Evaluating Fitness Functions for Automated Code Transformations

NATAŠA SUKUR and DONI PRACNER, University of Novi Sad

FermaT is a program transformation system based on the WSL language, and has been used in software evolution applications, mainly for legacy systems and conversions of low level code into high level structures. One option is to automate the process of code transformation as much as possible by using fitness functions to evaluate the improvements made. This paper presents some initial experiments with different fitness functions and tries to evaluate the correlation between using a software metric as a guide and improvements made for that specific metric.

1. INTRODUCTION

Software engineering is a very broad field, where numerous disciplines focus on software maintenance and evolution. They are important due to the problem of today's need for constant changes in software. In the early days of computer science and technology, the need for change was not so frequent, with mostly major releases and significant improvements, but now it seems like the need for changing never stops, and the deadlines are a matter of days and weeks, rather than months and years.

Software quality is the key element a piece of software should have. Nowadays, software is present everywhere, new software is also being produced and it needs to be constantly maintained in order to preserve its quality. Maintenance can emerge in perfective, adaptive and corrective form, based on what kind of changes it focuses on. However, despite the effort to develop software that conforms to the requirements, some errors show when the software goes into operation. It is possible that some errors were not discovered in time and software has to evolve in the sense of functionalities, environment and scale, in order to have a long life. These changes have to be performed quickly and because of tight schedules, the cost can be quite high. Sometimes, when software systems need significant changes, or have been changed too many times, they need to be reengineered. Software evolution is often defined as repeated reengineering, or as a single cycle of a possibly infinite process of creating a better system. The process of evolution in a dynamic system is never completely finished and every step of it has to be done in as fast as possible. When a need for reengineering emerges, the main thing is to fully understand the software and its functionalities, so that the modifications do not cause any new defects, but rather cause improvement and that they are done in a logical manner [Yang and Ward 2003; Tripathy and Naik 2014].

This paper presents research based on the usage of FermaT program transformation system (started in 1989, and developed, improved and reimplemented several times since then) and tools that extend

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

In: Z. Budimac (ed.): Proceedings of the SQAMIA 2018: 7th Workshop of Software Quality, Analysis, Monitoring, Improvement, and Applications, Novi Sad, Serbia, 27–30.8.2018. Also published online by CEUR Workshop Proceedings (http://ceur-ws.org, ISSN 1613-0073)

existing functionalities and introduce some new ones. One of these tools is *mjc2wsl* [Pracner and Budimac 2017], whose main purpose is translating MicroJava bytecode to WSL (Wide Spectrum Language), which is later transformed by FermaT. Another tool is based around the automated transformation process of translated low level programs into much more understandable and high level structures. The process is based around the idea of a *fitness function* that is used to evaluate the programs abstract quality, and is then used as a guide when selecting software transformations. *Fitness* is a term from evolutionary computing. As the name suggests, evolutionary computing draws inspiration from the process of natural evolution. The evolutionary processes relate to computing problems in the following way: if we defined fitness as a quality of individual to achieve its goals (to have a better survival and multiplying chance) in the environment, that could apply to computing problems as the quality of a candidate solution for actual solving of the problem at hand. The quality of these candidate solutions determines the chance that they will be kept and used as seeds for constructing further candidate solutions [Eiben et al. 2003]. The focus of this paper is on evaluating some possible fitness functions and their impact on the end results, but also on the process itself.

The rest of the paper is organized as follows: Section 2 focuses on formal methods in reengineering, and particularly on FermaT, WSL and code transformations. In Section 3, the translation tool *mjc2wsl* is briefly described, as well as the software transformation process. Section 4 deals with the actual experiments in improving the fitness function. Finally, Section 5 draws some conclusions based on obtained results and some options for future work.

2. FORMAL METHODS AND WSL

Formal methods can be used in different stages of software life cycle. They can be applied to specifications, models or source code and influence the overall product reliability. Formal methods are widely used in the process of software reengineering, which consists of three phases: reverse engineering, functional restructuring and forward engineering. Various formal methods can be used for the purpose of forward engineering (assertional methods, temporal logic, process algebra and automata), as well as functional restructuring. They have also been used for reverse engineering in some processes, such as formally specifying and verifying existing systems, introducing new functionalities and improving the systems design techniques. The question of using formal methods in the development of systems raised various questions and many different opinions. Some say that formal methods are vital to a high quality process of software development. On the other hand, some say that it is impossible to completely rely on usage of formal methods in (re)engineering and that it is very costly. Nevertheless, systems and their complexity are constantly growing and it is necessary to have a reliable methodology and approach to designing and developing them.

