IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 12 March 2024, accepted 8 April 2024, date of publication 15 April 2024, date of current version 26 April 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3389500

RESEARCH ARTICLE

Climbing the Hill to Understand the Code

DONI PRACNER[1], MARTIN P. WARD[2], NATAŠA SUKUR[1], AND ZORAN BUDIMAC[3]
[1]Faculty of Sciences, Department of Mathematics and Informatics, University of Novi Sad, 21000 Novi Sad, Serbia
[2]Software Migrations Ltd., St. Albans, AL3 4DH Hertfordshire, U.K.
[3]Faculty of Sciences, Department of Mathematics and Informatics, University of Novi Sad, 21000 Novi Sad, Serbia *(Deceased)*

Corresponding author: Doni Pracner (doni.pracner@dmi.uns.ac.rs)

This work was supported by the University of Novi Sad, through the Ministry of Science, Technological Development and Innovation of the Republic of Serbia under Grant 451-03-66/2024-03/200125 and Grant 451-03-65/2024-03/200125.

ABSTRACT Software maintenance takes up a disproportionately large amount of time in the modern software life cycle. One of the common problems is understanding the original code that is being restructured and improved and this is especially true with low-level code. This paper investigates the results and properties of an automated process that can raise the abstraction level of code from low-level operations to high-level structures. The process is made of independent components and can be adapted to different scenarios. The automated improvements implementation relies on the program transformation system FermaT and its catalogue of semantics-preserving transformations. The process uses hill climbing and a metric for the fitness function of the programs. This component was made to work on general inputs, without explicit knowledge of the type of origin of the program. The paper explores how different inputs are actually handled by the system, what are the properties and how these can be used for further improvements. Two main types of inputs are shown, x86 assembly and MicroJava bytecode. These two have many operational differences, and the translator tools introduce some more, but nonetheless, the same process can handle all of these and, on average, improve the Structure metric (a good approximation of the complexity of the code) by around 85%.

INDEX TERMS Automated code transformation, bytecode, assembly, program comprehension.

I. INTRODUCTION

Computers are integrated in almost every aspect of our lives, and with them a huge amount of software that they run. The software that runs even on relatively simple devices can be quite complex. Since software is hard to develop, and easy to copy this in turn leads to a lot of software reuse and dependencies. Maintenance becomes harder as time goes on, and often the first hurdle to overcome is to understand what the original logic of the code is. This is especially true when working with low-level code, partly due to the inherent difficulties of the logic of such programs, but also due to fewer and fewer developers having practical experiences with such languages.

This paper analyses a previously presented approach that enables automatic restructuring and raising the abstraction level of low-level source code. A higher-level representation makes it easier to understand the underlying logic, which in turn reduces maintenance time. The process consists of several stages and mostly independent tools, designed with the concepts of flexibility and adaptability in mind (Fig. 1).

The core part of the process is an automated transformation system. It relies on a hill climbing algorithm, where progress is always made towards a better program, as decided by the fitness function used. The system uses semantics-preserving transformations, which means the logic is always maintained and there is no need for a strong test suite to verify the end results, while the process is guided towards a higher-level program. The process uses the *FermaT* transformation system [1] which operates on and is written in *WSL (Wide Spectrum Language)*. *FermaT* has been successfully used in restructuring large scale industrial programs, for instance transforming legacy assembly into maintainable C or COBOL code. The automated transformation component in this process is called *HCF*, short for hill climbing in Fermat.

The associate editor coordinating the review of this manuscript and approving it for publication was Muhammad Ali Babar.

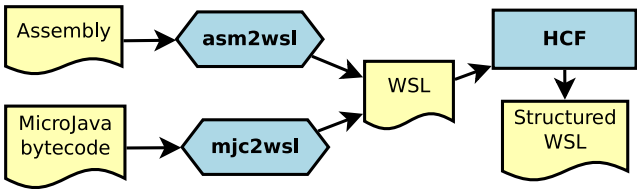


FIGURE 1. Overview of the process discussed.

Apart from the automated transformations, translators to WSL are needed for each input program type. This paper will present experiments on two main input groups, one written in x86 assembly, and the other in MicroJava bytecode [2]. The appropriate translations result in low-level programs in WSL, which will then be transformed to higher-level versions.

The core transformation system is independent from the input languages and makes no assumptions about how the input WSL was created. The end result of the presented process is, in a best case scenario, a high-level version of the program. At worst, only a part of the reengineering is done automatically, but it still results in a program that is easier to understand and it can save many hours of work on manual restructuring.

In this paper the focus is on the ability of the central part of the process (HCF) to handle different types of inputs, and to better understand the process itself. This should give us an insight of the strengths and weaknesses of the system, where it can be applied in the future, how it can be improved, and what alternatives can be used.

The rest of this paper is organised as follows. Section II presents the related work and approaches. Section III shows more details about FermaT, the system that is used in this process and some of the previous usages of the system. Section IV presents the details of the main process of this paper, the input programs, and the results achieved. It also gives the details on how to reproduce the results. Finally, Section V gives a short overview of the paper and options for future work.

II. RELATED WORK AND APPROACHES

The work of restructuring programs to high-level constructs is in many ways the same as that of decompilers. They try to reconstruct the original program that was the source for the executable code at hand. Decompilers are generally written with a very specific input language and targeted structures to be recognised. They also tend to have a fixed output language, although sometimes it is an intermediary representation and can be exported to multiple languages. The main difference in this paper is that the central steps of automated restructuring should be largely independent from the inputs (as will be shown in later sections).

Some of the earliest work in decompilation dates back to the 1960's, but most of the modern understanding of decompilers is mostly traced back to Cristina Cifuentes' PhD thesis which presented a 80286 decompiler to C [3]. One of the big problems for decompilers is the optimisations done by compilers which make it harder to reconstruct the original source code. For instance, a tool developed to produce C language code from SPARC assembly, *asm2c*, was able to reduce the size of programs by 66% on non-optimised assembly and by about 5% on optimised versions [4]. An important aspect of the end results is the correctness, since some abstractions can be wrongly interpreted. *Phoenix*, a GNU C decompiler, was proven on a set of programs from *coreutils* to outperform the *de facto* industry standard tool *IDA Hex-Rays*[1] [5]. Phoenix is based on the CMU Binary Analysis Platform (BAP), which translates assembly into an intermediate language to help program verification [6]. Large efforts are spent on removing jumps in code. *Dream* was written as an IDA plugin, with the goal of producing GOTO-free output, and made more compact results on the GNU coreutils utilities than both Phoenix and Hex-Rays [7].

Decompiling machine code generated for virtual machines, such as the bytecode generated by Java compilers, provides somewhat different problems than when working with assembly made directly for processors. Some tasks are easier, due to the more direct organisation of many structures and more embedded data related to classes, method names, parameters, etc. There is less optimisation done by the compiler itself, since some of them are done by the virtual machine itself when doing just-in-time translations. This again makes the job of decompiling easier. On the other hand the object oriented nature of code, with run-time types, inheritance, interfaces and exception handling adds new layers of problems.

While many developers believe that Java bytecode can be fully decompiled in most cases to the levels of human written code, even the best decompilers are not always successful. Research on a sizeable sample set showed that the best can decompile about 80% of the programs, while the rest can include faults or even not be compilable [8].

