Fitness Functions and Transformations in an Automated Process

NATAŠA SUKUR, DONI PRACNER and ZORAN BUDIMAC, University of Novi Sad

An important aspect of program maintenance is the understanding of the original code. One option is to automate the process with code transformations as much as possible by using fitness functions to evaluate the improvements made. The process presented focuses on low-level code and relies on FermaT, a program transformation system based on the WSL language. It has been used in software evolution applications, mainly for legacy systems and conversions of low-level code into high-level structures. This paper presents experiments with different fitness functions and compares their influence on the end result. The main focus is on the number of tried and applied transformations.

1. INTRODUCTION

In modern environments software is almost constantly in a state of change, adapting to the new needs. There is a strong need for tools that can help in various stages of maintenance. One of the problems is that even just understanding the original logic of the code can be hard. The experience has shown that understanding a piece of code one has written can often be cumbersome, let alone when the work is of someone else. Not understanding the original code and its functionalities correctly can lead to creating new errors in software rather than its improvements.

Software maintenance is a very important part of the software life cycle. It is the longest phase of the life cycle and it can be used for perfective, adaptive and corrective purpose, depending on the needs. When maintaining the software, it is very important to keep the quality of the software at the same level or to improve it. The perseverance in quality preservation is important for providing a long life for the software at hand. The deadlines are short and the demands for changes are high, especially compared to the early days of computer science and technology. By introducing new functionalities and correcting the existing ones, there is a lot of room for creating errors.

Software evolution and reengineering are another important aspect of the software lifetime. It is almost inevitable that software will have to adapt and evolve from the functionalities, environment and scale point of view over time. If the software has had many changes or if there is a need for significant changes, it is often necessary to completely reengineer it. Again, without understanding the functionalities of the software, reengineering cannot be done properly. Software evolution is repeated reengineering working towards creating a better system.

This work is partially supported by Ministry of Education and Science of the Republic of Serbia, through project no. OI174023: "Intelligent techniques and their integration into wide-spectrum decision support";
Author's address: Nataša Sukur, Doni Pracner and Zoran Budimac, University of Novi Sad, Faculty of Sciences, Department of Mathematics and Informatics, Trg Dositeja Obradovića 4, 21000 Novi Sad, Serbia; email: natasa.sukur@dmi.uns.ac.rs, doni.pracner@dmi.uns.ac.rs, zjb@dmi.uns.ac.rs

Copyright © 2019 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).
In: Z. Budimac and B. Koteska (eds.): Proceedings of the SQAMIA 2019: 8th Workshop of Software Quality, Analysis, Monitoring, Improvement, and Applications, Ohrid, North Macedonia, 22–25. September 2019. Also published online by CEUR Workshop Proceedings (http://ceur-ws.org, ISSN 1613-0073)

CEUR-WS.org/Vol-2508/paper-suk.pdf

If these perfective processes are done by hand and rely purely on the expertise and experience of the engineer, as well as his understanding of the problem, there is still a possibility for creating errors. The ideal scenario would be having a helper tool that would provide automation or at least semi-automation of software evolution.

This paper presents experiments with an automated source code transformation process. The foundation of this research is based on FermaT program transformation system and *mjc2wsl*, a tool that translates MicroJava bytecode to WSL (Wide Spectrum Language). Upon translation, the resulting WSL code is transformed by FermaT. The transformation process in this research is automated by another tool, from the translated low-level program to a higher-level and more understandable code. This approach uses the *hill climbing* algorithm [Russell and Norvig 2016], a search algorithm which moves "uphill", which means that it constantly tries to move towards the increasing value until it reaches a peak. In this case, the peak would be the best program possible, meaning that no further transformation could lead to a better program. In order to determine which values are better, this process is guided by a *fitness function*, a concept which originates from evolutionary computing. The ideas of evolutionary computing come from the natural process of evolution, where the qualities of an individual determine its chances of survival. The same could be applied to the solving of the computing problems: a *fitness function* should point to the most suitable candidate solutions which seem to have the best chances for solving the problem at hand. In this case, a "better" program is the one which is simpler and more understandable. The quality of the candidate solutions usually reflects in the values of metrics, where the better ones tend to be lower. That is the reason why the built-in WSL metrics were the obvious first candidates for fitness functions.