Different types of formal methods have their advantages, but also flaws. Some of the quality criteria that is considered is whether it has supporting automated tools for its development, whether it is reliable, concurrent, if there is any proof system and so on. The conclusion by [Yang and Ward 2003] is that the choice of formal methods depends on the nature of the problem we are trying to solve. Some formal methods are better for large industrial applications (Z) [ISOZ 2002], some are good for reasoning about concurrency and communication (different process algebras) and some are good for visual representation (net-based formalisms). As previously mentioned, formal methods are not used often as a theoretical basis for reverse engineering. WSL [Ward and Bennett 1995] is a language based on formal methods which is highly useful for reverse engineering of sequential systems. In fact, the whole WSL/FermaT system was designed with the purpose of reverse and forward engineering.

Code transformations can be performed by using formal methods. The whole purpose and goal of code transformation is cost reduction, which can be in terms of anything from performance and memory

usage to portability. Apart from using transformations for evolving existing software, they can also be useful when new software is being developed. A program transformation can be defined as an operation which, when applied to a program, generates an equivalent program.

WSL gets its name from *wide spectrum language*, which means that it contains both abstract mathematical specifications and low-level programming constructs. Program transformations use this wide spectrum language, in order to create programs from specifications, to perform the reverse engineering of programs and get the specifications and to analyze properties of a program. Aside from traditional language functionalities, such as commands and structures, it also contains operations that work on WSL programs themselves, which are called *MetaWSL*. These are the base for a variety of transformations that are provided with the system and whose correctness can be automatically checked. For these reasons, it is very useful for the activities performed in the process of restructuring [Yang and Ward 2003]. It has been used in several industrial projects of converting legacy assembly code to human understandable and maintainable C and COBOL [Ward 1999][Ward 2004][Ward et al. 2004][Ward 2013]. Another tool that was made for assembly translation, with a slightly different focus is *asm2wsl* [Pracner and Budimac 2011].

FermaT transformation system also contains a set of metrics, which can be applied to a certain program or its parts. These metrics are McCabe Cyclomatic Complexity, Essential Complexity, Size (the size of abstract syntax tree), Statements (the number of statements in a program), Control Flow and Data Flow (the number of variable accesses and updates, combined with the number of procedure calls and branches), and Structure metric (a custom WSL metric for representing the complexity of program structures, gives different weights to various types of structures) [Yang and Ward 2003]. These metrics will be used later on in the paper in the experiments to improve the quality of the transformation process.

3. TRANSLATION AND TRANSFORMATION

The first step in using FermaT to transform source code not originally written in WSL is to provide the translators of this code to WSL. One such tool that was created was *mjc2wsl*, which works with MicroJava [Mössenböck 2018], a subset of Java programming language. The tool itself works directly with the compiled MicroJava bytecode obtained from the high level source codes. The low level structures and operations are translated at the same level of abstraction with the operational semantics preserved through a “virtual” MicroJava Virtual Machine. WSL has a built in structure called *action system* that is made out of separate actions, which are essentially procedures without parameters which are very efficient at representing the low level code that is filled with jumps. Translation tools such as these do not need to prioritize the reduction of size and complexity of the outputs, since this will be handled during the transformation part of the process.

Once the code is translated to WSL, FermaT can be used for manual or automated transformations [Pracner and Budimac 2017]. The approach that this paper will be dealing with is using a *hill climbing* algorithm with a *fitness function* that measures the quality (however defined as being a simpler and/or more understandable) of the code. Hill climbing is a well-known search algorithm that constantly moves “uphill” – in the direction of the increasing value and terminates once it reaches a peak, which means that no neighbor has a higher value [Russell and Norvig 2016]. The function can be the value of some simple metric, such as the length of the program, or an arbitrarily complex function. The implemented *hill climbing* script tries one transformation at time, but if none of them shows an improvement, it advances to combining two transformations before reevaluating the result. All the successful intermediate steps are recorded in separate files, allowing for a more detailed analysis of the process when needed. The existing automated script uses the Structure metric, which is built into WSL and gives a weighted sum of the structures in the program, as the fitness function.