Another major branch of research in code maintenance is automated program repair. While having different goals than the approach in this paper, which focuses on understanding the code, there are some similarities. Specifically, we will mention some approaches that also use fitness function to guide and evaluate their processes.

The *GenProg* system has an approach to automated software repair via *genetic programming* [9]. The code itself is represented as a sequence of statements and the genetic operators are applied on this level, while the fitness of the programs is decided based on positive and negative run time tests which represent the desired behaviour and the faults. The mutation operators were made with the assumptions that it is very likely that elsewhere in the program there are correct versions of the faulty code. This means that they do not generate entirely new code, but insert other statements from the program, which also reduces the search space significantly. The system has been successfully used with C, assembly and Java bytecode programs including real world

[1] https://www.hex-rays.com/products/decompiler/index.shtml

examples such as web servers and Unix utilities [10], [11]. The main advantages of the approach are that it can be fully automated and has been shown to fix real bugs at relatively low costs.

ARJA-e [12] is another evolutionary program repair system, aimed at Java code repair. Its predecessor is ARJA [13], a tool which works on GenProg's search space, but brings certain enhancements in terms of the genetic programming algorithm. ARJA showed its strength in performance when compared to GenProg as well as in multi-location bug repair. ARJA-e tries to provide smoother gradients for exploring the search space by defining the degree to which a modified program is close to passing a test case.

Another common problem for automated program repair is the tendency to overfit the patch to the tests. This was proven with formal methods on a set of tools and Java programs annotated with Java Modelling Language (JML) [14].

Hill climbing is not a frequent approach in automated program restructuring. It has been briefly considered by Arcuri and Yao [15] and not analysed into depth due to its tendency towards local optimums. Despite of this, the results of this paper prove in practice that good results are possible with it.

A program transformation can in general be defined as any action that takes a program as an input and gives a different program as its output. This can then be something as simple as a change of variable, swapping branches of a conditional statement, or a complete rewrite. For instance, a patch for a software fault could be viewed as a transformation. Many systems use a self modifying (or meta-programming) approach, which means the same language is used to write the program being modified and the program and transformations that modifies it, Meta-programming is in some contexts used in general for languages which are primarily meant to write other programs.

Rascal is an example of a domain specific language (DSL) built around the idea of meta-programming and building tools for program manipulation [16]. It can be used for refactoring and analysing existing program, or construction of other DSLs. It has been used on C, Java, and PHP among others [17].

A special class are transformations that preserve the *semantics* of the program. These can then be applied with no fear of changing what the programs does, while allowing changes in the structure. On the other hand, these can not be used exclusively for applications such as program repair or introducing new features, since this inherently has to change the semantics. Transformations can be used to improve existing programs (or specifications, or models) or to add new features to them. For example, it is possible to extract models from source code that can give a better insight into the original code and enable the usage of high level transformations. *Gra2MoL* is a tool that can extract models from any text that conforms to grammars [18]. Formal methods can also be used for verifying model transformations done in non-formal ways [19].

The *FermaT* transformation system and the language WSL are an example of a self modifying (or meta-programming) approach, which means the same language is used to write the program being modified and the program that modifies it [20]. More detail about this system is given in Section III. One of the advantages of the system is that it is based on formal proof of semantic equivalence. Provided the transformations have been implemented correctly, they can be applied freely with a guarantee that semantic equivalence is preserved, and therefore that no security holes have been introduced, or any loss of functionality was caused by the transformations. In contrast, many systems work by making changes to the code and then running regression tests to see if the functionality has changed. This cannot catch changes to the functionality which are not explicitly tested for and can miss subtle changes to the behaviour of the program. For example, if the program includes security code checks which are not tested for in the regression tests, then a transformation which removes these checks (and therefore introduces a security hole) will be considered a ''success'': it has reduced the size of the program while still (wrongly) passing all tests. An example of such a test is code which sanitises user input to prevent SQL injection attacks. If the regression tests do not explicitly test for SQL injection, then this code will likely be removed and a security hole created.

III. PROGRAM TRANSFORMATIONS IN FERMAT

This section covers the existing transformation system FermaT that is used in the main transformation component HCF. It also presents some of the previous usages of the system for code transformations and restructuring.

FermaT is the current implementation of WSL (*Wide Spectrum Language*) [21]. FermaT provides a whole catalogue of semantics preserving transformations that are at the core of the system. It also provides structures that are able to represent everything from low-level structures up to high-level abstract specifications.

An *action system* is a specific structure in WSL with the goal of representing messy (''spaghetti'') code with a lot of jumps and go-to style commands in such a way that can be mixed with the higher-level structures and eventually converted into them. This type of program is common for legacy libraries and low level code in general. An action system consists of a number of actions, each defined by its name and containing a block of code. It can be viewed as a collection of procedures with no parameters. Additionally, the execution of the whole system can be interrupted at any point. Action systems were originally introduced by Arsac [22], [23]. Two examples are shown in Fig. 2. The action whose name is used at the beginning of the system is the one from which the execution starts, while `CALL` commands are used to jump to other actions. Once that action ends, execution will be returned to the caller. The whole action system stops its execution once the start action finishes, or if there is a call to a special name ''Z'' which terminates the system.

```
ACTIONS start:
  sidequest ==
    PRINT("sidequest")
  END
  start ==
    PRINT("start");
    CALL sidequest;
    PRINT("we are back
        ");
    CALL final;
    PRINT("the end")
  END
  final ==
    PRINT("final")
  END
ENDACTIONS
```

```
ACTIONS start:
  middle ==
    PRINT("middle");
    CALL final
  END
  start ==
    PRINT("start");
    CALL middle;
  END
  final ==
    PRINT("final");
    CALL Z
  END
ENDACTIONS
```

FIGURE 2. A recursive and a regular action system.

Depending on how the jumps and returns are used there are three types of action systems. The first one is *recursive* in which all the calls are returned normally, like the first example shown in Fig. 2. This is what is commonly expected out of recursive procedures. The second one is the *regular* system in which none of the calls ever return – the system is terminated by a `CALL Z` (second example in the Figure). Regular action systems have some advantages in transformations. For instance, since the calls are not returning, then any code that comes after a `CALL` command can be ignored, and `CALL` commands are therefore just simple jumps. There are transformations in FermaT that can only be applied to regular systems and in turn produce regular systems (with obvious exceptions such as converting an action system into ''normal'' code). The third type is the *hybrid* action system which is any combination of both returning and non-returning calls.

The tools used in this paper (*asm2wsl* and *mjc2wsl*) translated low-level code to both regular and recursive action systems, as will be shown in Section IV-A.

FermaT and WSL are successfully used by Software Migrations Ltd for migrating large, industrial assembly systems to C or COBOL, sometimes having millions of lines of code [1], [21]. The process typically involves three stages, similar to the process presented in this paper. First the assembly source code is translated to WSL. The translators used in these projects generated regular action systems, and would try to capture the semantics of assembler programs, without worrying about efficiency at this stage. Additional files that map the memory structures from assembly would also be created at this stage. In the second stage, the WSL is restructured. This part relies on the strengths of the transformations and the expertise of the developers to select the appropriate ones. This led to the development of ''meta transformations'', that primarily combine other transformations in useful ways. The main result of these improvements was the Fix_Assembler transformation that would automate most of the needed work. The migration process is totally automated (so the developers do not have to select transformations), but the translation and transformation process may need to be ''tuned'' to accommodate new styles of programming or new style requirements on the generated code (for example: do not nest IF statements more than N levels deep). Finally, the last stage is to translate WSL to the target language. This part starts with additional transformations to the WSL code to bring it closer to the target language. For instance, COBOL has a lot of restrictions and it is easier to make initial transformations than to write a more complex translator. The translators would combine the initial stored data about the memory mappings to produce the final high-level code.