In our previous work [Sukur and Pracner 2018], the main question we tried to answer was whether using a certain metric as a fitness function would lead to the best improvements of the said metric in the end result. Our assumption was that will not be the case for all metrics. The experiments performed confirmed our expectations, since the best results for a metric were not always accomplished by the same metric as a fitness function. Also, there was no single fitness function which would give the best results by far. The best fitness functions were metrics whose initial values were quite high and whose values changed frequently.

In this paper, we are trying to analyze another aspect of using different metrics as fitness functions. The main observations in this paper will be in regard to comparison of the efficiency of the process when using different fitness functions and the differences in the number of transformation that the process tries and succeeds to apply. The final, transformed code is often very similar when compared across all variants of the process. However, there are noticeable differences in the number of tried and applied transformations.

The rest of the paper is organized as follows: related work is given in Section 2; the automated transformation process and experiment setups are explained in Section 3; the results of the experiments and their discussion are given in Section 4; finally the conclusions and options for future work are given in Section 5.

2. FOUNDATIONS AND RELATED WORK

Formal methods are important for the overall software reliability and are often used in various stages of its life cycle. They can be applied to software artifacts such as specifications, models or source code. Formal methods are used in different fields of software engineering, such as specification or development, but also for the purpose of software reengineering. Software reengineering consists of reverse engineering, functional restructuring and forward engineering. It relies on numerous formal methods, such as assertional methods, temporal logic, process algebra and automata throughout forward engi-

neering and functional restructuring. In reverse engineering, it sometimes relies on formal methods for activities such as formal specification and verification of existing systems, as well as introduction of new functionalities. It is, however, still debatable whether formal methods should be used in system development and to what extent. On one hand, formal methods are very important for the reliability of the systems and for the quality of the entire development process. In other opinion, they are not 100% reliable in (re)engineering processes and using them can be very costly. What is obvious is that there is, beyond any doubt, a strong need for a reliable methodology which would take care of the quality of processes throughout the entire life cycle [Yang and Ward 2003].

Formal methods are very versatile and there is not a single formal method suitable for all purposes. There are different quality criteria of formal methods, some of which are supporting automated tools for development, reliability, concurrency and existence of a proof system. Although many formal methods have their advantages and disadvantages in regard to these criteria, the conclusion by [Yang and Ward 2003] is that formal methods should be chosen depending on the nature of the problem at hand. Depending on the scale and nature of the problem, one should choose a suitable formal method, for example, *Z* in the case of large industrial applications [ISOZ 2002]; different process algebras for reasoning about concurrency and communication; and net-based formalisms for visual representation. However, formal methods are not so frequently used in reverse engineering. WSL [Ward 2013] is a language for reverse engineering of sequential systems, based on formal methods. The main idea of the entire WSL/FermaT system is strongly based around reverse and forward engineering.

Formal methods can also be suitable for code transformations. A program transformation is an operation which, once applied to a program, produces a program with the same external behavior [Ward 1989]. The idea around code transformation is to achieve cost reduction – for example, improvements regarding performance, memory usage or even portability. Code transformations are not only useful for evolution of existing software, but also in the development phases of new software.

FermaT is one such system that offers program transformations. It is based around the WSL language, which stands for *wide spectrum language*, meaning that it contains both abstract mathematical specifications and low-level programming constructs. WSL contains standard language functionalities, such as commands and structures. Apart from that, another important aspect of WSL is *MetaWSL*, a set of operations that work on WSL programs themselves, as the name suggests. One of the main purposes of *MetaWSL* is its role in transformations. Program transformations are a part of the system and their correctness can be automatically checked. Transformations can be used for creating programs from specifications, performing reverse engineering of programs and getting specifications, as well as analyzing properties of a program. WSL was shown as very useful in various restructuring activities [Yang and Ward 2003], including industrial projects, where the aim was to convert legacy assembly code to human understandable and maintainable C and COBOL [Ward 1999; Ward 2004; Ward et al. 2004; Ward 2013]. Another tool that was made for assembly translation, with a slightly different focus is *asm2wsl* [Pracner and Budimac 2011].