Table I. Part of the generated comparison file, showing the improvement of McCabe Cyclomatic Complexity in percentages

filename	fit-cfdf	fit-mccabe	fit-o1	fit-o2	fit-size	fit-stat	fit-struct
ArrayTest.wsl	50	16	50	50	50	50	50
ArraysTest.wsl	37	12	37	37	50	50	37
InOut1.wsl	50	50	50	50	50	50	50
InOut2.wsl	50	50	50	50	50	50	50
InOut3.wsl	50	25	50	50	50	50	50
Rek1.wsl	25	25	25	25	25	25	25
RekFac.wsl	33	16	66	66	66	66	66
RekFib.wsl	66	16	66	66	66	66	66
chrtest.wsl	50	50	50	50	50	50	50
div0.wsl	66	66	66	66	66	66	66
div2.wsl	66	66	66	66	66	66	66
eratos.wsl	50	7	57	57	57	57	57
fields.wsl	50	12	50	50	50	50	50
linkedlist.wsl	38	5	27	38	55	50	38
pos-neg.wsl	56	6	62	62	62	62	62
while-print.wsl	50	25	50	50	50	50	50
Average	49.19	27.94	51.38	52.06	53.94	53.63	52.06

The question whether hill climbing is suitable and optimal for solving the problem of automatic program repair was only discussed before [Arcuri and Yao 2008], and this paper is an attempt to get empirical data. The idea of using fitness functions for code improvement is not a novelty. There has been a lot of research on automated software repair which also relies on fitness functions and focused at first on C programs [Forrest et al. 2009] and assembly programs [Schulte et al. 2010], but later evolved to applicability to any kind of code in general [Le Goues et al. 2012]. There is also research on how to design these fitness functions in order to get the best results of the automated bug detection [Fast et al. 2010; de Souza et al. 2018], which also shows that designing various fitness functions with respect to the problem at hand can be greatly beneficial.

4. EXPERIMENTS AND RESULTS

The main goal of these experiments was to compare several *fitness* functions, and in order to do that, different versions of the *hill climbing* script were generated and run with the same input sample sets. Following that, a number of metrics were calculated on all of the results. The goal was to compare whether changing the evaluation function made difference in the metric values and if scripts with some specific evaluation functions were more suitable for specific kinds of programs. Comparison of a part of these values of metrics is shown in Table I, with the combined results of McCabe Cyclomatic complexity measured on all transformed WSL, where each of the values represents the percentage of improvement for this metric. In general most of the comparisons were done using percentages, due to the differences in individual sample sizes.

Most fitness functions in this experiment are based on the value of one of the built in metrics in WSL (described in Section 2), and these were named correspondingly: *fit-cfdf*, *fit-mccabe*, *fit-size*, *fit-struct* and *fit-stat*. The remaining two fitness functions used in this experiment were more complex – *fit-o1* and *fit-o2*, which contain multiple checks, such as number of actions and number of calls. The experiment was run on a sample set which consisted of 16 different programs, later referred to as the *alpha* sample set.

4.1 Overview and Analysis

In this section focus will be given to some results which were significantly different in comparison to others and brief analysis of the possible reasons for these differences. One of the questions that we wanted to answer with these experiments was if using a certain metric for the fitness function would result in the best results for that specific metric in the final program (compared to other fitness functions). The assumption was that it does not have to be the case, and that in a general case using some other fitness function could give similar or even better results.

4.1.1 *Simple fitness functions.* The best improvement percentages were accomplished by using two fitness functions which used only one metric for evaluation, *fit-size* and *fit-stat*. Very similar results were also obtained with the originally used *fit-struct* fitness function. Other simple fitness functions showed less improvement in the results. Following are some notes on other fitness functions that were tested.

The results of the *Control flow*/*Data flow* fitness function show that changes of all metrics values are usually slightly less than the ones measured on the results of other, more efficient fitness functions mentioned above. Specifically, this also includes the values of Control flow/Data flow metrics measured on all transformed WSL code, which confirms our assumption that the metric used for the progress of the fitness function does not guarantee the best final result of the same metric. In general results of all metrics will get better or remain the same in the final results, but there was also one result which showed that the result can actually get worse than in the beginning. When using this fitness function on the ArraysTest sample WSL code, the value of Essential complexity was increased from one in the translated WSL to two in the transformed WSL. In general this could happen with other samples, since an improvement in one metric can lead to an unintentional deterioration of another metric. For the sample at hand there was an 85% improvement of the CFDF metric.

