VAASAN YLIOPISTO

TIMO MANTERE

Automatic Software Testing by Genetic Algorithms

ACTA WASAENSIA

No. 112

Computer Science 3

UNIVERSITAS WASAENSIS 2003

Supervisor	Professor Jarmo T. Alander Department of Electrical Engineering and Industrial Management University of Vaasa FIN-65101 Vaasa Finland
Reviewers	Professor Olli Nevalainen Department of Computer Science University of Turku FIN-20520 Turku Finland
	Dr. Harmen Sthamer DaimlerChrysler AG Alt-Moabit 96A D-10559 Berlin Germany
Opponent	Dr. Hans-Michael Voigt Gesellschaft zur Förderung angewandter Informatik (GFaI) Rudower Chaussee 30 D-12489 Berlin Germany

ACKNOWLEDGEMENTS

"So you feel like you ain't nobody
Always needed to be somebody
Put your feet on the ground
Put your hand on your heart,
Lift your head to the stars
And the World's for your taking"

I would like to express my gratitude to several people and institutions for their contribution to this thesis. First of all, I would like to thank my supervisor, Professor Jarmo Alander, for his co-operation, support and guidance throughout this study. Without his personal support this challenging study would never have seen the light of day. I would like to thank Professor Matti Linna for his pertinent comments concerning this study. I would also like to thank all the other co-authors of our publications: PhD Pekka Turunen, Lic.Sc. Ghodrat Moghadampour, M.Sc. Jukka Matila, M.Sc. Tero Pyylampi, M.Sc. Jari Virolainen, and research colleagues Professor Jouni Lampinen and M.Sc. Timo Rautakoura. Some of their ideas are inevitably mixed with mine and appear in some form in this study.

I would also like to thank Dr. Harmen Sthamer and Professor Olli Nevalainen for reviewing the thesis and their valuable comments and remarks concerning the text. I thank Mrs. Lilian Rautiainen (née Grahn) for proofreading this thesis and many of our research papers.

I would like to thank the organizations that have funded my research. This work was carried out at the University of Vaasa, in the Department of Information Technology and Production Economics, during the years 1996-2001, and the Department of Engineering Sciences in 2002. In 2002, the funding was provided by the Research Institute for Technology at the University of Vaasa. The work was partly carried out in the research project funded by the Technology Development Centre of Finland (TEKES), with the corporate partners ABB Transmit Oy, and FBM Limited Oy. The Finnish Cultural Foundation supported the study by granting research stipends from the Maili Autio Fund and TietoEnator fund. The author has also received travel grants to international conferences from TUCS (Turku Centre for Computer Science), USENIX (Advanced Computing Systems Association), AAAI (American Association for Artificial Intelligence), and TRM (European Commission Training and Mobility of Researchers Programme).

Furthermore, I wish to thank all the people who have supported or assisted me during this work, all my colleagues, and all my former teachers in all educational institutions I have attended by this day, and all friends and relatives.

Vaasa, Finland, April 8, 2003

"So you feel like it's the end of the story
Find it all pretty satisfactory
Well I tell you my friend
This might seem like the end
But the continuation is
Yours for the making....."
(Brian May)

Timo Mantere

CONTENTS

FIGURES AND TABLES

LIST OF APPREVIATIONS

AAAI	American Association for Artificial Intelligence
ABB	Asea Brown Bower, company
ANSI	American National Standards Institute
API	APplication Interface
ATE	Automatic Test Equipment
B&B	Branch and Bound
CA	Cultural Algorithm
CAN	Controller Area Network
CEC	Co-Evolutionary Computation
DE	Differential Evolution
EA	Evolutionary Algorithm
ES	Evolution Strategies
FBM	FBM limited Oy, company
GA	Genetic Algorithm
GP	Genetic Programming
IEEE	Institute of Electrical and Electronics Engineers
ISO	International Organization for Standardization
Lic.Sc.	Licentiate of Science, degree between a Master's and Doctor in Finland
LON	Local Operating Network
PhD	Doctor of Philosophy
SA	Simulated Annealing
SAT	SATisfiability problem
TEKES	Technology Development Centre of Finland
TRM	European Commission Training and Mobility of Researchers Programme
TSP	Traveling Salesman Problem
TUCS	Turku Centre for Computer Science
USENIX	Advanced Computing Systems Association
VHDL	Very high scale integrated circuits Hardware Description Language
W/BCET	Worst (WCET) and Best (BCET) Case Execution Times

ABSTRACT

Mantere, Timo (2003). Automatic Software Testing by Genetic Algorithms. *Acta Wasaensia* No. 112, 151 p.

This study examines automatic software testing by using genetic algorithm optimization for test data generation. The tested software set consists of some small subroutines, toy problems, image processing filters, measurement software, and a large time-critical embedded software. The goal was to verify software quality by finding extreme situations, bottlenecks or faulty behavior. Genetic algorithms are an optimization method that adapts to the given problem, thus they are also able to adapt to the software to be tested, and after finding some slightly suspicious input parameter combinations, start to evolve more dangerous parameter combinations.

Testing is both technically and economically an important part of high quality software production. It has been estimated that testing accounts for half of the expenses in software production. Much of the testing is done manually or using other labor-intensive methods. It is thus vital for the software industry to develop efficient, cost effective, and automatic means and tools for software testing. Searching for software errors with genetic algorithms might be one of the many steps needed towards this goal.

This study proposes that genetic algorithms can be used in automatic software testing to generate test data for system and module testing. In this study the genetic algorithms based test data generation was applied to different kinds of software systems in order to verify the proposed approach. Based on these experiments, the genetic algorithm finds suspicious parameter combinations more efficiently, compared to the random testing, and further more it reveals more serious error situations containing these parameters. In system verification genetic algorithms can be used to reveal input data characteristics that are problematic for the object software. The method can also be applied to search for more real software error tolerances than the classic error limit estimation. Co-evolutionary genetic algorithms can be applied to develop software parameters simultaneously with the testing.

Timo Mantere, Department of Electrical Engineering and Industrial Management, University of Vaasa, PO Box 700, FIN-65101 Vaasa, Finland. Email: timo.mantere@uwasa.fi

Keywords: Genetic algorithms, optimization, reliability, software testing, test data generation.

LIST OF PUBLICATIONS

This thesis consists of the introductory part and the following seven papers:

I [AM00a] Alander, Jarmo T. and Timo Mantere (2000). Genetic algorithms in software testing - experiments with temporal target functions. In: *MENDEL 2000 6th International Conference on Soft Computing*, June 7–9, 2000, Brno, Czech Republic, 9–14. Ed. P. Ošmera. Brno: Brno University of Technology & PC-DIR.

II [AMMM98] Alander, Jarmo T., Timo Mantere, Ghodrat Moghadampour and Jukka Matila (1998). Searching protection relay response time extremes using genetic algorithm – software quality by optimization. *Electric Power Systems Research* **46**, 229–233.

III [AM99] Alander, Jarmo T. and Timo Mantere (1999). Automatic software testing by genetic algorithm optimization, a case study. In: *SCASE'99 – Soft Computing Applied to Software Engineering*, 11.–14.4.1999, Limerick, Ireland, 1–9. Eds C. Ryan and J. Buckley. Limerick: University of Limerick.

IV [MA00] Mantere, Timo and Jarmo T. Alander (2000). Automatic image generation by genetic algorithms for testing halftoning methods. In: *Intelligent Systems and Advanced Manufacturing: Intelligent Robots and Computer Vision XIX: Algorithms, Techniques, and Active Vision, volume SPIE-4197*, Boston, MA, 5–8 November 2000, 297–308. Ed. D. P. Casasent. Bellingham, Washington, USA: SPIE Optical Engineering Press.

V [MA01d] Mantere, Timo and Jarmo T. Alander (2001). Testing a structural light vision software by genetic algorithms – estimating the worst case behavior of volume measurement. In: *Intelligent Systems and Advanced Manufacturing: Intelligent Robots and Computer Vision XX: Algorithms, Techniques, and Active Vision, volume SPIE-4572*, Newton, MA, 29–31 October 2001, 466–475. Eds D. P. Casasent and E. L. Hall. Bellingham, Washington, USA: SPIE Optical Engineering Press.

VI [MA02a] Mantere, Timo and Jarmo T. Alander (2002). Developing and testing structural light vision software by co-evolutionary genetic algorithm. In: *QSSE 2002 The Proceedings of the Second ASERC Workshop on Quantative and Soft Computing based Software Engineering*, Feb 18–20 2002, Banff, Alberta, Canada, 31–37. Edmonton: Alberta Software Engineering Research Consortium (ASERC) and the Department of Electrical and Computer Engineering, University of Alberta.

VII [MA03a] Mantere, Timo and Jarmo T. Alander (2003). *Testing digital halftoning software by generating test images and filters co-evolutionarily*. Unpublished draft, 14 pages (a revised version of [MA02c]), submitted to SPIE's Photonics East, 27-31 October, Providence, Rhode Island, USA.

Papers I–VI have been accepted with the referee practice and have been published in proceedings of international conferences or in scientific journal (II). Paper VII is a draft

version, and is not yet accepted or published at the date of this thesis, submitted as [MA03a]. The published papers are used here with the permission of their original publishers.

The articles themselves are included unchanged in their original published form. No changes or corrections were made to the original text. In case of known or significant errors, they are reported in errdata or Chapter 6, which includes the introduction to each paper.

Author's contribution to the publications

Most of the work, discussed in this thesis and the research papers included, was carried out by the author, while working as a member of the Genetic Algorithms Research Group of the University of Vaasa, under the guidance of Professor Jarmo Alander. Jarmo Alander was a co-author of all papers included in this thesis. He was acting as the supervisor of my thesis work and also as the leader of the research group. He checked all draft papers and stated the essential changes needed and made suggestions for improvements.

According to the research contract between the University of Vaasa and the industrial partner ABB Transmit Oy, the papers [AMMM98, AM99] were inspected by the representatives of ABB in order to prevent publishing any confidential information. Jukka Matila acted as project manager at ABB, and he was a co-author for the paper [AMMM98].

The author of the thesis is the principal author of all the included papers. The author carried out the required research work, programming, testing, experiments, and the problem solving reported in the articles. The required computer programs, including the genetic algorithms were programmed and improved to better meet the task by the author. One exception was the paper [AMMM98] where Ghodrat Moghadampour did the analysis of the test results. The author did the test setup part and edited the results in the paper.

1 INTRODUCTION

Software testing is an essential task when trying to achieve high software quality. Testing is both technically and economically an essential part in high quality software production. It has been estimated [Mye78, Nor93, Kit95] that testing causes about half of the expenses related to software production. Much of the testing is done manually or using other labor-intensive methods. It is thus vital for the software industry to develop efficient, cost effective, and automatic means and tools for this task. Even partial automation of the testing with an effective tool can bring considerable savings. Searching for software errors by genetic algorithms might be one step towards this goal. New measurement techniques and metrics are really needed for assessing the quality and reliability of software as well as for the prediction and measurement of software production. According to the US Trade Ministry [NIS02] software errors cost the U.S. economy $59.5 billion a year, approximately 0.6 percent of gross domestic product. They estimate that more than 30% of these costs could be eliminated by earlier and more effective identification and removal of software defects with an improved testing infrastructure.

Today, more and more new industrial products include microprocessors. The programs of these embedded systems grow in size and become more complex, and therefore problems with their software become more difficult and more usual. From the viewpoint of Finnish export, the most important programming tasks today are related to embedded electronic devices. Performance is also an important quality and a competitiveness factor of computer systems.

This thesis investigates the possibilities of applying genetic algorithms to software testing. *Genetic algorithms* (GA) [Hol75] are optimization methods that are based on an artificial computational model of evolution in nature and the Darwinian evolution process [Dar59]. A kind of artificial ecosystem is generated in the memory of the computer where the virtual beings compete for the chance of reproducing according to their fitness function values. The fitness function is deduced from the optimization problem for which the artificial ecosystem

tries to adapt. The individuals of the virtual ecosystem should offer relatively good solutions to the original problem after several generations of evolving.

Software testing in this work is applied by using the so-called "*black box*" approach [Bei90], in which the program code or its execution is not traced or followed in any way. The only test information we have from the execution is the input we feed to the software interfaces and the output (response) we get from the software.

Genetic algorithms are used as a *test data generator*, which generates test cases for a tester program that feeds them further to the tested software. A tester program traces software interfaces and recognizes outputs caused by input data. A fitness function is based on the selected software metric that we are trying to optimize. Genetic algorithms then generate new individuals based on the fitness values that each individual obtains during the evaluation process. After several generations have been tested, the GA should have adapted to the software and generated test cases that find problematic situations for the software to be tested.

1.1 Overview of the thesis

The framework of this thesis is based on the publications [AMTV96, AMT97a, AMM97, AMMM97, **AMMM98**, AMP98, **AM99**, AMP99a, AMP99b, **AM00a**, AM00b, **MA00**, MA01a, MA01b, MA01c, **MA01d**, **MA02a**, MA02b, MA02c, Man03, **MA03a**, MA03b] that have been published between 1996–2003. The papers shown in **bold** are included in this thesis in full. This work is also a continuation of the Lic.Sc. thesis [Man99] that studies the same problem. The references [Man96, ML96] from the author also study the use of genetic algorithms with another problem. Chapter 4 of the introduction is edited to the form of a review article about evolutionary software engineering [MA03b].

A major part of the research work behind this thesis was done in co-operation with Professor Jarmo Alander. During the years 1996–98 research was done within a research program called "*Adaptive and Intelligent Systems Applications*" of the Finnish Technology Development Centre – TEKES. The industrial partners in this project were ABB Transmit Oy 1996–98, and FBM Limited Oy in 1998. ABB Transmit Oy is a major manufacturer of

protection, control and automation products for medium voltage electrical distribution networks. FBM Limited Oy is the leading manufacturer of mailing systems in Finland.

1.1.1 Structure of the thesis

Chapter 1 introduces the contents of this study, research objectives and the main research questions and hypothesis. Chapter 2 outlines the basic terms, methods, and describes the history of software testing. Chapter 3 describes the concept of an evolutionary algorithm and different types of evolutionary algorithms are presented. Chapter 4 reviews the related work in the field. Chapter 5 gives some examples of applying genetic algorithms to the software testing problem. Chapter 6 presents and reviews the previously published articles included in this thesis. Finally Chapter 7 summarizes the conclusions and suggests some areas for further studies.

1.1.2 Background

In order to state the foundation of this thesis, it is appropriate to discuss the background and objectives of this study together with the justification for some fundamental decisions on how this work was carried out.

The original motivation for this research was our industrial partner's need for developing new software testing methods to verify that the time critical embedded software in their product meets its tight response time requirement under all possible input situations. A kind of stress testing was needed. The optimization approach was adapted from earlier research projects done in our group. Genetic algorithms were adapted *e.g.* to several problems dealing with elevator group control [AYT95], optimization of PID-controller parameters [AMT97b], and cam shape design for a diesel engine [Lam99].

The motivation for including image processing and halftone filter design in this study originated from our other industrial partner, FBM Limited Oy. They had a need for adding figures like company logos to address labels used in direct advertising. These labels were printed in bulk with a very fast, but low-resolution ink jet marking machine. The machines used did not support good graphics, so we tried to generate better threshold matrix halftone filters for them with GAs [AMP98, AMP99]. That research was extended to the test image

generation in order to connect our earlier research on software test data generation and image filters in order to do experiments on the idea of testing the image processing software quality by GA generated test images.

The testing of the accuracy of a measurement software using machine vision was due to the planned cooperation with another research project going on at our laboratory. The solder paste inspection based on machine vision is studied in this project [Rau00].

The idea of applying co-evolutionary methods was motivated by our previous experience with 1) generating halftoning filters with GA 2) generating test images for testing halftoning methods with GA, 3) generating software test data by GA 4) the goal of achieving better software. The goals 1–2 and also 3–4 seem natural co-evolutionary pairs for which the simultaneous optimization against the contradictory goal might lead to co-development.

Software testing is a field of engineering, where the gap between the state of the art and the state of the practice is exceptionally wide. Software testing has yet to become a fundamental component of the software engineering curricula of universities. Papers presenting leading-edge ways of improving the software quality are published. They are however, often not read by those who might benefit most from them. Unfortunately, many of the state-of-the-art methods are unproven in the field, and often they omit some important real-world dimensions like return on investment [Kit95]. Kaner *et al.* [KFN99] state that "*We have yet to meet a computer science graduate who learned anything useful about software testing at the university*", and they do not expect to meet many of them over the next decade. Tamres [Tam02] states that "*software testing is not intuitive, and one must learn how to do it*", "*not all the good programmers are good testers*", "*testers usually know more about programming than programmers about testing*", and she also describes many real-life problems within the testing profession.

Table 1 [Paa02] presents some of the discrepancies between the software testing goals in the research community and testing practices in the industry. The industry usually has practical problems that are too concise or technical to be the targets of scientific research. On the other hand some widely researched methods like formal proofing of the code is too complex and thus impractical to be applied in real industrial software production.

Table 1. Some reasons for the differences between the research and practice [Paa02].

Research community:	Industry:
Scientific problems, new scientific results	Business, new saleable products
General, profound, theoretical basis	Swiftness, operational, inherited environment
Radical new methods	Conservative, old familiar methods that work
Model or method important, not the realization	The realization decisive, not models or methods
Scientific community evaluates the quality	Customers evaluate the quality

This work was seen as a chance to see whether industry and researchers could benefit from co-operation in software testing. The industrial partners in this project offered test cases by which we could test our genetic algorithm based testing ideas in practice.

1.2 Research objectives and contributions

The research of this thesis was motivated by the rapid growth of the software industry. So-called embedded systems are becoming more and more a part of our daily lives, and all these systems have software that is becoming increasingly complex. Therefore, there is a need for studying new efficient and reliable automatic software testing methods. Genetic algorithms were chosen as a testing method, because they have grown in popularity in optimization of engineering problems. At the beginning of this study there were only a handful of research papers that used genetic algorithms for software testing. Therefore we found this territory relatively new, unexplored and challenging. During these six years GA has become more popular in this context and, as Chapter 4 indicates, several papers on the application of GA in testing have been published. The use of heuristic optimization methods in software testing has been called "*search-based software engineering*" [HJ01] and "*evolutionary testing*" [MW98].

The original general research question was: "*is GA capable of discovering software weaknesses, such as finding software bottlenecks, or parameter combinations that lead to delayed responses or to no responses at all?*"

The first PhD thesis on software testing with GAs was by Harmen Sthamer [Sth95], it concentrated on structural testing with white box methods. Therefore white box methods and test coverage testing were left out of this thesis. In 1996 we decided to concentrate on testing the response and execution times of software with GA. Coincidentally, the first publication

with similar idea of temporal testing with GAs was published at that same year [WGG+96]. Later, Hans Gross wrote PhD thesis [Gro00] concentrated on the same topic as chosen by us, so we decided to expand our application area. The topic of my thesis was expanded from temporal testing to include other testing types as well, *e.g.* testing image processing algorithms with test images, testing measurement software accuracy with GA generated objects, and applying co-evolution for developing software parameters simultaneously with testing.

The research questions and hypothesis of this thesis are as follows:

- "*Is genetic algorithm based temporal software testing capable of finding the worst case execution times of the software?*"
- "*Is a genetic algorithm able to generate test images and test surfaces that can be used for testing the quality of image processing software and a software system performing measurements by the aid of machine vision?*"
- "*Is it possible to apply a co-evolutionary genetic algorithm so that the changeable software parameters can be optimized simultaneously with the testing process?*"

Our hypothesis in this study is that the answers for all these three research questions are positive.

The main contributions of this thesis are publications dealing with worst-case execution time (WCET) optimization with GAs. These are among the earliest publications in this field. There are published papers about image and surface generation by GAs, but our papers seem to be the only published ones on generating these images and surfaces for testing purposes. We also do not know of [Ala95a] any previous published papers on applying co-evolutionary GA for developing software during the testing process.

An extensive literature search on applying evolutionary algorithms for optimizing software testing and software quality problem was also done. We like to call this research area "*Evolutionary Software Engineering*" [Ma03b] and review the most important papers in this area in Chapter 4. This study was limited to the black box testing strategy: the research on white box testing is only briefly reviewed in Chapter 4.

2 SOFTWARE TESTING

In this study it is assumed that the reader has a basic knowledge of software testing. The essentials of software testing, and the details of many commonly used techniques can be found in *e.g.* [Het73, Het88, Kit95, Mar95]. Therefore, this thesis only briefly describes the history and motivation behind software testing, and the methods and concepts that were directly used in this study.

The quality of a software system is primarily determined by the quality of the software design and implementation process that produced it. Likewise, the quality and effectiveness of the software testing process are primarily determined by the quality of the test processes used. Typically, more than half of the errors are introduced to the system in the requirements phase. The cost of errors is minimized if they are detected in the same phase as they are born, and an effective test program prevents the migration of errors from a certain development phase to the subsequent one. It has been estimated that repairing an error caught during the system specification phase may be about 50 times cheaper than for an error detected in the system testing phase [Jon78]. Boehm [Boe74] reports that 12% of the errors discovered in a software system over a three-year period were due to errors in the specification of the original system requirements. In practice we often do not have a mechanism to detect these errors in place until much later – often not until in the function and system testing phase. [Kit95]

Software errors are human errors, software is written by people, and people make mistakes. There are practically no commercial software systems without any errors. Some errors are more disturbing, visible or costly than others, and the testing can never reveal all of them. We cannot therefore totally prevent the occurrence of software errors. So the best we can do is to try to locate them as early as possible, and at least find and fix the most critical ones.

2.1 History of software testing

In the early days of software development, testing was regarded as "*debugging*", or fixing the known errors in the software, and the developers themselves usually performed it. There

were rarely any specific resources dedicated to testing [Kit95]. Early works on program testing can be traced back to 1950 [Mil80]. During that time, even advanced software systems had very limited interaction, if any, with other systems [Bab82]. *Automatic test equipment* (ATE) dates back to the mid-1950, when the maintenance of U.S. military electronics started to face formidable problems due to complexity. The solution was the concept of multipurpose ATE, which promised testing at computer speeds, fully automatic operation by less-skilled operators, virtual elimination of maintenance documents, and universal designs adoptable to any test problem through the flexibility of programming [Lig72].

In the late 1950's software testing was distinguished from debugging and became regarded as detecting the errors in the software [Het88]. But the testing was still an after-development activity. The underlying objective was to show that the given product worked and then ship it to the customer. Researchers of computing science did not talk much about testing either. Computer science curricula dealt with numerical methods and algorithm development, but not with software engineering or testing. [Kit95]

The term "*software engineering*" was invented in the late 1960's. At that time there was a "*software crisis*", software being expensive, bug ridden, and impossible to maintain [SW99]. By the 1970's the term software engineering was used more often, though there was little consensus as to what it really meant [Kit95]. The first formal conference on testing was held at the University of North Carolina in 1972 [Het88].

In 1978 Myers [Mye78] defined testing as "*the process of executing a program with the intent of finding errors*". He pointed out that if our goal is to show the absence of errors, we will discover very few of them. Establishing the proper goal and mind-set has a profound effect on testing success. Baber [Bab82] states that by the 1970's the cases of "*fully tried and tested*" software that were found to be useless were no longer surprising, since they were fairly common throughout the 1960's. The assumption that they represented a transient phenomenon was beginning to prove false.

By the early 1980's "*quality*" became the popular theme in industry. Software development professionals and testers started to get together to talk about software engineering and testing.

Groups were formed to eventually create the many standards we have today (IEEE, ANSI, ISO). International standards in full-published form are becoming too vast and detailed to be guidelines for everyday practical purposes. However, they include important guidelines, a baseline for contracts and provide invaluable references [Kit95]. In 1981 Browne and Shaw [BS81] stated that at the present *software engineering* is a technical activity for which we have developed a large set of *ad hoc* engineering techniques without a corresponding scientific foundation.

Since the mid-1980's testing tools and automatic testing have been the center of focus with growing quantity and also due to increasing quality. These tools include *e.g.* coverage analyzers, test planning systems, and test design aids that can help to do the testing job more efficiently and cost effectively. [Dra99]

Table 2. The four eras of testing [Lai02].

I. The process pioneers
• Test management and tailored tools
II. Initiation of commercial testing tools
• Test coverage and white box testing
III. Current testing practice
• Regression and black box testing
• Static test automation
IV. Next generation testing
• Testing process and strategy
• Dynamic test automation

Today, only about 10% of the cost of a large computer system lies in the hardware, while it was over 80% in the 1950's [SW99]. It has been reported that software costs are growing 15% annually, while productivity is increasing at a rate less than 3% [DG81]. Despite the enormous advances in the last 30 years or so, the software development and testing process is still very immature in most companies. The complexity and criticality of the software has increased. Even many well-proven methods are still largely unused in industry today, and the development of software systems remains inordinately expensive. [Kit95]

Laine [Lai02] present the software testing tool industry point of view of the four eras of software testing practices shown in Table 2. Currently, the practices are moving from the third era to the fourth. The focus is to move from static to dynamic test automation and also to test processes and strategies behind software production. He also claims that software size and therefore also test requirements are increasing exponentially. The scalability of current software testing practices are increasing linearly; causing the expanding gap between testing requirements and testing practices.

2.2 Terminology

In colloquial language there are several terms used to describe *program errors (bugs)*. According to the IEEE/ANSI standard [IEE90] they mean:

- Mistake: A human action that produces an incorrect result.
- Fault: An incorrect step, process, or data definition in a computer program. The outgrowth of a mistake.
- Failure: An incorrect result. The result of the fault.
- Error: The amount by which the result is incorrect.

The definition of testing according to the IEEE/ANSI standard is: "*The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component*" [IEE90]. The definition of software testing according to the IEEE/ANSI standard is: "*The process of analyzing a software item to detect the difference between existing and required conditions and evaluate the features of the software items*" [IEE83].

2.2.1 Verification and validation

Verification and validation are two basic forms of testing. *Verification* is human testing, because it involves looking at documents, specifications and the code on paper. Verification is the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of the phase [IEE86]. *Validation*, on the other hand, normally involves executing the actual software or simulation. Validation is computer-based testing, and it usually exposes symptoms of errors. Validation is the process of evaluating a system or component during or at the end of a development

process to determine whether it satisfies specified requirements [IEE86]. Verification is closer to static and white box testing, whereas validation is closer to dynamic and black box testing.