The experiments in this paper use the hill climbing approach and rely on fitness functions for automated reengineering. There have already been some attempts to answer the question whether hill climbing is adequate and optimal approach for automatic program repair in [Arcuri and Yao 2008]. However, the conclusions did not show a lot of optimism for this approach, due to hill climbing tendency towards local optimums. Fitness functions have also been used for code improvement. Extensive research on automated software repair using fitness functions has been done, firstly focusing on C programs [Forrest et al. 2009] and assembly programs [Schulte et al. 2010], which resulted in applicability to any kind of code in general [Le Goues et al. 2012]. In this paper, we also tried to show that using different fitness function can lead to change in results and that these functions should be selected based on properties of the problem at hand. The research which focuses on the automated bug

detection [Fast et al. 2010; de Souza et al. 2018] also speaks in that favor and tries to give directions for designing these fitness functions in order to get the best results.

Rascal is a domain specific language (DSL) for metaprogramming, which can be used for static analysis, program transformation and implementation of other DSLs [Klint et al. 2011]. It has been used, inter alia, on C, Java and PHP [Hills and Klint 2014].

SmaCC (Smalltalk Compiler-Compiler) is a parser generator, successfully used to write custom refactoring and transformation tools for languages such as Java, C#, and Delphi [Brant and Roberts 2009]. The purpose of these tools varies from small scale refactorings to large scale migration projects. SmaCC was also ported to *Pharo*, which made usage of *Moose* analyzers and other *Pharo*-based software available [Brant et al. 2017; Ducasse et al. 2000].

3. AUTOMATED TRANSFORMATION PROCESS

The automated code transformation is done by a tool [Pracner and Budimac 2017] which uses a *hill climbing* algorithm, where the progress relies on the results of a *fitness function*. Simply explained, the process tries to apply code transformations as long as it is improving the structure of the input program, and the fitness function determines whether the program is improved after a transformation is applied. The fitness function can be some numeric value which indicates the complexity of the program (various software metrics). The program which has better *fitness* is usually the one whose complexity is less than of the original. The *hill climbing* script which was used in this research tries to apply one transformation at a time. However, if there is no improvement, the script attempts to achieve it by combining two transformations. The process is finished once it has reached the program of the highest quality possible. The original automated script used the structure metric as fitness. It is a custom WSL metrics which gives a weighted sum of the structures in the program.

The process records all intermediate steps which have resulted in some improvement in separate files, which gives more insight into the details. The process is also fully recorded in logs, which means that the order of transformations that were tried and successfully applied is also available for further inspection.

Code transformation is done by FermaT. However, since FermaT transforms only code written in WSL, it is necessary to translate the original source code to WSL by corresponding translation tools. The reduction of size and complexity of the outputs is not highly important for these tools, since the transformation part of the process takes care of that. The translation is also done in such a manner so that the low-level structures and operations retain the same level of abstraction. The translator tool for MicroJava, a subset of Java programming language [Mössenböck 2018], is *mjc2wsl* [Pracner and Budimac 2017]. The tool does not work with the code written in MicroJava, but rather with the compiled bytecode obtained from the original high-level source code.

In this paper, the hill climbing process with a number of different fitness functions was tested on a set of MicroJava programs called *alpha-mj* [Pracner 2019]. This set of code samples was carefully created with the idea to cover different properties of code and the virtual machine – recursion, in/out operations, some erroneous situations (division by zero) and similar. Previous experiments have shown that changing the fitness function can influence the end results of the process. Some common properties of the fitness functions that led to best code improvements were usually software metrics which had high starting values and which had tendencies to change their values easily when transformations were applied. Changing values easily meant that the process could make a lot of progress by trying and successfully applying many transformations, whereas the process was significantly longer and less successful when these properties were not present (i.e, *McCabe's cyclomatic complexity*) [Sukur and Pracner 2018].