IV. HILL CLIMBING APPROACH TO AUTOMATISING

This section presents the main process analysed in this paper and its components. It also introduces the sample sets that the experiments were ran on and the results of their transformations. Then some of the properties of the process are analysed in more depth.

The process analysed in this paper transforms a low-level source code to a higher-level version of it. The overview of the process is shown in Fig. 1. It is structured such that it has relatively independent components which gives it flexibility. The basics of most of the components were presented in earlier papers ([24], [25]), and in this paper they are all brought together for a deeper analysis of the whole.

There are two basic steps, translation and transformation. Any input program must first be translated to WSL, which is the language that is used by the FermaT transformation system. To achieve this, translators were written for two main input types which will be presented further in this section. Once the code is presented as a low-level WSL code, it can be transformed to a higher-level version.

Transformations can be chosen and applied ''manually'' by a user through a graphical UI such as FME [26], using command line tools or by writing programs that will apply them. This requires a developer who has experience in using FermaT and preferably understands (at least to an extent) the original languages and concepts.

The transformation component used here offers an automated approach, that requires no domain knowledge. Fermat transformation catalogue offers a large number of transformations that are semantics preserving, which mean they can be freely applied without the risks of changing the logic of the code. On the other hand, the code itself can change, which can be used to gain more understandable programs. To automate the process, the hill climbing algorithm was implemented, which tries to make improvements until it reaches the top of a ''hill'', i.e., no transformations lead to a ''better'' program. The implementation has a list of possible transformations that it can apply to the program at hand. These were hand picked by the authors to be those that will lead to higher-level structures and some others that can help these first ones be more efficient. The main loop of the algorithm tries a transformation and checks if the new program is better than the current one, and if so it becomes the current program and the loop resets. The whole process ends when all transformations are tried on the ''current'' program and none

of them leads to an improvement. An inherent problem to this algorithm is that it can result in local optimums, which will be discussed later.

A central problem to this approach (independent of the algorithm) is how to measure which program is more *high-level* and *understandable*. In this paper we define a *fitness function* that represents this relation. The term is used in genetic algorithms for the same purpose. This is the function that the algorithm will call to check whether the program is better now.

The ease of understanding a piece of code is hard to measure. Even the terms such as *understandability* and *readability* are not universally applied. Finding metrics that correlate well with human cognition is an ongoing task that still has no final conclusions [27]. For the purposes of this paper, we will not go into separating such terms and use them in a general way.

The implementation in this paper uses a fitness function that will compare the metrics of two programs. The *Structure* metric is defined in FermaT with the purpose of showing the elusive property of structuredness. The implementation of the metric is a weighted sum of the elements of the program. The least weight of 1 is given to simple operations such as addition and multiplication, operands such as division and exponent have weight 2 and other expressions are weighted at 4. Branch statements are given a value of five times the number of branches they have, while all types of calls and jumps have even higher values. The numbers were assigned according to the domain knowledge of the authors of FermaT. In practice, this choice leads to good results in most cases and is therefore used in the paper as an approximation of the ease of understanding the programs. For instance a program that has a WHILE loop instead of a number of conditional jumps will have a lower value and will therefore be picked as better.

The nature of the algorithm guarantees that every next iteration of the program will have a lower Structure metric number. An example of the changes in each successive version is shown in Fig. 3. This shape is pretty common for all the samples, the angle of the decline will somewhat oscillate, with usually one really big step down. This is usually the moment in the process when the action systems get removed, or are significantly restructured. This point is commonly before the half point of the process.

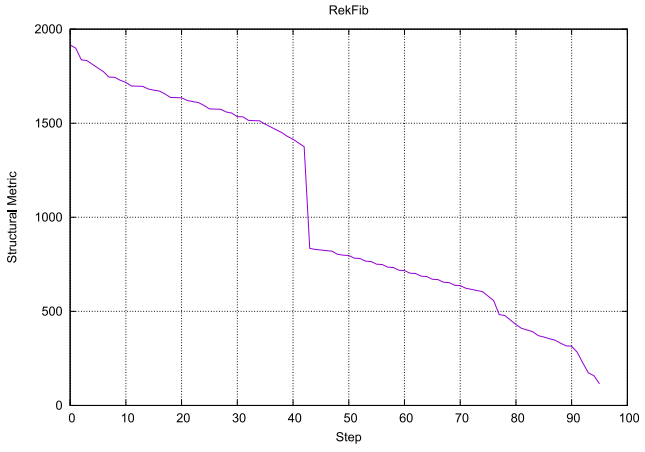


FIGURE 3. Metrics change during the process (RekFib example).

A. INPUT PROGRAM GROUPS AND TYPES

This paper presents results of transformations on programs which come from two different low-level languages: x86 assembly and MicroJava bytecode. Since these need to be translated to WSL for the transformations to be applied, two different translation tools are used as well. The tools themselves have some similarities in approaches, but also some important inherent differences in the way they generate code.

1) X86 ASSEMBLY PROGRAMS

One of the tools developed for this research is *asm2wsl*, which translates a subset of x86 assembly to WSL, enabling the further steps of transformation [24]. More specifically it supports the MASM (Microsoft Assembly) and TASM (Turbo Assembly) dialects of assembly languages [28], [29]. The basic principle of the translation is to have a virtual processor, whose state (registers, stacks, etc) are presented by global variables. Then the individual commands can be translated as operations on these, with all of their side effects (for example, subtraction will set several flags in the processor). There is a limited support for procedures in the translator. Inputs and outputs tend to be very reliant on interrupts in these types of programs. To make the translator simpler, and comparisons more fair, several macro names were defined for these operations which are then recognised and the calls are directly translated to the appropriate print or read statements. When the programs are later compared, the macro definitions that were not translated are not counted in the metrics.

Labels and jumps are presented with the usage of actions in an action system, with the presumption that none of the calls will return, and instead the special *CALL Z* is used to end the execution. Therefore, the resulting action system is of a *regular* type and this can make some of the transformations very efficient on them. Examples of translated jumps and a loop are given in Fig. 4.

The resulting programs usually have several times more statements than the assembly counterparts, due to the ''spelled out'' nature of all the effects of the commands. This is not a problem, since this verbosity is handled by the transformations, as will be shown in later sections.

The sample set used in this paper is named *asm-a*, and contains programs that are mostly classic algorithms like the Greatest Common Divisor, sometimes with variants included to analyse how the same programs will be transformed if recursion is used, or different inputs are included.