2.2.2 The white, gray, and black box testing

Two fundamental testing strategies are black box and white box testing. These represent external or internal perspectives [Het88]. Black box testing is closer to dynamic testing, whereas white box testing is closer to static testing. In *black box testing* the tests are derived from the functional design specification, without any regard to/or knowledge of the internal program structure, or code. The code is understood from the specification point of view, *i.e.* what the software is supposed to do, but not how the code is actually constructed. Black box testing will not test hidden functions, nor find errors hidden in them. White box tests require knowledge of the internal program structure and tests are generated with the help of code specification [Kit95]. In *white box testing* the code could be statically analyzed against the specification. When the code is executed with white box testing, it is traced, and the clause, decision, condition, multiple condition, and path coverage are measured, also the dead code can be detected. It can also be executed symbolically. *Gray box testing* is an intermediate form, where code is not examined, but the black box testing is supplemented with the knowledge of program structures.

2.2.3 Stress testing

In *stress testing* we load the system until it starts to paralyze, *e.g.* when response times become too long, the system ignores some inputs or is totally collapsing. The key parameters of the system are strained with maximal load. According to Schäfer [Sch96], stress testing includes testing of performance, maximal speed, same inputs, requests or faults to all lines simultaneously, difficult searches, inappropriate values, *etc.* The purpose of stress testing is to identify peak load conditions under which the system fails.

2.2.4 Fault seeding

Error seeding is a well-known method, in which some specific errors are seeded into the software being tested. The goal is to find these seeded errors, and then to statistically

estimate the amount of real errors in the software by measuring how many of the seeded errors were found.

2.2.5 Automation of software testing

Software testing is an expensive and time-consuming task. The purpose of automatic test data generation is to reduce costs and human work. Normally, the large number of possible test cases is a problem, and automatic testing can work round-the-clock and reduce routine work. The profitability of automatic testing is achieved through repeating the tests for newer versions or different configurations of the software.

2.2.6 Testing temporal behavior

The operation of real-time software is often presented with the help of the so-called *state-machine model*. In a simple state-machine only the state, in which the impulse occurs, affects the response. In the real software there are several processes and impulses that overlap, causing the responses also to overlap, sometimes this results in unexpected delays. Load variations also have their side effects. It is not easy to predict the behavior of the system, however, and embedded real-time software usually is required to meet some time constraints. Temporal behavior testing tries to find those input situations where the system either fails or nearly fails to meet its time constrains.

2.3 Research problem

Normally, when using genetic algorithms for function optimization, the GA creates trial solutions, which are then tested by a static fitness function in order to evaluate their fitness as a problem solution. When we are generating test data with GAs the roles of the system parts (GA and fitness function) change. Now the GA generates test cases, that are passed to the object system and there is no longer a static fitness function that directly would evaluate the goodness of the trial. Instead, the object system under testing executes its functionality according to the received data, and the fitness value is assigned by observing what the tested system does.

2.3.1 Software testing as an optimization problem

When testing software by a GA we optimize the input according to some criteria. We must therefore define or choose the software metric we are optimizing. This metric should be measurable from the software, either directly or indirectly.

We could use evolutionary testing with either white box or black box testing techniques. In the first case, we can define the optimized metric to be *e.g.* some test coverage metric; code, condition, or path coverage. There we need an application that traces the software execution, and we try to generate a test set that gives the best coverage. If we choose the black box approach, we will not trace the software execution, but instead trace what happens in the software interfaces. The optimized metric could be error based, *e.g.* amount of warnings, calculation or rounding errors, leakage of memory, *etc.*, or temporal based *e.g.* best or worst execution times or response times (B/WCET).

2.3.2 Software testing by genetic algorithms

It is important to observe that a GA does not find any single error from the software with any higher probability than random search. However, if the error situation is composed of a combination of input parameters, or a sequence of operations by these inputs, then it is possible that GA gains advantage. In practice this means that a GA tester uses trials with several parameter combinations that cause minor faults and constructs new input sequences that have more errors than pure random tests. When testing nondeterministic time critical software with a temporal fitness function, GA recognizes inputs that cause delays; it starts to favor the parameter values of these inputs, and hopefully eventually generates input that causes maximal response delay. Note that the extreme execution times found in this way are not necessarily globally maximal/minimal. There are methods like static analysis that can be used in order to validate the extreme cases, but with large software their use becomes virtually impossible [MW98].

The GA based testing does not need to separately test faults like the index boundaries or division by zero. If a division by zero happens, the GA recognizes the possible symptoms of the software and should soon discover that error situation. The GA has also been applied to

mutation testing [Sth95], where it effectively killed the mutants that were generated by changing the boundary values.

The advantages of GAs over random testing include:

1. needs less human analysis, the GA pre-analyses the software according to the fitness function,
2. automatically tests combinations of suspicious input parameters, and
3. may find the combination of parameters that leads to more severe fault behavior.

The testing setups and GA parameters of this thesis are problem specific. Therefore we omit the setup details and GA parameters in this introductory part, instead they are discussed in the individual papers in detail. This thesis does not concentrate on finding optimal GA parameters, however, paper I deal with this problem briefly. We believe there is no universal good GA parameter setting, but instead the optimal evolutionary algorithm and its parameter set is different for each problem. As a rule to find an error is much more important than to trying to find optimal GA parameters.

Software testing in this thesis is done as a validation process, using the black box testing strategy and dynamic testing.

3 EVOLUTIONARY ALGORITHMS

Evolutionary algorithms (EA) belong to a branch of evolution inspired heuristic optimization methods, the most well known being: evolution strategies, genetic algorithms, and genetic programming [Mit98]. Methods used in this study were genetic algorithms, differential evolution, and co-evolution. Last two of these are special variants, differential evolution, sometimes called differential genetic algorithm, is not based on Holland's original idea and is not exactly GA at all, it is a rather new concept and method. Co-evolution contains two or more GAs, in which population co-evolves either with predator-prey relationships, symbiotic relationships, or parasitic relationships.

The history of genetics may start from Lamarck's [Lam09] multifaceted theory of evolution. Lamarck proposed that the obtained characteristics of an organism are inheritable by its offspring. If a person lives in a mountainous terrain and develops muscular legs, their offspring would inherit muscular legs. So the specialized traits for surviving in the environment are passed to the offspring. The Darwinian [Dar59] theory is that a person who has muscular legs has a genetic tendency for muscular legs, and will probably have offspring with a similar genetic tendency. There does exist EA versions that are based more on the Lamarckian theory [Gre91], however, the great majority of EAs are based on the Darwinian view.

3.1 Evolution strategies

Evolution strategies (ES) [Rec73] are a group of widely applied optimization algorithms based on the evolutionary principles. The hypothesis is that during the biological evolution the laws of heredity have been developed for the fastest phylogenetic adaptation. In contrast to the genetic algorithms, ES imitate the effects of genetic procedures on the phenotype. Features of an optimized object are parameterized as vectors of numbers. The presumption for coding the variables is the realization of a sufficiently strong causality; small changes must affect small changes. ES did not originally contain crossover, thus the idea was that children would have a single parent from which they are mutated. Later, a variety of crossover operations have been added. Populations of these vectors are evolved via selection

and variation processes over a number of generations. Fitness values, defined according to the optimization, decide the individual's survival probabilities from generation to generation. The significant part of ES theory is the "*evolution window*"; evolutionary progress takes place only within a narrow band of the mutation step size. This fact leads to the necessity for a rule of self-adaptation of the mutation step size, *c.f.* simulated annealing.

3.2 Genetic algorithms

Genetic algorithms (GA) [Hol75, Gol89] are another group of evolutionary algorithms, that use evolution principles, like selection, mutation, and recombination, that were originally proposed by Charles Darwin [Dar59]. Genetic algorithms form a kind of electronic population that fights for survival, adapting to its environment as well as possible, which is an optimization problem. Surviving and crossbreeding possibilities depend on how well individuals fulfill the target function. GAs are used to solve complex optimization tasks, they do not require the optimized function to be continuous or derivable, or even be a mathematical formula, and that is perhaps the most important factor why they are gaining more and more popularity in practical technical optimization. The difference between evolutionary strategies and genetic algorithms is not always clear. GA and ES have become more close to each other as population and recombination have been added to ES. The main differences that remain are: in GA the individuals are encoded as a string of values (binary strings, integers, floating point numbers) called a genome, whereas ES use whatever representation fits the problem. EAs can work on the actual parameters that are encoded as real numbers, GAs perform mutations and recombination on an encoded version of the parameters. In EA mutations are normally distributed perturbations, frequent small changes and infrequent large changes and the mutation rate decreases as the run time elapses, whereas in GA mutation is usually fixed size changes and always with the same rate. In ES offspring replace parents only if they are more fit, only good individuals breed and no randomness is involved in the selection, whereas in GA there is randomness in selection of survival and reproduction.

3.3 *Genetic programming*

Genetic programming (GP) [Koz92] is an automated method for creating computer programs, algorithms, or digital circuits. GPs require a high-level presentation of a problem. Then by genetically breeding a population of computer programs and using the principles of Darwinian selection they perform automatic program synthesis using biologically inspired operations, such as recombination, mutation, inversion, gene duplication, and gene deletion. Usually GPs use trees as representation of individuals (fig. 1). Genetic operations are applied to trees, branches and leafs. A leaf represents a single program command or a variable value, while a branch represents some programming language construct of commands and variables, usually subroutine calls.

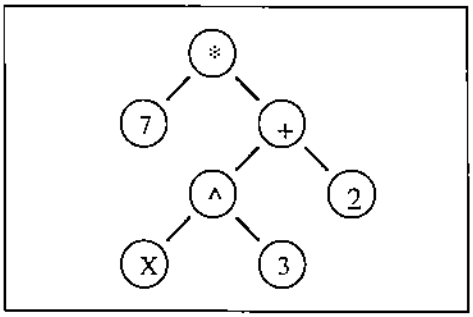


Figure 1. Representation of a GP solution as a parse tree, which is read from left to right. The tree represents arithmetic expression: $7*X^3+2$.

Genetic programming is related to the evolutionary software testing in the sense, that when generating software code with GP, we have a definition of what the resulting routine should do. GP generates trial solutions, which are then tested against the specification, the closer the output is to the desired output, the better the fitness.

3.4 Differential evolution

Differential evolution (DE) [SP95] is an optimization method that was created by Kenneth Price, while trying to solve the Chebychev polynomial-fitting problem that was posed to him by Rainer Storm. He came up with the idea of using vector differences for perturbing the vector population. The discussions between them and several computer simulations brought many substantial improvements, which made DE a versatile and robust tool.

DE is a very simple population based, stochastic function minimizer. It turned out to be the best genetic type algorithm for solving the real-valued test function suite of the 1st ICEO (International Contest on Evolutionary Computation)[BDL+96]. The essential idea behind DE is a scheme for generating trial parameter vectors. Basically DE adds the weighted difference between two population vectors to a third vector, this third vector is then recombined with a selected vector x to create a new trial. A trial is tested and if its better than x, it replaces x in the population. There is no need for an additional mutation operator, and thanks to the weighted differences the scheme is self-organizing.

3.5 Co-evolution

Co-evolutionary computation (CEC) [DY97] generally means that an evolutionary algorithm is composed of several species with different types of individuals, while a standard evolutionary algorithm has only one single population of individuals. In CEC the genetic operations, crossover and mutation are applied to only one species, while selection can be performed among individuals of one or more species. When we deal with an optimization problem, the environmental conditions are either stochastic or immeasurable. We can then try to develop the environmental conditions concurrently with the problem. Change in one population causes an environmental change in the other population. Trial solutions implied by one species are evaluated in the environment implied by another species. The goal is to accomplish an upward spiral, an arms race, where both species would achieve ever better results.

4 RELATED WORK

When we are interested in finding ways of making software reliable and faultless, or developing new automatic means and tools for software testing, we find many interesting connections between our research and previous research in the field of software testing. We could thus review a large number of related studies, but in the following we will discuss only works that are most closely related to our subject.

The generation of test data for software by EAs is the main related topic. In addition to this, we are also interested in applying EAs as software quality assistance and in using EAs for other testing purposes, *e.g.* testing digital circuits.

The growing interest in using metaheuristic search techniques in software engineering problems has generated a research network: Software Engineering using Metaheuristic Innovative Algorithms (SEMINAL). Their homepage <http://www.discbrunel.org.uk/seminalproject/index.html> presents the researchers involved in this area and lists research topics. Another site <http://www.systematic-testing.com/> concentrates on functional and evolutionary software testing and their automation. The research community has also started to generate a benchmark problem library. However, currently it contains only a few short C-routines. There is also Software Testing Online Resources at the Mtsu site STORM <http://www.mtsu.edu/~storm/> that includes a directory of researchers working in the field of software testing.

4.1 Genetic algorithms in software testing

In this chapter we discuss previous research on applying evolutionary algorithms to software testing and software quality assistance.

4.1.1 Early works

The short history of applying genetic algorithms to software testing problems can be traced back to 1992. The earliest referenced paper is Ellis *et al.* [XESLK92], in which the testing

prototype TAGGER is introduced. The system was used for generating test data for programs written in Pascal.

Shultz *et al.* used GAs for testing behavior based control software of autonomous vehicles [SGD92, SGD93, SGD95, SGD97]. Their goal was to find a minimal set of faults that can be tolerated without significant performance loss of the control system. A chromosome represents a set of initial conditions followed by rules, which specify various fault modes that could be present in the control system. A GA was used to search for potential faults in the software. The object controller software was designed for aircraft and an autonomous underwater vehicle.

The first PhD thesis in the area was by Sthamer [Sth95] who studied the use of GA as a test data generator for structural testing. The example programs are small procedures written in ADA, including triangle classification, linear search, remainder calculation, and direct sort. Sthamer applies GA for branch, boundary, and loop testing, and also for mutation testing. He observed that "*GAs show good results in searching the input domain for the required test sets, however, other heuristic methods may be as effective, too*". "*GAs may not be the final answer to the software testing problem, but do provide an effective strategy*".

4.1.2 Coverage testing

Pei *et al.* [PGGZ94] concentrated on pathwise test data generation. By using test data generation by GA they try to define if the selected subpath is feasible or not. They compare their system with Korel [Kor76] and believe that a GA based system works better because it processes the whole path simultaneously. Pei *et al.* also claim that their system is superior when compared with many commonly used methods.

Watkins [Wat95] deals with path coverage optimization by GA, using the popular triangle classification problem as an example. GA reached the same coverage as the random method while sampling a smaller percentage of the complete search space.

Roper *et al.* [RMB+95] have also studied optimization by GA, trying to find a set of data that will test the program until the required level of coverage is met. Roper [Rop96, Rop99] has

also stated that using a GA often neatly avoids many of the problems of automatic test data generation encountered by other methods.

Smith and Fogarty [SF96] studied test coverage optimization by a hybrid version of GA and hill-climbing local search. Their application was also the triangle classification problem. They claim that their system can generate test sets that fully satisfy the given metric and reduce the size of evolved test sets. Smith *et al.* [SBF97] continue that work, by generating with a GA test programs for verifying the design of a microprocessor. The test problem is a VHDL model of hardware, the test coverage is optimized and the results are compared against the random method. The distribution of the results for a GA is significantly more skewed towards higher fitness values than for the random method.

Warfield [War98] has received a United States patent for an automatic software testing tool that generates test scripts based on state machine definitions. The system measures the code coverage that each test script achieves when fed to the tested user interface of API (Application Program Interface). Coverage metric is used as the fitness value.

Another patent has been issued to Whitten [Whi98] at Sun Microsystems for a method for selecting a set of test cases which may be used to test software program products. The set of test cases is generated by the designer in the form of software that exercises as many of the code blocks in the product as possible. A GA is applied to determine which subset of test cases to use. This is done on the basis of the fitness value using a combination of time and coverage measurements. The aim is to determine the set of test cases that exercises a maximum number of code blocks in the minimum time.

Gounares and Sikchi [GS01] at Microsoft Corporation have received a patent for a system for adaptively solving sequential problems in a target software system utilizing modified GAs. Stimuli to the target system are presented as actions, and a sequence of actions is a GA chromosome. These chromosomes are applied to the target system one action at a time and the changes in properties of the target are measured. The fitness value is defined so that successive generations of chromosomes will converge upon the desired characteristics. For software testing these characteristics are defect discovery and code coverage. Evolving ever-

shorter chromosomes that produce the same defect minimizes defects in a target software system, and the defect discovery rate is thereby maximized.

Michael *et al.* [MM98, MMS98] have developed the so-called GADGET (Genetic Algorithm Data Generation Tool) system that is fully automatic and supports all C/C++ constructs. The system is used to obtain condition/decision coverage. Michael *et al.* use triangle classification and an autopilot control program for a Boeing 737 as example problems. They also studied the performance of different GA variants and compared results with the random method — GAs gained a much higher coverage than the random method.

Pargas *et al.* [PHP99] have experimented with genetic algorithm based test data generation for statement and branch coverage using a control-dependence graph to guide optimization. They tested six relatively small test programs and compared the results to the random method. Their approach clearly outperformed the random method for three of the six test programs, for the other three programs both methods find the optimal solution in the initial population. They suggest that the use of GA could be more beneficial for complex programs.

Bueno and Jino [BJ00] have studied the possibility of using a GA to identify the potentially infeasible program paths. They propose that monitoring the progress of the GA search could identify an infeasible path. Their approach combines earlier works by other authors and introduces a new fitness function using control and data flow information to guide the search. They use the so-called "*path similarity metric*" as their fitness function. Results with their six small test programs were very promising. The GA based approach reached 100% success rate in unfeasible path identification in tenth of the amount of command executions and time needed by random search to reach 70% success rate.

4.1.3 Test data generation

Hunt [Hun95] used a GA for testing cruise control system software. In his implementation a GA chromosome represents the input and expected output. The fitness value is assigned, if the measured output differs from the expected output. The greater the difference, the higher the fitness value. The expected output is derived from the original software specification. Hunt states that software is often developed by a third party, and the tester only has the software, which he treats as a black box and tests against the corresponding requirement

specification. A GA chromosome must be able to represent all input values that the software can process, as well as the values that its single output can have. He claims that the chromosome must be able to represent both the valid and erroneous inputs. In his approach the GA is used as an aid for a human tester. The GA identifies failure scenarios, but it is up to the human tester to identify the faults that led to the failure.

Yang [Yan98, JSXE95] have written a PhD thesis about using genetic algorithms to derive test cases and test data from the formal Z specifications in order to test the functional behavior of the software. His aim was to show the conformance of the implementation to its specifications, *i.e.* the correctness of the implementation with respect to the set of test data with which it was exercised.

Minohara and Tohma [MT95] have used a GA for parameter estimation of a so-called "*hyper-geometric distribution software reliability growth model*", where the increase of the number of errors is observed as a function of time. Their GA chromosome represents a set of parameter values. The fitness value is evaluated by testing errors between the observed and the estimated test-and-debug data. They are trying to minimize the amount of errors. Their results suggest that the GA may be a more stable method to get the estimates.

Lin and Yeh [LY01] have also studied automatic test data generation by a GA for a chosen subpath. Their method uses so-called "*normalized extended Hamming distance*" to guide the optimization process and to test the optimality of the candidate solutions. Fitness function, so-called SIMILARITY defines how similar the traversed path is to the target path, is used to choose the surviving test cases. "*Optimality*" means that the test case (*i.e.* a particular input) forces the program to follow the wanted path of program statements when executed. They claim that a GA is able to significantly reduce the time required for automatic path testing.

4.1.4 Testing program dynamics

Kasik and George [KG96] have used a GA for emulating software inputs in an unexpected, but not totally random way. The GA is used as a repeatable technique for generating user events that drive conventional automated test tools, so that the system can mimic different forms of novice user behavior. The system tries to represent how a novice user learns to use an application. The fitness value is given according to how much the actions performed are

guided by the chromosome to resemble novice-like behavior. The novice behavior is described by a special reward system that is build based on observations.

Boden and Martino [BM96] used a GA to generate API tests. They concentrated on the operating system error treatment routines. The fitness function was a weighted sum of various factors of a test response with an attempt to assess the sequences of operating system calls. Boden [Bod98] has also received a US patent for an order-based GA based automated testing of software application interface, object method and command. The GA is used to search and detect symptoms of software errors by generating test sequences.

Wegener *et al.* [WGG+96, WGJ97, JW98, GW98, MW98, WBS01] have studied the search of the execution time extremes of real-time software with a GA. They have compared their results to the random testing and static analysis. Their object software has mainly been some small examples or DaimlerChrysler embedded automotive electronics software. They think that the static analysis and evolutionary testing together can effectively find the lower and upper execution time limits. They claim that there is not much support for temporal testing, and often testers just use the methods that are designed to test the logical correctness. In their research the GA based testing was much more effective than the random testing, and particularly effective when a problem has many variables and a large input domain. In their studies they measure execution times as processor cycles, so that interruptions *etc.* would not have an effect on results. A few times their GA found more extreme time than was previously known, they verified the time by analyzing the control flow graph. They also introduced the term "*evolutionary testing*", which by their definition means: "*the use of metaheuristic search methods for test case generation*".

Puschner and Nossal [PN98] have applied a GA for test data generation for testing Worst-Case Execution Times (WCET). Tests were executed in a simulation environment on a workstation and compared against random testing, best effort data generation, and static WCET analysis. The GA results compared well with the static WCET analysis, and clearly outperformed the random testing. They conclude that the GA is well suited for WCET tests, and with large input spaces the GA based method proved to be particularly favorable.

Ostrowski and Reynolds [OR99] present the implementation of the so-called Cultural Algorithms (CA) embedded with both the white and black box testing techniques. Cultural algorithms are GAs that has the so-called belief space that is used to pass the culture component, *e.g.* the acquired knowledge or accumulated experiences, from generation to generation. The idea is that the faults diagnosed by CA that does black box testing are passed to the CA that does the white box testing. The goal is automatic detection and isolation of program faults.

Pohlheim *et al.* [WSP99, Poh01] applied extensions of evolutionary algorithms (EA), called different strategies and competing subpopulations to automatic software testing. Several variants of GA are competing to each other and the best results is selected as the final solution of this technique. The object software was a DaimlerChrysler engine control software module, and the goal was to perform structure-oriented testing and temporal testing of real time software modules. The EA results were compared to the results obtained by the software developer with white box testing. The EA based automated test found always equal or even better execution times. It was observed that different EA strategies are successful with different software modules, each of the strategies were particularly successful at a specific point in time. By using competitive subpopulations with different EA strategies one can exclude unsuccessful ones.

The PhD thesis of Gross [Gro00] concentrates on temporal B/WCET testing of real-time systems. He tries to measure "*evolutionary testability*" by studying if there is a relationship between the complexity of the test objects and the quality of the outcome produced by evolutionary testing. The example programs are short routines written in C++. Gross states that complexity as it is 'seen' by the evolutionary algorithm is not much different from the way humans may experience it. This means that programs that were difficult for human analysis were also difficult for evolutionary testing.

The PhD thesis by Tracey [Tra00] deals with automatic test data generation for testing safety-critical systems. He uses simulated annealing and genetic algorithms, but also random search and hill climbing as the optimization methods. He defines the framework on how to use them for generating test data for temporal WCET testing, assertion based testing, and structural testing. It is observed that "*genetic algorithm based approaches for structural test*

data generation have a number of weaknesses that restricts their application to real software industry". On the other hand, GAs seems to be, on average, the most effective and efficient of the techniques he implemented in his work.

4.1.5 Black box testing

Bingul *et al.* [BSPZ00] apply a GA to test the war simulation software THUNDER with the black box method. They applied multiobjective optimization with the Pareto method, and define three different ways to assign fitness values. They try to optimize software behavior, war strategies, and the running time. They claim that the GA was able to provide optimal or near optimal solutions.

In addition to our own research there does not seem to be much work on black box testing applications with GA. The applications and methods we have applied are reviewed and explained in more detail in Chapter 6.

4.1.6 Software quality

Hochman *et al.* [HKAH96, HKAH97] applied a GA to optimize neural network architectural, learning, and training parameters. They call the result "*evolutionary neural networks*", and use them to detect fault-prone and not-fault-prone software modules. They compare their method against discriminant analysis to discover software reliability problems. Statistical analysis of their results seems to confirm that the performance of their approach is better.

Mansour *et al.* [ME97, BM97] applied a GA to the optimal regression-testing problem. They try to determine the minimum number of test cases for revalidating modified software in the maintenance phase. They compare the GA based method with the branch and bound (B&B) and simulated annealing (SA) based methods. The B&B method was the fastest, when testing small modules, but as the size increases, and the number of test cases grows, the GA based method becomes fastest. They conclude that in contrast to analysis-based optimization methods, the complexity of the GA and SA does not grow exponentially with module sizes. Results show that the GA and SA find an optimal or nearly optimal number of retests in a reasonable time.

Baisch and Liedge [BL97, BL98] used GAs for tailoring a fuzzy rule base of an expert system used for software quality prediction. Their object software is from a large real-time telecommunication system. The GA is used to classify the software modules into two classes: 1. few faults (<5), 2. many faults (>20). The authors discovered that additional factors, like fault history, change history, and size should be utilized. There were also several faults that cannot be predicted by the system. They claim that the proposed system helps to decrease faults by up to 50% after changes in the modules.

Evett *et al.* [EKCA98] used genetic programming (GP) approach for software quality prediction. Their system predicts the relative quality of each module, instead of a classic classification into fault-prone and not-fault-prone modules. Their GP used the size of code, degree of reuse, and faults in the previous releases to predict the number of expected faults in each module. Their target was two actual industrial softwares, a large military communication system, written in ADA, and a large telecommunication system, written in a Pascal like language. Both contain approximately 200 modules, from which they use two thirds as training data and the other third for validating the predictive accuracy of the best model developed. Their conclusion was that GP is able to generate software quality models based on data collected earlier in the development phase.

Burgess and Lefley [BL01] applied GP to the estimation of a software project effort. They use data collected from existing software projects, and generate estimation models for these with GP. The estimation is done by using data available from the specification stage. They compare their GP based method against the statistical and neural network based methods. It is concluded that while the GP and ANN (Artificial Neural Networks) are able to provide better accuracy, they require more effort for set up and training.

