The developer need only provide μScalpel with these simple annotations and suitable test cases and the reminder of the transplantation process is entirely automated.

We used several different test suites to validate our transplant. First, we execute the original regression test suite, available with `Kate`. Since this test suite does not execute the organ, we augmented it with test cases specifically aimed at executing the organ. Second, we generated an acceptance test suite, aimed at checking the transplanted functionality when executed in the host. As with all approaches to GP, the test cases are used to guide the search for suitable code.

Provision of these test cases remains the responsibility of the software engineer. Such tests, or a large subset thereof, would be likely required by a human transplantation process in any case, so this is not a significant additional burden. Furthermore, even were such costs attributed to the autotransplantation process, it would be likely easier, in many cases, to define suitable test cases to *check* a transplant than it would be to *generate* one from scratch while ensuring it is sufficiently tested. In order to estimate the human cost, we recorded the elapsed developer time required to construct the isolation, acceptance, and regression$++$ test cases that are specifically required to validate the transplantation, thereby providing an upper bound on human effort. Our estimation for annotations and the test suites is one hour for both of the transplants.

For all our test suites we provide coverage information. Tab. 2 shows the results of our tests for both donor programs. We also manually generated the ice-box test suite, used by the GP in the process of transplanting the feature.

[4] https://techbase.kde.org/Development/Tutorials/Kate/KTextEditor_Plugins

4 Experiments and Results

We seek to answer three research questions. **RQ1)** Can we transplant the two desired features into `Kate`, without breaking the original functionality of `Kate`? **RQ2)** Do the transplanted features (organs) provide the desired functionality inside `Kate`? **RQ3)** What is the computational effort required by automatic transplantation? We repeat each of the transplantation experiments 20 times, executing μSCALPEL on a Ubuntu 14.10 machine, 64-bit architecture, 16 GB ram, and 8 cores processor.

Tab. 2 presents the number of runs in which for every test suite, all test cases pass. We report the number of successful runs for the regression, augmented regression, and acceptance test suites individually and also report the coverage achieved by test cases (of the entire `Kate` system, and of just the transplanted organ). This coverage data is that reported by the publicly available coverage metric tool `gcov`. Tab. 2a shows the results of the test suites for `cflow` donor transplant, while Tab. 2b shows the results of the test suites for `Indent` donor transplant. For `cflow`, 16 out of 20 runs where unanimously successful, while for `Indent` 18 out of 20 runs were unanimously successful.

We deem a transplantation attempt to be successful if (and only if) all the test cases from the corresponding test suite passed. The row labelled 'Unanimously' reports the number of transplantation attempts in which *all* test cases passed in all test suites. The line 'Isolation' reports the results of the isolation test suite, which is used by the GP algorithm for evolving the organ (as opposed to being used for valuation purposes).

Observe that even were we to find that automated transplantation was only successful one a few of the 20 attempts, then this would be sufficient to demonstrate the feasibility of autotransplantation in general. The testing process can be used to validate any transplantation attempt, allowing the software engineer to discard any and all failed attempts. As a result show, autotransplantation achieves a much higher success rate than this, minimal, feasibility requirement. Overall, we have evidence that autotransplantation is feasible for the popular real world system `Kate`. We now turn to the specific research questions, we posed to answer them.

RQ1 Tab. 2 revels that for both of the transplants, all the regression test cases passed. However, the organs were not executed by the existing regression test suites, so we manually augmented them to generate the regression++ test suites. Organ coverage for the regression++ test suites is: 59% for `cflow`, and 58% for `Indent`. For `cflow` 17 out of 20 transplantation attempts passed all test in these augmented test suites, while for `Indent`, 18 out of 20 pass all. Clearly one can never do enough regression testing, but these results provide release some confidence. In future work we plan to use automated search based software testing [5] to further improve autotransplantation regression testing.

Table 1. Runtime data, averaged over 20 runs.

Donor	Time(min.)	
	Avg	Std Dev
GNU `cflow`	101	31
GNU `Indent`	31	6

RQ2 For `cflow` 18 out of 20 transplants passed all acceptance tests, while for `Indent` 19 out of 20 pass, giving confidence that μSCALPEL has successfully transplanted code, such that the desired functionality is available to host program.

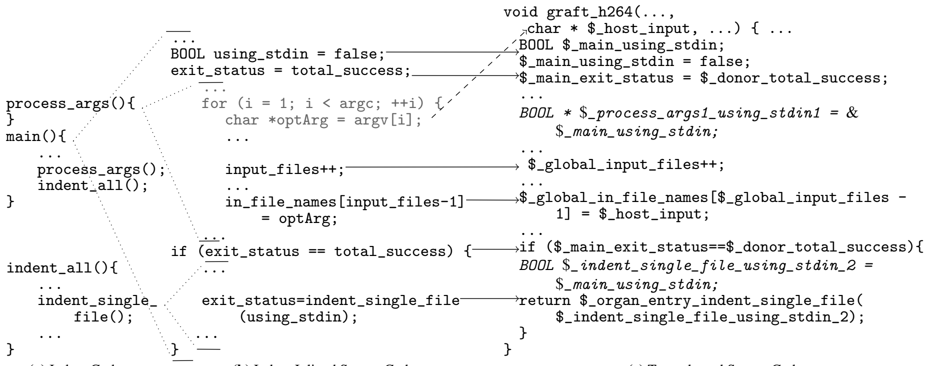
Table 2. Transplantation results. Figures marked with * exclude regression test cases that failed before the transplantation (only one for `Kate`).