2) MICROJAVA BYTECODE

The second type of inputs are based on a virtual machine bytecode. In particular a tool for translating MicroJava

```
je    exit        →
                    IF flag_z = 1 THEN CALL exit
                        FI;

ja    greater     →
                    IF flag_z = 0 AND flag_c = 0
                        THEN CALL greater FI;

jmp   compare     →
                    CALL compare;


    ....
theloop:
    pop ax
    add dx, ax      →
    loop theloop        ....
    ....            CALL theloop
                     END
                    theloop ==
                    POP(ax, stack);
                    overflow := 65536; dx := dx
                        + ax;
                    IF dx >= overflow
                    THEN dx := dx MOD overflow;
                        flag_o :=1; flag_c :=
                        1;
                    ELSE flag_o :=0; flag_c :=
                        0; FI;
                    cx := cx - 1;
                    IF cx>0 THEN CALL theloop
                        ELSE CALL dummy21 FI
                    END
                    dummy21 ==
                        ....
```

FIGURE 4. *asm2wsl* **translations – examples of jumps and label handling.**

```
                                14: enter 0 1
                                17: const_0
                                18: store_0
                                19: load_0
                                20: const_5
program P                       21: jge 13 (=34)
{                               24: load_0
  void main()                   25: const_0
  int i;                        26: print
  {                             27: load_0
    i = 0;                      28: const_1
    while (i < 5) {             29: add
      print(i);                 30: store_0
      i = i + 1;                31: jmp -12
    }                               (=19)
  }                             34: exit
}                               35: return
```

FIGURE 5. **MicroJava code and the translated bytecode ("while-print" program).**

```
C:"#15 (const_0)";
PUSH(mjvm_estack,0);

C:"#23 (add)";
VAR < tempa := 0, tempb := 0, tempres := 0 > :
POP(tempa, mjvm_estack);
POP(tempb, mjvm_estack);
tempres := tempb + tempa;
PUSH(mjvm_estack,tempres);
ENDVAR;
```

FIGURE 6. **Examples of translation of bytecode instructions to WSL (original address and instruction is given in a comment).**

Bytecode, *mjc2wsl* was developed [25]. MicroJava is a subset of Java, developed mainly for teaching purposes, with a few reductions and simplifications made to make it a doable project for students, while still retaining the main features of a complete language [2]. An example of MicroJava and its compiled version is shown in Fig. 5.

The programs used here were written in MicroJava, but only their compiled, low-level bytecode versions, are used in the process. The end results are later on compared with the original programs as another evaluation of the efficacy of the process.

The translator has similarities and differences to the previously mentioned *asm2wsl*. It does use similar concepts of making a virtual machine with the generated code, using variables that represent the state of stacks and execution (some examples shown in Fig. 6). The commands are then translated to show all of their side effects. A full translation (and the later transformation) of the program shown in Fig. 5 is given in Appendix A. The main difference is that here almost each command is translated as its own action in an action system, and all of the calls are supposed to return to their sources, making this a recursive action system (Section III).

During the initial development of the tool, several ways to translate particular structures were introduced and made available through switches. This was in part made to see how these different translations of the same inputs will be handled by the transformation scripts. These include: making local variables in a procedure as an array, or separate variables; leaving temporary variables global or making them as local as possible; making stack operations use pop and push, or the direct list operations head and tail. The three mentioned switches were used in all the combinations, generating 8 types of translations of the same programs from the *alpha-mj* set. The resulting data set was named *alpha-wsl-v8*. All of these were then transformed to see which combinations are the most appropriate.

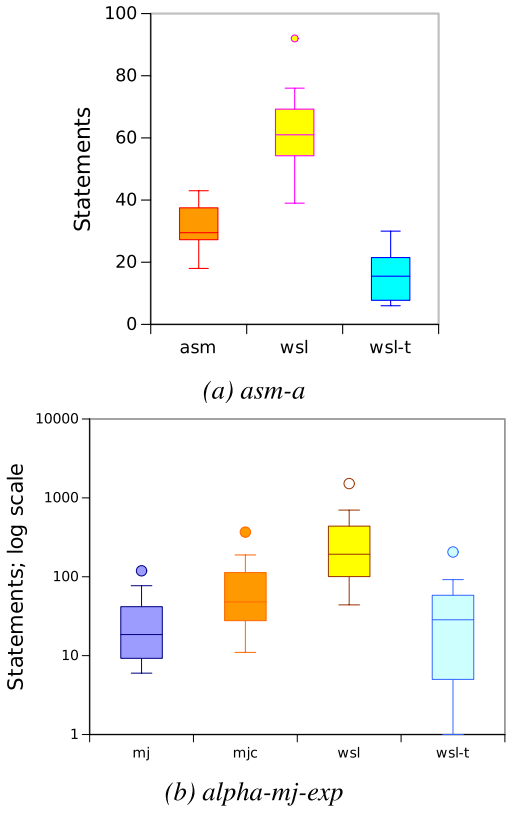
The experiments showed that the best combination, both in terms of average improvements of the Structure metric and in the length of the process is push-pop/local variable blocks/separate local variables (internally marked *pp-lo-sp*). A close second was to use local variables as arrays. Use of global variables would usually result in the same end programs, but the process would be much longer.

The sample set for MicroJava programs is named *alpha-mj-exp*, containing programs that vary from input/output demonstrations to sorting algorithms and palindrome checks.

B. OVERVIEW OF THE RESULTS OF TRANSFORMATIONS

This section will analyse the results of the hill climbing algorithm applied to the translated low-level version of the programs (shown in the previous sections).

As mentioned previously, the translators presented generate several times more statements than the original input programs. This is due to the more explicit handling of side effects of the original commands, such as setting processor flags, interacting with stacks, etc. The later transformation processes are not only able to reduce all of these expansions, but actually can result in shorter programs than the originals. The changes of program sizes in assembly and bytecode can be seen in Fig. 7. Both of the figures present the statement numbers of the samples as box plots in different stages. For assembly there are three stages shown, the original inputs, the translations, and the transformed programs. For MicroJava, there is an additional column which shows the size of the original high-level programs. However, it should be noted again that the process starts from the low-level programs, and the high-level versions are only used for later comparison and are not available to the process as any sort of a guide.



(a) asm-a

(b) alpha-mj-exp

FIGURE 7. Program sizes in different stages.

As explained in the previous sections, the translations add statements to simulate the side effects of the original code on a virtual processor, and are several times larger than the originals. For assembly, this means on average each instruction will be translated to just a bit over 2 WSL statements. With bytecode, this is closer to 4 statements in the translations. The automated transformations are very good at removing this added complexity, and the final results are smaller than the input low-level programs (0.42 times smaller for assembly and 0.52 for bytecode).

```
BEGIN
  PRINFLUSH(@Format(3, a14(0)));
  PRINFLUSH(@Format(3, a14(2)));
  PRINFLUSH(@Format(3, a14(7)))
WHERE
  FUNCT a14(par1) ==
    VAR <  >:
    SKIP;
    (IF par1 = 1
      THEN 1
      ELSE IF par1 <> 0
        THEN a14(par1 - 2)
             + a14(par1 - 1)
        ELSE 0 FI FI) END
END
```

FIGURE 8. Recursive Fibonacci example – automatically restructured WSL.

As an example input, a recursive Fibonacci implementation in MicroJava translates about 80 bytecode statements into 250 WSL lines. HCF can then reduce this to the high-level version shown in Fig. 8. Appendix A shows the full translation and the transformed end result of the program "while-print" (shown in Fig. 5).

The core part of the presented process is the automated transformation. The improvements made can be compared using the available metrics in FermaT. These can be seen in Table 1 for assembly samples. The data is given as averages with standard deviations over all of the samples. It shows the original translation of the samples to WSL, then the transformed versions, and finally it gives the average of the improvements made on samples as a percentage to normalize the data. Significant drops are noticable in almost all metrics. The exception is McCabe's Essential metrics, which is usually unchanged and in one example it increases due to some of the introduced loops.