Aguilar-Ruiz *et al.* [ARRT01] used evolutionary algorithms to estimate software development projects. They use a software project simulator that generates a database from the software project. EA is then used to produce a set of management rules. The aim is that these rules will help the project manager to keep the project within the budget, and to reach the quality and duration targets. The EA generated rules are generated against rules generated by a commonly used C4.5 tool that uses a recursive algorithm that optimizes rules from

decision trees. The practical results seem to demonstrate that the approach using the EA finds better solutions.

Jones *et al.* have written several papers on using GA in testing [JSE95, JSXE95, JSE96, JES98, JW98, GJE99, GJE00, SJE94, WGJ97, and HJ01]. In the article [HJ01] they call this new field of software engineering research "*search-based software engineering*". They argue that software engineering is ideal for the application of metaheuristic search techniques. They also note that the search-based technique must outperform the random technique in order to be qualified as worthy of even being considered a successful application. The random method therefore provides the lowest benchmark. If the metaheuristic method does not outperform the random method, it is likely to be poorly implemented. They also expect to see a dramatic growth in the field of search-based software engineering within the next few years. They list the likely application areas and the developments that the growing research capacity will provide.

The researchers in the field of "*search-based software engineering*" usually test their own object software that is obtained from their industrial partners *etc.*, therefore comparing results is problematic. The only commonly used benchmark problem seems to be the tiny triangle classification problem. The problem is to classify whether three given numbers a, b and c (length of triangle edges) form a triangle and of which type; sharp, straight or obtuse-angled. The problem is usually used with the white box testing strategy, while the path or condition coverage is usually used as the optimization target.

Researchers that have used a GA on the software testing problem have usually reported "*good*", "*excellent*", "*positive*" or "*encouraging*" results. This might be partly due to the fact that researchers are not so willing to publish negative results. Many researchers that have earlier worked with traditional software testing methods now have focused their interest on GAs, see [Ala95a] for further references of evolutionary methods in software engineering.

4.2 VLSI testing

GAs for automatic test data generation has had great success in VLSI circuit test pattern generation [HSP94, AMTB95, OA95, LHRP96, HRP96, HRP98, KHS+97, RPGN97, BYFR00, YWR00]. The goal of VLSI testing is to find the smallest test set that could test the

circuit completely. Circuit and software testing differ from each other in that only one program instance needs to be tested while each individual piece of hardware must be fully tested thoroughly with the same test set. When the software version is acceptable it can be copied infinitely without errors and without any need to test each peace of code individually. See [Ala95b] for more references.

5 EXAMPLES OF USING GENETIC ALGORITHMS IN SOFTWARE TESTING

This chapter introduces two examples of how evolutionary algorithms can be applied to software testing. The first example also illustrates the difficulty of selecting a proper software metric, a problem that can occur even with relatively small and simple problems.

5.1 Example 1: Analyzing a faulty bubble sort routine

We give an example of software testing using genetic algorithm for automatic test data generation. The software to be tested is a simple bubble sort routine that was earlier recognized to be faulty, *i.e.* in some cases it does not sort the given input sequence correctly. Several different target functions, fitness functions, were adapted and tested in order to recognize this fault behavior and its severity. The goal was to find the most erroneous situations.

Sorting algorithms are used to arrange the given material (a sequence of n numbers) into either an increasing or decreasing order. Sorting algorithms are important, because sorting is a common task in computing. Almost all data must be sorted before analyzing it or displaying it more illustratively. The well-known bubble sort scans the list looking for consecutive items that are in the wrong order, swapping them and continuing the search after taking one step backwards. This scanning is done until the list is ordered, *i.e.* no consecutive numbers are in the wrong order. The bubble sort is a simple and efficient algorithm needing $O(n)$ steps if only a few items are out of order. If the material is originally in a more or less random order, bubble sort is slow and in the worst case, when the input is in the reverse order, $O(n^2)$ steps are needed. Actually, it does not make much sense to automate the testing of such an elementary subroutine as the bubble sort routine. However, this simple example was chosen because it can be analyzed from several points of view, making it easier to understand the method and its potential in more real software testing.

In this case we use pure black box type testing that does not require any additional tools to trace the software under execution. The target function is the number of errors in the sorted

sequence. Therefore, the fitness measure is easy to calculate, because it is simply the amount of pairs at consecutive array elements violating the ordering relation.

One of the most beneficial features of a GAs is that they can be simply applied to many different problems. This applies to software testing also. The automated testing algorithm can be easily adapted to test other subroutines: the optimization algorithm itself does not need to be modified at all. Only the fitness function and the part of the program that sends test data into the target software must be revised so that it is compatible with the interface of the new target. The bit string or array of numbers that form individuals or chromosomes in the GA, are converted into the form of tables, arrays or variables that can be used as the call parameters of to the subroutine to be tested. Hence the cost to automate testing of every new routine in a software project with a GA should be lower than for traditional methods [Kan97, Mar97, Mar98, Pet96]. The GA based approach may be best suited for the timing and stress testing that are also considered the best suited testing types for automation in general.

The GA solves optimization problems by calculating fitness values and favoring those individuals that get a good fitness value. The idea of using a GA is self-analytic in the sense that new individuals are created automatically based on fitness values without any human intervention during the test run. However, the need for manual analysis of the test data cannot totally be eliminated, and usually some statistical analysis is needed after the test runs.

The last generation of a GA contains more or less good solutions, here test cases that are difficult or have caused faults. The test cases of the last generation can be saved and rerun after the faults have been fixed, to see if the problems were really removed.

Testing can be done by the following two approaches. The first is to feed strictly proper data, and the second is to feed also improper data to the system. The second data type is useful because it shows how the program reacts to improper data. The first data tells whether there are some faulty responses to the formally correct input. The idea is to give the GA the bounds, within which the proper data should be. We do not need any additional boundary testing, because the GA should recognize if the near boundary values cause problems and find the problematic boundary by itself. Also the test run will be different each time, because

the GA reproduction operators (crossover points) and mutations (do we mutate and how) are controlled by a random number sequence.

The goal to find as many errors as possible sounds straightforward, but even in our simple example of bubble sort we can come up with several different target functions. This might even lead the tester to less illustrative (of error behavior) testing results.

The bubble sort subroutine (written in C++) is called with two integer arrays: the numbers to be sorted *A* and an index array for the final sorting order *I*. In addition the function call includes three integer parameters: the indices of the first (*i*) and the last (*j*) items to be sorted and the last index (*limit*) of *A*. The GA parameters used in this example are: population size 20, elitism 10, only uniform crossover, and mutation probability 2%.

5.1.1 Testing with improper data

Improper data could mean that we are calling the subroutine with the wrong data type, but that is sometimes prevented by the compiler (*etc.* C/C++ or Java). However, if the variables *i*, *j*, and *limit* are of the wrong data type the compiler may just make a proper conversion. When testing the routine with improper data, the occurring of runtime error should be allowed, so it cannot be compiled into the same executable as the GA tester itself. It could be a totally different executable, with which the GA communicates, or a dynamic link library or class that is called in order to execute some subroutine.

The improper calls that are easy to generate include calling with too wide limits (calling with *limit* or *j* larger than the length of the reserved array); this usually leads to a runtime error. Calling the subroutine with $j > limit$ caused a runtime error. Calling with $j < i$ the execution simply returns back from the subroutine. Test cases including improper data are difficult for an optimization-based tester. If a runtime error occurs immediately after an improper input, it is difficult to formulate a proper fitness function. The GA needs some properties to build on and increase in order to optimize the problem.

The use of GAs is more straightforward when searching for errors with formally proper test data, *i.e.* when the given input is formally correct and we try to find whether the tested program might handle the correct input in a wrong way. It should be noted that a runtime

error can occur with formally correct input also, but then we have probably encountered an error of different type.

5.1.2 Error is detected if a greater number follows a smaller one

In order to find the maximum number of sorting errors that the tested routine causes for an input sequence we need a target function. The first target function (1) is

$$\sum_{i}^{I-1}(A[I[i]] < A[I[i+1]]) \tag{1}$$

where we have used the C/C++ conversion that the comparison operator returns either to 0 or 1. This target function has the advantage that we do not need to know the right order, against which to compare the results.

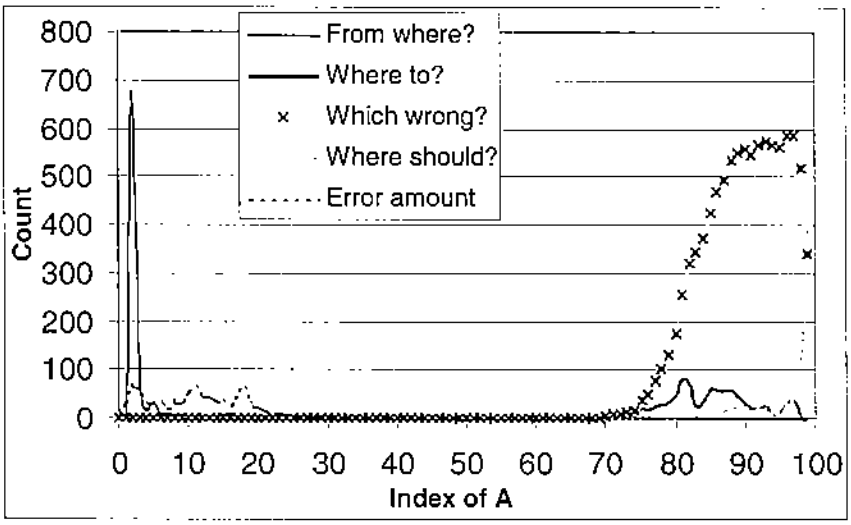


Figure 2. Error histograms with GA optimization with the fitness function (1), counting the number of elements sorted incorrectly.

Figure 2 represents the test results with function (1), when sorting a fixed length array of 100 integers. Testing was usually done with free length arrays, but for practical reasons it is easier to visualize the behavior of the erroneously sorted items using fixed length representation. In Figure 2 *From where* denotes the index where the erroneously sorted item originally was in the input vector, *Where to?* tells where it was located in the output vector after the sorting operation. *Where should?* indicates the correct position and *Error amount* tells distance from the right position. *Which wrong* tells us in which indexes the erroneously

sorted items have been detected (see fitness function 2 below). The y-axis gives the corresponding count of cases.

The histograms tell that the faulty items were usually originally at the beginning of the array, while they should be near the end in the sorted array. Actually, if we look at the target function, we note that this is expected, because we are only looking for small numbers ahead of a greater number, so the formulation seems to be inadequate to reveal the worst possible case.

Let us consider the possible sorting errors. This example shows 10 numbers {0, 1, 2, ..., 9}, their correct order and three possible error types.

Table 3. Sorting error cases.

Correct order:	9 8 7 6 5 4 3 2 1 0
Error case 1:	9 8 7 **4** 6 5 3 **1** 2 0
Error case 2:	9 8 7 <u>4</u> **3** 6 5 2 1 0
Error case 3:	9 8 7 3 **<u>6</u>** **<u>5</u>** **<u>4</u>** 2 1 0

Table 3 shows the possible types of sorting error. Error case 1 represents the errors (marked **bold**) that our first target function detects, *i.e.* smaller numbers ahead of greater ones. Error case 2 has two errors of which the other (<u>underlined</u>) is not detected by the first target function. This is because the erroneous numbers are correctly ordered with respect to each other, but they are in a wrong position. For error case 3 the definition of error is changed so that we count as erroneously sorted all the numbers that are not in their exactly right positions. Thus if a smaller number is too early (**bold**) in the sequence, it pushes all the other numbers further back (**<u>underlined</u>**) into wrong position with respect to the correct order.

5.1.3 Error is detected if the number is not in exactly the correct place

We can formulate a different model for the fitness function (2), if we count as erroneously sorted also those numbers that are incorrectly pushed back one place, because one smaller number has jumped ahead of them. Then there is an error if

$$\sum_{i}^{j} (A[I[i]] \neq A[I'[i]]) \qquad (2)$$

where $A[I[i]]$ represents the output sequence and $A[I'[i]]$ the right order. This model also detects the type 2 and 3 errors. However, for this target function we need the correct order to compare with, which may in the real world software testing case be too strict a prerequisite. Optimizing with this fitness function formulation leads to the results represented in Figure 3.

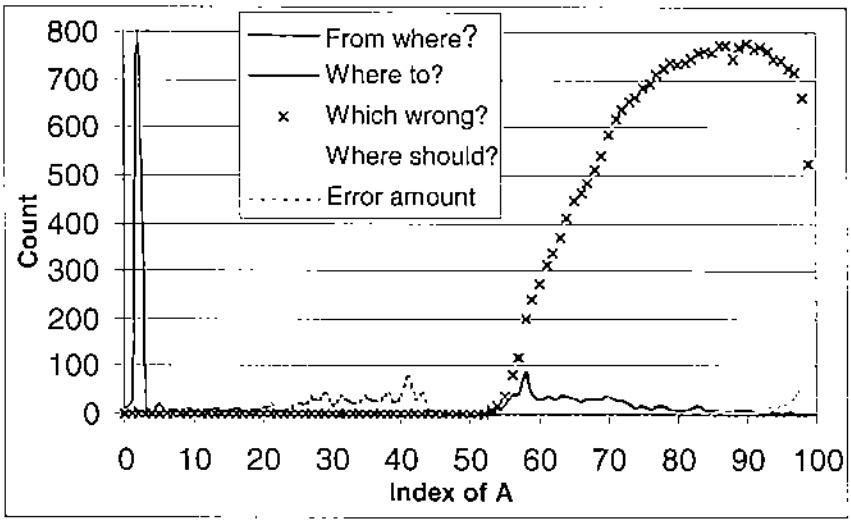


Figure 3. Error histograms with GA optimization with (2), for denotations see after Figure 2.

What is notable in Figure 3 is that the ranges of indexes, where errors have been detected are moved towards the beginning. The *Error amount* curve that tells the one error caught by the first target function can now be sorted much closer to the beginning of the outcome array. Obviously that is also what this target function aims for the optimization to do. Most items will be out of order, if one small item is near the beginning of the output array, then the subsequent items are pushed one position to the right.

This target function was designed to find out how badly things can go wrong, in other words, what is the highest percentage of wrongly sorted items that a sequence can cause. This target function can be used with either fixed length arrays, when we optimize the amount of faulty sorted items in an array, or we can optimize with freely changing array lengths, when the target function is proportional: amount of errors divided by the length of the array. For the latter case we found the worst case to be 87.5% (7 of 8) of the items being sorted in the wrong place.

5.1.4 Finding the shortest array being sorted as faulty

The third target function was designed to find the shortest array that is sorted faulty. The fitness formula is simply minimizing the array length while keeping sorting errors from occurring. Finding the shortest possible array that causes an improperly sorted output is also handy for a programmer, when (s)he later tries manually to find where the tested routine made the error. The less data you need to handle in order to reveal the error the better.

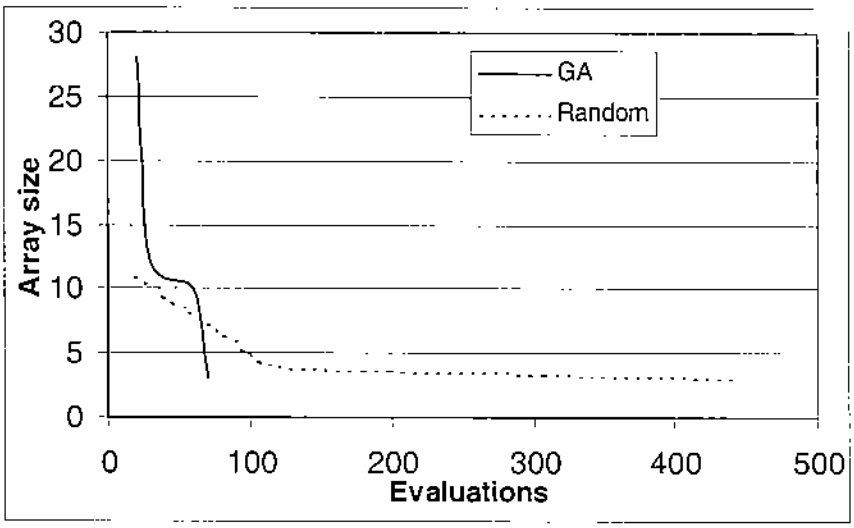


Figure 4. The development of the shortest array being sorted as faulty.

Figure 4 shows that an array of 3 numbers was the shortest array found to be sorted faulty. In this case, both the GA and the random search reach the same result, however test runs have large variations; so the figure shows only one example. It seems that the GA is able to find the minimal array consisting of only three numbers using much less evaluations than random search.

5.1.5 Comparing the GA approach to the random testing

The GA approach is often compared to random testing; this might be partly because of the "*no free lunch theorem*" [WM95] and the skeptics that claim that these methods are not any better than a good or educated guess or random search. The no free lunch theorem says that no method is better than any other method, if compared over the space of all possible problems. Notice that the space of all problems contains all the known problems like TSP,

SAT and a super astronomical number of other problems that can be created using basic combinatorial operations. Hence, in practice most of the problems in this meta set are irrelevant for practical purposes. In any case the random testing represents the lowest benchmark limit to compare with. If the GA is unable to outperform the random method with some problem, there is no sense to use it for that problem.

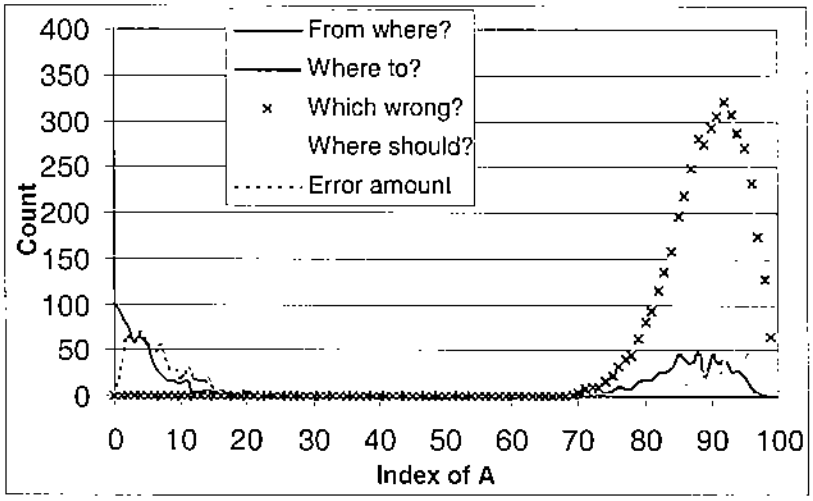


Figure 5. Error histogram with random testing, using target functions (1) and (2).

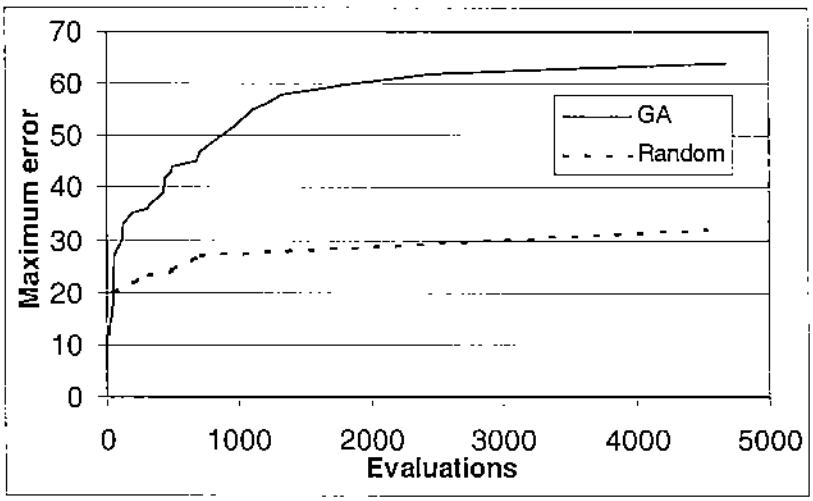


Figure 6. Evolution of fitness with the GA and random testing, using target function (2).

Figure 5 shows the error histograms for the random testing method. The target function formulation makes no difference with the random testing method, so the curves are practically the same with the target functions (1) and (2). Here we also notice, that the error

situation seems to be the case, where a number from the beginning of the array has not been sorted as near the end as it should be. That wrongly placed number is pushing others from their position and causing the other errors.

Figure 6 compares how the best fitness function values develop with the number of evaluations when using the GA optimization and the random testing with the target function (2). The GA result seems to develop logarithmically, while random testing finds more cases only slowly and obviously it would take a large amount of evaluations before it reaches the same error detection rate as the genetic algorithm.

Table 4. Number of erroneously sorted arrays of 1000 test cases generated, using target function (1).

	Sorted wrong	
Array size	Random	GA
3	165	867
10	416	806
100	592	803

We generated 1000 random permutations of 3, 10 and 100 items, fed them to the sorting program, and noted the number of items in output not in a decreasing order, see Table 4. We also let GA to optimize 1000 times each of these test array cases. The numbers in Table 4 shows the means of 10 random or GA runs with at each array size. Table 4 further indicates that when creating arrays randomly, the length of the array greatly affects the sorting correctness. With GAs there seems to be a negative correlation. This might be due to a less uniform population with larger arrays or due to the fact that optimization becomes much harder when the problem size grows. The reason is partly due to the selection procedure in the GA that reduces diversity more quickly with shorter chromosomes. A GA creates a much higher number of faulty sorted arrays than the corresponding random method for all tested array sizes.

5.1.6 Concluding remarks

After the above analysis we still do not know anything about the internal structure of the tested program (black box testing) and what code lines caused it to malfunction. However,

this is not necessarily the tester's problem because the higher task is to test the given software as well as possible and report as clearly as possible to the programmer what has been found. Software developers can then do the searching, inspection, and fixing of the erroneous code lines. As complete and accurate error reporting as possible, essentially facilitates the code-repairing phase.

It is difficult to state which of the fitness functions is generally the "*best*", because they are defined for different purposes. The recommendation is that one considers what one wants to find from the software and then choose the optimized software metric based on deliberation.

In this example a rather simple routine was tested using both GA and random methods. The example demonstrates how to use a GA in black box testing. When optimizing with GA, it is advantageous to record how the best fitness function value develops. It is also recommended to store the test data and results. These can then easily be transformed to *e.g.* spread sheet figures that help the tester to analyze and visualize test results and discover the reasons for the errors. In this case such recorded the following testing history data: in which indices errors have been detected and also in which locations the erroneously sorted items were originally. For more about this example, see [AM00b]. Other researchers have worked with generating sorting algorithms with evolutionary methods [Kin93, KBH+97].

5.2 Example 2: Test images for halftoning methods

This example introduces a GA for automatic test image generation to test the quality of an image processing software. The goal was to reveal, whether genetic algorithm is able to generate images that are difficult for the object software to halftone, *i.e.* to find if some prominent characteristics of the original image disappear or ghost features appear due to the halftoning process.

There does not seem to be much research in the field of test image evaluation. There are many open questions to be solved. How does one determine a good test image? What are the essential characteristics of a good test image? How do we determine that a particular image is good for testing some specific image-processing algorithm? Researchers rely usually on commonly used and very limited test image sets. We encountered this problem, when we wanted to test the image-processing system we implemented for an ink jet marking machine

[AMP98, AMP99b]. In this study genetic algorithms are used for software testing purposes. The principle of using a GA for generating test images is rather similar to the use of GA to generate software test data.

This example is a continuation to that given in paper IV [MA00], and a prelude to paper VII [MA03a]. The preliminary results in the study [MA00] show that the background colour of the test images was most susceptible to the distortions. We therefore concentrate on finding out how dominant the background colour is. In addition, we represent a new implementation that has no longer just one parameter for determining the background color for each image, but several parameters that define background segments and their colors.

The genetic algorithm in this study was written in Java. One of the advantages of Java is its easiness to use image handling procedures. However, the execution speed of Java may not be the best possible.

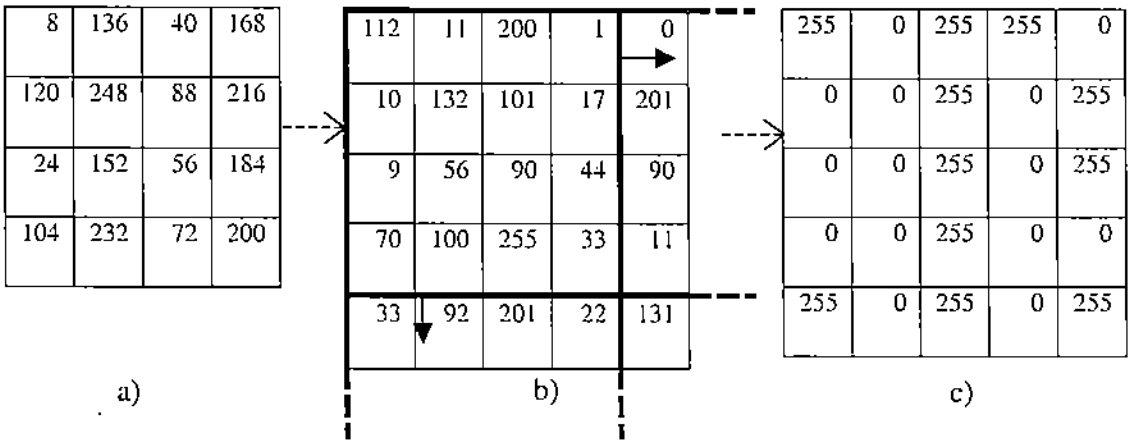


Figure 7. Halftoning process with threshold matrix.
a) Threshold matrix, b) gray image, c) halftoned black and white image.

Digital halftoning [Kan99], or dithering, is a method used to convert continuous tone images into images with a limited number of tones, usually only two: black and white. The main problem is to halftone so that the bi-level output image does not contain artifacts, such as alias, moiré, lines or clusters, caused by dot placement [Bar99]. The average density of the halftoned dot pattern should interpolate as precisely to the original image pixel values as possible. Evolutionary methods have been applied to generated halftoning patterns *e.g.* in [KS96, NB97], for more references see bibliography [Ala95c].