Most of the tested fitness functions were basic metrics built-in to WSL [Yang and Ward 2003]. These include McCabe's cyclomatic complexity, marked as *fit-mccabe*; the number of statements (*fit-stat*); the size of the abstract syntax tree (*fit-size*); control flow and data flow – the number of variable accesses and updates, combined with the number of procedure calls and branches (*fit-cfdf*); structure metric – a custom WSL metric for representing the complexity of program structures, gives different weights to various types of structures (*fit-struct*).

As an initial experiment into more complex fitness functions, two combinations of "simple" metrics were also used. One tries to combine many different aspects of the program in a sequence. It is named *fit-o1* and evaluates the new program as better if it has, in order, less actions, calls, McCabe's complexity, statements or expressions. The other one, *fit-o2*, is a simple expansion of the originally used metric. It first compares the number of calls, and then the *structure* metric.

Additionally, another fitness function (*fit-max*) was used to see the maximal number of transformations tried. Basically it is a function that always returns the same result for any program, therefore the hill climbing process can never advance and will just go through all of the possible transformations.

We also define several groups of these fitness functions, based on the results that will be discussed later. The two complex functions, *fit-o1* and *fit-o2*, are in group *o-fit*; functions *fit-size*, *fit-struct* and *fit-stat* are in group *s-fit*; and the union of these two groups is named *so-fit*.

4. EXPERIMENT RESULTS

The main focus of this paper is the comparison of efficiency of the process with different fitness functions, mainly how many transformations were tried and how many were applied to achieve the end result. However, the fitness functions can and will lead to different final programs and these need to be compared quality-wise first to have a useful comparison of efficiency. The analysis here will focus on the *structure* metric of the end results, since most of the other metrics have very similar relative values. It was chosen since it gives a good approximation of the abstraction level of the program. The improvements of WSL programs from their original low-level version to the final, more abstract version are expressed as the percentage of improvement, to overcome the differences in program sizes.

It is obvious from the definition of *fit-max* that it will lead to no improvements in the programs, so it will not be discussed in this context further. The least improvements in programs is always with *fit-mccabe*, usually by a significant margin compared to the other fitness functions. The main reason for this is that the values of McCabe's cyclomatic complexity are not easily changed by a transformation, and therefore there are very few that will lead to forward steps in the hill climbing process. This is not the case with the other fitness functions tested. The differences between end result metrics for the other fitness functions were relatively small, as shown in Table I. The *so-fit* group of fitness functions have nearly identical metrics, with a difference from the best being often 0, and rarely, on same samples up to 5 or 6 percentage points. Following them is the *fit-cfdf* function, which was consistently somewhat lower in its results, but not by a significant amount, mostly just a few percentage points, sometimes up to 8.

Table I. Differences of fitness functions from best results for *alpha-mj*

	fit-mccabe	fit-cfdf	fit-o1	fit-o2	fit-size	fit-stat	fit-struct
avg	81.75	3	0.5	0.44	0.25	0.56	0.06
stdevp	11.22	1.9	1.46	1.22	0.75	0.93	0.24
min	53	1	0	0	0	0	0
max	90	8	6	5	3	3	1

Differences are expressed in percentage points; less is better

One of the aspects that is important is the length of the process itself. The number of transformations tried in the whole process is very good for this purpose since these numbers are hardware independent and are not influenced by any other processes that might be running on the same machine. Figure 1 shows all of the tested fitness functions with a log scale of the transformations tried. The highest numbers are always with *fit-max*, as they should be, since this is a “fake” fitness function that never leads to improvements, and was meant to get an idea of how long the process can be. The highest number of transformations on these samples was more than 3 million, while the average values exceeded half a million. Next is *fit-mccabe*, which is always significantly worse than the other “real” fitness functions, generally almost as bad as *fit-max*. The rest of the functions have much more intertwined results, with a lot of variations between samples. Table II shows the ratio of transformations tried compared to *fit-max* per fitness function. In this aspect the *s-fit* group was the best, with overall very similar results among them. Following them are the two *o-fit* functions, with similar averages, but higher deviations and significantly higher maximums. Finally *fit-cfdf* had a slightly higher average, but low deviation numbers and a better maximum than the *o-fit* functions.

