Transforming Low-Level Variants of Greatest Common Divisor Algorithm: A Case Study

Doni Pracner[1,*], Nataša Sukur[1,*]

[1]University of Novi Sad, Faculty of Sciences, Department of Mathematics and Informatics, Trg Dositeja Obradovića 4, 21000 Novi Sad, Serbia

Abstract

A large portion of maintenance effort is spent in understanding the logic of the original code. Our research presents an option to automatically transform a low-level program into a semantically equivalent version with high-level structures. Making the code more understandable with high-level structures can lead to faster and easier maintenance. This paper presents a case study on several implementations of the greatest common divisor in two low-level languages (assembly and MicroJava bytecode) and their transformations.

Keywords

Automated Maintenance, Hill Climbing, Fitness Function, Greatest Common Divisor

1. Introduction

One of the most significant properties of software is its need to change due to new requirements or changes of the environment. By constantly changing a software product, the need for maintenance becomes more and more important, but at the same time more and more challenging. However, the maintainers are often left with low-level code or binaries as the only source of information about the program. Trying to understand the code and maintain it according to the needs can be a very costly process in the sense of necessary time and effort. This process often also relies on the experience and skills of the maintainer, which opens possibilities for additional errors. New developers are generally only trained in high-level languages, emphasising the problem. One solution for this is automation of the entire process by using helper tools which would make the original software more readable without causing new faults (of course given that the tools themselves are not faulty).

The research presented in this paper is based on FermaT transformation system and WSL (Wide Spectrum Language) [1], strongly based around the idea of software maintenance. In this paper, translation and semantics preserving transformations are applied to the input low-level code in an automated manner. The goal of this automated process in general is to derive higher level structures from the low-level code.

SQAMIA 2022: Workshop on Software Quality, Analysis, Monitoring, Improvement, and Applications, September 11–14, 2022, Novi Sad, Serbia

*Corresponding author.

doni.pracner@dmi.uns.ac.rs (D. Pracner); natasa.sukur@dmi.uns.ac.rs (N. Sukur)

https://perun.pmf.uns.ac.rs/pracner/ (D. Pracner)

0000-0002-3428-3470 (D. Pracner); 0000-0003-4701-9289 (N. Sukur)

This paper presents a case study on several implementations of the Euclidean algorithm (greatest common divisor), both in MicroJava bytecode and assembly. Being one of the first algorithms, it seems suitable to be one of the initial case studies of this research. The same automated transformation process has been applied to each of these samples and the results of the experiments have been presented.

The rest of the paper is organised as follows: Section 2 presents the case study of this paper, the greatest common divisor algorithm; in Section 3 the foundations of this work and the automated transformation process are explained; in Section 4 the related work is presented; Section 5 introduces the experiments and the results of this research and finally, in Section 6 we present the conclusions and the future work based on the observations made on current results.

2. Greatest Common Divisor Algorithm

The greatest common divisor algorithm (further referred to as *GCD*), or Euclidean algorithm is considered to be one of the oldest nontrivial algorithms recorded. Its first written occurrence was in Euclid's *Elements* and it explains the calculation of the greatest common divisor of two numbers [2][3].

The easiest and most natural approach to finding the GCD of two numbers would be to factor both numbers and then multiply the common factors, raised to according powers. However, this is far from the most efficient way of calculating the GCD and the Euclidean algorithm solves it in a more efficient manner [4].

The original Euclidean algorithm is based on continuous subtraction. If `A` and `B` are two positive integers, their greatest common divisor can be found by continuously subtracting smaller from the larger value until they are equal. After the process is finished, any of the two numbers can be returned as the result.

The algorithm can also be represented as repeated calculation of the remainder of one number divided by another. The result is reached once the remainder is `0`, and the repeated subtraction of the original algorithm can simply be replaced by remainder calculation in this improved version.