Dithering methods include static methods, where each pixel is compared to a threshold value that is obtained *e.g.* from a threshold matrix, generated randomly. An other possibility is to use a static median value as the threshold. Depending on the matrix this method can create both frequency (size of the pixel size is static, but the distance between pixels vary) or amplitude (the distance between dots are static, but the size of these dots vary) modulated halftones. There are also error diffusion methods, such as Floyd-Steinberg and Jarvis-Judge-Ninke coefficients [Kan99]. In these methods the rounding error of the current pixel is spread into those neighboring pixels for which the bi-level value has not yet been determined.

Figure 7 shows an example of halftoning with a threshold matrix. The matrix is placed over gray-value image matrix, and each pixel is compared with the corresponding threshold value. If a pixel value is higher than the threshold it gets the value 255 (white), and otherwise 0 (black). The threshold matrix is then moved forward one matrix width and the comparison is repeated. Figure 8 shows the idea of error diffusion dithering. X marks the pixel to be dithered. If its value is higher than 127 we assign 255 for it, and zero otherwise. The rounding error is then spread to the neighboring pixels proportionally to the error diffusion coefficients. When the current pixel has been halftoned we proceed to the next pixel.

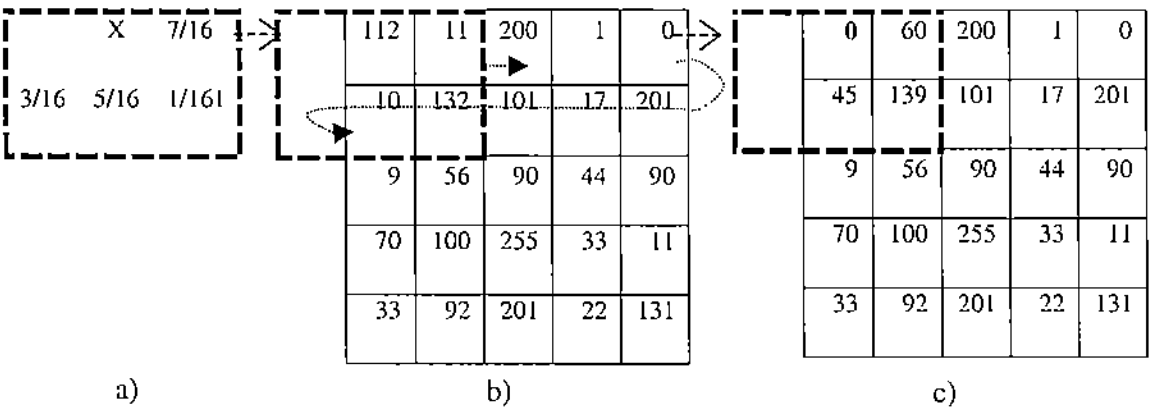


Figure 8. Halftoning process with error diffusion method.
a) Error diffusion coefficients, b) gray tone image, c) image b after dithering the first pixel.

This example concentrates only on frequency modulated halftoning methods. The three halftoning methods used here are Floyd-Steinberg (FS) and Jarvis-Judge-Ninke (JJN) error diffusions and thresholding (TH) with a 16×16 ordered threshold matrix [Kan99].

To compare a dithered image with the original one is obviously a challenging problem. One cannot simply use pixel by pixel comparison, since dithered images usually have only two tones. The minimum difference by that measure would be achieved if every gray tone was rounded to the nearest tone (black or white), which unfortunately usually results in a poor image. Better image comparison methods have been developed [Kan99, Nil99]. One alternative is to sum the pixel values from the corresponding areas ($n \times n$ window) over the images to see if the average gray tones have been preserved. With this method we can compare the images directly.

Also a set of methods called inverse halftoning [Kan99] have been developed. From these the most common seems to be the low pass filtering method, in which images are first low pass filtered and then the resulting images are compared pixel by pixel. The problem with lowpass filtering is that the high frequencies will disappear and the images get a somewhat blurred overall appearance. However, this method is easy to implement and it enables pixel by pixel comparison. In a way the blurring by low pass filtering also resembles human eye perception: when we look at the image from a distance the small details disappear and the visual observation of larger objects is averaged out from the small details. Sullivan *et al.* [SMR91] have developed a low pass filter model that is based on the human eye transform function.

Line by line comparison of the difference between the current and the previous pixel was also tried. This method has been introduced in [EG99]. For example if we take a chess board and the mirror image of it, and compare the difference between consecutive pixels the images the result is that the two images are almost identical. The result is totally the opposite when comparing the images pixel-wise. If the images are not compared properly the received divergence value between images may as well depend on a comparison used as the actual difference between the images or the dithering method i used.

Several fitness functions *i.e.* image comparison methods were tested. In this example we used the average density at the corresponding image areas (SW), pixel by pixel comparison using low pass filtered images (LP), and the tone difference between consecutive pixels (LS). The fourth method used is a hybrid (HYB) of the three previous methods. The hope was that it would inherit the advantages of all three but not their shortcomings. The hybrid was

generated by first running each of those three methods (index *i*) individually five times (index *j*) and then calculating the fitness gain from the best value F_{j0} from the first population to the last F_{jN}. The three methods were then weighted by w_i so that the gain/best result proportion of all three methods was equal (3).

$$w_i = \frac{5}{3 \times \sum_{j=1}^{5} \left(F_{ij_N} - F_{ij_0} \right)} \tag{3}$$

5.2.1 The Implementation of the proposed system

The GA runs as an independent program and optimizes a set of parameter vectors which are used by an image generator to create images. These images are then sent to the object software, that halftones them and returns the resulting images to a pixelgrapper program. The pixelgrapper reads pixels from both the test image and its halftoned transformation image and sends an 8 bit pixel arrays of both images to the fitness function evaluator of the GA which calculates the sum of differences. GA generates the new parameter vectors by using crossover and mutation, favoring those parent chromosomes that previously had gotten a high fitness value.

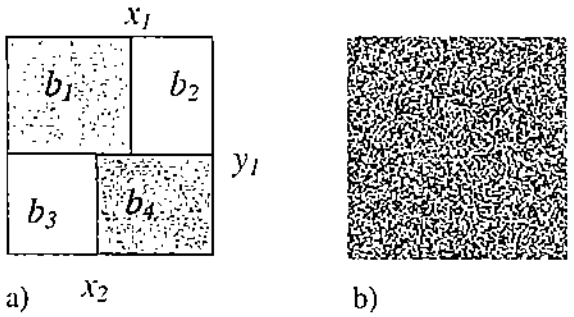


Figure 9. An example of how the background and noise for the synthetic test images were constructed. a) Background segments, b) chaotic data to be added on image a.

Test images in this example are created by optimizing parameters, such as place, size and color of elementary graphical objects, like lines, rectangles, circles and fonts (ASCII characters), together with the background tiles and colour. All these objects are encoded as a GA chromosome. This kind of artificial images was chosen on the basis of our application of testing the dithering methods aimed for company logos.

In paper IV the images contained five lines, one rectangle, one circle and two characters. That coding was rather inflexible and resulted in monotonic images. It required the chromosome length of 50 bytes (1 byte for background color, 5 bytes per each line, rectangle and ellipse, and 7 bytes for each character). The population size was 50, elitism 50%, a total 550 evaluations (initial population + 20 generations) were done, uniform crossover [Sys89] was used and the mutation rate was 2%.

The quite complex chromosome consisted of a total of 79 parameters. From those the first 7 parameters were for background, three of them (x_1, x_2, and y_1 showed in fig. 9a) divide the background into four segments and the other parameters (b_1, b_2, b_3, and b_4) determine the tone of each background segment. This way, one parameter does not dominate optimization. However, the background might still become monotone if one sector takes the whole space or the tone parameters b_i are equal.

The next 70 parameters were divided into 10 groups of 7 parameters, each group defines an elementary image object in the following way:

1. Image object (line, rectangle, oval, ASCII character); for characters also the font style,
2. color,
3. x-coordinate of the starting point,
4. y-coordinate of the starting point,
5. length in x-coordinate direction or character font size,
6. length in y-coordinate direction or character font type,
7. not used or the character value (only printable ASCII characters were used).

All objects are opaque and may cover earlier created objects, background is created first and then the other objects on it.

The images generated this way were still quite monotonous. One reason for this is that a natural image usually has more variation between neighbouring pixels. Hence, the test images were further diversified by adding chaotic data (see fig. 9b) with the Verhulst [Add97] logistic equation:

$$x_i = a \times x_{i-1} \times (1 - x_{i-1}) \tag{4}$$

The chaotic data was used rather than white noise in order to control the noise diversity and to keep the added noise repeatable. The last two parameters of the chromosome consist of a 16-bit value a for the Verhulst function that was scaled to be a decimal number in the range [2, 4]. The optimization process usually favoured chaos parameters that generated striped patterns rather than patterns that resemble white noise (fig. 9b). The size of the generated image was 256×256 pixels, so that the values of most parameters fit into 8 bits. The population size was 50, elitism was 40%, in total 3050 evaluations (initial population + 100 generations) were done, uniform crossover was used, and the mutation rate was 1%. The transparent implementation of images by using a 98 parameter long chromosome was also tested, however, the transparent objects did not seem to bring anything new to the results, so the transparent implementation is not discussed further in this example.

5.2.2 Experimental results

The results were generated by running five test runs with each dithering method and a comparison method, so there were altogether 12 dithering/comparison method pairs and in total 60 GA test runs. Each dithering/comparison method pair was also tested by randomly generated test images. The number of random images was equal to that of the GA.

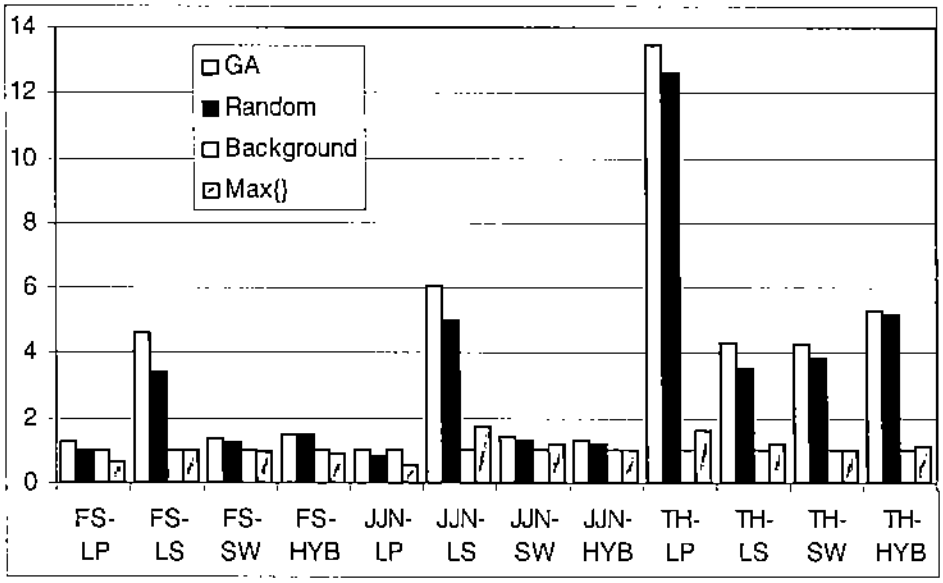


Figure 10. Comparison of the best values with different test image sets and dithering/comparison method pairs. The fitness values are normalized so that the best Background value for each pair is exactly one.

In paper IV the tests runs with different dithering methods and image comparison method tend to produce images with nearly the same dominating background tone *b*. It is therefore interesting to see if the background tone in fact explains the whole result.

The results of GA optimization were compared to the results obtained by three other methods: 1) with the randomly generated test images, 2) testing all possible monotone images with tones between 0 and 255, 3) images from the commonly used, "*standard*", test image set {Lena, Bird, Boat, Goldhill, Mandrill, Peppers} available *e.g.* in <http://sipi.usc.edu/services/database/Database.html>.

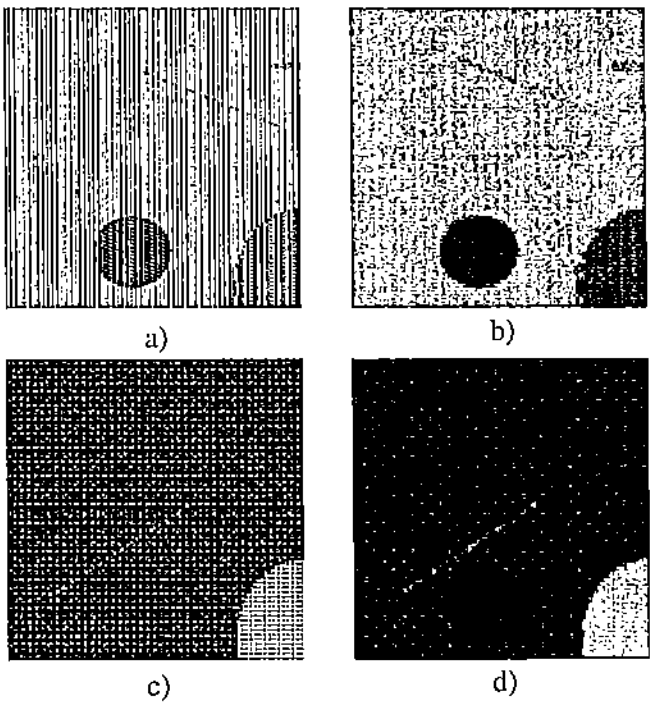


Figure 11. Test image found by GA causing the highest difference between the original and dithered images. a) The best solution for TH-LP, b) low pass filtered image a, c) dithered image a, and d) low pass filtered image c.

Figure 10 shows a comparison of the best values obtained by using the test images generated by GA (GA), the random method (Random), testing all possible monotone images (Background) or the test image set (Max{}). The figure collects the results from all dithering methods used in this study combined with all comparison methods, altogether 12 halftoning/comparison method pairs. The best value found with each test set is divided by the

best value obtained with the monotone images (Background), so the best value obtained with the background test set is exactly 1 in each pair

From Figure 10 we can see that for each of the 12 dithering/comparison method pairs the GA has reached the best fitness value for each item of the four image sets. The Random method has usually generated nearly as good solutions as the GA. Six times the GA and the Random method generated images that have gotten a considerably higher fitness value than the monotonic images (Background) or images from the standard test set (Max{}). The results confirm that the background tone does not explain the whole difference, but that also the other objects in the image are important for explaining the high fitness result.

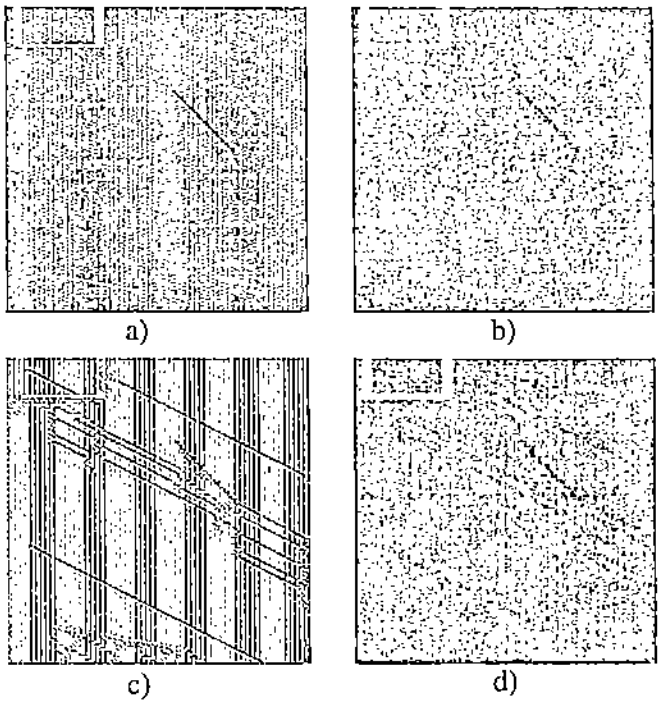


Figure 12. An example of a shadow image that appears during the halftoning.
a) best solution for JJN-LS, b) low pass filtered image a,
c) dithered image a, and d) low pass filtered image c.

The biggest difference between the images generated by the GA and the standard test images was obtained for the combination of using the threshold matrix and low pass filtering (TH-LP). The big difference in this particular case seems to be due to the fact that the GA is able to find weaknesses from the ordered dithering matrix and it generates image patterns that result in considerable differences between the compared images.

Figure 11 shows an example of how the difference in TH-LP is composed. The GA has found such chaos parameters that result in vertical stripes. These stripes happen to be in such a position that when dithered by the ordered threshold matrix the background becomes considerably darker. This phenomena is possible because the ordered dispersed dithering threshold matrix based on Bayer's [Bay73] original model has different threshold value sums in vertical colums, but the row sums are equal. If we are able generate striped patterns that fits perfectly to these differences, we are able to get either much darker or much lighter toned vertical lines than in the original image. Interestingly also the circle in the low right corner has become much lighter. The dark circle has thus totally disappeared into the background. The left hand side images are the original and the dithered, while the right hand side images are their low pass filtered forms.

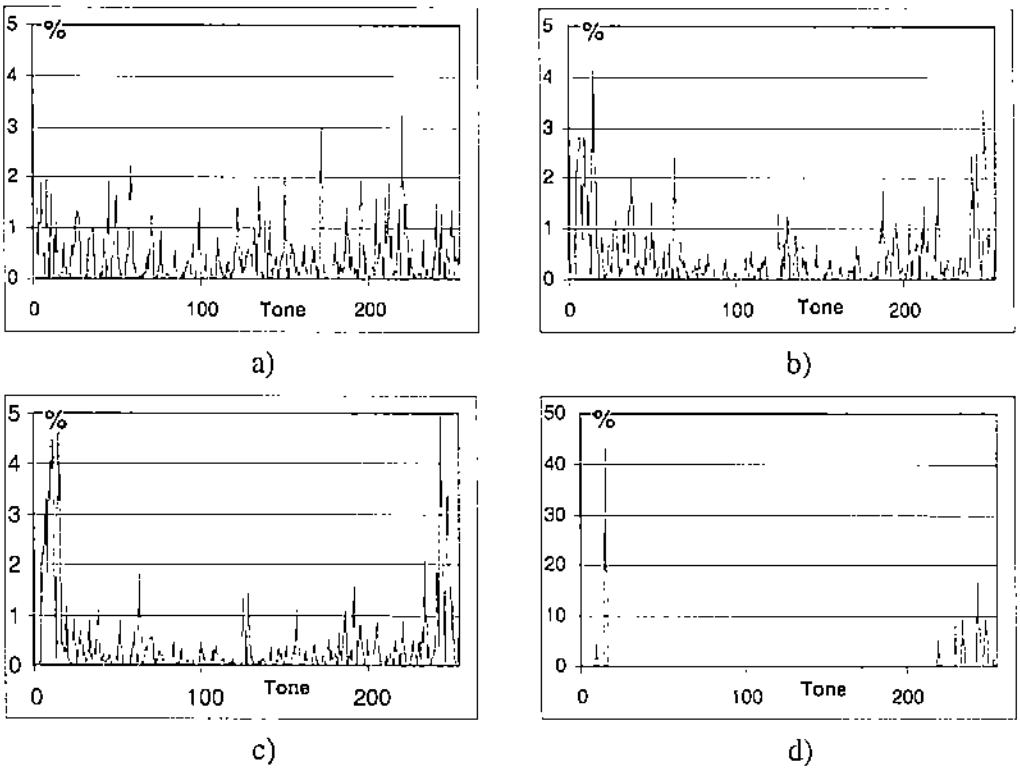


Figure 13. Gray level histogram development during the optimization process
a) initial population, b) 10th generation population,
c) final population, and d) the best solution found in 100 generations.
Note the different scale of the y-axis in d.

Figure 12 shows an example of the typical shadow images that dithering generates. Figure 12 is from the JJN-LS pair that generates the second largest fitness difference between the GA optimized images and the standard test images

Figures 13a-c show an example of how the gray level histograms develop during the optimization run. The gray tones that generate a higher difference between the original and the dithered image are rapidly increasing. In the original population tones are fairly randomly distributed (fig. 13a). In the last generation tones are clustered around the dominating tones 8 and 246 (fig. 13d).

5.2.3 Concluding remarks

The results seem to confirm that the GA is capable of generating high quality test images for halftoning methods. Either some features of the original image disappear or some objects appear. The changes were perceived either by comparing the original and the dithered image, or by comparing low pass filtered versions of the images. The preliminary results of this work show that the background tone is by far the most significant factor when testing the dithering methods. However, background tone is clearly not the only factor. This study confirms that objects in the background usually generate a larger difference than the solid one tone image.

Statistical analysis of the generated image parameters should be done in order to fully determine possible correlating parameters. In this example the conclusions were drawn by observing the images and analyzing only a few variables. After a satisfying fitness function has been found, the obvious application of the above testing method is automatic design of dithering methods. Then, a GA is devoted to generate halftone filters while another GA tries to create the hardest test image for each filter candidate. The best filter being the one where the hardest test image is closest to the original after dithering. That approach is implemented in paper VII. In general this kind of GA based approach could be used in the design and testing of demanding software.

6 INTRODUCTION TO THE ORIGINAL PUBLICATIONS

This chapter introduces the seven papers written over a five-year period. The papers have been published in a scientific journal (II) and conference proceedings (I, III—VI). The last (VII) has been submitted for publication.

The papers include references to many application areas: automation field buses (CAN and LON), Bezier curves, electric network protection relays, ESIM simulator, fitness landscape, image processing methods, the polyomino problem, structured light vision, *etc.* that are not explained in detail in the introductory part of the thesis. For more details please see the original paper and references therein.

6.1 *Paper I: Experiments with temporal target functions*

The paper introduces a temporal fitness function. The experiments reported in this paper were performed in 1998–99 during the process of testing a protection relay software, but the results were published later. The first part of the paper illustrates the risk associated with a small population size. In many reports where good and fast results have been received with a small population size, the results are the best of several test runs. What is often omitted is the information on how many times the run did not reach any reasonable solution. This leads to the consideration of optimum population size, which is the compromise between small population size, which may lead to good results faster, if we are lucky, and a larger population size that usually leads to good results more slowly but more reliably, see [Ala92, Ala99]. The results showed that GA optimization results are fairly insensitive to its parameters other than the population size. This emphasizes the risk of not finding the optimal area with too small a population size.

The second part of the paper concentrates on a popular method of tuning GA parameters with another GA. This time this kind of meta-GA system is built so that the only fitness information that the higher level GA receives from the lower level GA is its execution time, *i.e.* how long the lower level GA needs to optimize the given task with the GA parameter set given to it as input from the higher level GA. This construction also simulates in a way the

testing of real-time software. With a certain input domain the execution time of the system under test is faster than with others. Even with the same input domain, the system does not spend an equal amount of time, because the GA operation is highly nondeterministic. The old individuals are retested in the new generation, and they usually then receive a different fitness value from that of the earlier evaluations.

This kind of simulation was done in order to see if our GA can handle the temporal optimization with some other, maybe even more nondeterministic target function than the protection relay software. The results seem to confirm that GA is capable of handling stochastic or nondeterministic target functions, and is able to find "*good*" or "*bad*" parameter combinations that cause long or short response times. The optimization of these times corresponds to the W/BCET optimization of the real-time software. These results indicate that the GA has a good chance of being applicable for temporal black box testing of large embedded relay software.

6.2 **Paper II:** *Searching protection relay response time extremes using genetic algorithm*

This paper was originally published as [AMMM97] and the version included in this thesis was later published slightly revised as [AMMM98]. The object software under testing in this study is a large embedded software system, designed for a protection relay of an electrical network. The product was not yet on the market, when this study was done, thus the software was still under development, and continually changing.

The goal was to find response time extremes of that software, when the input domain consisted of CAN and LON communication messages, and digital measurement inputs generated with the genetic algorithm. The software did not have other means to communicate with the outside world, since the interface and other cards in the system communicated via CAN messages.

The testing was done using the pure black box testing scheme: no code was traced, only the communication that the software made with the outside world and the timing of those communications. The only fitness value GA received was the response time, the time which

the software needed to reply to the input values generated by GA. See the Lic.Sc. theses [Man99, Mog99] for a more detailed report of this research.

The results indicate that GA is capable of finding parameter combinations that cause responses with long delays. In comparison to the random tests, the GA generates many more test cases, where the deadlines of responses are violated with a great amount. However, Table 1 shows that the longest response time found in this test series did occur with the random testing. We must emphasize that the tested system was highly nondeterministic and that one extremely high response time with the random testing was probably due to some inherent event, and the repetition of that input domain did not give as high response time again. The author's Lic.Sc. thesis [Man99] shows how the response time of the worst test case varied when fed repeatedly, due to the nondeterministic nature of the target system.

The goal of this research was not to find the one extreme response time, but some characteristics of the input domain that did cause long response times. GA succeeded to reach this goal, because it generated considerably more test cases in the high end of the response time scale than the random testing did.

6.3 *Paper III:* Automatic software testing by genetic algorithm optimization, a case study

The paper is a continuation of the previous paper dealing with black box testing of the relay software communication module and the temporal testing. This paper analyses how the lower priority LON messages interfere with the higher priority CAN message handling.

When this study was done the protection relay product had just came onto the market and its software was close to the first production version. The object software was substantially larger and more versatile than in the previous study. These facts were taken into consideration when the tested features were expanded to cover a larger part of the software. In the previous paper we could not implement the whole software due to several problems related to platform dependent reasons and therefore a kind of mutilated software version was used. In this study the tested software version was more elegantly borderlined and interfaced from the whole relay software. Due to this it was impossible to include a lot of hardware specific functions of the relay software the test system. We also tried to classify the possible

situations into a larger number of groups, in order to get more information on the new software version.

The results in this and the previous paper show that a GA is able to learn input domain values that lead to longer response times than just by using the random testing. It seems that after finding the optimal search area, the GA focuses on it and is able to find those more detailed small parameter changes that lead to the final fitness peaks. The GA did find the time-slide when the sending of the LON message had the strongest effects on the response time of CAN message handling. The findings in this paper further show that the GA is applicable for temporal black box testing of a nondeterministic system. From the results we were also able to assess on what effect the messages, and their timings had on each other.