(a) GNU `cflow` Donor

Category	Pass Rate	Coverage (%)	
		All	Organ
Unanimously	16	-	-
Isolation	18	-	-
Regression	20*	62	0
Regression++	17	74	59
Acceptance	18	52	59

(b) GNU `Indent` Donor

Category	Pass Rate	Coverage (%)	
		All	Organ
Unanimously	18	-	-
Isolation	19	-	-
Regression	20*	66	0
Regression++	18	78	68
Acceptance	19	48	68



```
process_args(){
}
main(){
    ...
    process_args();
    indent_all();
}


indent_all(){
    ...
    indent_single_
        file();
    ...
}
```

(a) Indent Code

```
...
BOOL using_stdin = false;
exit_status = total_success;
    ...
    for (i = 1; i < argc; ++i) {
       char *optArg = argv[i];
       ...

       input_files++;
       ...
       in_file_names[input_files-1]
          = optArg;
    ...
if (exit_status == total_success) {
    ...

    exit_status=indent_single_file
       (using_stdin);
   ...
}
```

(b) Indent Inlined Source Code

```
void graft_h264(...,
   char * $_host_input, ...) { ...
  BOOL $_main_using_stdin;
  $_main_using_stdin = false;
  $_main_exit_status = $_donor_total_success;
  ...
  BOOL * $_process_args1_using_stdin1 = &
      $_main_using_stdin;
  ...
   $_global_input_files++;
  ...
  $_global_in_file_names[$_global_input_files -
      1] = $_host_input;
  ...
  if ($_main_exit_status==$_donor_total_success){
  BOOL $_indent_single_file_using_stdin_2 =
      $_main_using_stdin;
  return $_organ_entry_indent_single_file(
      $_indent_single_file_using_stdin_2);
  }
}
```

(c) Transplanted Source Code

Fig. 1. Transplant operation in `cflow` donor transplant. Code snippet from the beginning of the graft. ···| means function inlining; `optArg` is mapped to `$_host_input`; → means statement replacement under α — renaming; grayed statements are deleted.

RQ3 Tab. 1 reports the timing information for the transplants. On average, transplanting the call graph feature from `cflow` took 101 minutes, while the layout feature from `Indent` took 31 minutes. In less than 44 hours total time, we were able to complete all 40 repetitions of the two experiments. The human effort required to incorporate these two new features would surely have been considerably greater.

A Flavour for the Transplants Produced by μ**SCALPEL** Fig. 1 provides a flavour of the `Indent` transplant. Fig. 1a shows portion of the vein, identified in the static slicing processing. The vein starts at the function `main()`, and ends at organ entry; the function `indent_single_file()`. The vein contains the function `process_args()`, which initialises globals, based on the command line parameters originally used in the donor `Indent`. Fig. 1b shows the resulting code after the inlining process. The brackets capture the code corresponding to each original function. Fig. 1c shows the code transplanted into `Kate` by one of the successful transplants. An α–renaming scheme is used to avoid namespace conflicts within the host, and between the inlined functions.

Some organ statements must be removed, due to failed test cases or incorrect binding to host variables. Some variables may even be unbindable, leading to an uncompilable (or crashable) transplant. For example, the variables `argc` and `argv` simply cannot be bound to host variables, because `Kate` has no concept of 'command line argument'. Fortunately, GP discovers such issues. It removes the first `for` statement in Fig. 1b . The variable `optArg` is used for parsing the command line parameters of `Indent`. This variable was mapped at the variable `$_host_input$`, thereby correctly using input from `Kate` call graph computation.

5 Conclusions

We demonstrated that search based automated transplantation (a form of genetic improvement) can be used to automatically transplant non-trivial features (that are requested by users, but hitherto unimplemented by developers) into the large real-world system `Kate`.

References

1. Barr, E.T., Harman, M., Jia, Y., Marginean, A., Petke, J.: μscalpel. http://crest.cs.ucl.ac.uk/autotransplantation/MuScalpel.html (2014)
2. Barr, E.T., Harman, M., Jia, Y., Marginean, A., Petke, J.: Automated software transplantation. In: ISSTA (2015), to appear
3. Bruce, B., Petke, J., Harman, M.: Reducing energy consumption using genetic improvement. In: GECCO 2015 (2015)
4. Cadar, C., Sen, K.: Symbolic execution for software testing: Three decades later. CACM 56(2), 82–90 (2013)
5. Harman, M., Jia, Y., Zhang, Y.: Achievements, open problems and challenges for search based software testing (keynote). In: ICST (2015)
6. Harman, M., Langdon, W.B., Jia, Y.: Babel pidgin: SBSE can grow and graft entirely new functionality into a real world system. In: SSBSE (2015)
7. Harman, M., Langdon, W.B., Weimer, W.: Genetic programming for reverse engineering (keynote paper). In: WCRE (2013)
8. Langdon, W.B., Harman, M.: Evolving a CUDA kernel from an nVidia template. In: IEEE CEC (2010)
9. Langdon, W.B., Harman, M.: Optimising existing software with genetic programming. TEVC 19(1), 118 – 135 (2015)
10. Le Goues, C., Forrest, S., Weimer, W.: Current challenges in automatic software repair. SQJ 21(3), 421–443 (2013)
11. 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(2014)
12. Poli, R., Langdon, W.B., McPhee, N.F.: A field guide to genetic programming. Published via http://lulu.com and freely available at http://www.gp-field-guide.org.uk (2008)
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. 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)
15. Sitthi-amorn, P., Modly, N., Weimer, W., Lawrence, J.: Genetic programming for shader simplification. ACM TOG 30(6), 152:1–152:11 (2011)
16. Wu, F., Harman, M., Jia, Y., Krinke, J., Weimer, W.: Deep parameter optimisation. In: GECCO 2015 (2015)