3. Automated Transformations

Current demands for software development are that it is delivered in the shortest time possible, yet still with high level of quality and reliability. In order to preserve the reliability of the software product, it is important to find means which would assure reliability throughout different stages of the development process. As a solution to this problem, formal methods can be used. Formal methods can be very useful for the overall reliability of software, since they can be applied in different stages of software development and to different software artefacts [5].

WSL is a language used for reverse engineering of sequential systems [1]. It is based on formal methods and together with FermaT program transformation system, it makes a platform for successful reverse and forward engineering. The idea of program transformations is some sort of cost reduction, for example in terms of performance, memory usage or portability. FermaT's main feature is the support for transformations which are formal and semantics preserving [6].

WSL stands for *Wide Spectrum Language*, which describes its wide usage from abstract specifications to concrete runnable implementations. Apart from standard language constructs, WSL has another important extension, *MetaWSL*, used for writing code transformations in the language itself. Transformations are included in the system and their correctness can be automatically checked. Transformations can be of great use in processes such as creating programs from specifications, analysing programs and their properties or getting specifications from reverse engineering of programs.

Action systems are a special structure in WSL. It was designed to be able to emulate jumps and goto statements that are common to low-level programs. It consists of a collection of parameter-less procedures. One of them is marked as the start procedure and they can call each other with no limits. Once a procedure is finished, the control is returned to the call site. The only exception to this is the reserved name Z, a call to which immediately ends the execution of the whole action system. There can be multiple action systems in a program, and the flow of the program normally continues with the next statement when an action system terminates. There are three types of systems in relation to how the calls are used. The first one are *recursive* action systems, in which all of the calls return, and the system is terminated when the starting action terminates. The second one are the *regular* action systems in which the system is always terminated by a call to Z and none of the calls return to their sites. This type is important for many transformations, since this special property can be used for simplifications. The third type are the *hybrid* ones, which are a mix of the previous two, or, in other words, any that don't fall into the previous categories.

Two tools were developed by the authors for translation of the original low-level code to WSL – *asm2wsl*, which translates a subset of x86 assembly in the MASM/TASM dialect [7] and *mjc2wsl*, which translates MicroJava bytecode [8]. Both of these tools replicate the operational semantics of the original processor/virtual machine. This is achieved through a set of additional variables that represent the state of the machine, such as registers, flags and stacks. The first tool is a bit limited when it comes to the span of assembly programs it can handle. For instance there is an assumption that only labels are jump targets. It also recognises some special macro names for inputs and outputs, instead of dealing with interrupts directly. The WSL translation consists of a *regular* action system in which actions mostly start where the labels were, and the final statement is a call to Z. On the other hand, *mjc2wsl* translates practically any valid program for the MicroJava virtual machine. It generates a *recursive* action system, in which every original operation has an action associated with its address. This enables more flexible jumps.

Transformations can be applied manually, by relying on one's experience and expertise or automatically, by using an existing helper tool [9]. This tool automates the process, starting from the translated original low-level code and resulting in higher level code, improved in terms of its structure and understandability. This automation tool is based around the *hill climbing* algorithm.

Hill climbing [10] is a search algorithm. It gets its name due to its approach and logic – it tries to move in the direction of increasing value, that is, towards better solutions. That is accomplished by starting from an arbitrary solution and making changes to it in order to reach a better one. This way, the process constantly moves "uphill", with the best solution being the hilltop. Finding the best solution means that no immediate neighbour has a better value.

This also means that the algorithm has an implicit problem of ending up in local optimums. In this use case, the algorithm tries to apply transformations to a program and if the program is improved based on some criteria, it is kept as a base for further transformation. The process applies transformations to the code as long as they are leading to better programs. The end of the process means that the "best" program has been reached and that application of additional transformations will not result in a better one.

After a transformation has been applied in the hill climbing process, the quality of the potential solution is measured by a *fitness function*. The term draws its origin from evolutionary computing. Generally speaking, the role of a fitness function is to define qualities which should be met by a population. It defines the criteria for selection, which also means that it defines the meaning of improvement. In terms of problem-solving, it represents the task which is solved in evolutionary manner [11].