6.4 ***Paper IV:*** *Automatic image generation by genetic algorithms for testing halftoning methods*

This paper was our first attempt to expand the GA based testing system to a wider application area. The field of image processing was selected, because we worked on that field in our other research [AMP98, AMP99a, AMP99b, Pyy99], where we generated halftoning filters with GA, and we wanted to link that research to the study of GA based software testing.

Test image generation with GAs was inviting also because there is little research done on this subject in general. GAs has been applied to image generation earlier [Sim91], but not for testing purposes. We felt that there is no reason, why a GA could not be used to generate synthetic images. Our idea was to use these images for testing image processing systems, algorithms and software.

Usually, research studies in the image processing area use the same limited set of test images. No thorough research has been done on test images to define what makes some test images good, and what characteristics they should possess to be appropriate for testing a particular image processing method or algorithm. However, it is not within the scope of this thesis to find answers to these difficult questions, but keep our focus on test data generation.

The principle of using a GA for generating synthetic test images resembles software test data generation. A GA generates a test case, an array of parameters, that is converted into a test image (in this study) or input domain (software testing) and sends it to the tested software.

The environment then has the measurement system, that in this case compares the original test image and the result image after it has been processed by the image processing software.

The first implementation of test image generation by a GA introduced in this paper was simple and it was later extended [MA01a]. However, we include it here because it was the first one on the subject, and it represents the original idea and shows early results. In addition the paper introduces the basic idea in detail and points out the relation to software testing.

The preliminary results in this paper show that synthetic test images generated by a GA can reveal image characteristics that can be problematic for digital halftoning algorithm/software to reproduce. In most cases the images causing the highest difference value in each test run with the same halftoning method/image comparison system pair were fairly similar. This finding does hint that the problematic test images have some properties in common that can be recognized or found with GA optimization. The research introduced in this paper is further developed in Chapter 5.2 (example 2) and paper VII included in this thesis.

6.5 ***Paper V:*** *Testing a structural light vision software by genetic algorithms – estimating the worst case behavior of volume measurement*

This paper is a further expansion of the GA based testing method. The GA is used for generating simulated 3-D test surfaces in order to determine the error bounds of the proposed 3-D vision measurement software. Structured light vision software was chosen because it is illustrative, visual, and easy to understand operation of an illumination based profile measurement method.

The idea was to generate simulated test surfaces with a GA so that the measurement error of the structured light vision software is maximal. In this way one can draw conclusions about which kind of surfaces cause the largest measurement error, and also the error bounds of the proposed measurement system and software.

Many factors influence the measurement error, and therefore it is not easy to determine the maximum error. In a real-life application there are many factors that affect the measurement error. These factors include accuracy of the camera used, illumination, reflections, imaging or illumination angles, shadows and hidden shapes. In our simulated version many real-life

problems were simplified, but still the accurate calculation of measurement error remains a difficult task. There were also some restrictions on the shape being measured, because the simulated shapes must at least to some extent correspond to the real-life problem at hand [Rau00]. The focus of interest was not on the measurement error bounds of all possible shapes, but only the shapes that fulfill the restrictions fixed *a priori*.

The measurements were also performed in order to collect data on the accuracy and speed of the proposed 3-D measurement method. This was done in order to see if it is applicable to our real-life problem. The results show that accuracy and speed are linked together, by using a more accurate camera and more imaging angles the accuracy improves, but the processing speed greatly increases.

The results show that a GA is capable of generating test surfaces that extend the error bounds. The GA does find parameters that cause the generated test surfaces to have characteristics that make them difficult for the measurement software. We cannot say how far or close to the largest possible error bounds are reached, but we know that the real-life objects that are to be measured with the system are not as varied, so the error bounds found with the GA are most probably higher than those expected within the real-life application.

This research also revealed that the proposed measurement software has a problem with combining surface height matrices obtained from several directional scans. This was obvious because it was most accurate with scans from two directions, and the GA testing was able to find test surfaces that were measured with higher relative error when using three or four scan directions.

6.6 **Paper VI:** *Developing and testing structural light vision software by co-evolutionary genetic algorithm*

The paper is a continuation of the previous one. The goal was to develop the measurement software, so that the increase of scanning directions increases the accuracy. The reason for this is that they provide more height information (see the problem explained in the previous chapter). The emphasis was on the idea of developing the software simultaneously with the testing. In order to do that, the object software must have a number of parameters that we can change.

The hypothesis here was that if we can tune software parameters simultaneously with the testing, we can use a co-evolutionary GA that generates the worst test case for the current population of software versions (different parameter setups), and at the same time it optimizes the parameter setups, so that the best current individuals of the software population handles the worst test cases as well as possible.

In order to test that hypothesis, the software is changed so that a dynamic rule base controlled the method of combining different directional heights got from several directional scans. The co-evolutionary GA that simultaneously tried to generate harder test cases and better rule combinations optimized the rule base. In order to handle the test cases as well as possible. We thought that the more dynamic parameters we provide, the more accurate the system should get, and the results confirm this natural presumption.

In this paper random walks and simple fitness landscape analyses [Wei90] were used in order to analyze how many steps away the optimal peak is from the normal optimization landscape. These tests were preliminary and such analyzing methods need further consideration and study. On the basis of this study, it seems that the optimal peak is closer to the normal landscape in the optimized system than in the non-optimized one. Therefore, the optimized system is more sensitive to the parameter changes. The optimal area is more narrow in the optimized system and the random test cases cause less significant accuracy errors, hence the test cases that cause large errors are more rare and extreme.

The results also showed that co-evolutionary tuning did clearly improve the software accuracy. However, we cannot prove that this improvement was due to the co-evolution and could not be achieved otherwise. In order to further study and consider the benefits of using the co-evolutionary method it was also applied to the problems of image filter and test image generation, see research paper VII.

6.7 **Paper VII:** *Testing digital halftoning software by generating test images and filters co-evolutionarily*

This paper is a continuation of paper IV, focusing on the application of co-evolutionary GAs. The aim is to develop test images and halftoning filters concurrently with the co-evolutionary genetic algorithm. The fitness of each filter is evaluated against all test images and *vice*

versa. The best filter is the one with which the hardest test image, when dithered, differs least from the original. Similarly, the hardest test image is the one that, when halftoned with the best image filter, differs the most from the original.

We test the hypothesis that co-evolutionary development of image filters and test images will lead to better optimization results than just optimizing image filters and test images separately against some static test image set or filter set. The results show that if bad image filters appear in the filter population they are soon killed from the population due to their weak response to the test images.

When using the co-evolutionary method, we can expect that the filters and images interact in such a way that the test image set is developed so that it is hard for the current filter set, and *vice versa* the filter set evolves so that it can handle the current test image set as well as possible. However, if the optimum is not in the area where these species start to converge, the mutation operator will cause a new type of filter or image individuals to appear and the interacting species roam into a different equilibrium situation. The co-evolutionary system should therefore be able to identify if there exists some hard test image characteristics or good filters that cannot be found just by optimizing both separately against some static set or fitness function.

The other goal of the research was to find further evidence that by generating test images we can find some image characteristics that are not repeated satisfactorily with the halftoning software. This time we tried to uncover these characteristics by doing error seeding, where the errors were caused by some image characteristics, and their severity depends on other image characteristics. The power of the GA optimization based test system is that after it finds some problematic or suspicious parameters or combinations of them, it start to focus on them by favoring them and gradually building the parameter combinations that cause ever greater problems.

7 CONCLUSION, DISCUSSION AND FUTURE

This thesis discusses genetic algorithm-based software testing. Several test cases and problems were implemented and the corresponding test results were reported. Each example and research paper included in the thesis had a slightly different research topic, introduced in them individually. This introductory part reviewed the main research framework and overall research questions the individual test cases tried to answer. The literature related to the research problem was reviewed.

The first research question was on the applicability of genetic algorithms in temporal testing of software. It was studied in papers I, II and III, and more generally in [AMTV96, AMT97a, AMM97, AMMM97, AMMM98, AM99, AM00a, MA01c]. The research indicates a positive answer to the question. In addition, the PhD thesis concentrating on the same topic by Gross [Gro00] seems to have reached similar results independently. During the years when we studied this research topic, many other researchers also published a number of papers on related research [*e.g.* WGJ97, PN98, MW98, GJE99, Tra00] and which have positive results with GAs in temporal testing.

The second research question focused on testing image-processing software with the test images generated by genetic algorithms. Papers IV and VII, and more generally [MA00, MA01a, MA01b, MA02b, MA02c, MA03a] discussed the topic. Also papers V and VI [MA01d, MA02a] are extensions of the topic as they explore the testing of a machine vision based measurement software with the simulated test surfaces generated by a GA. These studies seem to confirm that the GA is capable of generating test images and surfaces that reveal some weaknesses in image processing or measurement software. In paper VII error seeding was tried to see if deliberately generated errors, that simulated the kind of errors we were trying to find for real, could be found by our method. We found that these seeded errors were effectively revealed by using these test images. The test surface generation also revealed the type of surfaces that were measured most unsatisfactorily.

The third research question concentrated on whether it is possible to optimize the software parameters simultaneously with the testing with a co-evolutionary genetic algorithm. Two applications of this method were presented in papers VI and VII [MA02a, MA03a]. In both studies the system seems to tune better with co-evolution than by just using the static test set. We could not really prove that this gain was caused by the co-evolutionary optimization, and that it could not have been obtained without it. However, the results were not as good, when we tried to optimize both problems separately without the co-evolution.

The object problems are such that a direct comparison of the results with other studies is not possible. The only commonly used benchmark problem (triangle classification) has usually been used with white box testing methods, but those methods were omitted from this study.

This research has concentrated on evolutionary methods. Other heuristic methods like tabu search, simulated annealing, *etc.* were not used for comparison due to the schedule and financing of the research. However, it was observed that the GA outperforms the random method in all our applications, so the lower benchmark limit was always reached. Furthermore, the Tracey's PhD thesis [Tra00] includes comparisons between random testing, hill climbing, simulated annealing and genetic algorithm based approaches for software test data generation and he came to the conclusion that GA was, on average, the most effective method of these.

It seems that a GA is capable of finding different kinds of software weaknesses, and co-evolution is beneficial in the tuning of software parameters. There is a lot of space for future research in this field. Our results are encouraging. We expect that more studies on evolutionary optimization based software engineering will appear in the following years. Implications for this is the rapidly increasing number of references in the field [MA03b].

A GA is well applicable to the problems that are discrete and those that have no exact mathematical expression or model, as is the case of software testing problem. Jones *et al.* [HJ01] claim that software engineering is ideal for evolutionary based optimization; none of the findings here contradict this claim.

The use of GAs is more up to the implementation; how the problem is encoded. If a GA does not seem to outpower the random method it has been usually implemented poorly or wrong, or the GA operators: crossover, mutation, selection schemes have been wrongly selected or they have pathological values. However, small changes in GA parameter values do not usually affect the optimization result much. When properly implemented, a GA is highly applicable for software testing, including testing coverage, timing, parameter values, or finding calculation tolerances, bottlenecks, problematic input combinations and sequences. If the program parameters are changeable on the fly, the co-evolutionary GA can tune the tested software during the testing.

We propose that a GA based automatic testing tool can be used to automatically generate test data for module and system testing. Although a random generator can just as simply generate the test data, a GA based testing more easily reveals problematic parameters and combinations of them and starts to construct more erroneous situations using only a fraction of the test cases that the pure random search method generates.

Testing the temporal behavior and test coverage simultaneously is a topic of further research. When testing temporal problems as WCET or BCET, there is not much guarantee about test coverage and how much of the software is tested. One could combine the black and white box testing views, and create a multi-objective fitness function, that includes the time-based part and the test coverage based part. This kind of approach has not yet been discussed in the literature. The only studies suggesting something in this direction were by Whitley [Whi98], where the goal was to exercise a maximum amount of code in the minimum time, Mueller and Wegener [MW98] suggested combining temporal testing with structural coverage, and Pohlheim [Poh01] who used both structural and temporal testing goals, but only the other one of them seemed to be the optimization target each time. Since white box methods and coverage testing was omitted from this thesis, it was not within the research framework to try this kind of hybrid.

The testing of the quality of an image processing software with test images would require more research. We applied our GA generated test images only for testing halftoning methods. There are several other research questions in the image processing field, where the artificial images might also be used.

The co-evolution approach has been applied mostly in game-playing and artificial life simulations. There is very little research on applying co-evolution to software engineering in general. Our implementation of co-evolution to the simultaneous software development and testing seems quite unique, and therefore it is mostly just a proposal at this point. However, due to the encouraging results got the possibilities of applying this method to software engineering deserve more research.

REFERENCES

Keys in alphabetical order.

[Add97] Addison, Paul S. (1997). *Fractals and Chaos: An Illustrated Course*. Bristol, Philadelphia: Institute of Physics Publishing.

[Ala92] Alander, Jarmo T. (1992). On optimal population size of genetic algorithms. In: *CompEuro 1992 Proceedings, Computer Systems and Software Engineering, 6th Annual European Computer Conference*, The Hague, 4.–8. May 1992, 65–70. Eds Patrick Dewilde and Joos Vandelwalle. Silverspring, MD: IEEE Computer Society Press.

[Ala95a] Alander, Jarmo T. (Feb. 13, 2002). *An Indexed Bibliography of Genetic Algorithms in Computer Science. Vaasa:* Department of Information Technology and Production Economics, University of Vaasa. Report Series No. 94–1-CS [cited 8.4.2003]. Available: ftp://garbo.uwasa.fi/cs/report94-1/gaCSbib.ps.Z.

[Ala95b] Alander, Jarmo T. (July 10, 2002). *An Indexed Bibliography of Genetic Algorithms in Electronics and VLSI*. Vaasa: Department of Information Technology and Production Economics, University of Vaasa. Report Series No. 94–1-VLSI, [cited 8.4.2003]. Available: ftp://garbo.uwasa.fi/cs/report94-1/gaVLSIbib.ps.Z.

[Ala95c] Alander, Jarmo T. (Oct. 4, 2002). *An Indexed Bibliography of Genetic Algorithms in Optics and Image Processing*. Vaasa: Department of Information Technology and Production Economics, University of Vaasa. Report Series No. 94–1-OPTICS [cited 8.4.2003]. Available: ftp://garbo.uwasa.fi/cs/report94-1/gaOPTICSbib.ps.Z.

[Ala99] Alander, Jarmo T. (1999). Population size, building blocks, fitness landscape and genetic algorithm search efficiency in combinatorial optimization: an empirical study. In: *Practical Handbook of Genetic Algorithms*, 459–485. Ed. Lance D. Chambers. Boca Raton, FL: CRC Press LLC.

[AM00a] Alander, Jarmo T. and Timo Mantere (2000). Genetic algorithms in software testing – experiments with temporal target functions. In: *MENDEL 2000 6th International Conference on Soft Computing*, June 7–9 2000, 9–14. Ed. P. Ošmera. Brno, Czech Republic: Brno University of Technology & PC-DIR.

[AM00b] Alander, Jarmo T. and Timo Mantere (2000). Genetic algorithms in automatic software testing – analysing a faulty bubble sort routine. In: *SteP 2000 – Millennium of Artificial Intelligence, The 9th Finnish Artificial Intelligence Conference*, August 28–31 2000, SteP 2000, vol. 2, 23–32. Ed. H. Hyötyniemi. Helsinki University of Technology, Espoo, Finland: Publications of the Finnish Artificial Intelligence Society 16.

[AM99] Alander, Jarmo T. and Timo Mantere (1999). Automatic software testing by genetic algorithm optimization, a case study. In: *SCASE'99 – Soft Computing Applied to Software Engineering*, April 11–14 1999, 1–9. Eds C. Ryan and J. Buckley; Limerick, Ireland: University of Limerick.

[AMM97] Alander, Jarmo T., Timo Mantere and Ghodrat Moghadampour (1997). Testing software response times using a genetic algorithm. In: *Proceedings of the Third Nordic Workshop on Genetic Algorithms and their Applications (3NWGA)*, August 18–22 1997, 293–298. Ed. J. T. Alander. Helsinki, Finland: Finnish Artificial Intelligence Society (FAIS).

[AMMM97] Alander, Jarmo T., Timo Mantere, Ghodrat Moghadampour and Jukka Matila (1997). Searching protection relay response time extremes using genetic algorithm – software quality by optimization. In *Proceedings of the Fourth International Conference on Advances in Power System Control, Operation & Management (APSCOM-97), volume 1*, November 11–14 1997, 95–99. Hong Kong: IEEE.

[AMMM98] Alander, Jarmo T., Timo Mantere, Ghodrat Moghadampour and Jukka Matila (1998). Searching protection relay response time extremes using genetic algorithm – software quality by optimization (a revised version of [AMMM97]). *Electric Power Systems Research* **46**, 229–233.

[AMP98] Alander, Jarmo T., Timo Mantere and Tero Pyylampi (1998). Threshold matrix generation for digital halftoning by genetic algorithm optimization. In: *Intelligent Systems and Advanced Manufacturing: Intelligent Robots and Computer Vision XVII: Algorithms, Techniques, and Active Vision*, volume SPIE-3522, Boston, MA, 1–6 November 1998, 204–212. Ed. D. P. Casasent. Bellingham, Washington: The SPIE Press.

[AMP99a] Alander, Jarmo T., Timo Mantere and Tero Pyylampi (1999). Digital halftoning optimization via genetic algorithms for ink jet machine. In: *Euroconference: Parallel and Distributed Computing for Computational Mechanics 1999, EURO-CM-PAR, 20–25 March 1999, Abstracts, Lecture and Research Presentations*, 83–84. Weimar, Germany.

[AMP99b] Alander, Jarmo T., Timo Mantere and Tero Pyylampi (1999). Digital halftoning optimization via genetic algorithms for ink jet machine (An extended version of [AMP99a]). In: *Developments in Computational Mechanics with High Performance Computing*, 211–216. Ed. B. H. V. Topping. Edinburg, UK: CIVIL-COMP Press.

[AMT97a] Alander, Jarmo T., Timo Mantere and Pekka Turunen (1997). Genetic algorithm based software testing. In: *Artificial Neural Nets and Genetic Algorithms, Proceedings of International Conference (ICANNGA97)*, Norwich, UK, April 1997, 325–328. Eds G. Smith, N. Steele and R. Albrecht. Wien, Austria: Springer-Verlag (Printed 1998).

[AMT97b] Alander, Jarmo T., Ghodrat Moghadampour and Pasi Törmänen. Evaluating the benefit of fuzzy logic for PID-control by means of genetic algorithms - case: frequency controller. In: *Proceedings of the Third Nordic Workshop on Genetic Algorithms and their Applications (3NWGA)*, August 18–22 1997, 321–331. Ed. J. T. Alander. Helsinki, Finland: Finnish Artificial Intelligence Society (FAIS).

[AMTB95] Almaini, A. E., J. F. Miller, P. Thomson and S. Billina (1995). State assignment of finite state machines using a genetic algorithm. In: *IEEE Proceedings on Computers and Digital Techniques*, Volume 142:4, 279 –286.

[AMTV96] Alander, Jarmo T., Timo Mantere, Pekka Turunen and Jari Virolainen (1996). GA in program testing. In: *Proceedings of the Second Nordic Workshop on Genetic Algorithms and their Applications (2NWGA)*, 19.–23. August 1996, 205–210. Ed. J. T. Alander. Vaasa, Finland: Proceedings of the University of Vaasa, Reports Nr. 11.

[ARRT01] Aguilar-Ruiz, J. S., I. Ramos, J. C. Riguilme and M. Toro (2001). An evolutionary approach to estimating software development projects. *Software Technology* **43**:14, 875–882.

[AT96] Alba, E. and J. M. Troya (1996). Testing software using order-based genetic algorithms. In: *Proceedings GP-96 Conference*, Stanford, CA, July 28–31 1996. Eds J. R. Koza, D. E. Goldberg, D. B. Fogel, R. L. Riolo. Cambridge, MA: MIT Press.

[AYT95] Alander, Jarmo T., Jari Ylinen and Tapio Tyni (1995). Optimizing elevator group control parameters using distributed genetic algorithms. In: *Artificial Neural Nets and Genetic Algorithms*, Ales, France, 19–21 Apr. 1995, 400–403. Eds D.W. Pearson, N.C. Steele and R.F. Albrecht. Wien, Austria: Springer-Werlag.

[Bab82] Baber, Robert L. (1982). *Software Reflected*. Amsterdam, Netherlands: North-Holland Publishing Company.

[Bar99] Barten, Peter G. J. (1999). *Contrast Sensitivity of the Human Eye and Its Effects on Image Quality*. Bellingham, Washington: SPIE Optical Engineering Press.

[Bay73] Bayer, B. E. (1973). An optimum method for two-level rendition of continuous-tone pictures. In: *IEEE International Conference on Communications* 1, June 11-13 1973, 11-15. New York, NY: IEEE.

[BDL+96] Bersini, H., M. Dorigo, S. Langerman, G. Seront and L. Gambardella (1996). Results of the first international contest on evolutionary optimisation (1st ICEO). In: *Proceedings of 1996 IEEE International Conference on Evolutionary Computation (ICEC '96)*, May 20–22 1996, 611–615. Nagoya: Japan: Nagoya University.

[Bei90] Beizer, B. (1990). *Software Testing Techniques*. 2. New York, NY: Van Nostrand Rheinhold.

[BJ00] Bueno, P. M. and M. Jino (2000). Identification of potentially infeasible program paths by monitoring the search for test data. In: *Proceedings ASE 2000, the Fifteenth IEEE International Conference on Automated Software Engineering*, 209–218. Grenoble, France: IEEE.

[BL01] Burgess, C. J. and M. Lefley (2001). Can genetic programming improve software effort estimation? A comparative evaluation. *Information and Software Technology* **43**, 863–873.

[BL97] Baisch, E. and T. Liedtke (1997). Comparison of conventional approaches and soft-computing approaches for software quality prediction. In: *1997 IEEE International Conference on Systems, Man, and Cybernetics, Computational Cybernetics and Simulation* 2, 1045 –1049. Orlando, FL: IEEE.

[BL98] Baisch, E. and T. Liedtke (1998). Automated knowledge acquisition and application for software development projects. In: *Thirteenth IEEE Conference on Automated Software Engineering*, 13–16 October. 1998, 306–309. Honolulu, Hawaii: IEEE Computer Society.

[BM96] Boden, E. B. and G. F. Martino (1996). Testing software using order-based genetic algorithms. In: *Proceedings GB-96 Conference*. Stanford. CA. 28–31 July 1996, 461–466. Eds J. R. Koza, D. E. Goldberg, D. B. Fogel and R. L. Riolo. Cambridge, MA: MIT Press.

[BM97] Baradhi, G. and N. Mansour (1997). A comparative study of five regression testing algorithms. In: *Proceedings of the Australian Software Engineering Conference 1997*, 174–182. Sydney: IEEE Computer Society Press.

[Bod98] Boden, E. B. (1998). *Automated Testing of Software Application Interfaces, Object Methods and Commands*. U.S. patent no. 5,708,774. Issued January 13, 1998.

[Boe74] Boehm, B. W. (1974). Some steps towards formal and automated aids to software requirements analysis and design. In: *IFIB74*, 192–197. Amsterdam, Netherlands: North-Holland Publishing Company.

[BS81] Browne, J. C. and Mary Shaw (1981). Toward a scientific basis for software evaluation. In: [*PSS81*].

[BSPZ00] Bingul, Z., A. S. Sekmen, S. Palaniappan and S. Zein-Sabatto (2000). Genetic algorithms applied to real time multiobjective optimization problems. In: *Proceedings of the IEEE SoutheastCon 2000*, 95 – 103. Nashville, TN: IEEE.

[BYFR00] Boschini, M., X. Yu, F. Fummi and E. M. Rudnick (2000). Combining symbolic and genetic techniques for efficient sequential circuit test generation. In: *Proceedings of the IEEE European Test Workshop 2000*, 105–110. Cascais, Portugal: IEEE Computer Society Press.

[Dar59] Darwin, Charles (1859). *The Origin of Species: By Means of Natural Selection or The Preservation of Favoured Races in the Struggle for Life*, A reprint of the 6th edition, 1968. London: Oxford University Press [cited 8.4.2003]. Available: http://www.literature.org/authors/darwin-charles/the-origin-of-species.

[DG81] Denicoff, Marvin and Robert Grafton (1981). Software metrics: a research initiative. In: [*PSS81*].

[DMS94] Davies, E., J. McMaster and M. Stark (1994). *The use of genetic algorithm for flight test and evaluation of artificial intelligence and complex software systems*, Report AD-A284824. Patuxent River, MD. USA: Naval Air Warfare Center.

[Dra99] Drabick, Rodger (1999). *Growth and maturity in the testing process*. International Software Testing Institute [cited 8.4.2003]. Available: <http://www.softtest.org/articles/rdrabick3.htm>.

[DY97] Darwen P. J. and X. Yao (1997). Speciation as automatic categorical modularization. *IEEE Transactions on Evolutionary Computation*, 1:2, 101–108.

[EG99] Eklund, Patrick and Fredrik Georgsson (1999). Unraveling the Thrill of Metric Image Spaces. In: *Discrete Geometry for Computer Imagery*. LNCS 1586. Eds G. Bertrand, M. Couprie and L. Perroton. Marne-la-Vallée, France: Springer-Verlag.

[EKCA98] Evett M., T. Khoshgoftar, P. Chien and E. Allen (1998). GP-based software quality prediction. In: *Proceedings of the Third Annual Conference of Genetic Programming 1998*, 60–65. Madison, Wisconsin: Morgan Kaufmann.

[GJE00] Gross, H. G., B. F. Jones and D. E. Eyres (2000). Structural performance measure of evolutionary testing applied to worst-case timing of real-time systems. In: *IEEE Proceedings on Software* 147:2, 25–30.

[GJE99] Gross, H. G., B. F. Jones and D. E. Eyres (1999). Evolutionary algorithms for the verification of execution time bounds for real-time software. *IEE Colloquium on Applicable Modelling, Verification and Analysis Techniques for Real-Time Systems (Ref. No. 1999/006)*, 8/1 –8/8.

[GN96] Graham, P. and B. Nelson (1996). Genetic algorithms in software and in hardware – A performance analysis of workstation and custom computing machine implementations. In: *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, 216 – 225. Napa Valley, CA: IEEE Computer Society Press.