TABLE 1. *asm-a* transformation metrics.

Metric	WSL	WSL-t	% diff
McCabe Cyclo	7.40 ± 3	3.60 ± 1	41.60 ± 30
McCabe Essential	1.00 ± 0	1.10 ± 0	-10.00 ± 32
Statements	62.30 ± 15	16.00 ± 9	73.90 ± 14
Expressions	84.20 ± 28	31.50 ± 20	63.80 ± 14
CFDF	88.60 ± 16	22.80 ± 9	73.90 ± 8
Size	327.40 ± 71	95.30 ± 51	70.90 ± 13
Structure	947.70 ± 199	206.90 ± 119	78.20 ± 10

The comparison for bytecode samples is given in Table 2 (the columns are same as in the previous table). Here, all the metrics show reductions in the process. The lowest changes are again with McCabe's metrics, around 60%. This is mostly due to both of those not having high starting numbers. The number of expressions gets reduced by 73%, while the other metrics are in the high 80% ranges.

Overall, even though the two types of inputs have a lot of differences in them (recursive and regular (non-recursive) action systems being the biggest), the improvements made by HCF are similar and comparable.

TABLE 2. ***alpha-mj-exp*** **transformation metrics.**

Metric	WSL	WSL-t	% diff
McCabe Cyclo	12.29 ± 11	5.07 ± 4	60.29 ± 15
McCabe Essential	3.50 ± 2	1.04 ± 0	62.32 ± 15
Statements	300.36 ± 306	38.64 ± 44	88.50 ± 6
Expressions	242.25 ± 251	73.89 ± 86	73.61 ± 11
CFDF	440.00 ± 457	59.32 ± 77	89.46 ± 8
Size	1368.04 ± 1365	225.04 ± 247	85.21 ± 7
Structure	4206.64 ± 4255	475.96 ± 519	89.61 ± 5

C. PROPERTIES OF THE PROCESS

After giving an overview of the metrics improvements in the experiments, this subsection will try to answer some more general questions about the properties of the process.

RQ1: *What is the quality of outputs of the automated system in general?*

The transformation script is designed so that it works with any general WSL program, not just with the ones obtained from the mentioned translation tools.

The translated programs that are the input for the transformations had a lot of differences between them. With *asm2wsl* the resulting program is a *regular* action system, where none of the calls return, and the execution ends when the predefined **z** action is called. On the other hand, *mjc2wsl* creates the opposite, a *recursive* action system, in which all of the calls return and the execution ends when the first called action is done. With *asm2wsl* individual actions represent labels in the original program, and can consist of any number of instructions, while *mjc2wsl* creates a single action for each of the instructions in the original bytecode.

Both tools feature switches that can change some aspects of the output code and the way it is organised. There are versions with mostly global variables that are reused a lot, and others with a lot of small scope local variables. Local variables for a procedure (*stack frames*) were also treated in different ways. There were different ways in which the stacks were accessed. However all of these types of outputs were automatically transformed with the same transformation script, admittedly with some variation in the quality of end results.

The same script was also used successfully on some handwritten WSL programs, including the transformer itself.

The quality of the end result of automated transformations is heavily reliant on the used fitness function that guides the process. In these experiments this was the custom *structure* metric. It was successful in restructuring and reducing the metrics of the programs generated with the tools presented in this paper. However, with other input programs it could prove to be less successful. In general, it is impossible to find a universally good fitness function. The "no free lunch" theorem for search and optimisation states that for any search algorithm that is well suited for one class of problems, there is a separate class that will offset this advantage [30]. Of course, this holds when all possible inputs are taken into consideration, and there is no prior knowledge of the input space at hand. For this process in particular, the worst possible input would be one where no transformation can lead to the reduction of the fitness function. In that case the end result would be the same as the original input. This also means that the process can not "corrupt" a program and lead it to a worse state, at least not according to the fitness function.

When considering the input programs that were the main target of this research, there are characteristics that can be taken into account. They do not feature high-level structures and in general have a large amount of labels and jumps that can be restructured. Since these start programs will have higher values for the *structure* metric than even a modestly structured equivalent program, it is likely that, on average, this fitness function is adequate for this class of problem. Of course, it is always an open question whether there is a better function and algorithm.

There is no guarantee of the end result being the global optimum, which is due to the inherent problems of the hill climbing algorithm and it being able to be "trapped" in local optimums.

For many of the samples the end results could be further improved by humans. For instance, manual transformations on one sample could lead to a further simplification in 4 steps, but the metrics stay the same or even go higher during these steps and only the final result is "better" according to the fitness function. For this to be achievable by the automated process it would need at least 4 steps of look-ahead. However, it is impossible to know for a given program how many steps are needed for an improvement, or if there is indeed such a number, since this is a variation on the halting problem [31]. A potential improvement for these types of problems lies in using more sensitive fitness functions and potentially more complex transformations that try more things at once. Both of these can be very sample specific and also increase the usage of resources in the process.

RQ2: *Can the process lead to worse programs, or get stuck in an infinite loop?*

The current implementation of the process can only result in improvements to the readability of the program, as defined by the fitness function. The worst case scenario is an input program on which no improvements can be made, but this should imply that it is already well structured and therefore in no need to be used in the process.

The process is guaranteed to finish on any input program, given a fitness function that defines strict improvements (does not allow programs of the same quality to be used), and that the values are decreasing and can not be negative. This is due to several facts – the set of transformations available is finite, so they can be exhausted on a single position in the program. The algorithm implementation only allows a forward step if the result is better, therefore there is no possibility of a loop. Finally, the used fitness function prefers more structured and shorter programs, so there is no possibility of an infinitely increasing program.

RQ3: *What are the expected execution times of the process?*

Execution times of the transformation process were compared with the metrics of the input programs on both the assembly and MicroJava samples (Fig.s 9 and 10). The graphs present the metrics in different colours for the samples used and maps them to the execution times measured. The growth of execution time was not directly proportional to any of the metrics. That is, there were significant outliers to any of the trends, and programs with similar metrics can have wildly different transformation times. The properties of the presented process are such that it is more important what the structures are in the samples, rather than just how long they are. The hill climbing algorithm will try to apply transformations to any part of the programs to find an improvement. If the input is such that large improvements are found earlier, there will be fewer places to test transformations later on, significantly reducing the execution time.

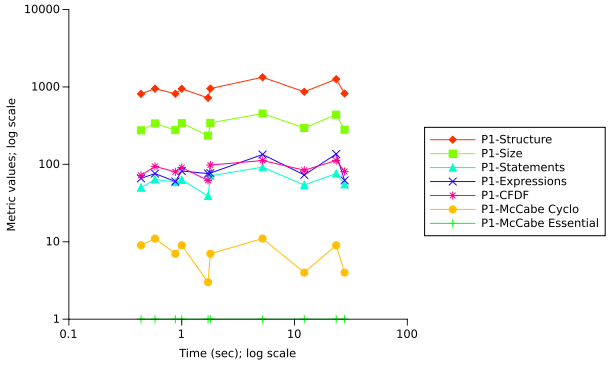


FIGURE 9. Transformation execution times for assembly samples.