The fitness function indicates if the newly created solution shows an improvement, which from the computing point of view means that it has better qualities for solving the problem at hand. For this purpose, a simpler and more understandable program is the better one. Since the fitness function represents the quality of a program as numeric values, the first choice are various metric values. These metrics can be good indicators whether a solution is better structured and therefore more understandable. In our previous work [12], it was shown that selecting different fitness functions on the same input programs leads to different end results. One of the important observations of this work was that the the best results were not accomplished by a single fitness function across all input programs. Based on that, the conclusion was that the selection of the fitness function should probably vary based on the properties of the samples. The experiments were run with *structure* metric by default, which is a custom weighted metric for WSL, but different fitness functions were also used.

Both of the translation tools, as well as the transformation program are available under the terms of the GNU Public Licence on the project site[1].

4. Related Work

Although using hill climbing and fitness functions for automated reengineering is not so widespread, there have been some tools and papers which share some of the ideas and objectives. Hill climbing has not been frequently used for automatic program repair, however this approach has been considered by Arcuri and Yao in [13]. There has not been a lot of confidence in this idea, since the algorithm shows tendency to gravitate towards local optimums. Hill climbing was also compared to different approaches for solving similar problems and as a result, genetic algorithms showed as better [14]. However, this approach is different than the one described in this paper, since the order of transformations was being optimised in search for a better solution.

Fitness functions have shown as useful in automated software repair for C programs [15] as well as assembly [16]. Later on, that research resulted with the possibility to be applied to any kind of code [17]. There has also been research which further confirms our hypothesis that the selection of the fitness function should vary based on the problem which it solves. The topic of

[1] https://perun.pmf.uns.ac.rs/pracner/transformations

that research is automated bug detection [18, 19] and tries to give guidance how to create them and get the best results.

FermaT has previously been successfully used in several industrial projects, where the original assembly code was transformed to C and COBOL code. However, this process was significantly adapted to the specific problems it was solving, unlike the hill climbing process which is always the same and fully automated [1].

5. Transforming GCD Programs

5.1. Manual Application of Transformations

Using FermaT (possibly with a helper tool like FME), an experienced maintainer can raise the level of abstraction of low-level code. An example can be a direct implementation of subtracting the smaller number from the larger until they are the same in assembly, shown in Figure 1. It features macros with specific names that get recognised by the translator, as mentioned before. When this code is translated into WSL using *asm2wsl* it will expand into about 50 statements, which is an expected increase. A rather optimal path could be to apply the following transformations: *Remove Redundant Vars*, *Flag Removal*, *Collapse Action System*, *Floop to while*, *Constant Propagation*. The end result is a very clear implementation of Euclid's algorithm in only 5 statements, shown in Figure 2. Note that the pretty printer for WSL is not in a C-like style, but leans much more toward a LISP-like indention, where the closing of a branching or a loop will be at the end of the block, on the same line, not in a new line on the same level as the opening.

```
start:
        mov   ax,12
        mov   bx,8
compare:
        cmp   ax,bx
        je    exit ;exit since they're equal
        ja    greater
        sub   bx,ax
        jmp   compare
greater:
        sub   ax,bx
        jmp   compare
exit:
;exit out of the program
        print_num ax
        end_execution
```

Figure 1: Assembly code for gcd-predef

Of course, since this is a program with hard coded starting values, the result is also always the same. Further simplification if possible by using *Unroll loop* and *Constant propagation* transformations repeatedly to just assign the final value 4 to ax and bx, and then even further into a simple PRINFLUSH(4).

Reaching this optimal path requires a lot of expertise from the maintainer and although very

```
VAR < ax := 12, bx := 8 >:
WHILE ax <> bx DO
  IF ax >= bx THEN ax := ax - bx ELSE bx := bx - ax FI
OD;
PRINFLUSH(ax) ENDVAR
```

Figure 2: gcd-predef transformed

good final results could be achieved by manual selection and application of transformations, it is usually a much better option to use automated tools for these kinds of activities. Although the automated process might not reach the best solution using the same path as an expert would, the manual approach is in general less efficient by far in terms of the cost of the process.