[Gol89] Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*. New York, NY: Addison-Wesley.

[Gre91] Grefenstette, J. J. (1991). Lamarckian learning in multi-agent environments. In: *Fourth International Conference on Genetic Algorithms*. San Diego, CA, 303–310. Eds R. Belew and L. Booker. San Mateo, CA, USA: Morgan-Kaufmann.

[Gro00] Gross, Hans-Gerhard (2000). *Measuring Evolutionary Testability of Real-Time Software*. PhD thesis. Pontypridd, Wales: University of Glamorgan.

[GS01] Gounares, A. and P. Sikchi (2001). *Adaptive Problem Solving Method and Apparatus Utilizing Evolutionary Computation Techniques*. U.S. patent no. 6,282,527. Issued August 28, 2001.

[GW98] Grochtmann, M. and J. Wegener (1998). Evolutionary testing of temporal correctness. In: *Proceedings of the 2nd International Software Quality Week Europe (QWE 1998)*. Brussels, Belgium, November 1998. San Francisco, CA: SR/Institute, Inc.

[Het73] Hetzel, W. (1973). *Program Test Methods*. Englewood Cliffs, NJ: Prentice-Hall.

[Het88] Hetzel, W. (1988). *The Complete Guide to Software Testing*, second edition. New York, NY: John Wiley & Sons, Inc.

[HJ01] Harman, M. and B. F. Jones (2001). Search-based software engineering. *Information and Software Technology* 43, 833–839.

[HKAH96] Hochman, R., T. M. Khoshgoftaar, E. B. Allen, and J. P. Hudepohl, (1996). Using the genetic algorithm to build optimal neural networks for fault-prone module detection. In: *Proceedings of the Seventh International Symposium on Software Reliability Engineering*, Oct 30 – Nov 2 1996, 152–162. White Plains, New York: IEEE Computer Society Press.

[HKAH97] Hochman, R., T. M. Khoshgoftaar, E. B. Allen and J. P. Hudepohl, (1997). Evolutionary neural networks: a robust approach to software reliability problems. In: *Proceedings of the Eight International Symposium on Software Reliability Engineering*, 13–26. Albuquerque, New Mexico: IEEE Computer Society Press.

[Hol75] Holland, John (1975). *Adaptation in Natural and Artificial Systems*. Ann Arbor, MI: University of Michigan Press, Reissued by The MIT Press, 1992.

[HRP96] Hsiao, M. S., E. M. Rudnick and J. H. Patel (1996). Automatic test generation using genetically-engineered distinguishing sequences. In: *Proceedings of 14th VLSI Test Symposium 1996*, 216 –223. Princeton, NJ: IEEE Computer Society Press.

[HRP98] Hsiao, M. S., E. M. Rudnick and J. H. Patel (1998). Application of genetically engineered finite-state-machine sequences to sequential circuit ATPG. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **17**, 239–254.

[Hun95] Hunt, J. (1995). Testing control software using a genetic algorithm. *Engineering Applications of Artificial Intelligence* 8:6, 671–680.

[IEE83] IEEE/ANSI (1983, re 1991). *IEEE standard for software test documentation*, IEEE Std. 829–1983.

[IEE86] IEEE/ANSI (1986, re 1992). *IEEE standard for software verification and validation plans*, IEEE Std. 1012–1986.

[IEE90] IEEE/ANSI (1990). *IEEE standard glossary of software engineering terminology*, IEEE Std. 620.12–1990.

[JES98] Jones, B. F., D. E. Eyres and H. H. Sthamer (1998). A strategy for using genetic algorithms to automate branch and fault-based testing. *The Computer Journal* **41**:2, 98–107.

[Jon78] Jones, T. C. (1978). Measuring programming quality and productivity. *IBM Systems Journal* 17, 39–63.

[JSE95] Jones, B. F., H. H. Sthamer and D. E. Eyres (1995). Generating test data for ADA procedures using genetic algorithms. In: *First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications, GALESIA*, 65–70. Scheffield, UK: Iee Conference Publication, No 414.

[JSE96] Jones, B. F., H. H. Sthamer and D. E. Eyres (1996). Automatic structural testing using genetic algorithms. *Software Engineering Journal* 11, 299–306.

[JSXE95] Jones, B. F., H. H. Sthamer, X. Yang and D. E. Eyres (1995). The automatic generation of software test data sets using adaptive search techniques. In: *3rd International Conference on Software Quality Management*, 435-444. Seville, Spain: Computational Mechanics Publications.

[JW98] Jones, B. F. and J. Wegener (1998). Measurement of extreme execution times for software. *IEE Colloquium on Real-Time Systems* (Digest No. 1998/306), 4/1–4/5.

[Kan97] Kaner, Cem (1997). Improving the maintainability of automated test suites. In: *Proceedings of the Tenth International Quality Week*. San Francisco, CA: Software Research.

[Kan99] Kang, Henry R. (1999). *Digital Color Halftoning*. Bellingham, Washington: SPIE Optical Engineering Press & New York: IEEE Press..

[KBH+97] Koza, John R., Forest H. Bennett III, Jeffrey L. Hutchings, Stephen L. Bade, Martin A. Keane and David Andre (1997). Evolving sorting networks using genetic programming and rapidly reconfigurable field-programmable gate arrays. In: *Workshop on Evolvable Systems. International Joint Conference on Artificial Intelligence*, Nagoya, Japan, 27–32. Ed. Higuchi, Tetsuya.

[KFN99] Kaner, Cem, Jack Falk and Hung Quoc Nguyen (1999). *Testing Computer Software, 2nd Edition*. New York, NY, USA: John Wiley & Sons.

[KG96] Kasik, D. J. and H. G. George (1996). Toward automatic generation of novice user test scripts. In: *Proceedings 1996 Conference on Human Factors in Computing Systems, CHI 96*, Vancouver, BC, Canada, April 13–18 1996, 244–251. New York, NY: ACM Press.

[KHS+97] Krishnaswamy, D., M. S. Hsiao, V. Saxena, E. M. Rudnick, J. H. Patel and P. Banerjee (1997). Parallel genetic algorithms for simulation-based sequential circuit test generation. In: *Proceedings of the Tenth International Conference on VLSI Design*, 475–481. Hyderabad, India: IEEE Computer Society Press.

[Kin93] Kinnear, Kenneth E. Jr. (1993). Generality and difficulty in genetic programming: evolving a sort. In: *Proceedings of the Fifth International Conference on Genetic Algorithms*, San Mateo, CA. Ed. S. Forrest. Los Altos, CA: Morgan Kaufmann.

[Kit95] Kit, Edward (1995). *Software testing in the real world – improving the process*. New York, NY, USA: Addison-Wesley.

[Kor76] Korel, Bogdan (1976). Automated software test data generation. *IEEE Transactions on Software Engineering* **16**:8, 870–879.

[Koz92] Koza, John. R. (1992). *Genetic Programming*. Cambridge, MA: The MIT Press.

[KS96] Kobayashi, N. and H. Saito (1996) Halftoning technique using genetic algorithms. *Systems and Computers in Japan* **27**, 89–97.

[Lai02] Laine, Antti (2002). *Testaus ja testauksen laatu*. Presentation in Testaus ja laatu seminar 3.–5. 9. 2002, Silja Serenade [cited 8.4.2003]. Available: http://www.pcuf.fi/sytyke/yhdistys/arkisto/risteily2002/Conformiq_Sytyke_pres.pdf.

[Lam09] Lamarck, J. B. (1809). *Zoological Philosophy*. Originally published as *Philosophie Zoologique*. Translated, with an introduction, by Hugh Elliot. London: Macmillan and Co., Ltd., 1914 [cited 8.4.2003]. Available: http://members.aol.com/evomech/index.html.

[Lam99] Lampinen, Jouni (1999). *Cam Shape Optimization by Genetic Algorithms*. Acta Wasaensia, No. 70, PhD thesis. Vaasa, Finlad: University of Vaasa.

[LHRP96] Lee, T., I. N. Hajj, E. M. Rudnick and J. H. Patel (1996). Genetic-algorithm-based test generation for current testing of bridging faults in CMOS VLSI circuits. In: *Proceedings of 14th VLSI Test Symposium 1996*, 456–462. Princeton, NJ: IEEE Computer Society Press.

[Lig72] Liguori, Fred, ed. (1972). *Automatic Test Equipment: Hardware, Software, and Management*. New York, NY, USA: IEEE Press.

[LY01] Lin, Jin-Cherng and Pu-Lin Yeh (2001). Automatic test data generation for path testing using GAs. *Information Sciences* **131**, 47–64.

[MA00] Mantere, Timo and Jarmo T. Alander (2000). Automatic image generation by genetic algorithms for testing halftoning methods. In: *Intelligent Systems and Advanced Manufacturing: Intelligent Robots and Computer Vision XIX: Algorithms, Techniques, and Active Vision*, Volume SPIE-4197, Boston, MA, November 5–8 2000, 297–308. Ed. D. P. Casasent. Bellingham, Washington: SPIE.

[MA01a] Mantere, Timo and Jarmo T. Alander (2001). Testing halftoning methods by images generated by genetic algorithms. In: *Arpakannus 1/2001, Special issue on Bioinformatics and Genetic Algorithms*, 39–44. Espoo, Finland: Finnish Artificial Intelligence Society c/o Technical Research Centre.

[MA01b] Mantere, Timo and Jarmo T. Alander (2001). Automatic test image generation by genetic algorithms for testing halftoning methods – comparing results using a wavelet filtering. In: *Proceedings of the 2001 Finnish Signal Processing Symposium FINSIG'01*, Helsinki University of Technology, Espoo, Finland, June 5, 2001, 55–58. Eds J. Tanskanen and J. Martikainen. Espoo: IEEE Finland Section.

[MA01c] Mantere, Timo and Jarmo T. Alander (2001). Automatic software testing by optimization with genetic algorithms – introduction to the method and consideration of the possible pitfalls. In: *MENDEL2001 7th International Conference on Soft Computing*, June 6–8, 2001, Brno, Czech Republic, 19–23. Eds R. Matousek and P. Ošmera. Brno, Czech Republic: Brno University of Technology & Kuncik.

[MA01d] Mantere, Timo and Jarmo T. Alander (2001). Testing a structural light vision software by genetic algorithms – estimating the worst case behavior of volume measurement. In: *Intelligent Systems and Advanced Manufacturing: Intelligent Robots and Computer Vision XX: Algorithms, Techniques, and Active Vision*, volume SPIE-4572, Newton, MA, October 29–31 2001, 466–475. Eds D. P. Casasent and E. L. Hall. Bellingham, Washington: SPIE Optical Engineering Press.

[MA02a] Mantere, Timo and Jarmo T. Alander (2002). Developing and testing structural light vision software by co-evolutionary genetic algorithm. In: *QSSE 2002 The Proceedings of the Second ASERC Workshop on Quantative and Soft Computing based Software Engineering*, Feb 18–20 2002, 31–37. Banff, Alberta, Canada: Alberta Software Engineering Research Consortium (ASERC) and the Department of Electrical and Computer Engineering, University of Alberta.

[MA02b] Mantere, Timo and Jarmo T. Alander (2002). Testing halftoning methods using genetic algorithms – comparing results using Haar wavelet filtering. In: *MENDEL2002 8th International Conference on Soft Computing*, June 5–7, 2002. Brno, Czech Republic, 103–108. Eds R. Matousek and P. Ošmera. Brno: Brno University of Technology & Kuncik Jan.

[MA02c] Mantere, Timo and Jarmo T. Alander (2002). Generating and testing halftoning filters co-evolutionarily. Unpublished, accepted to be published In: *Proceedings of 2002 WSEAS International Conference on Electronics, Control and Signal Processing*. Singapore, December 9–12, 2002, 6 pages (Not yet printed at the date of this thesis).

[MA03a] Mantere, Timo and Jarmo T. Alander (2003). *Testing digital halftoning software by generating test images and filters co-evolutionarily*. Unpublished, included in this thesis, a revised version of [MA02c], submitted to SPIE's Photonics East: Intelligent Robots and Computer Vision XXI, 27-31 October, Providence, Rhode Island, USA, 14 pages.

[MA03b] Mantere, Timo and Jarmo T. Alander (2003). *Evolutionary software engineering*, a review. Unpublished, preliminarily accepted to *Applied Soft Computing*, 24 pages.

[Man03] Mantere, Timo (2003). Software Testing by Evolutionary Algorithms. Unpublished, accepted to be published In: *Proceedings of Southeastern Software Engineering Conference*, April 1st - 3rd, 2003, Huntsville, Alabama, USA, extended abstract draft version 3 pages (Not yet printed at the date of this thesis).

[Man96] Mantere, Timo (1996). *Sähkönhankinnan optimointi* [*The optimization of the unit commitment problem in electrical power distribution*]. M.Sc. thesis, 100 pages. Vaasa, Finland: University of Vaasa, Department of Information Technology and Production Economics.

[Man99] Mantere, Timo (1999). *Automaattinen ohjelmien testaus optimoimalla geneettisillä algoritmeilla* [*Automatic software testing by optimizing with genetic algorithms*]. Lic.Sc. thesis, 125 pages. Vaasa, Finland: University of Vaasa, Department of Information Technology and Production Economics.

[Mar95] Marick, Brian (1995). *The Craft of Software Testing – Subsystem Testing*. Englewood Cliffs, NJ: Prentice Hall.

[Mar97] Marick, Brian (1997). Classic testing mistakes. In: *Proceedings of STAR 97, Sixth International Conference on Software Testing, Analysis, and Review*, May 1997. San Jose, CA: Software Quality Engineering.

[Mar98] Marick, Brian (1998). When should a test be automated?. In: *Proceedings of the 11th International Software Quality Week Conference (QW'98)*, May 1998. San Francisco, CA: Software Research Inc.

[ME97] Mansour, N. and K. El-Fakih (1997). Natural optimization algorithms for optimal regression testing. In: *Proceedings of the COMPSAC '97, The Twenty-First Annual International Computer Software and Applications Conference, 1997*, 511–514. Los Alamitos, CA: IEEE Computer Society Press.

[Mil80] Miller, Ed (1980). Program testing – An overview for managers. In: *IEEE software testing tutorial*. IEEE Computer Society Press.

[Mit98] Michell, Melanie (1998). *An Introduction to Genetic Algorithms*. Cambridge, MA, USA: MIT Press.

[ML96] Mantere, Timo and Ilpo Lindfors (1996). Comparison of unit commitment methods. In: *Proceedings of the Second Nordic Workshop on Genetic Algorithms and their Applications (2NWGA)*, 19.–23. August 1996, Vaasa, 245–250. Ed. J. T. Alander. Vaasa, Finland: Proceedings of University of Vaasa, Nro. 11.

[MM98] Michael, C. and G. McGraw (1998). Automated software test data generation for complex programs. In: *Proceedings of 13th IEEE International Conference on Automated Software Engineering*, 136–146. Honolulu, HI: IEEE Computer Society Press.

[MMS98] McGraw, G., C. Michael and M. Schatz (1998). *Generating Software Test Data by Evolution*. Technical Report RSTR-018–97–1, RST Corporation, Suite #250, Ridgetop Circle, Sterling, VA 20166. February 9, 1998.

[Mog99] Moghadampour, Ghodrat (1999). *Using Genetic Algorithms in Testing a Distribution Protection Relay Software – A Statistical Analysis*. Lic.Sc. thesis. Vaasa, Finland: University of Vaasa, Department of Information Technology and Production Economics.

[MT95] Minohara, T. and Y. Tohma (1995). Parameter estimation of hyper-geometric distribution software reliability growth model by genetic algorithms. In: *Proceedings of the Sixth International Symposium on Software Reliability Engineering*, 324–329. Toulouse, France: IEEE Press

[MW98] Mueller, F. and J. Wegener (1998). A comparison of static analysis and evolutionary testing for the verification of timing constraints. In: *Proceedings of the 4th IEEE Real-Time Technology and Applications Symposium*, June 1998, 144–154. Denver, USA: IEEE Press.

[Mye78] Myers, G. J. (1978). *The Art of Software Testing*. New York, NY: John Wiley&Sons.

[NB97] Newbern, J. and V. M. Bove, Jr. (1997). Generation of blue noise arrays using genetic algorithm. In: *Human Vision and Electronic Imaging II*, Feb. 1997, San Jose, Volume SPIE-3016, 401–450. Eds B.E. Rogowitz and T.N. Pappas. Bellingham, Washington, USA: SPIE.

[Nil99] Nilsson, Fredrik (1999). Objective quality measures for halftoned images. *Optics, Image, Science and Vision* **16**:9, 2151–2162.

[NIS02] NIST (2002). *The Economic Impacts of Inadequate Infrastructure for Software Testing*, US Department of Commerce, National Institute of Standards and Technology (NIST), Planning Report 02–3, May 2002, 309 pages [cited 8.4.2003]. Available: http://www.nist.gov/director/prog-ofc/report02-3.pdf.

[Nor93] Norman, S. (1993). *Software Testing Tools*. London: Ovum Ltd.

[OA95] O'Dare, M.J. and T. Arslan (1995). Generating test patterns for VLSI circuits using a genetic algorithm. *IEE Electronics Letters* **30**:10.

[OR99] Ostrowski, D. A. and R. G. Reynolds (1999). Knowledge-based software testing agent using evolutionary learning with cultural algorithms. In: *Proceedings of the 1999 Congress on Evolutionary Computation, CEC 99*, 1657–1663. Washington, DC: IEEE Press.

[Paa02] Paakki, Jukka (2002). *Testaus teoriassa ja käytännössä*. Presentation in: Testaus ja laatu seminar 3.–5. 9. 2002, Silja Serenade [cited 8.4.2003]. Available: http://www.pcuf.fi/sytyke/yhdistys/arkisto/risteily2002/sytyke-09-02.pdf.

[Pet96] Pettischord, Bret (1996). Success with test automation. In: *Proceedings of the Ninth International Quality Week*. San Francisco, CA: Software Research Inc.

[PGGZ94] Pei, Min, Erik D. Goodman, Zongyi Gao and Kaixiang Zhong (1994). *Automated Software Test Data Generation Using a Genetic Algorithm*. Technical report 6/2/1994. Beijing, China: University of Aeronautics and Astronautics.

[PGK99] Pal, Sankar K., Arhish Ghosh and Malay K. Kundu (1999). *Soft Computing for Image Processing*. Heidelberg, New York: Physica-Verlag.

[PHP99] Pargas, Roy, Mary Harrold and Robert Peck (1999). Test-data generation using genetic algorithms. *The Journal of Software Testing, Verification and Reliability* **9**, 263–282.

[PN98] Puschner, P. and R. Nossal (1998). Testing the results of static worst-case execution-time analysis. In: *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS '98)*, 1998, 134 – 143. Madrid, Spain: IEEE Computer Society Press.

[Poh01] Pohlheim, H. (2001) Competition and cooperation in extended evolutionary algorithms. In: *GECCO'2001 – Late Breaking Papers*. Ed. L. Spector. San Francisco, CA: Morgan Kaufmann.

[PSS81] Perlis, Alan, Frederick Sayward and Mary Shaw, Eds (1981). *Software Metrics: An Analysis and Evaluation*. Cambridge, MA: The MIT Press.

[Pyy99] Pyylampi, Tero (1999). *Rasterointimenetelmän kehittäminen mustesuihkutulostimelle geneettisellä algoritmilla [Developing Digital Halftoning Methods for Ink Jet Printer with Geneting Algorithms]*. M.Sc. thesis. Vaasa: Finland: University of Vaasa, Department of Information Technology and Production Economics.

[Rau00] Rautakoura, Timo (2000). *Suunnitelma pastanpainon laaduntarkkailujärjestelmästä www-pohjaiseen tutkimus-, opetus- ja koulutuskäyttöön [A Plan for Solder Paster Quality Control System for WWW-based Research, Teaching and Training Use]*. M.Sc. thesis. 61 pages. Pori, Finland: Tampere University of Technology.

[Rec73] Rechenberg, I. (1973). Evolutionsstrategie: *Optimierung Technischer Systeme nach Prinzipien der Biologischen Evolution*. Stuttgart, Germany: Fromman-Holzboog.

[RHSP94] Rudnick, E. M., J. G. Holm, D. G. Saab and J. H. Patel (1994). Application of simple genetic algorithms to sequential circuit test generation. In: *Proceedings of EDAC European Design and Test Conference 1994*, 40–45. Paris, France: IEEE Computer Society Press.

[RMB+95] Roper, Marc, Iain MacLean, Andrew Brooks, James Miller and Murray Wood (1995). *Genetic Algorithms and the Automatic Generation of Test Data*. Technical report RR/95/195 [EfoCS-19–95]. Glasgow, UK: University of Strathclyde.

[Rop96] Roper, M. (1996). Cast with GAs –automatic test data generation via evolutionary computation. In: *IEE Colloquium on Computer Aided Software Testing Tools*, vol. IEE Digest No. 1996/096, London, April 23 1996, 7/1–7/5. London, UK: IEE.

[Rop99] Roper, M. (1999). Software testing –searching for the missing link. In: *Information and Software Technology* **41**, 991–994.

[RP95] Rudnick, E. M. and J. H. Patel (1995). A genetic approach to test application time reduction for full scan and partial scan circuits. In: *Proceedings of the 8th International Conference on VLSI Design 1995*. 288 –293. New Delhi, India: IEEE Press.

[RPGN97] Rudnick, E. M., J. H. Patel, G. S. Greenstein and T. M. Niermann (1997). A genetic algorithm framework for test generation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **16**, 1034 –1044.

[SBF97] Smith, J. E., M. Bartley and T. C. Fogarty (1997). Microprocessor design verification by two-phase evolution of variable length tests. In: *IEEE International Conference on Evolutionary Computation 1997*, 453–458. Indianapolis: IEEE Neural Networks Council.

[Sch96] Schäfer, Hans (1996). *World of Software Testing*. Seminar publication 15–16. Apr. 1996. Tampere, Finland: Pirkanmaan tietojenkäsittely-yhdistys ry.

[SF96] Smith, J. E. and T. C. Fogarty (1996). Evolving software test data – GA's learn self expression. In: *Evolutionary Computing*. LNCS 1143. Ed. T. C. Fogarty. Berlin, Germany: Springer-Verlag.

[SGD92] Schultz, A. C., J. J. Grefenstette and K. A. De Jong (1992). Adaptive testing of controllers for autonomous vehicles. In: *1992 Symposium on Autonomous Underwater Vehicle Technology*, June 1997, 158–164. Washington DC: IEEE.

[SGD93] Schultz, A. C., J. J. Grefenstette and K. A. De Jong (1993). Test and evaluation by genetic algorithm. *IEEE Expert* **8**, 9–14.

[SGD95] Schultz, A. C., J. J. Grefenstette and K. A. De Jong (1995). Applying genetic algorithms to the testing of intelligent controllers. In: *The Workshop on Applying Machine Learning in Practice at IMLC-95*, 9 July 1995, 41–48. Tahoe City, CA: Naval Research Laboratory, Techical Report AIC-95–023.

[SGD97] Schultz, A. C., J. J. Grefenstette and K. A. De Jong (1997). Learning to break things: adaptive testing of intelligent controllers. In: *Handbook of Evolutionary Computation*, G.3.5:1–10. Eds T. Bäck, D. Fogel and Z. Michalewicz. Oxford, UK: Oxford University Press.

[Sim91] Sims, K. (1991). Artificial Evolution for Computer Graphics. In: *Computer Graphics (Siggraph '91 Proceedings)*, July 1991, 319–328. New York, NY: ACM Press.

[SJE94] Sthamer, H. H., B. F. Jones and D. E. Eyres (1994). Generating test data for ADA generic procedures using genetic algorithms. In: *Proceedings of the ACEDC 1994*, 134–140. Plymouth, UK: University of Plymouth.

[SP95] Storm, R. and K. Price (1995). *Differential Evolution – A simple and efficient adaptive scheme for global optimization over continuous spaces*. Technical report TR-95–012, ICSI, March 1995 [cited 8.4.2003]. Available: ftp://ftp.icsi.berkeley.edu/pub/techreports/1995/tr-95-012.pdf.

[SRM91] Sullivan, J., L. Ray and R. Miller (1991). Design of minimum visual modulation halftone patterns. *IEEE Transactions on Systems, Man, and Cybernetics* **21**:1.

[Sth95] Sthamer, Harmen-Hinrich (1995). *The Automatic Generation of Test Data Using Genetic Algorithms*. PhD thesis. Pontypridd, Wales: University of Glamorgan.

[SW99] Singpurwalla, Nozer D. and Simon P. Wilson (1999). *Statistical Methods in Software Engineering – Reliability and Risk*. New York, NY: Springer-Verlag.

[Sys89] Syswerda, Gilbert (1989). Uniform crossover in genetic algorithms. In: *Proceedings of the Third International Conference on Genetic Algorithms*, George Mason University, June 4–7, 1989, 2–9. Ed. J. D. Schaffer. San Mateo, California: Morgan Kaufmann Publishers, Inc.

[Tam02] Tamres, Louise (2002). *Introducing Software Testing*. London, UK: Pearson Education Ltd., Addison-Wesley.

[Tra00] Tracey, Nigel James (2000). *A Search-Based Automated Test-Data Generation Framework for Safety-Critical Software*. PhD thesis. York, UK: University of York.

[War98] Warfield, R. W. (1998). *Automated Software Testing Tool*. U.S. patent no. 5,754,760. Issued May 19, 1998.

[Wat95] Watkins, Alison L. (1995). The automatic generation of test data using genetic algorithms. In: *Proceedings of the 4th Software Quality Conference*, July 4–5 1995. Eds I. M. Marshall *et al.* Dundee, Scotland: University of Abertay.

[WBS01] Wegener, J., A. Baresel and H. Sthamer (2001). Evolutionary test environment for automatic structural testing. *Information and Software Technology* **43**, 841–854.

[Wei90] Weinberger, E. D. (1990). Correlated and uncorrelated fitness landscapes and how to tell the difference. *Biological Cybernetics* **63**, 325–336.

[WGG+96] Wegener, J., K. Grimm, M. Grochtmann, H. Sthamer and B. Jones (1996). Systematic testing of real-time systems. In: *Proceedings of the 4th European Conference on Software Testing, Analysis & Review EuroSTAR 1996*. December 1996. Amsterdam, Netherlands: Software Quality Engineering.