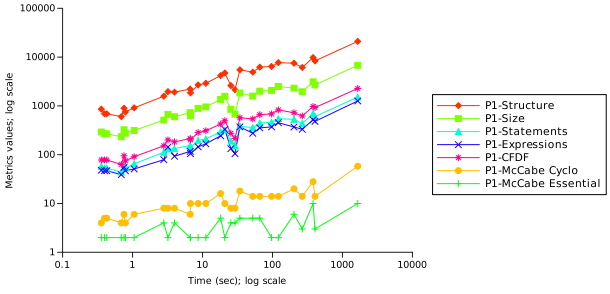


FIGURE 10. Transformation execution times for the MicroJava *alpha-mj-exp* set.

It is early to make generalised statements about execution times for all possible input programs. While the samples here had many differences between them (different languages, action system handling, etc), there could be other types of inputs that behave significantly different. With that taken into consideration, we can say that there is an expectation that larger and more complex programs will take longer to transform, it is just not clear, or predictable how much longer, since there is no direct proportions to any of the metrics.

Apart from execution times in seconds, the length of the process could be measured in the number of transformations. This gives a machine independent figure where performance is not important. In the tests in this paper, shorter programs need a few thousand transformations tried (depending on the computer, performed in less than a second), while the more complex tested samples take almost three million transformations tested. The number of transformations selected to be applied from all of those tested varies a lot between samples. It is usually in the order of magnitude of 1 in a 1000. On some of the simplest samples it can be as high as 1 in 40, again showing that the inherent complexity of the samples influences the behaviour of the whole process.

D. REPRODUCIBILITY

The experiments presented in this paper used FermaT version 18g, available from out git repository.[2] The translation tools are available from their repositories: asm2wsl, version 0.81[3]; mjc2wsl, version 1.1.1,[4] parameters used were "pp-lo-sp". The used HCF script is also available as part of the *mjc2wsl* repository in the "src-wsl" folder.

The data sets are available at in the repositories of the translators, in the "samples" folders. They are also available combined with the translations and transformations discussed in this paper on the project web site.[5]

V. CONCLUSION AND FUTURE WORK

This paper presents a new analysis of an automated approach to restructuring low-level code into high-level structures. The main goal is to help understanding systems that are only available as low-level source and starting the maintenance and/or migration processes of those systems, but the approach could be applied to any system where there is a need to restructure and comprehend the program.

The HCF process relies on using the hill climbing algorithm combined with trying transformations from a selected set in the FermaT catalogue on the input programs and choosing those that lead to better programs based on the *Structure* metric. The system uses semantics preserving transformations, therefore guarantees no loss of functionality and does not rely on external tests to verify the end results. The process itself was presented on a smaller set of MicroJava samples in an earlier paper [25].

This paper analyses the process in more depth. It presents the results achieved on two input groups, obtained from different languages and translators that produced different types of action systems (structure that emulates the jump heavy low-level programs). The goal is to better understand the limits of the current process, where can it be best applied and what improvements could be made.

[2]https://gitlab.com/clatu/fermat3/-/tags/fermat-18-g
[3]https://gitlab.com/clatu/asm2wsl/-/tags/v0.81
[4]https://github.com/quinnuendo/mjc2wsl/releases/tag/v1.1.1
[5]https://perun.pmf.uns.ac.rs/pracner/transformations#datasets

Experiments that were run on several types of input programs showed that the results can be on a level of human written high-level code. When comparing the end results with low-level original source code, there was always significant improvement in the metrics (the assembly code had around 78% on average, while MicroJava bytecode had improvements at around 89% on average). For some input samples the original high-level source code was available for comparison. For these, the end result metrics of the automatically transformed programs were similar to the originals. Some samples ended up being better (metrics-wise) than the originals, while there are some that retained some of the low-level complexities. The details of the experiments are given in Section IV.

The transformation process used make no assumptions about the input programs, i.e., they can be applied to any type of input and are not dependant on the translators used. The end results can vary in quality for different types of programs, since the transformations were chosen specifically to raise the level of abstraction. This means that for some specific inputs the improvements might be minimal, but even then the programs will keep their semantics, in other words, the program will not be ''ruined''.

The fitness function used was a built-in metric that gives the ''weight'' of structures in a program. This was used as an approximation of the understandability of a program, and while it can not be said to be perfect, it showed good results in practice in guiding the process. The developed tools generate different types of low-level programs, and also feature parameters to additionally change the way some structures are translated. All of these variations were handled with the same automated transformation program. The different translations of the same programs had different percentages of metrics improvements. Better results were almost always achieved in significantly less time. This is a direct consequence of the algorithm that tries all the transformations on all parts of a program. As soon as there is success, the process is sped up since there are fewer application targets.

Some of the closest types of processes to these presented in the paper are decompilers. They have a very similar task to what is presented here – start from a low-level program and give an approximation of the original source code. The process in this paper is a more general version of what a typical decompiler is, since it makes no assumptions on the inputs. On the other hand this can be a limiting factor, since sometimes decompilers can obtain better results based on their domain knowledge of the inputs. Neither of the approaches can guarantee the end results to be ''perfectly'' readable, though the advantages of our approach are that they do guarantee the operational semantics of the program to remain the same, while decompilers sometimes produce wrong or useless outputs.

A. FUTURE WORK OPTIONS

By the very definition of the hill climbing process, it is very dependant on the choice of its fitness function. This paper presents results based on the custom *Structure* metric. The metric was giving good results, while also being computationally very efficient. However there is a lot of room to explore other fitness functions and potentially combine metrics in them depending on some attributes of the sample under hand. Some of this work has already been started in [32], where it was shown that many of the alternative metrics and even some combinations of metrics do not lead to significant improvements over the Structure metric. Further research into other options for fitness functions is still needed.

The hill climbing process itself could also be further improved with changes to the transformations used. There is a large amount of data about the transformations tried during these experiments which could be analysed to change these sets. Different sets are tried in the stages of the process and are sometimes grouped up. These choices were made based on the domain knowledge of the author of the transformation system and some early experiments, but some further tweaks and experiments based on the collected data could also be used. The process could also be expanded to test more steps ahead, although this would lead to an exponential increase in time needed with the current implementation. Therefore, heuristics are needed to cut short the exponential time increase by prioritising transformations that have proved useful in combinations in similar code and stopping the search after a specified amount of time has elapsed or a specified number of transformations have been tested with no improvements found.

With an exhaustive search, as with the current system which tries every transformation and each of the selected pairs of transformations at every position, the search order only affects efficiency since any valid transformation will eventually be found. But with a deeper look ahead, an exhaustive search is impractical and search order therefore becomes more important.

With the current system, multiple cores are only exploited to process multiple programs in parallel, but the search process is ''embarrassingly parallel'' since at any point the set of transformations and application points to be searched can be divided equally between multiple processors [33].

Other searching approaches could also be tested in the future to avoid the tendency towards local optima that hill climbing has. One of these is the *tabu* algorithm, which is similar, but allows steps towards worse solutions where no better ones can be found [34]. Variations of evolutionary algorithms could also be tested. The transformations could be used as mutation operators, while the metrics would remain to be the fitness evaluation. Another replacement could an artificial intelligence search technique. Initial experiments were already made with Reinforcement Learning as a replacement for the hill climbing approach [35].

The current process works on WSL directly, which requires programs to be translated to it, and in many cases it would be convenient to translate the results back to the starting language. To be able to use this system more widely, developing more translators is a necessity. In general,

experiments with more programs from different sources are needed to check what the outcomes can be.