5.2. Automated Transformation Selection

There are several variants of the algorithm implementation that were used in these experiments. Some are written in assembly, and some in MicroJava and then compiled into bytecode. Many of these programs come in pairs, a variant with predefined values (i.e., hard coded) and a variant with user input for ax and bx. These are then usually marked with suffixes predef and input, respectively.

The first variant that will be presented is the one that was already discussed for a manual approach to transformations (Figure 1). The hill climbing process will automatically transform the program into the same version as already shown in Figure 2. It will do so by automatically finding and applying 27 transformations. This is a lot more than with the manual approach, but those were hand picked and targeted very precisely, and some of them are very complex, while here there is a lot of smaller and cheaper transformations for the same end result. During the process about 2300 transformations are tried, which took less than half a second on a 3GHz Intel processor.

The process that ran with the default *structure* metric as the fitness function did not simplify the program further into a simple print, which was shown in the manual transformation example. This is due to the fact that unrolling the loop increases the metric of the new program, and it is therefore discarded by the process. In this case it would be required to look ahead 4 transformations to achieve an improvement of the fitness of the program. The problem, of course, is that in a general case it is undecidable to know how many steps are required, or indeed if there is a finite number of steps that would lead to an improvement, since this is a variation of the *halting problem* [20].

Another variant of this program is with user inputs, instead of hard coded values. Macros are used for input operations, to avoid the specifics of interupts. The increase in the number of statements in the translated program is about ten. The transformed version of the program with inputs is shown in Figure 3. There are somewhat verbose parts that load the user input, but the core part of the algorithm is exactly the same as in the predef version. The process selected 34 transformations to achieve this version, which is inline with the slight increase of statements. On the other hand it tried almost 6000 transformations for this program, which is a

significant increase. The time elapsed also doubled to about a second. However, this is natural for the process since it has a lot more statements to try transformations on.

```
PRINFLUSH(”num?$”);
VAR < ax := 0, bx := 0, t_e_m_p := 0 >:
@Read_Line_Proc( VAR t_e_m_p, Standard_Input_Port);
bx := @String_To_Num(t_e_m_p);
PRINT(””);
PRINFLUSH(”num?$”);
@Read_Line_Proc( VAR t_e_m_p, Standard_Input_Port);
PRINT(””);
ax := @String_To_Num(t_e_m_p);
WHILE ax <> bx DO
  IF ax > bx THEN ax := ax - bx ELSE bx := bx - ax FI
OD;
PRINFLUSH(<”res,$”>);
PRINFLUSH(ax) ENDVAR
```

Figure 3: Assembly program gcd-input transformed

The next experiment was to wrap the algorithm into a procedure, with a single call to it with predefined values. During the process the procedure gets inlined into the main program, the end result was the same as with gcd-predef. The increase in the number of transformations was low for this variant.

Another variant is a recursive one, with a procedure that will keep subtracting the smaller from the larger number and call itself until the numbers are the same. A difference to the version seen in gcd-predef is that this variant always ensures that ax is the larger number by swapping them when needed, which means that it always subtracts bx from ax. When this version is automatically transformed, the recursion is reduced to a while loop, as shown in Figure 4. WSL has a simultaneous assignment environment that can be seen in the Figure. Again, as with the previous examples, both the predef and input versions get transformed to the same core loop, and, again, the number of transformations tried with the second version is more than double.

```
WHILE ax <> bx DO
  IF ax <= bx
    THEN < ax := bx, bx := ax > FI;
  ax := ax - bx OD;
```

Figure 4: Assembly program rek-gcd-input transformed, core loop

There are also several versions of the algorithm implemented in MicroJava, again with pairs of predefined values and user inputs.