[WGJ97] Wegener, J., M. Grochtmann and B. Jones (1997). Testing temporal correctness of real-time systems by means of genetic algorithms. In: *Proceedings of the 10th International Software Quality Week (QW '97)*. May 1997. San Francisco, CA: Software Research, Inc.

[Whi98] Whitten, T. G. (1998). *Method and Computer Program Product for Generating a Computer Program Product Test that Includes an Optimized Set of Computer Program Product Test Cases, and Method for Selecting Same*. U.S. patent no. 5,805,795. Issued September 8, 1998.

[WM95] Wolpert, D. and W. Macready (1995). *No Free Lunch Theorems for Search*. Technical report 95–02–010. Santa Fe Institute.

[WSP99] Wegener J., H. Sthamer and H. Pohlheim (1999). Testing the temporal behaviour of real-time tasks using extended evolutionary algorithms. In: *Proceedings of the 7th European Conference on Software Testing, Analysis and Review (EuroSTAR '1999)*, November 1999. Barcelona, Spain: Software Quality Engineering.

[XESLK92] Xanthakis, S., C. Ellis, C. Skourlas, A. Le Gall and S. Katsikas (1992). Applying genetic algorithms to software testing. In: *Proceedings of the 5th International Conference on Software Engineering & It's Applications*. Toulouse, France.

[Yan98] Yang, Xile (1998). *Automatic Testing From Z specifications*. PhD thesis. Pontypridd, Wales: University of Glamorgan.

[YWR00] Yu Xiaoming, Jue Wu and E. M. Rudnick (2000). Diagnostic test generation for sequential circuits. In: *Proceedings of the International Test Conference 2000*. 225–234. Atlantic City, NJ: IEEE Computer Society Press.

Referred WWW-sites [cited 8.4.2003]

Software Engineering using Metaheuristic Innovative Algorithms http://www.discbrunel.org.uk/seminalproject/index.html

Selected Papers on Software Testing http://www.systematic-testing.com

Software Testing Online Resources http://www.mtsu.edu/~storm

The USC-SIPI Image Database http://sipi.usc.edu/services/database/Database.html

REPRINTS OF THE PUBLICATIONS

ERRDATA

Paper II:

Unfortunately this paper includes some mistakes that occurred during the publication phase: the list of references is in the wrong order, the right order is to renumber 1, 2, 3, .. , 23 to the following order: 2, 3, 6, 8, 5, 21, 12, 4, 1, 16, 17, 14, 11, 18, 20, 13, 15, 9, 10, 7, 22, and 23. The Mean (ms) for the GA-total in Table 1 should be 223.45 as it is in [AMMM97]. However, these errors do not affect the general findings of this paper.

Paper III:

This paper does include some typo's caused by some kind of Windows and Unix LaTeX version incompatibility, that made parts of the texts that were commented off from the manuscript appearing, like lonely words "*of*" in Chapter 5., "*whole*" in Chapter 5.1, the word "*linked*" before ESIM in Chapter 5.1, and the mark ¿ in Chapter 6.

Paper VI:

This paper includes one found mistake: when the test run represented in Figure 3 was made, there were not zeros in the max and min clauses of equation 2. This does not affect the main results, but explains why the *Surface* value can be lower than the *Parameters* value in Figure 3.

GENETIC ALGORITHMS IN SOFTWARE TESTING - EXPERIMENTS WITH TEMPORAL TARGET FUNCTIONS

Jarmo T. Alander and Timo Mantere

University of Vaasa, Dept. of Information Technology and Production Economics, P.O. Box 700, FIN-65101 Vaasa
Email: firstname.lastname@uwasa.fi

In this paper we simulate the software response time by a genetic algorithm based system. The object was to determine whether a genetic algorithm could handle largely nondeterministic target functions. This study was a part of a software testing project, where the goal was to test large time critical embedded software with genetic algorithms. The behavior of the embedded system was recognized to be highly nondeterministic, therefore it was considered necessary to test the method with some other highly nondeterministic problem. We are also considering the question of good GA parameters, optimal population size and the risk of not finding the optimal solution.

Keywords: genetic algorithms, software testing, test data generation, response time, optimal parameters, population size, nondeterministic fitness

1. Introduction

Software testing is an essential task when trying to achieve high software quality. Manual testing is usually time demanding and expensive. Therefore there is a strong demand for creating new testing techniques. It has been estimated [1, 2], that software testing costs over 30% of total expenses in software system development. So even a partial testing automation with an effective tool can produce considerable savings.

In order to automate software testing by genetic algorithms (GA) we need to define some target function, software metric. In our earlier study [3, 4] the target function was simply the response time of the tested software. Genetic algorithm was used as the test data generator. The test data was mainly timing related information. GA sent the information to the tested software and measured the response time from certain input to some specific output observed at the software interface.

The problem encountered in the above test experiment was the large nondeterminism, identical input sequences resulted in quite different outputs. To further study whether that kind of nondeterminism can be acceptable and whether the obtained test results still are valid we decided to model nondeterministic systems by using only the response time as the optimization target.

1.1 Genetic algorithms

Genetic algorithms [5] are optimization methods that mimic evolution in nature [6]. Genetic algorithms form a kind of electronic population that fights for survival, adapting as well as possible to its environment, which is an optimization problem. Surviving and crossbreeding possibilities depend on how well individuals fulfill the target function. GA reproduces new individuals by crossbreeding good individuals, it also adds some mutations, i.e. bit changes to the individuals. The set of the best solutions is usually kept in an array called the population.

1.2 Related work

Genetic algorithms have previously been adapted to automatic software test data generation in several studies [7, 8, 9, 10, and 11]. Parameter optimizations with GAs have been dealt f.e. in [12]. Genetic algorithms have been adapted to a nondeterministic problem, f.e. in [13]. Optimizing GA with another GA have been dealt f.e. in [14, 15,16, and 17]. See also the bibliography [18].

2. Representation of test problems

In this chapter we briefly present the test problems that were used to pre-evaluate the GA test approach. These tests were also used to analyze if there exist good GA parameter combinations. In the next chapter that concentrates on time based testing the goal was to find GA parameter combinations that leads either to acceleration or retardation of the optimization speed. The test problem set was introduced in [19]. All problems are encoded into a 34 bit long chromosome.

2.1 Polyomino problem

In the polyomino problem [19] the goal is to cover a 4×4 rectangular area with seven building blocks, from which two are 1×1 blocks, two 1×2 tiles, one 2×2 tile and two L-shaped tiles (2×2-1×1). The coordinates of the polyominoes were encoded into a 34 bit long chromosome. These building blocks are presented in figure 1. This problem belongs to the set of layout design problems.

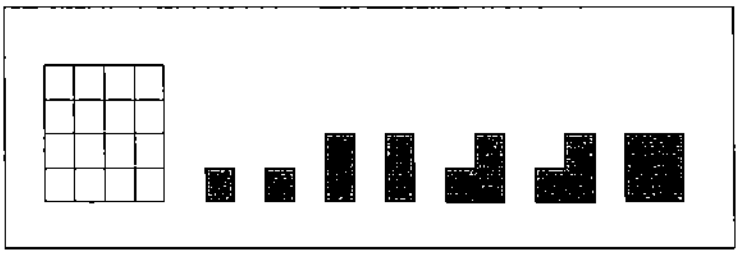


Figure 1: The building blocks in the polyomino problem and the rectangular 4×4 area they should cover

The fitness function of the polyomino problem is the area covered by the polyominos. The optimum is 16. The fitness function only counts how much of the area is covered; it doesn't penalize for overlapping tiles or parts that surpass the boundaries of the area. This problem has several different optimal solutions, i. e. the parts can be ordered in several ways, while still getting the total area covered. GA optimizes the problem by moving and rotating the given seven building blocks.

When optimizing a polyomino problem with different GA parameters, it was discovered that the problem is solved the quickest with a smaller population. Other parameters did not have such a clear influence on the processing speed. This finding confirms results presented in Alander [19].The crossover rate [20] showed a tendency to be quite high. Many of the best solution had this parameter around 85%.

The fastest optimal solutions were found with relatively small population sizes. Figure 2 shows the number of trials needed in the different GA optimization runs as the function of population size. There was a stopping condition of 20.000 tested solutions. If the optimum had not been found before reaching it, the search would be given up.

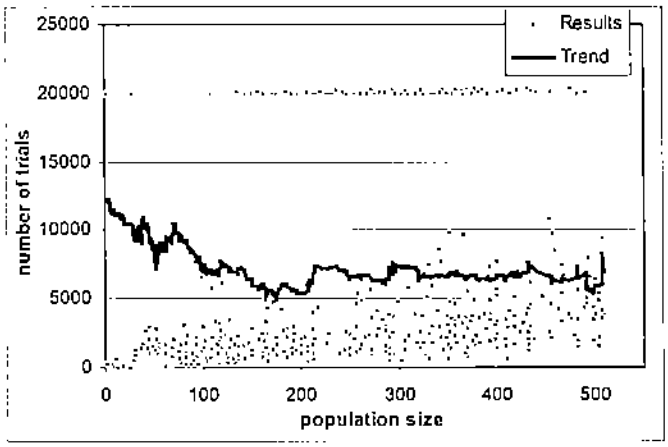


Figure 2: The amount of trials as the function of population size when solving the polyomino problem

Figure 2 illustrates that quite often the optimum is not found before reaching the stopping condition. The figure also shows a moving average trend, the trend clearly illustrating that actually the speed increases when population size decreases. This is because the risk of not finding the solution increases with decreasing population size. This illustrates the importance of population size on finding the optimum and also the risk involved. In practice the optimal population size should be a compromise between efficiency and tolerable risk.

2.2 All ones and max sum problems

In the all ones problem (onemax) [21] the fitness function is the number of one bits of the 34 bit long chromosome. This problem has one maximum, which obviously has the value 34. This problem has been largely used to test GAs.

The fitness function of the max sum problem is the sum of certain fields of the parameter array, seven pieces: 4, 4, 5, 5, 6, 6 and 4 bits long each. Actually the fields correspond to those used to encode the polyomino tile coordinates. The optimal chromosome for both the all ones and the max sum problem is identical; all 34 bits assigned one. However the fitness function is different. Max sum has one maximum, which is 233 (15+15+31+31+63+63+15). This problem may sound easy, but it isn't necessarily the case for GA, because even smaller numbers may have several ones, f.e. 15 have 4 ones, but 16 only one, so there are several local optimae. This problem is linear, and thus relatively easy to solve.

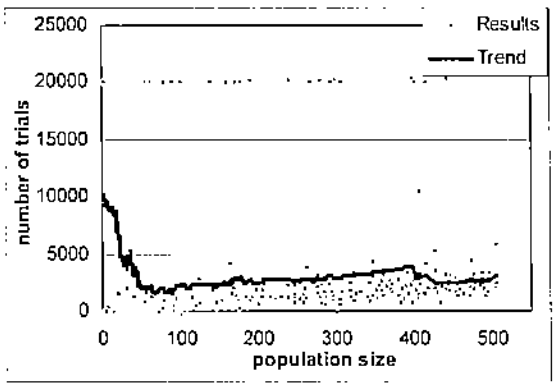


Figure 3: Distribution of trials as the function of population size when solving the all ones problem

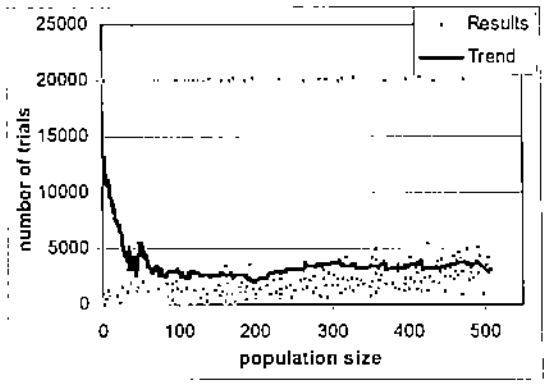


Figure 4: Distribution of trials as a function of population size when solving the max sum problem

In conclusion figures 2-4 show the interesting phenomenon, that the smaller the population sizes the faster the processing. Unfortunately at the same time the risk of not finding the optimal solution increases. This observation can be made also from the trend curves.

The all ones problem didn't seem to be sensitive to any other tested parameters than population size. The max sum problem also didn't seem to be very sensitive to any other GA-parameter values than population size, perhaps with the exception that with this problem GA seems to favor large mutation probability. All these observations lead to the conclusion that the population size has a strong influence on processing speed and the risk involved. The other GA parameters didn't seem to have such great effect. GA is robust and capable of finding a good solution, even if the parameters were widely changed.

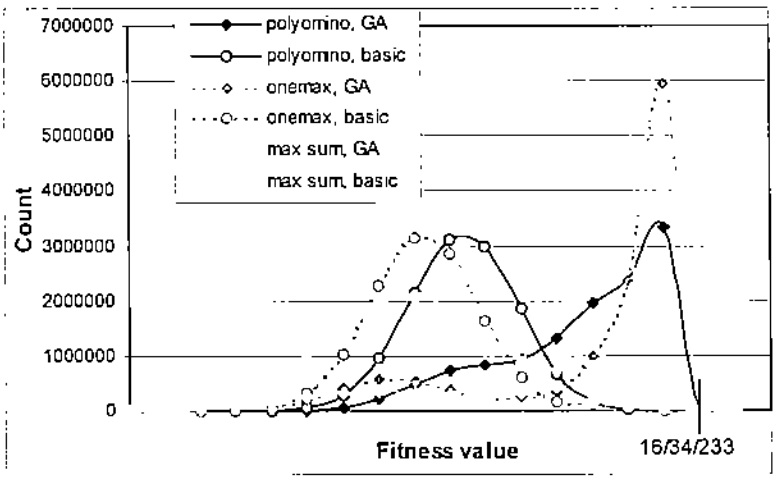


Figure 5: Basic solution (o) distributions of the test problems, and the corresponding distributions due to GA optimization (♦).

When optimizing these three problems with GA, we achieve the typical solution distributions due to GA optimization (figure 5), which differs quite a lot from basic distributions (also shown in fig. 5), which seems to be

almost Gaussian. As expected GA heavily favors the solutions near the high end of the solution space. Figure 5 was created by making 1500 GA optimization runs with random GA parameters for each of the three example problems. For practical reasons parameters were limited: population size between 2 to 511, elitism between 1 to population size-1, and the other parameters (crossover, mutation, change-operation and creating new rates) between 1 to 99 %. The basic distributions of the function values were created by generating as many random trials as GA used in the corresponding case.

As stated earlier the onemax and max sum have the same optimal chromosome, however the distribution of the values of the max sum function is wider and has less kurtosis. However, under GA optimization these two problems achieve nearly identical trial distribution.

3. Simulating temporal fitness functions

The goal was to create temporally nondeterministic fitness functions. This was realized by letting the fitness function be another (secondary level) GA that solved one of the above example problems. GA-optimization run is strongly nondeterministic taking different amounts of time (figs 2-4), even with the same GA-parameters, because of the different solution paths. The top level GA then optimizes parameters of the second level GAs.

The fitness function for these tests was the time that genetic algorithm needed to solve the given problem with the given input parameters: population size, amount of elitism, crossover-, mutation-, changing-, and creating new probabilities. This setup results a quite nondeterministic target system, a system that was though simulate e.g. the testing of response times of real time embedded systems.

A system can be optimized with two different ways, either minimizing or maximizing the response time of the secondary level target system. This corresponds to either trying to optimize the parameters of the secondary GA, so that the target problem is solved as fast as possible, or to find bad GA parameters that lead to long problem solving time. The latter one corresponds more to the testing of real time systems and finding the bottlenecks and problematic parameter combinations that further lead to long response times. Figure 6. shows the results of minimizing and maximizing the target system response time while using the polyomino problem as the second level target problem.

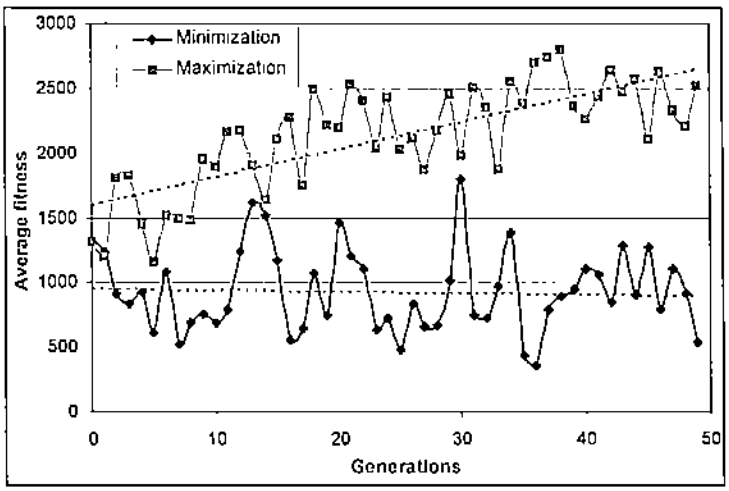


Figure 6: The average response times via generations when the secondary GA solves the polyomino problem.

The maximizing of the response time seems to be quite easy, i.e. bad GA input parameters are easy to find. The trend when maximizing response time is clearly increasing. While when minimizing the response time progress fairly slowly, hence it seems that good GA parameters are quite hard to optimize. Fig. 6 also shows that with both optimizations the average response time is similar at the start. The curve of generation average has large fluctuations, which is due to the nondeterministic nature of the problem. Because of the nondeterminism the survivors were tested again in the next generation.

In figure 7 we can see similar trends as in fig. 6, but this time the target problem is all ones. This problem seems to be more sensitive to the GA parameters than the corresponding polyomino case, because maximized response time takes a large leap at the beginning. Hence bad GA parameters are even easier to find with this problem. The averages of the response times for the minimizing and maximizing cases are much further apart after the start than for the polyomino problem.

Figure 8 represents further the evolution for the max sum problem. In the maximization case the response time increases strongly, so it seems that GA easily finds bad parameters.

Let us take a closer look at the test run log and especially at the input parameters that caused the long response

times: The bad input parameter combination for the max sum problem seems to be high population size and low elitism. For the other problems the bad combination seems to be such in which the elitism is either low or almost as high as the population size. Other parameters didn't seem to have any clear effect on response time. When using the all ones problem the bad parameter combination seems to lead to quite similar response time each time. Other problems seem to be more nondeterministic; hence the same parameter combination causes more variety to the measured response time.

The operation system adds even more nondeterminism into the system and that causes sometimes irregular abnormally long response times.

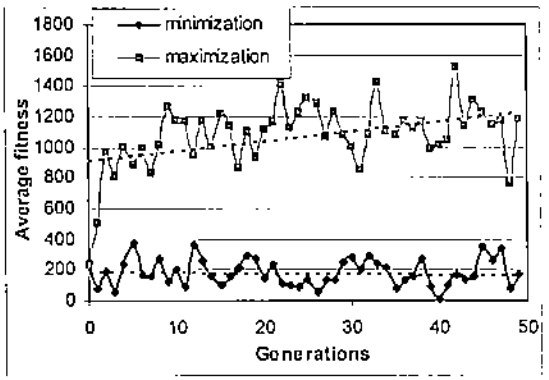


Figure 7: Response times trends when using the all ones problem as the secondary level target problem

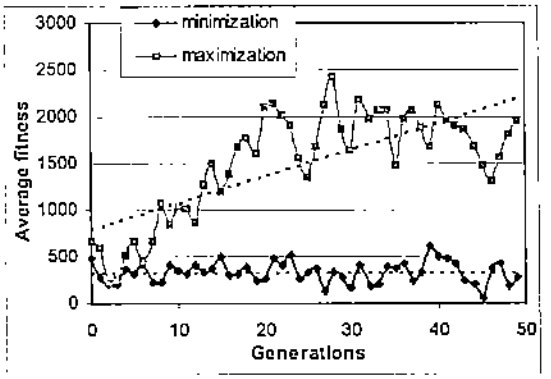


Figure 8: Response times trends when solving the max sum problem as the secondary target function

All the above tests show that even with a nondeterministic target function GA can find input parameter combinations that cause a long response . This finding is promising when designing response time testing of critical real time software. Hence it seems that these tests confirm our earlier findings with real time software testing [2, 3].

4. Discussion and conclusions

This simulation and modeling was mainly done to see if response time can be used as an optimization goal and if GA is able to find problematic parameter values when the target system is nondeterministic. When using only response time as the target function there is clearly a risk that the whole search base is not checked, i.e. it doesn't by any means verify high code or path coverage. However, when the goal is not so much to find bugs, but to confirm fast response to all possible signals and messages, this optimization method can be used to find bottlenecks, i.e. difficult input parameter combinations that cause long response times.

The results confirm that GA based testing is able to find problematic parameter combination even with nondeterministic target functions.

Acknowledgements

The research work of Timo Mantere has been funded by the Finnish Cultural Foundation. Lilian Grahn is acknowledged for her help with the proofreading of this paper.

References

[1] Myers, G. J. (1978). *The Art of Software Testing*. John Wiley&Sons. New York.
[2] Norman, S. (1993). *Software testing tools*. Ovum Ltd. London.
[3] Alander, Jarmo T., Timo Mantere, Ghodrat Moghadampour and Jukka Matila. Searching protection relay response time extremes using genetic algorithm – software quality by optimization. *Electric Power Systems Research 46*, 1998, pp. 229-233.
[4] Alander, Jarmo T., Timo Mantere. Automatic software testing by genetic algorithm optimization, a case study. In Conor Ryan and Jim Buckley (eds.) *SCASE'99 - Soft computing applied to software engineering*, 11.-14.4.1999, Limerick, Ireland, pp. 1-10.
[5] Holland, John (1975). *Adaptation in Natural and Artificial Systems*. University of Michigan Press. Ann Arbor, MI, Reissued by The MIT Press, 1992.

[6] Darwin, Charles (1859). The Origin of Species: By Means of Natural Selection or The Preservation of Favoured Races in the Struggle for Life. Oxford University Press, London. A reprint of the 6th edition, 1968.

[7] Lachut-Watkins, Alison (1995). The automatic generation of test data using genetic algorithms. In Marshall, I. M. et. al. (eds.) (1995), *Proceedings of the 4th Software Quality Conference 4-5 July 1995*. University of Abertay, Dundee, Scotland.

[8] McGraw, Gary, Christoph Michael and Michael Schatz (1998*). Generating Software Test Data by Evolution.* Technical Report RSTR-018-97-1, RST Corporation, Suite #250, Ridgetop Circle, Sterling, VA 20166. February 9, 1998.

[9] Pei, Min, E.D. Goodman, Zongyi Gao, and Kaixiang Zhong (1994*). Automated Software Test Data Generation Using A Genetic Algorithm.* Technical Report GARAGe of Michigan State University June 1994.

[10] Roper, Marc, Iain MacLean, Andrew Brooks, James Miller and Murray Wood (1995). *Genetic Algorithms and the Automatic Generation of Test Data.* Technical report RR/95/195 [EfoCS-19-95]. University of Strathclyde, Livingstone Tower, Richmond Street, Glasgow G1 1XH, U.K.

[11] Smith, Jim and T. C. Fogarty (1996). Evolving software test data - GA's learn self expression. In T. C. Fogarty, editor, *Evolutionary Computing*. LNCS 1143, Springer-Verlag, Berlin.

[12] Bäck, T. and Schwefel, H.-P (1993). An overview of evolutionary algorithms for parameter optimization. *Evolutionary Computation*, 1(1), pp. 1-23.

[13] Lankhorst, Marc M. (1995). *A Genetic Algorithm for the Induction of Nondeterministic Pushdown Automata.* Computing Science Report CS-R 9502. University of Groningen, Netherlands.

[14] Grefenstette, John J. (1986). Optimization of control parameters for genetic algorithms. *IEEE Trans. Systems, Man, and Cybernetics*, SMC-16(1), pp. 122-128.

[15] Friesleben, B. and Hartfelder, M. (1993). Optimisation of genetic algorithms by genetic algorithms. In Albrecht, Reeves, and Steele (editors), Artificial Neural Networks and Genetic Algorithms, pp. 391-399. Speinger Verlag.

[16] Alander, Jarmo T. (1991). On finding the optimal genetic algorithms for robot control problems. *In Proceedings IROS '91 IEEE/RSJ International Workshop on Intelligent Robots and Systems '91*, volume 3, pp. 1313-1318, Osaka (Japan), 3.-5. November 1991. IEEE Cat. No. 91TH0375-6.

[17] Alander, Jarmo T. (1992). On optimal population size of genetic algorithms. In Patrick Dewilde and Joos Vandelwalle, editors, *CompEuro 1992 Proceedings, Computer Systems and Software Engineering, 6th Annual European Computer Conference*, pp. 65-70, The Hague, 4.-8. May 1992. IEEE Computer Society, IEEE Computer Society Press.

[18] Alander, Jarmo T. (2000). *An Indexed Bibliography of Genetic Algorithms Theory and Comparisons*, Draft March 16, 2000. Department of Information Technology and Production Economics, University of Vaasa, Report Series No-94-1-Theory. (Available via ftp: ftp.uwasa.fi/cs/report94-1/gaTHEORYbib.ps.Z)

[19] Alander, Jarmo T. (1999). Population size, building blocks, fitness landscape and genetic algorithm search efficiency in combinatorial optimization: an empirical study. In Lance D. Chambers, editor, *Practical Handbook of Genetic Algorithms*, CRC Press LLC, Boca Raton, pp. 459-485.

[20] Syswerda, Gilbert (1989). Uniform Crossover in Genetic Algorithms. In J. David Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, George Mason University, June 4-7, 1989. Morgan Kaufmann Publishers, Inc. San Mateo, California.

[21] Höhn, Cristian and Colin Reeves (1996). The crossover landscape for the onemax problem. In Alander, Jarmo T., Editor, *Proceedings of the Second Nordic Workshop on Genetic Algorithms and their Applications (2NWGA)*, Proceedings of the University of Vaasa, Nro. 11, Vaasa (Finland), 19-23. August 1996.