On the other hand, the approach with hill climbing could also be used with other transformation systems, not just FermaT. This could be useful for systems that already operate on the target languages. It is important to note that any system that does not feature semantic preserving transformations would also need additional verification that the behaviour of the program has not changed. For instance, this could be done with sets of test inputs and outputs ran after each change. Such verification is potentially resource intensive, while at the same time a challenge to actually prove that the new program is equivalent in behaviour.

APPENDIX A
CODE EXAMPLE

An example of the difference from the translated, low-level version of a program, and the transformed high-level code that is achieved after applying the Hill Climb process. The original MicroJava program and the compiled version were shown in Fig. 5.

A. LOW-LEVEL BYTECODE TRANSLATED TO WSL:

```
C:" This file was automatically
   converted from microjava bytecode
using mjc2wsl v1.1.1
-options:
popPush         (pp/ht):true
localVarBlocks  (lo/gl):true
localsAsArrays  (ar/sp):false
inlinePrint:            false
";
BEGIN
VAR < mjvm_flag_jump := 0,
mjvm_arrays := < >,
mjvm_objects := < >,
mjvm_estack := < >,
mjvm_mstack := < > >:
SKIP;
ACTIONS a14:
a14 == PUSH(mjvm_mstack, mjvm_locals_0);
    CALL a17 END
a17 == PUSH(mjvm_estack, 0); CALL a18
   END
a18 ==
VAR < tempa := 0 >:
POP(tempa, mjvm_estack); mjvm_locals_0
   := tempa
ENDVAR;
CALL a19 END
a19 ==
VAR < tempa := 0 >:
tempa := mjvm_locals_0; PUSH(mjvm_estack
   , tempa)
ENDVAR;
CALL a20 END
a20 == PUSH(mjvm_estack, 5); CALL a21
   END
a21 ==
VAR < tempa := 0, tempb := 0 >:
POP(tempa, mjvm_estack);
POP(tempb, mjvm_estack);
IF tempb >= tempa
THEN mjvm_flag_jump := 1
ELSE mjvm_flag_jump := 0 FI ENDVAR;
IF mjvm_flag_jump = 1 THEN CALL a34 ELSE
    CALL a24 FI;
SKIP END
a24 ==
VAR < tempa := 0 >:
tempa := mjvm_locals_0; PUSH(mjvm_estack
   , tempa)
ENDVAR;
CALL a25 END
a25 == PUSH(mjvm_estack, 0); CALL a26
   END
a26 ==
VAR < tempa := 0, tempb := 0 >:
POP(tempa, mjvm_estack);
POP(tempb, mjvm_estack);
Print_MJ(tempb, tempa) ENDVAR;
CALL a27 END
a27 ==
VAR < tempa := 0 >:
tempa := mjvm_locals_0; PUSH(mjvm_estack
   , tempa)
ENDVAR;
CALL a28 END
a28 == PUSH(mjvm_estack, 1); CALL a29
   END
a29 ==
VAR < tempa := 0, tempb := 0, tempres :=
    0 >:
POP(tempa, mjvm_estack);
POP(tempb, mjvm_estack);
tempres := tempb + tempa;
PUSH(mjvm_estack, tempres) ENDVAR;
CALL a30 END
a30 ==
VAR < tempa := 0 >:
POP(tempa, mjvm_estack); mjvm_locals_0
   := tempa
ENDVAR;
CALL a31 END
a31 == CALL a19; SKIP END
a34 == POP(mjvm_locals_0, mjvm_mstack);
   CALL a35 END
a35 == SKIP END
b35 == SKIP END ENDACTIONS;
SKIP ENDVAR
WHERE
```

```
FUNCT CHR(num) ==
VAR <  >:
SKIP;
(@List_To_String(<num>)) END
PROC Print_MJ(val, format) ==
PRINFLUSH(@Format(format, val)) END
PROC Print_MJ_CHAR(val, format) ==
PRINFLUSH(@Format(format, CHR(val))) END
END
```

B. TRANSFORMED HIGH-LEVEL END RESULT:

```
VAR < mjvm_locals_0 := 0 >:
WHILE mjvm_locals_0 < 5 DO
PRINFLUSH(mjvm_locals_0);
mjvm_locals_0 := mjvm_locals_0 + 1 OD
    ENDVAR
```

REFERENCES

[1] M. P. Ward, "Pigs from sausages? Reengineering from assembler to C via FermaT transformations," *Sci. Comput. Program.*, vol. 52, nos. 1–3, pp. 213–255, Aug. 2004.

[2] H. Mössenböck, "Language specification and compiler construction," in *Handouts for the Course*. Linz, Austria: Johannes Kepler Univ., 2024.

[3] C. Cifuentes, "Reverse compilation techniques," Ph.D. thesis, School Comput. Sci., Queensland Univ. Technol., Brisbane, QLD, Australia, 1994.

[4] C. Cifuentes, D. Simon, and A. Fraboulet, "Assembly to high-level language translation," in *Proc. Int. Conf. Softw. Maintenance*, Washington, DC, USA, Nov. 1998, p. 228.

[5] E. J. Schwartz, J. Lee, M. Woo, and D. Brumley, "Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring," in *Proc. USENIX Security Symp.*, 2013, pp. 353–368.

[6] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "BAP: A binary analysis platform," in *Computer Aided Verification*, G. Gopalakrishnan and S. Qadeer, Eds. Berlin, Germany: Springer, 2011, pp. 463–469.

[7] K. Yakdan, S. Eschweiler, E. Gerhards-Padilla, and M. Smith, "No more gotos: Decompilation using pattern-independent control-flow structuring and semantics-preserving transformations," in *Proc. Netw. Distrib. Syst. Secur. Symp.* San Francisco, CA, USA: The Internet Society, 2015, p. 15.

[8] N. Harrand, C. Soto-Valero, M. Monperrus, and B. Baudry, "Java decompiler diversity and its application to meta-decompilation," *J. Syst. Softw.*, vol. 168, Oct. 2020, Art. no. 110645.

[9] C. Le Goues, S. Forrest, and W. Weimer, "Current challenges in automatic software repair," *Softw. Quality J.*, vol. 21, no. 3, pp. 421–443, Sep. 2013.

[10] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A generic method for automatic software repair," *IEEE Trans. Softw. Eng.*, vol. 38, no. 1, pp. 54–72, Jan. 2012.

[11] E. Schulte, S. Forrest, and W. Weimer, "Automated program repair through the evolution of assembly code," in *Proc. IEEE/ACM Int. Conf. Automated Softw. Eng.*, Sep. 2010, pp. 313–316.

[12] Y. Yuan and W. Banzhaf, "Toward better evolutionary program repair: An integrated approach," *ACM Trans. Softw. Eng. Methodol.*, vol. 29, no. 1, pp. 1–53, Jan. 2020.

[13] Y. Yuan and W. Banzhaf, "ARJA: Automated repair of Java programs via multi-objective genetic programming," *IEEE Trans. Softw. Eng.*, vol. 46, no. 10, pp. 1040–1067, Oct. 2020.

[14] A. Nilizadeh, G. T. Leavens, X. D. Le, C. S. Pasareanu, and D. R. Cok, "Exploring true test overfitting in dynamic automated program repair using formal methods," in *Proc. 14th IEEE Conf. Softw. Test., Verification Validation (ICST)*, Apr. 2021, pp. 229–240.