One version is the same as seen in assembly – a while loop with subtraction of the smaller number from the larger number. This high-level code is then compiled into bytecode, which is translated into WSL. Unlike *asm2wsl* which created *regular* action systems, *mjc2wsl* creates

recursive action systems, in which every call returns. It is then transformed with the same hill climbing program. The automatically transformed version (Figure 5) is again a very clear high-level loop at the core part of the program, although it does leave a few additional stack operations. The names of the local variables are not encoded in the compiled files, so they get automated sequential names. These can mostly be simplified with additional manual transformations. The main reason they are left in the program is that sometimes the introduction of a VAR block can be a greater increase in the metrics than the removal of the stack operations.

Another version is the same as this one, just contained in a procedure. It gets transformed to the same core code as in Figure 5.

```
WHILE mjvm_locals_0 <> mjvm_locals_1 DO
  IF mjvm_locals_0 > mjvm_locals_1
    THEN mjvm_locals_0 := mjvm_locals_0 - mjvm_locals_1 FI;
  IF mjvm_locals_0 < mjvm_locals_1
    THEN mjvm_locals_1 := mjvm_locals_1 - mjvm_locals_0 FI
OD;
```

Figure 5: MicroJava gcd-input transformed, core loop

The alternative version of this program, with predefined inputs, can actually be transformed into a single PRINFLUSH statement – that is the end result is correctly deduced in the process. This is dependant on the starting predefined values. The current implementation of the process is able to unroll the loop if there was only one pass through it, for instance for the starting values of 12 and 8. Already for two passes the metrics do not drop after an unroll, for instance for 20 and 8, and the result will be the same core code as with the input version.

This is different to the assembler programs since the implementation used two IF branches, instead of the IF/ELSE that is inherent to the assembly version. This difference is enough for the metrics to change when the loop is unrolled. An additional MicroJava program with this other type of branching was tested, and it didn't get fully simplified either. It is arguable which of these is "better" high-level code; the first one is clear that there is no middle option and is explicit about what it does, like a specification should be. The other one is more optimal in execution since it will not do an additional check.

Next is a recursive version, shown in Figure 6, where the values are swapped if needed to ensure a is larger than b. This is the same logic as the recursive assembly program. However, unlike that version, which was transformed into a WHILE loop (Figure 7), here the process resulted in a different implementation of the same algorithm, with two recursive calls in which the parameters get subtracted as needed. It was not transformed into a function with a return value, since the process removed the in between stack operations and the result is only once put on the stack and only taken from the stack in the main program.

Other recursive version of the algorithm were also tried. Sadly, they do not get properly transformed to fully high level code. One problem is that the procedure does not get recognised as having two parameters, only one of them is transformed, while the other is still passed through the stack. We believe this is due to the intertwined stack operations that are sometimes prematurely optimised in the process and are then hard to fit into the expected patterns. It is

```
int gcd(int a, int b)
int t;
{
        if (a == b)
                return a;
        if (a<b) {
                t = a;
                a = b;
                b = t;
        }
        return gcd(a-b,b);
}

void main()
{
        print(gcd(8,12),3);
}
```

Figure 6: rek-gcd-swap MicroJava program

```
VAR < mjvm_estack := < >, mjvm_mstack := < > >:
BEGIN
  a14(12, 8);
  VAR < tempa := 3, tempb := 0 >:
  POP(tempb, mjvm_estack);
  PRINFLUSH(@Format(tempa, tempb)) ENDVAR
WHERE
  PROC a14(par1, par2) ==
    IF par1 > par2
      THEN a14(par2, par1 - par2)
    ELSIF par1 <> par2
      THEN a14(par1, par2 - par1)
      ELSE PUSH(mjvm_estack, par2) FI END
END ENDVAR
```

Figure 7: rek-gcd-swap, fully transformed

somewhat ironic that among these is a MicroJava implementation that is almost the same as the successful transformation shown in Figure 7.

5.3. Properties of The Process

Assembly programs get translated to more than 2 times longer WSL programs; the transformed high-level versions are more than 3 times shorter on average, on these samples. It should also be noted that this is with assembly programs that have the macro definitions and bodies removed. If those were included, that ratio of improvement would be even higher.