In general, functions that have much worse end results are also the ones with significantly more transformations tried. This also holds for *fit-cfdf*, having somewhat higher averages than the functions in the *so-fit* group. The reason for this behaviour is that a successful transformation will reduce the size of the program, and in turn reduce the number of places where the following transformations can be tested. This means that, in a general case, a less successful transformations process naturally leads to longer search times.

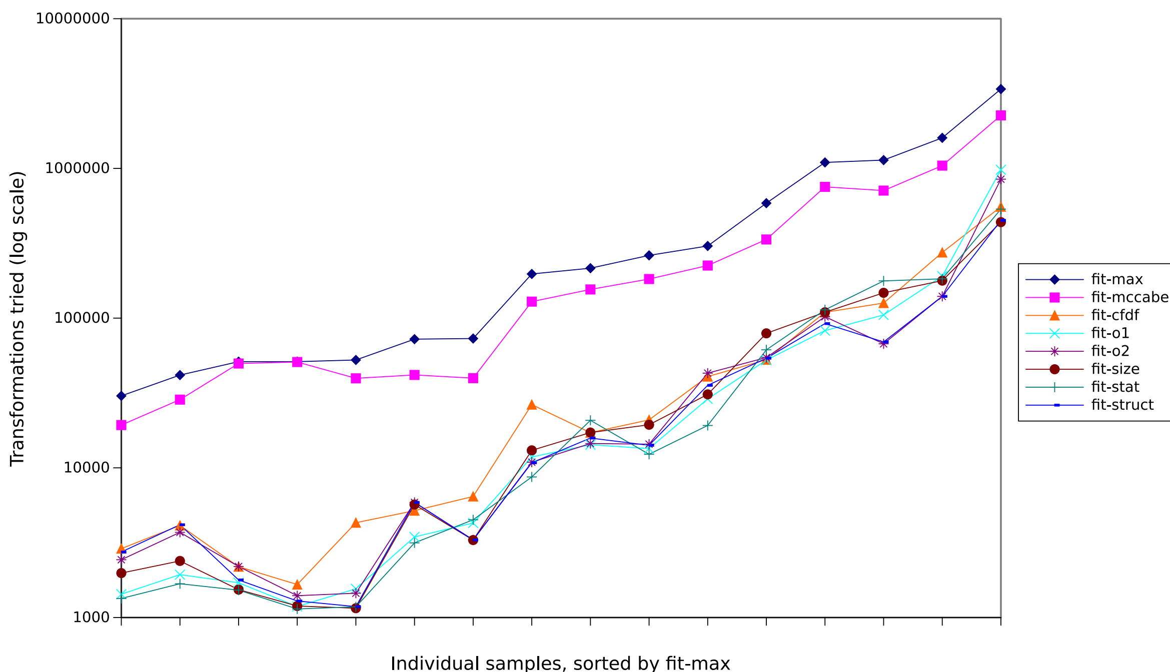

Fig. 1. Number of transformations tried, per fitness function, on *alpha-mj*

In the comparison of the number of transformations that were applied during the process *fit-max* and *fit-mccabe* will not be included, since one results in no changes, and the other shows very little

Table II. Ratio of transformations tried compared to *fit-max*

	fit-mccabe	fit-cfdf	fit-o1	fit-o2	fit-size	fit-stat	fit-struct
avg	69.86	9.84	7.65	8.10	7.80	7.20	7.22
stdevp	12.17	3.68	6.05	5.19	3.62	4.31	3.08
min	54.36	3.25	2.32	2.73	2.20	2.23	2.25
max	99.45	17.15	28.90	25.00	13.50	15.77	13.27

All numbers are percentage of fit-max

improvements compared to the others. Figure 2 shows the variations of the fitness functions using a log scale. The lowest number of applied transformations was almost always with *fit-cfdf*, but this fitness function also gave slightly worse end results. Functions *fit-o1* and *fit-o2* mostly had the highest values, on average over 60% more transformations. On average *fit-struct* had about 30% more transformations than the minimum, but would sometimes match the *o-fit* group. Function *fit-size* was mostly 10% above the minimum. Whereas, *fit-stat* was sometimes the minimum, and on average just 4% above it. The conclusion is therefore that using the number of statements as a guide will, for most cases, lead to a process with the lowest number of applied transformations for the same end result.