An obvious example that defies the initial hypothesis is applying *McCabe Cyclomatic Complexity* metric in the fitness function. In fact, when using this metric to evaluate the programs, it was noticeable that all the other fitness functions have lead to better results for McCabe, and in many cases with a large margin (Table I). Another important observation was that when this fitness function was used it always gave the least percentage of change not just for itself, but also for all other metrics. This leads to a conclusion that using McCabe Cyclomatic complexity is probably not suitable. Analysis of the results indicates that the reason for this is the rare changes of cyclomatic complexity in the transformation process, and therefore, the function can rarely make progress, and the whole process will get stuck. It has only shown slightly better results on div0 and div2 examples, but that is still not as good as other fitness functions. It also executes for a significantly longer period of time than other fitness functions, mostly because of the large number of transformations performed with no apparent improvements.

Not all available metrics were used for creating and comparing different fitness functions. *Essential Complexity* was not used, for example. There were attempts at running it on the samples, but there were some technical difficulties and the process was very long even for the simplest programs when successful and the results were not very promising, and therefore the testing was left for future work. The problem with essential complexity is similar to that already encountered with cyclomatic complexity – it rarely changes and it is very difficult for the function to make progress with no changes in value.

4.1.2 *More Complex Fitness Functions.* Comparing results of different fitness functions indicates that more complex fitness functions, *o1* and *o2* usually give the same or similar values as some of the simple fitness functions: the ones using size, structure and number of statements metrics. Perhaps

the more complex evaluations cause longer transformation time, due to more checks. However, it is possible that they would show better results in the analysis of more complex code. Therefore, the question is whether more complex checks are necessary and if they are, how to implement them to get better results. Otherwise, the cost of performing multiple checks does not make much sense.

4.1.3 *Additional Observations of Metric Results.* In most cases, very similar results were obtained from a group of fitness functions that includes size, structure, number of statements and the two complex ones. However, the results of McCabe Cyclomatic Complexity on two samples (ArraysTest and LinkedList) are significantly better when using size and number of statements than the other ones. Another exception was that the o1, o2 and structure gave better results in terms of *Essential complexity* on the LinkedList example, with 80% improvement, while statements and size gave improvements of 60% and 40%, respectively.

Almost all fitness functions gave excellent results in reducing *Control flow / Data flow* metric, usually around 90%. In other words, the choice of fitness function is not that important for this metric. In cases of very simple programs such as chrtest, div0 and div2, the reduction of CFDF value was 100%. This happens mostly when all values can be calculated and propagated from the starting state of the program, which means the code will be simplified as much as theoretically possible and can lead to these high reductions of CFDF of up to 100%.

4.1.4 *Fitness Functions and Corresponding Metrics Values.* Special focus was given to question whether metrics when used as a fitness function lead to the best results for the corresponding metric. The improvement of a certain metric while using the same metric as the fitness function is compared to the values which represent the maximal average improvement of the same metric value. These results are presented in Table II, with the names of the appropriate "best" functions. Figure 1 additionally shows the average improvements over all of the tested fitness functions. Based on these, conclusions can be made that in most cases a metric used as the fitness function gives good results for the same metric on the code that is transformed. Sometimes better results can be obtained with other fitness functions, but usually the differences will not be great. The major exception to this statement is clearly using McCabe Cyclomatic Complexity, which did not show good results not only for itself, but also for any of the calculated metrics, as already stated earlier.

Table II. Metrics and improvements with the corresponding fitness function, and with the best fitness function

Metric Name	Improved	Best Fitness	Best improvement
McCabe Cyclo	27.94%	fit-size	53.94%
Statements	91.50%	fit-stat	91.50%
CFDF	93.12%	fit-struct	93.44%
Size	86.19%	fit-struct	86.25%
Structure	90.12%	fit-struct	90.12%

5. CONCLUSIONS AND FUTURE WORK

FermaT and WSL can be efficiently used to transform programs from low to higher levels of abstraction. It is possible to automate this process to a large extent, and one of the possible scripts that does this is called *hill climbing*. It relies on using a *fitness* function that will evaluate a program at hand and uses it to determine whether a given transformation improves the program or does not. It uses a predetermined set of transformations and tries to apply them as much as possible. This paper focuses