MicroJava gets compiled to about 2.5 times more statements, each of which then get translated to around 4 WSL statements. After the transformations the number of statements is mostly in line with the original high-level programs, or in other words it is capable of obtaining the same level of structures.

The length of the process in general rises with the length of the input program, but the correlation is not a direct one. There are exceptions both ways – some significantly longer programs can get transformed in the same time, and sometimes short programs will take a long time to process. Transformations with better end results are often achieved in less time. In general, if a program is successfully transformed and simplified there are less places to test future transformations on [21, 12], thus finishing faster.

6. Conclusions

One of the most important tasks in software maintenance is the understanding of the program at hand. This can especially be hard with low-level implementations. In this paper an approach to automated restructuring into a high-level version of the same program is presented. It relies on FermaT transformation system and WSL language, and its support for semantics preserving transformations. Two tools are in charge of translation of the low-level code to WSL, *asm2wsl* and *mjc2wsl*. The process is fully automated and is based around a hill climbing algorithm which advances towards the best solution with a fitness function as the measure of quality. All the tools are open source and available under the GNU Public Licence. The presented process consists of several steps and independent tools. It can be applied to a number of different maintenance tasks. Since the process is flexible it can be easily adapted or integrated into other processes.

The focus of this paper is a case study of several implementations in two different languages of the oldest written algorithm, *greatest common divisor*. Several versions got automatically transformed into practically the same high-level code. Even though some of these versions used loops and some used recursive procedures, some were regular action systems, and some recursive, and that the programs that translate them have other differences as well, they all get transformed with the same hill climbing program.

A notable transformation was made, in which one implementation of the algorithm got automatically transformed into another. The starting version was a MicroJava recursive implementation that swaps the values of the parameters when needed. The result was an implementation that has two calls and changes which parameters are passed into them (Figure 7).

On the other hand, some other recursive versions of MicroJava programs were not as much of a success. They do get transformed into a higher level and are easier to understand but there are still some low-level stack operations that remained. A maintainer who is somewhat familiar with the concepts of virtual machines that rely on stacks should be able to fully understand these.

Of course, there is a lot of space for expanding this work. More in-depth studies of different algorithms could reveal more about the properties of this process and how it can be improved. For instance more effort could be put into improving the transformations of stack operation to parameters and return values to recognise some of the already optimised versions. The order

of the transformations could also be changed to check the possibilities of obtaining better end results.

In general, a big question is how to formulate a fitness function that will be as universal as possible to define a program as being more *understandable*. In our experiments, on different inputs, different functions would give better results. Since this is an optimisation problem, the "no free lunch" theorem [22] is applicable – for the set of all possible input programs, there are always examples for which a fitness function would give worse results. On the other hand, there are some common properties of the input programs that were translated from low-level sources, so there is hope to at least find good candidates. A lot of what was learned from this can be invested into further experiments in combining the functions, or even combining multiple processes.

One of the known issues of the hill climbing algorithm is the tendency towards local optimums. This can be somewhat countered with starts from multiple random points and picking the best results from all the runs. Another option that is considered for future processes is to use a different searching algorithm.

Acknowledgments

The authors acknowledge financial support of the Ministry of Education, Science and Technological Development of the Republic of Serbia (Grant No. 451-03-68/2022-14/200125).

References

[1] M. Ward, Assembler restructuring in fermat, in: SCAM, IEEE, 2013, pp. 147–156. doi:http://dx.doi.org/10.1109/SCAM.2013.6648196.

[2] Euclid, Elements, Book VII, –, 300 BC.

[3] D. E. Knuth, The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[4] H. Cohen, A Course in Computational Algebraic Number Theory, Springer-Verlag, Berlin, Heidelberg, 1993.