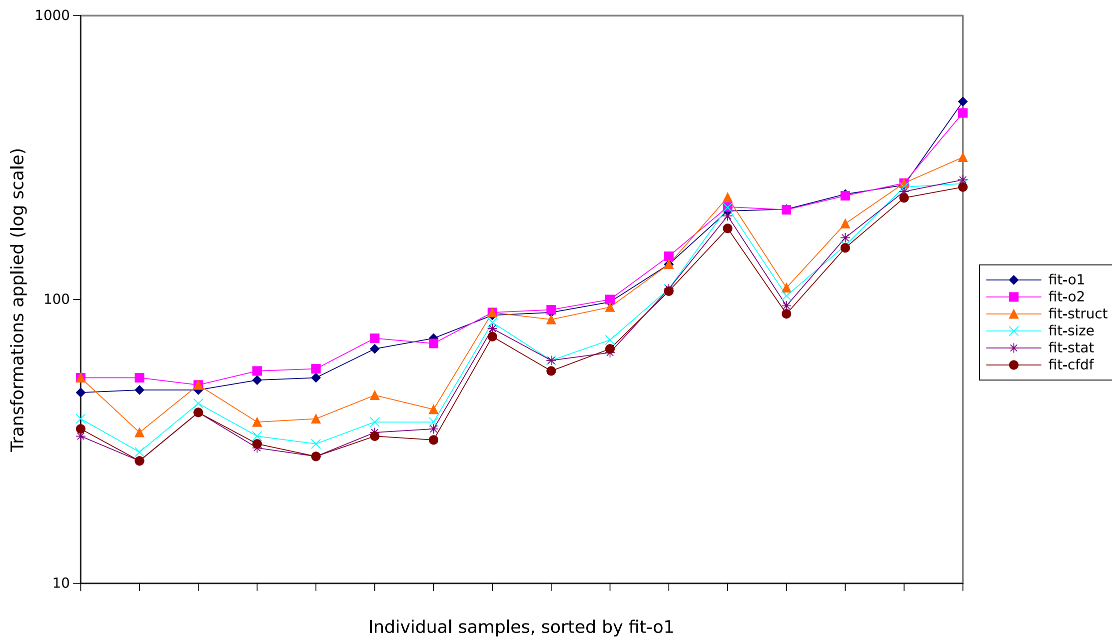

Fig. 2. Number of transformations applied, per fitness function, on *alpha-mj*

The percentage of transformations that were applied from all of those that were tested is in most cases around 1%, but can be as low as 0.05%, or almost 5% in some cases (excluding *fit-mccabe*, which was on average 0.0019%). Again the groupings are similar as in previous considerations. Function *fit-cfdf* is a bit lower than others (around 0.5%, 1.63 at most), the *o-fit* group is around 1.5%, with the highest values being almost 5%, while *s-fit* are around 1% and maximums of around 3%.

5. CONCLUSIONS AND FUTURE WORK

FermaT and WSL can be successfully used for code transformation from the low-level to a higher level of abstraction. In this approach, the entire process is automated by using a hill climbing algorithm which relies on a fitness function. The fitness function is a means which is used to evaluate the results of the applied transformation and help deduce whether applying a transformation leads to program improvement. The process tries to improve the input program as long is it possible, that is, as long as it can generate better versions of it by applying transformations. In this research, different fitness functions were used, from the obvious candidates – built-in metrics for WSL, to a few more complex ones, which consist of different combinations of the simple metrics.