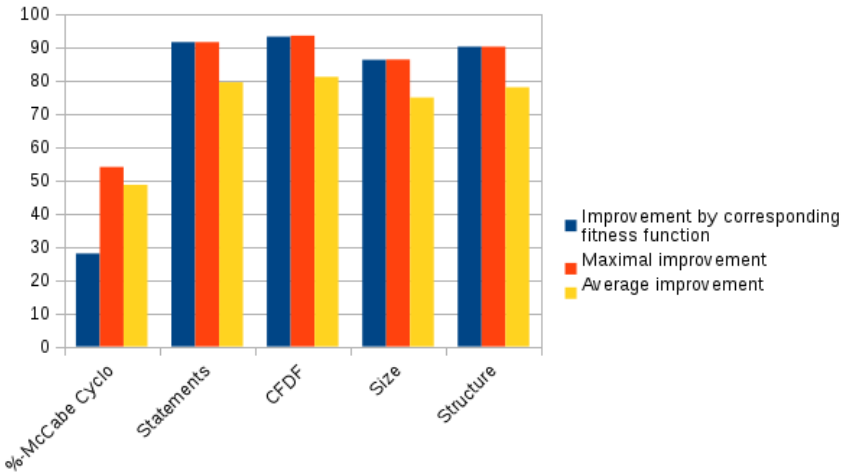

Fig. 1. Comparison of the results of the fitness function which corresponds to the observed metric, maximal metric improvement and average metric improvement.

on experiments with different fitness functions and their effects on the final result as well as the process itself. Mainly the selected functions were simple and consist of built-in metrics, but a few more complex ones were tested as well.

Results show that if some metric is used as the fitness function, the values of the same metric in the final transformed program do not have to be better than with any other tested function. Mostly the values will be similar to the best, but there were exceptions, most notably McCabe's Cyclomatic Complexity, which showed far worse results, but it gave poor results for all the other metrics as well. Overall, this confirmed the assumptions made.

Size and Structure generally showed the best results across the board, as well as number of statements. However, there is no single fitness function which always and absolutely had the best results. On the other hand, Control flow/Data flow always had slightly worse results than the group of these "better" fitness functions. McCabe's Cyclomatic Complexity had the worst results as a fitness function for all of the metrics. The more complex evaluation functions showed the best, or close to best results, but not significantly better, therefore not justifying the additional computation time.

Analysis shows that the starting values of the better metrics on the translated WSL before transformation usually had significantly higher values than the weaker ones, and that their change during the process was large as well. This is a big part of why they show better results – their values also change much more easily with many transformations, and therefore the algorithm succeeds to perform more steps and perform better code enhancements.

There is still a lot of room for analysis and improvement. These first steps only show some indications for that, but after more analysis it would be more clear which way these improvements should go. The process itself produces a lot of data that could be analyzed in more depth – specifically the intermediate steps in the transformation process, which metrics change the most, when do they change, which transformations are more often successful, what orders of application are best, etc.

While the more complex functions did not show excellent results, it is definitely worth experimenting with new combinations for fitness functions used in the transformation process. Based on these results it would be good to focus on using metrics whose values change more easily and in different situation which would make the algorithm more capable of finding good transformations. Execution times of all

fitness functions should also be considered – there are some that give similar results, while taking very different amounts of time to get to them. Apart from combining existing components into a good evaluation function, it is still an open question whether some new metrics should also be introduced that would prove to be good as a fitness function.

The input program samples should be improved and expanded to enable more and different experiments. Grouping samples with similar features could also give more insights into the quality of some fitness functions and possibly lead to conclusions about the best functions for certain types of programs.

The basic outlines of these experiments could also be applied to other transformation systems to obtain more data and to get a clearer picture of the applicability of fitness functions in different systems. One such system is Rascal [Klint et al. 2011], with support for both source to source transformations, as well as built-in metrics.

REFERENCES

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

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

Agoston E Eiben, James E Smith, and others. 2003. *Introduction to evolutionary computing*. Vol. 53. Springer.

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

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

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

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

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

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

Doni Pracner and Zoran Budimac. 2011. Understanding Old Assembly Code Using WSL. In *Proc. of the 14th International Multiconference on Information Society (IS 2011)* (2011-10), Marko Bohanec et al (Ed.), Vol. A. Institut "Jožef Stefan", Ljubljana, Ljubljana, Slovenia, 171–174. http://is.ijs.si

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

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

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

Priyadarshi Tripathy and Kshirasagar Naik. 2014. *Software evolution and maintenance*. John Wiley & Sons.

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

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

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

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

Martin P Ward and Keith H Bennett. 1995. Formal methods for legacy systems. *Journal of Software: Evolution and Process* 7, 3 (1995), 203–219.

Hongji Yang and Martin Ward. 2003. *Successful evolution of software systems*. Artech House.