[5] M. Roggenbach, A. Cerone, B.-H. Schlingloff, G. Schneider, S. A. Shaikh, Formal Methods for Software Engineering: Languages, Methods, Application Domains, Texts in Theoretical Computer Science. An EATCS Series, Springer Cham, 2022.

[6] M. Ward, Proving Program Refinements and Transformations, Ph.D. thesis, Oxford University, 1989.

[7] D. Pracner, Z. Budimac, Restructuring assembly code using formal transformations, in: T. E. Simos (Ed.), Proc. of Symposium on Computer Languages, Implementations and Tools (SCLIT 2011) held within ICNAAM 2011, volume 1389 of *AIP proceedings*, 2011, pp. 845–848.

[8] D. Pracner, Z. Budimac, Transforming low-level languages using FermaT and WSL, in: Z. Budimac (Ed.), Proceedings of the 2nd Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications, volume 1053 of *CEUR-WS.org*, 2013, pp. 71–78. URL: http://ceur-ws.org/Vol-1053/.

[9] D. Pracner, Z. Budimac, Enabling code transformations with FermaT on simplified bytecode, Journal of Software: Evolution and Process 29 (2017) e1857–n/a. URL: http://dx.doi.org/10.1002/smr.1857. doi:10.1002/smr.1857.

[10] S. J. Russell, P. Norvig, Artificial intelligence: a modern approach, Malaysia; Pearson Education Limited,, 2016.

[11] A. E. Eiben, J. E. Smith, et al., Introduction to evolutionary computing, volume 53, Springer, 2003.

[12] N. Sukur, D. Pracner, Evaluating fitness functions for automated code transformations, in: Z. Budimac (Ed.), SQAMIA 2018, 7th Workshop of Software Quality, Analysis, Monitoring, Improvement, and Applications, Faculty of Sciences, University of Novi Sad, Serbia, 2018, pp. 18:01–18:08. URL: http://ceur-ws.org/Vol-2217/.

[13] A. Arcuri, X. Yao, 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, 2008, pp. 162–168.

[14] D. Fatiregun, M. Harman, R. M. Hierons, Evolving transformation sequences using genetic algorithms, in: Source Code Analysis and Manipulation, Fourth IEEE International Workshop on, 2004, pp. 65–74. doi:10.1109/SCAM.2004.11.

[15] S. Forrest, T. Nguyen, W. Weimer, C. Le Goues, A genetic programming approach to automated software repair, in: Proceedings of the 11th Annual conference on Genetic and evolutionary computation, ACM, 2009, pp. 947–954.

[16] E. M. Schulte, S. Forrest, W. Weimer, Automated program repair through the evolution of assembly code, in: C. Pecheur, J. Andrews, E. D. Nitto (Eds.), Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE '10), ACM, 2010, pp. 313–316. doi:10.1145/1858996.1859059.

[17] C. Le Goues, T. Nguyen, S. Forrest, W. Weimer, Genprog: A generic method for automatic software repair, IEEE Trans. Software Eng. 38 (2012) 54–72. doi:10.1109/TSE.2011.104.

[18] E. Fast, C. Le Goues, S. Forrest, W. Weimer, Designing better fitness functions for automated program repair, in: Proceedings of the 12th annual conference on Genetic and evolutionary computation, ACM, 2010, pp. 965–972.

[19] E. F. de Souza, C. L. Goues, C. G. Camilo-Junior, 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, 2018, pp. 1443–1450. doi:10.1145/3205455.3205566.

[20] A. M. Turing, On computable numbers, with an application to the entscheidungsproblem, Proceedings of the London Mathematical Society s2-42 (1937) 230–265. doi:10.1112/plms/s2-42.1.230.

[21] D. Pracner, Translation and Transformation of Low Level Programs (Prevođenje i transformisanje programa niskog nivoa), Ph.D. thesis, Faculty of Sciences, University of Novi Sad, Serbia, 2019.

[22] D. H. Wolpert, W. G. Macready, No free lunch theorems for optimization, IEEE Transactions on Evolutionary Computation 1 (1997) 67–82. doi:10.1109/4235.585893.