This paper presents an analysis of these alternative fitness functions that can be used in the process and how they influence the end results, and especially how they change number of transformations tried and the number of transformations that were applied in the process. They were all run on a set of MicroJava programs called *alpha-mj*. A special fitness function, *fit-max*, which never advances the process was used to estimate the maximum number of transformations tried.

In terms of metrics, the fitness functions in the *so-fit* group resulted in very little differences, and generally gave the same end results. The general conclusion is that functions that have relatively high numbers which are more prone to change are better as a fitness function (as already analyzed in [Sukur and Pracner 2018]).

When comparing the number of transformations tried, there is a strong trend that better end results actually lead to fewer transformations tried. This is an inherent property of the process itself – the "better" programs are in general shorter, and therefore have less possible places for transformations to be applied at, which reduces the search space.

The number of applied transformations in the process was almost always lowest with *fit-cfdf*, but this function gave slightly worse end results. From the *so-fit* group, *fit-stat* had the lowest average number of transformations applied. This means that it tends to make larger steps forwards with individual transformations, but it is still inconclusive whether this is inherent to this fitness function, or is it a combination of the order of transformations tried and the sample set that was used. The number of applied transformations in proportion to the number of transformations tried was in general around 1% for all fitness functions (except *fit-mccabe*, which was much lower).

Overall, these results give more insights into the behaviour of the process, and how it might be improved, both in terms of end results, but also in terms of its length. It is not likely that there is an universal best fitness function that will lead to best results on any input program, as this is a case of the *no free lunch* theorem [Wolpert and Macready 1997]. However, recommendations can be made about these specific types of low-level programs, and there are trends to be observed.

There is a lot of room for improvements and additional analyses. The functions in the *so-fit* group have on average a very similar number of transformations tried, just like their end results are very similar. However, there is some per-sample variation that should be inspected in more detail, which could lead to more insights into the process and how it can be improved.

The knowledge of which fitness function leads to the lowest number of applied transformations for similar end results could be very interesting for general recommendations on what transformations should be applied both in manual and in automated scenarios.

There is a need for further experiments with more complex fitness functions. This paper used only two of these, one with a large number of parameters checked, and another that was a simple expansion of one of the basic ones. These initial results should be expanded with a larger set of functions created in a systematic approach. A deeper analysis of individual successful end results should also be used when combining the metrics into more complex ones.

The main drawback of the hill climbing algorithm is its tendency towards local optimums, which can hinder the process. Future versions could try to use one of the classical solutions, in which the process is started from multiple points and the final results are compared. It could also be replaced altogether with an algorithm which offers better results.

REFERENCES

Andrea Arcuri and Xin Yao. 2008. A novel co-evolutionary approach to automatic software bug fixing. In *Evolutionary Computation, 2008. CEC 2008.(IEEE World Congress on Computational Intelligence). IEEE Congress on*. IEEE, 162–168.

John Brant, Jason Lecerf, Thierry Goubier, and Stéphane Ducasse. 2017. Smacc: a Compiler-Compiler. http://books.pharo.org/booklet-Smacc/ The Pharo Booklet Collection.

John Brant and Don Roberts. 2009. The SmaCC Transformation Engine: How to Convert Your Entire Code Base into a Different Programming Language. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. ACM, New York, NY, USA, 809–810. DOI:http://dx.doi.org/10.1145/1639950.1640026

Eduardo Faria de Souza, Claire Le Goues, and Celso Gonçalves Camilo-Junior. 2018. A Novel Fitness Function for Automated Program Repair Based on Source Code Checkpoints. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '18)*. ACM, New York, NY, USA, 1443–1450. DOI:http://dx.doi.org/10.1145/3205455.3205566

Stéphane Ducasse, Michele Lanza, and Sander Tichelaar. 2000. Moose: an extensible language-independent environment for reengineering object-oriented systems. In *Proceedings of the Second International Symposium on Constructing Software Engineering Tools (CoSET 2000)*, Vol. 4. 24–30.

Ethan Fast, Claire Le Goues, Stephanie Forrest, and Westley Weimer. 2010. Designing better fitness functions for automated program repair. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*. ACM, 965–972.

Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. 2009. A genetic programming approach to automated software repair. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*. ACM, 947–954.

Mark Hills and Paul Klint. 2014. PHP air: Analyzing PHP systems with rascal. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*. IEEE, 454–457.

ISOZ 2002. Information Technology—Z Formal Specification Notation—Syntax, Type System and Semantics. (2002). https://www.iso.org/standard/21573.html International Standard 13568:2002.

Paul Klint, Tijs van der Storm, and Jurgen Vinju. 2011. EASY Meta-programming with Rascal. In *Proceedings of the 3rd International Summer School Conference on Generative and Transformational Techniques in Software Engineering III (GTTSE'09)*. Springer-Verlag, 222–289.

Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering* 38, 1 (2012), 54.

Hanspeter Mössenböck. 2018. Compiler Construction. (2018). Handouts for the course, available at http://ssw.jku.at/Misc/CC/.

Doni Pracner. 2019. *Translation and Transformation of Low Level Programs (Prevođenje i transformisanje programa niskog nivoa)*. Ph.D. Dissertation. Faculty of Sciences, University of Novi Sad, Serbia.

Doni Pracner and Zoran Budimac. 2011. Understanding Old Assembly Code Using WSL. In *Proc. of the 14th International Multiconference on Information Society (IS 2011)* (2011-10), Marko Bohanec, Matjaž Gams, Dunja Mladenić, Marko Grobelnik, Marjan Heričko, Urban Kordeš, Olga Markič, Jadran Lenarčič, Leon Žlajpah, Andrej Gams, Vladimir A. Fomichov, Olga S. Fomichova, Andrej Brodnik, Rok Sosič, Vladislav Rajkovič, Tanja Urbančič, and Mojca Bernik (Eds.), Vol. A. Institut "Jožef Stefan", Ljubljana, Ljubljana, Slovenia, 171–174. http://is.ijs.si

Doni Pracner and Zoran Budimac. 2017. Enabling code transformations with FermaT on simplified bytecode. *Journal of Software: Evolution and Process* 29, 5 (2017), e1857–n/a. DOI:http://dx.doi.org/10.1002/smr.1857

Stuart J Russell and Peter Norvig. 2016. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,.

Eric Schulte, Stephanie Forrest, and Westley Weimer. 2010. Automated program repair through the evolution of assembly code. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM, 313–316.

Nataša Sukur and Doni Pracner. 2018. Evaluating Fitness Functions for Automated Code Transformations. In *SQAMIA 2018, 7th Workshop of Software Quality, Analysis, Monitoring, Improvement, and Applications*, Zoran Budimac (Ed.). Department of Mathematics and Informatics, Faculty of Sciences, University of Novi Sad, Serbia, 18:01–18:08. http://ceur-ws.org/Vol-2217/

Martin Ward. 1989. *Proving Program Refinements and Transformations*. Ph.D. Dissertation. Oxford University.

Martin Ward. 1999. Assembler to C Migration using the FermaT Transformation System. In *IEEE International Conference on Software Maintenance (ICSM'99)*. IEEE Computer Society Press, 67–76.

Martin Ward. 2004. Pigs from Sausages? Reengineering from Assembler to C via FermaT Transformations. *Science of Computer Programming, Special Issue on Program Transformation* 52/1-3 (2004), 213–255. DOI:http://dx.doi.org/10.1016/j.scico.2004.03.007

Martin Ward. 2013. Assembler restructuring in FermaT. In *SCAM*. IEEE, 147–156. DOI:http://dx.doi.org/10.1109/SCAM.2013.6648196

Martin Ward, Hussein Zedan, and Tim Hardcastle. 2004. Legacy Assembler Reengineering and Migration. In *ICSM2004, The 20th IEEE International Conference on Software Maintenance*. IEEE Computer Society.

David H. Wolpert and William G. Macready. 1997. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation* 1, 1 (April 1997), 67–82. DOI:http://dx.doi.org/10.1109/4235.585893

Hongji Yang and Martin Ward. 2003. *Succesful Evolution of Software Systems*. Artech House, Norwood, MA, USA.