[15] A. Arcuri and X. Yao, "A novel co-evolutionary approach to automatic software bug fixing," in *Proc. IEEE Congr. Evol. Comput.*, Jun. 2008, pp. 162–168.

[16] P. Klint, T. van der Storm, and J. Vinju, "Easy meta-programming with rascal," in *Proc. 3rd Int. Summer School Conf. Generative Transformational Techn. Softw. Eng.* Berlin, Germany: Springer-Verlag, 2011, pp. 222–289.

[17] M. Hills and P. Klint, "PHP AiR: Analyzing PHP systems with rascal," in *Proc. Softw. Evol. Week IEEE Conf. Softw. Maintenance, Reengineering, Reverse Eng. (CSMR-WCRE)*, Feb. 2014, pp. 454–457.

[18] J. L. Cánovas Izquierdo and J. G. Molina, "Extracting models from source code in software modernization," *Softw. Syst. Model.*, vol. 13, no. 2, pp. 713–734, May 2014.

[19] L. Ab. Rahim and J. Whittle, "A survey of approaches for verifying model transformations," *Softw. Syst. Model.*, vol. 14, no. 2, pp. 1003–1028, May 2015.

[20] M. P. Ward and K. H. Bennett, "Formal methods for legacy systems," *J. Softw. Maintenance, Res. Pract.*, vol. 7, no. 3, pp. 203–219, May 1995.

[21] M. Ward, "Assembler restructuring in FermaT," in *Proc. IEEE 13th Int. Work. Conf. Source Code Anal. Manipulation (SCAM)*, Sep. 2013, pp. 147–156.

[22] J. J. Arsac, "Syntactic source to source transforms and program manipulation," *Commun. ACM*, vol. 22, no. 1, pp. 43–54, Jan. 1979.

[23] J. J. Arsac, "Transformation of recursive procedures," in *Tools and Notations for Program Construction*, D. Neel, Ed. Cambridge, U.K.: Cambridge Univ. Press, 1982, pp. 211–265.

[24] D. Pracner and Z. Budimac, "Transforming low-level languages using FermaT and WSL," in *Proc. 2nd Workshop Softw. Quality Anal., Monitor., Improvement, Appl.*, vol. 1053, Z. Budimac, Ed., 2013, pp. 71–78.

[25] D. Pracner and Z. Budimac, "Enabling code transformations with FermaT on simplified bytecode," *J. Softw., Evol. Process*, vol. 29, no. 5, p. e1857, May 2017.

[26] M. Ladkau, "A wide spectrum type system for transformation theory," Ph.D. thesis, Softw. Technol. Res. Lab., De Montfort Univ., Leicester, U.K., 2009.

[27] M. Muñoz Barón, M. Wyrich, and S. Wagner, "An empirical validation of cognitive complexity as a measure of source code understandability," in *Proc. 14th ACM/IEEE Int. Symp. Empirical Softw. Eng. Meas. (ESEM)*. New York, NY, USA: Association for Computing Machinery, Oct. 2020, pp. 1–12, Paper 5. [Online]. Available: https://dl.acm.org/doi/10.1145/3382494.3410636

[28] Intel Corp. (1987). *80286 Programmer's Reference Manual*. [Online]. Available: http://datasheets.chipdb.org/Intel/x86/286/manuals/286INTEL.zip

[29] *Turbo Assembler 2.0 User's Guide*, Borland Int., Austin, TX, USA, 1990.

[30] D. H. Wolpert and W. G. Macready, "No free lunch theorems for optimization," *IEEE Trans. Evol. Comput.*, vol. 1, no. 1, pp. 67–82, Apr. 1997.

[31] A. M. Turing, "On computable numbers, with an application to the entscheidungsproblem," *Proc. London Math. Soc.*, vols. s2–42, no. 1, pp. 230–265, 1937.

[32] N. Sukur, D. Pracner, and Z. Budimac, "Fitness functions and transformations in an automated process," in *Proc. 8th Workshop Softw. Qual., Anal., Monitor., Improvement, Appl. (SQAMIA)*, Z. Budimac and B. Koteska, Eds., 2019, pp. 17:01–17:10.

[33] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming* (Revised Ed.) Amsterdam, The Netherlands: Elsevier, 2012.

[34] F. Glover, "Tabu search—Part 1," *ORSA J. Comput.*, vol. 1, no. 2, pp. 190–206, 1989.

[35] N. Sukur, N. Milošević, D. Pracner, and Z. Budimac, "Automated program improvement with reinforcement learning and graph neural networks," *Soft Comput.*, vol. 28, no. 3, pp. 2593–2604, Feb. 2024.

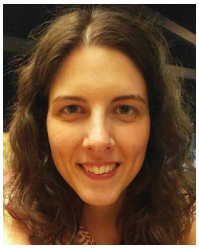
DONI PRACNER received the Diploma, M.S., and Ph.D. degrees in computer science from the University of Novi Sad, in 2007, 2010, and 2019, respectively.

From 2008 to 2019, he was a Teaching Assistant with the Faculty of Sciences, University of Novi Sad. Since 2019, he has been an Assistant Professor with the Faculty of Sciences. He is the author of one book and 20 scientific papers published at journals and conferences. His current research interests include software evolution and software testing, while also having experiences with data mining.

MARTIN P. WARD received the B.A. degree in mathematics and the D.Phil. degree in computation from Oxford University, in 1983 and 1989, respectively.

From 1991 to 1995, he was a Research Fellow with Durham University. From 1999 to 2017, he was a Reader in software engineering with De Montfort University. Since 1995, he has been the Chief Technology Officer of Software Migrations Ltd. He is the author of one book, 25 journal articles and 25 conference papers. He developed the FermaT program transformation system. His research interests include language oriented programming, program transformation systems, program slicing, and software migration.

Dr. Ward was awarded the Royal Society Industry Fellowship, in 2009.



NATAŠA SUKUR received the Bachelor Diploma degree in information technologies and the master's degree in computer science from the University of Novi Sad, in 2014 and 2016, respectively, where she is currently pursuing the Ph.D. degree in computer science.

She spent one semester with the University of Coimbra, Portugal, in 2020, as a part of Erasmus Ph.D. Student Exchange Program. From 2017 to 2023, she was a Teaching Assistant with the Faculty of Sciences, University of Novi Sad. Since 2023, she has been a Teaching Associate with the Faculty of Sciences. She has published 11 papers in international journals and conferences. She has participated in several national and international research projects. Her main research interest include software evolution and maintenance, and software quality in general.

ZORAN BUDIMAC was born in Sombor, Serbia, in 1960. He received the B.S. degree in computer science, the M.S. degree in mathematics, and the Ph.D. degree in computer science from the University of Novi Sad, in 1983, 1991, and 1994, respectively.

He was a Full Professor with the University of Novi Sad, in 2004. He has published more than 300 research articles in the areas of agent technologies, software engineering, learning technologies, 16 books, one international monograph, and presented his work at more than 180 international conferences. He delivered numerous invited speeches at different universities all over the world (Austria, Australia, Bulgaria, China, Germany, Hungary, Thailand, Poland, Portugal, Romania, Slovakia, Slovenia, and Turkey). He was the principal investigator of numerous national and international projects.