ELSEVIER

Electric Power Systems Research 46 (1998) 229–233

ELECTRIC POWER SYSTEMS RESEARCH

Searching protection relay response time extremes using genetic algorithm—software quality by optimization

Jarmo T. Alander [a,*], Timo Mantere [a], Ghodrat Moghadampour [a], Jukka Matila [b]

[a] *Department of Information Technology and Industrial Economics, University of Vaasa, PO Box 700, FIN-65101 Vaasa, Finland*
[b] *ABB Transmit Oy, Relays and Network Control, PO Box 699, FIN-65101 Vaasa, Finland*

Abstract

In this work we study possibilities to automate the searching and measuring of response time extremes of protection relay software using genetic algorithm optimization. The idea is to produce test cases to reveal potentially problematic situations causing processing time extremes in the software of an electric network protection relay. The testing was done using a relay software in a simulator environment. In the comparison performed, the genetic algorithm-based method was clearly better than a pure random test method.

Keywords: Genetic algorithms; Distribution systems; Real time control; Safety; Simulation; Software testing

1. Introduction

Failure in the software for a power distribution system may have severe consequences; thus, the requirements for precision and reliability are high. Testing is very laborious, because the amount of work grows exponentially with the code size, which is already considerable and steadily growing with new functions. Testing software manually is slow, and thus expensive, and in addition demands inventiveness and highly skilled personnel. Automated testing can reduce both time and costs needed for performing tests. Studies show that two thirds of the development costs of embedded systems are caused by software development, about half of these are testing costs [1]. The most common way of generating test data is random, which is considered a weak method [2]. For this reason, efforts have been made to optimize data using various methods, including heuristic ones.

In our method, genetic algorithm creates (generates, simulates) and sends data inputs to the tested relay program: CAN and LON messages, manual user commands and status information from the electric network. The time that tested programs take to process the sent information (response time) is used as a fitness function. These response time extremes are searched by genetic optimization.

* Corresponding author. E-mail: jarmo.alander@uwasa.fi

0378-7796/98/$19.00
PII S0378-7796(98)00013-3

2. Genetic algorithm

Genetic algorithm is a nondeterministic optimization method that imitates natural evolution. The possible solution space of optimized problem is encoded as bits (genes) and bit strings (chromosomes). In this case, one individual represents test cases, they contain data sections for CAN and LON messages, time variables for the time between two CAN messages and the time between a CAN message and a possible LON message. the individual also contains variables that tell the GA whether a LON load message is going to be sent and whether the CAN message is sent with the address of I/O card 1 or 2. The initial population is usually created at random. After the initial population is created, we test the individuals in it one at the time by sending the messages expressed in the individual to the tested relay program. We wait for the response message, usually a CAN message, that represents new values for I/O card outputs. The time between the sent CAN message and the response message is used as a fitness value (response time). After we received fitness values for all individuals, we replace half of the population by creating new individuals by crossover and mutations. This occurs so that we sort the population by fitness value and remove the half with the poorest fitness values and then we create new individuals by crossover, which means that

we select two individuals from the population that was left and mix their chromosomes (bit strings) by randomly selecting part of the one or the other parent, then we change some bits in the new individuals (mutation). The best individuals from the previous generation are kept for the new generation, so that their good qualities will not be lost (elitism).

In our case, population size had varied between 30 and 50. While creating the data of the Section 6, it was 40, from which 20 were renewed to the next generation. Mutation probability has varied between 4 and 30% depending on to which variable is under reproduction. Both single-point and multipoint crossovers were used. At first, old individuals were not tested again; but later they were. Because of the nondeterminism of the testing system, the tested individuals do not have the same fitness values every time.

GA is suitable when an optimized function cannot be solved with more accurate deterministic methods. In combinatorial optimization like this, it is impossible to prove that the found optimum is a global one, or to determine how close to the optimal solution we are. It is usually also quite difficult to predict the location of the global extremum from (random) samples.

2.1. Related work

Statistical testing of software is interesting but difficult. It involves excercising a system with random inputs, the input distribution and the number of test data being determined according to given test criteria. It is, unlike random testing, based on a probabilistic generation of test data. It may be appropriate in situations where, for instance, the mean execution time or the probability of violating a deadline are of interest. Since operational input distribution is not usually available at the program unit level, statistical structural test sets may be used as test data for testing the execution time [3]. In cases where the probability of applying an input does not change during the execution of the software, using statistical aspects of the testing process, such as reliability, mean time to failure (MTTF) and mean time between failures (MTBF) [4].

The automatic generation of test data using GA has been studied by Xanthakis et al. [5], Davies et al. [6], Jones et al. [7], Watkins [8], Kasik and George [9], Alba and Troya [10], Boden [11], Gunavathi and Shanmugam [12], Smith and Fogarty [13] and Roper [14]. See [15] for some further references. Studies have been mostly based on the white-box testing methodology. However, here we are using a black-box technique.

Recently there has been a growing interest to use GA-based methods to test VLSI circuits [16–19]. See [20] for further references.

3. Statistical software testing

Events that possibly cause software failure do not have the same probability of occurence. Consequently, the likelihood of different future events are assessed through a probability distribution, which is based, when possible, on observational data. The most interesting part in the probability distribution is the area which does not have observational data, but is of low probability and high consequences.

Extreme events are of low probability and high severity and their risk is represented by the tails of a probability distribution. Even though many common methods of distribution selection may represent the central tendency, however, they do not represent extreme events accurately.

Extreme event distributions may be used to assess risks in three different situations: situations of ample data, situations of sparse data and situations of no data. The idea of selecting extreme event distributions is to simulate occurrence of events with a low likelihood.

4. Network protection relay

The relay is a microprocessor-based embedded system and it normally contains several processing and interface units that communicate with each other, and relay and control units of the distribution network. It is used for protection, control, measuring and supervision of the radial electric network [21].

Protection functions of the relay terminal include, e.g. overcurrent (also directional), earth fault (also directional), residual voltage, over/undervoltage, thermal, breaker failure and automatic autoreclosing.

Control functions of the terminal include: controlling of network (e.g. circuit-breakers), status indication and interlocking.

Measurements that the terminal performs are: phase currents and phase-to-phase voltages, residual current and voltage, active and reactive power and energy demand values of current and power, frequency, power factor, disturbance recorder, harmonics and event buffer.

Supervision of the terminal: trip circuit supervision for two outputs, circuit breaker ready, SF6 gas pressure and internal self supervision of the relay.

The terminal has two independent serial ports for communication; they support the protocols: LON bus [22], SPA bus and VDEW6.

Because of the nature of the task, protection relay should work pretty fast, it should react within 30 ms (legal norm) from the discovery of a serious failure.

Time demand for the embedded relay software is even shorter, it should react in 20 ms. However, in the

simulated environment response times of relay software are naturally somewhat longer than those in the target environment, and in addition contain some noise.

5. Simulation system

In this work we have used an ESIM simulator environment, which is designed to be a test automation tool for embedded real-time software development. ESIM makes it possible to use a workstation PC for testing embedded system software written in the C or C++ programming languages.

ESIM provides a set of generic services (building blocks) for simulating real-time operating systems. The user describes the input and output system (the application-specific hardware). ESIM then simulates the I/O system and the operating system of the application, allowing the user to monitor what is happening in each of them. Application itself is unchanged, but the hardware drivers are modified so that instead of handling physical hardware objects, they call the service functions provided by ESIM [1].

The GA and the tested program run separately in their own ESIM tasks, which communicate with each other through simulated ESIM hardware ports. Both programs, GA and the tested program, run under their own ESIM applications, either in the same Windows NT workstation or both in their own NT, which are connected together by a LAN.

GA sends simulated input to the tested program and measures the response time (Fig. 1). The response time is the time it takes for the tested program to process the operations caused by the input parameters it received. When the tested program is complete, it sends a response signal. The response time is the fitness value, by which the search is guided.

GA is interfaced to the relay program via ESIM in three different interface points. It has direct access to the memory database (shared memory area) of the relay software, so that it can read and write information directly. Second, it is interfaced to the internal communication lines (CAN bus) of the relay terminal, where it can send and receive control messages. Third, it is interfaced to the LON bus connection point, where it can send and receive network messages.

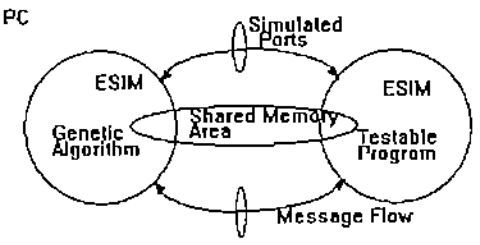


Fig. 1. Program testing by GA in a simulated environment.

Table 1
Descriptive statistics of the total response times for GA and random methods

	GA-total	Random-total
Mean (ms)	233.45	190.86
S.D. (ms)	116.66	87.85
Count	4750	4751
Minimum (ms)	90.00	80.00
Maximum (ms)	761.00	811.00
Range (ms)	671.00	731.00
Skewness	1.20	1.28
Kurtosis	1.39	2.61
Median (ms)	200.00	161.00

GA simulates

- internal communication inside the relay terminal between its CPU and digital input/output cards and manual commands interface card (CAN bus). GA sends and receives the internal messages to/from these cards through internal message lines.
- the electric network information, such as information from the conventional current and voltage transformers or new current and voltage sensors by writing information directly to memory database of the relay program.
- other relays in the LON network by sending network messages to the relay program.

6. Results

The first test case was to compare the GA-based method with a simple random testing method. The most important communication task in the relay is the communication between digital input/output cards and the CPU (CAN bus). In this test, load of the CPU is increased by sending lower priority LON network messages.

In order to examine the performance of the GA-based input data generator, we ran the GA and random tester 20 times on a simulator. In order to combine the input data cases generated by both methods, we used descriptive statistics (Table 1).

As Table 1 shows, it is obvious that the average and the median processing times of a test case generated by GA are longer than of the test case generated by the random method. S.D. in the GA method is also larger, but the processing time distribution range is narrower.

Notice that the values shown in Table 1 are from the simulated environment, and they are over 10 times longer than target environment times, but are precise enough proportionally for diagnostics and a good starting point for further analysis.

For the most reliable results, we need to repeat the experiment with the real relay. However, it envolves a lot of electronic and logical interfacing work.

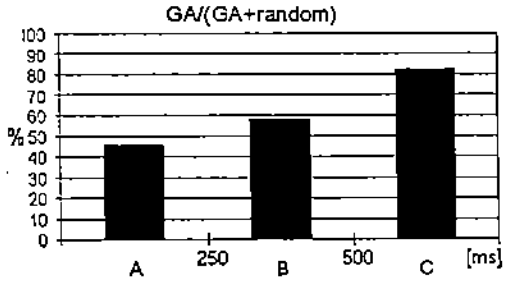


Fig. 2. Relative distribution of total response time for test case in three intervals.

The relative number of generated test cases GA/(GA + random) in three response time intervals (A = [0, 249] ms, B = [250, 499] ms, $C \geq 500$ ms) are shown in Fig. 2, when exactly the same amount of test cases were generated by the GA method and the random method. It seems obvious that GA generates more input data cases with longer response times. The random method generates 12% more cases in the shortest (poor) time range A, while in turn the GA method generates 41% more input data cases in the middle range B and 376% more cases in the extremal time range C.

By calculating the statistical significance of the difference between average processing times, we can be 95% confident that the average response time of the test case generated by GA is at least 15%, and at most 19%, longer than the average response time of the test case generated by the random method.

The second test case was to determine in which phase during the internal message handling and processing (messages between CPU and digital I/O cards) the LON bus message delays the response the most (Fig. 3).

In Fig. 3 it can be observed that if there is no LON bus message load for CPU, I/O card messages are handled faster than in cases that there are. Also, the state the LON message is received seems to have an effect on the response time. The response in case B is on average 77% longer that in case A, and respectably, in cases C and D, 27 and 53% longer than in case A.

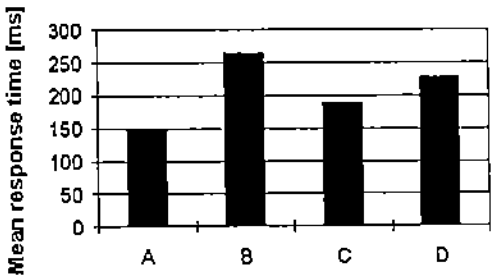


Fig. 3. Mean response times in four cases: A, no LON load for relay; B, LON message is sent simultaneously with I/O card message; C, LON message is sent after I/O card message; and D, LON message is received after CPU sends response message to I/O card.

7. Conclusions and future

It seems that GA-based search can be used for searching protection relay software response time extremes, when testing relay software and its reliability. The problems might be in finding a characteristic fitness function and factors that influence the test results. Problem complexity and nondeterminism might also cause problems, which increases the difficulty of software testing [8,23].

There seems to be significant differences between test data cases produced by GA and the random method, with GA clearly a better method.

Future research should concentrate on more precise analysis of message interruptions, study the impact of frequency of sending I/O card messages, and influence of the time between the sent I/O card message and the LON bus message.

The new version of the tested software is under implementation in the test simulation environment. It includes more modules from the actual target environment. Some modules were left off from the first implementation in order to make adjustment to simulated environment easier. Tests with the updated sortware version include first the comparison to the old measurements and then some new measurements that will be made possible by the new version.

The adjusting of the system into a test bench that contains an actual relay terminal is also underway. A test with the real protection relay unit should contain comparable tests with simulated environments first, and then more tests that become possible by using the whole relay unit.

Further information on this work can be found in our anonymous ftp server (`ftp.uwasa.fi`) in directory `cs/report97-5`.

Acknowledgements

This work is supported by Finnish Technology Development Center (Tekes) and ABB Corporate Research.

References

[1] G.J. Meyers, The Art of Software Testing, Wiley, New York, 1979.

[2] W. Schütz, P. Thévenod-Fosse, Using statistical testing to assess the execution time of program units, Tech. Rep., LAAS-CNRS, Toulouse, France, 1993.

[3] E. Davies, J. McMaster, M. Stark, The use of genetic algorithm for flight test and evaluation of artificial intelligence and complex software systems. Report AD-A284824, Naval Air Warfare Center, Patuxent River, MD, USA. 1994.

[4] A.L. Watkins, The automatic-generation of test data using genetic algorithms, in: I.M. Marshall, W.B. Samson, D.G. Edgar-Nevill (Eds.), Proc. 4th Software Quality Conf., vol. 2. University of Abertay Dundee, Scotland, 4–5 July 1995, pp. 300–309.

[5] S. Xanthakis, C. Ellis, C. Skourlas, A.L. Gall, S. Katsikas, K. Karapoulios, Application of genetic algorithms to software Engineering, Proc. 5th Int. Conf. on Software Engineering, Toulouse, France, 7–11 December 1992, pp. 625–636

[6] A.N. Partner, REF 541 Feeder Terminal, ABB Network Partner, Vaasa, Finland, 1996.

[7] K. Gunavathi, A. Shanmugam, Genetic algorithm-based performance analysis of two new distributed MAC protocols for LAN, in: P. Ošmera (Ed.), Proc 3rd Int. Mendel Conf. on Genetic Algorithms, Optimization problems, Fuzzy logic, Neural networks, Rough Sets (MENDEL '97), Technical University of Brno, Brno, Czech Republic, 25–27 June 1997, pp. 59–64.

[8] H.-D. Chu, J. Dobson, A statics-based framework for automated sotware testing, Tech. Rep., Department of Computer Science, University of Newcastle upon Tyne.

[9] Prosoft, ESIM—Testing environment for embedded sortware, User's Guide Version 2.1 for Windows NT. Prosoft, Oulu, Finland, 1995.

[10] J.H. Aylor, J.P. Cohoon, E.L. Feldhousen, B.W. Johnson, Compacting randomly generated test sets, In: Proc. 1990 IEEE Int. Conf. on Computer Design: VLSI in Computers and Processors, IEEE Comp. Soc. Press, Cambridge, MA. 17–19 September 1990, pp. 153–156.

[11] M.S. Hsiao, E.M. Rudnick, J.H. Patel, Automatic test generation using genetically engineered distinguishing sequences, In: Proc. 14th IEEE VLSI Test Symposium, Princeton, NJ. 28 April–1 May 1996, IEEE Comp. Soc. Press, Cambridge, MA, 1996, pp. 216–223

[12] M. Roper, CAST with GAs—automatic test data generation via evolutionary computation, In: IEE Colloquium on 'Compter Aided Software Testing (Cast) Tools', vol. IEE Digest No. 1996/096, London, 23 April 1996, IEE, London, 1996, pp. 7/1–7/5.

[13] E.B. Boden, G.P. Martino, Testing software using order-based genetic algorithms, in: J.R. Koza, D.E. Goldberg, D.B. Fogel, R.L. Riolo (Eds.), Proc. GP-96 Conf., Stanford, CA, 28–31 July 1996, MIT Press, Cambridge, MA. 1996.

[14] F. Corno, P. Prinetto, M. Rebaudengo, M. Sonza Reorda, GATTO: a genetic algorithm for automatic test pattern generation for large synchronous sequential circuit, IEEE Trans. Comput. Aided Des. Integrated Circuits 15 (8) (1996) 991–1000.

[15] J.T. Alander, Indexed bibliography of genetic algorithms in electronics and VLSI design and testing. Report 94-1-VLSI, Department of Information Technology and Production Economics, University of Vaasa, 1995. (ftp.uwasa.fi:cs/report94-1/gaVLSIbib.ps.Z)

[16] J. Smith, T.C. Fogarty, Evolving software test data—GAs learn self expression, In: Evolutionary Computing, Proc. AIS B96 Workshop, Brighton, UK, 1–2 April, 1996, pp. 227–235.

[17] J.T. Alander, Indexed bibliography of genetic alogorithms in computer science. Report 94-1-CS, Department of Information Technology and Production Economics, University of Vaasa, 1995. (ftp.uwasa.fi:cs/report94-1/gaCSbib.ps.Z)

[18] D.J. Kasik, H.G. George, Toward automatic generation of novice user test scripts, Proc. 1996 Conf. on Human Factors in Computing Systems, CHI 96, Vancouver, BC, Canada, 13–18 April 1996, ACM, New York, NY, pp. 244–251.

[19] E.M. Rudnick, J.H. Patel, Combining deterministic and genetic approaches for sequential circuit test generation, In: Proc. 32nd Design Automation Conf., San Francisco, CA, 12–16 June 1995. IEEE, New York, pp. 183–188.

[20] E. Alba, J.M. Troya, Genetic algorithms for protocol validation, In: H.-M. Voigt, W. Ebeling, I. Rechenberg, H.-P. Schwefel (Eds.), Parallel Problem Solving from Nature—PPSN IV, vol. 1141 of Lecture Notes in Computer Science, Springer, Berlin, Germany, 1996, pp. 870–879.

[21] B.F. Jones, H.-H. Sthamer, D.E. Eyres, Automatic structural testing using genetic algorithms, Software Eng. J. 11 (5) (1996) 299–306.

[22] Anonymous, LonTalk Protocol Specification, Version 3.0, Number 19550, Echelon Corporation, Palo Alto, CA. 1994

[23] A. Auer, J. Korhonen, State testing of embedding software, in: EuroStar-95, London, UK, 1995.

AUTOMATIC SOFTWARE TESTING BY GENETIC ALGORITHM OPTIMIZATION, A CASE SUDY

JARMO T. ALANDER AND TIMO MANTERE

ABSTRACT. In this work we have studied the feasibility of program testing automation by using optimization via genetic algorithm. The main objective of the study is to find potentially problematic situations by maximizing the response times of a rather large real-time embedded system. The results show that the genetic optimization is able to outperform pure random testing. The identification of the problematic input helps the programmer to fix performance bottlenecks.

1. INTRODUCTION

Performance and reliability are among the most important quality criterion in the planning, realization and use of computer systems. The objective is to guarantee at least the minimum performance in any case. For example in some real-time systems absolute response time constraints are set. In general speed is an important competition factor in software markets. Unfortunately it is difficult to reach good performance in practice.

Requirements for precision and reliability of embedded software are usually high, e.g. a failure to meet a response time constraints might have severe consequences both in economics and human terms. It has been estimated that about two thirds of the total costs of product development projects of embedded systems consist of software development work, of which about half is testing costs [17]. Thus software testing will cause up to one third of the total product development costs. Testing is laborious, because it tend to increase exponentially with the code complexity. Manual testing is slow and thus expensive. Software grows and becomes more and more complicated bringing with it problems related to its performance.

It has been said that when testing software, one should examine only small sections at a time, otherwise it easily results that somewhere is a fault, but it is extremely difficult to locate. However in this work the goal was to test the whole software as one piece. In larger software the performance problems are usually emphasized. There can be millions of lines of source code produced by hundreds of programmers. Even if there was no performance problem in the code written by the individual programmer, such will easily appear in the integration phase. Testing is a vital part of software development and automation makes it faster, more reliable and cost efficient.

1.1. **Research problem.** The object of this study is software testing automation; how to realize it using a combinatorial optimization method called genetic algorithm (GA). A real-time embedded system must usually react within a given response time. This can be seen as an optimization problem: find those i.e. problem cases, causing the longest response times.

The object of this study is a communication module of a rather large embedded software, consisting of about 11 Mb of source code written in C. We performed automatic black-box type stress testing, by using a GA, that optimizes input parameters, stimuli from the environment to

Key words and phrases. Software engineering, software testing, combinatorial optimisation, embedded systems, genetic algorithms, simulation.

the software. The objective was to find the extreme response times, in order to reveal bottlenecks, which must be corrected. A simulation environment was used mainly for cost reasons.

Related work. In the field of software testing GA has been applied mainly to the automatic test data generation [13, 14, 16, 18, 19, 20, 22, 25, 26, 27]. These studies have concentrated on the optimization of branch coverage; in other words finding test sets that covers all possible paths of the program. Especially reference [13] is interesting because in it several problems are dealt with both a GA and random method. For more references on GA in software testing see the bibliography [2]. A closely related application area of GAs is VLSI testing [3].

2. Genetic algorithms

The genetic algorithms have steadily reached growing popularity during the last twenty or so years as a general method to solve difficult optimization problems [2]. The method is based on the recombination of genes in the same way as it takes place in nature. This combined to selection leads to the growth of the average fitness of the population. GA is not perhaps the best possible method in a given task but it is robust and more or less suitable for most complex optimization tasks.

A GA forms a sort of artificial simple electric ecosystem, where digital beings (bit-strings, that represent parameter values) struggle for survival and reproducing possibilities. GA is a method especially suited for high power digital computing. It is at its best when facing problems that can't be solved with exact methods or at least within a given processing time. GA is often applicable when the number of parameters or their variations is considerable i.e. when combinatorial explosion prevents us from testing all possibilities. The reason for the growing popularity of genetic algorithms is probably the fact that GA does not set any *a priori* restrictions on the target function. It does not need to be smooth, meaning it does not need to be derivative or even be continuous and it can have several local optimum. The target function doesn't even need to be expressible in a mathematical form, if one can e.g. measure fitness somehow.

The study of genetic algorithms started in 1975, when John Holland developed the first GA at the University of Michigan [11]. He utilized evolutionism to number sequences that would live, reproduce and die like living organisms. He got the idea after having been convinced that living organisms can solve an optimization problem like adaptation to its natural environment better than even the most powerful supercomputers. In order to describe it, Holland borrowed terms from the glossary of genetics. However, all the concepts can be interpreted exactly and are easy to implement.

Benefits of GA include that it can easily be run on parallel processors and it can easily be adapted to different problems. It doesn't provide much programming and is usually relatively efficient in difficult problems. GAs belong to the so called soft computing methods, which include such methods as artificial neural nets and fuzzy logic.

3. Software testing

Software that has been compiled is syntactically correct, but it usually contains an unknown number of semantic errors. These errors are searched by code validation or by running the program using some testing method [12, 17, 24]. The amount of testing is not necessarily equivalent to the effectiveness of testing: a few carefully planned test cases can lead to better results than an intensive random experiment. However, one should not forget that even the best tests cannot save a badly designed program.

A comprehensive test would require the testing with at least every acceptable input. The test result depends not only on input but also on the state of the program. Thus input should be tested with all combinations of different states. Unfortunately comprehensive testing is impossible in practice. This does not mean that investment to testing would not be worthwhile, but that the functionality of the program should not be trusted too much.

3.1. **Automation of testing.** It has been estimated that software testing takes 30-50% of the working time of a software project [9]. In terms of manpower testing is expensive and susceptible to errors. Already partial testing automation will bring significant savings and improves program quality. Software testing automation can benefit from the following [7]:

- The testing can be prepared even before all the necessary tools are available; this was also partly the case in this study.
- It accelerates the testing essentially.
- The tests can be done at least partly unmanned.
- The repeatability helps to locate also infrequent error situations.
- It reduces the amount of routine work.
- Opportunity for remote testing.

There are also some drawbacks:

- The planning of tests and analysis of results may be laborious.
- Requires better trained testing tool developers and users.

Software can be so complex that one cannot easily deduce what response results from a given input. Thus determining the proper test data can be difficult. Software may be so non-deterministic that the response time is due to more the non-deterministic nature of the software and testing system, than the given input.

4. Embedded systems

Embedded system refers to a computer system that has been integrated into an electro-mechanical device. The electronics part of the embedded system controls electric and mechanical functions of the device. The processor of the system executes the so-called embedded software, which is to read impulses from the environment and to process responses to them. Usually the embedded software is real-time, this means that it has to respond within given time constraints.

4.1. **Real-time software.** The operation of any sequential program has a well defined beginning and end, while a real-time software is in an infinite loop constantly interacting with its environment. An impulse originating from the environment starts a series of events, which results in some sort of response. Real-time software is usually required to meet some response time constraints.

The operation of the real-time software is often presented with the help of the so-called state-machine model. In a simple state-machine only the state, during which the impulse occurs affects the response. The actual software is seldom in accordance with this simple model, however. The actual software can process several impulses overlapping causing also the responses to have tendcy to overlap, resulting in sometimes unexpected and unforeseeable interactions. The load variations can cause dramatic behavioral changes. Thus it is not usually easy to predict the behaviour of the system.

5. Experimental setup

The object embedded system has been designed to carry out many functions as a power distribution protection device (relay) control unit. It is responsible for the primary protection functions

of the device and auxiliary functions including controlling functions, measuring functions and surveillance procedures. The device contains actually several processing and interface units, which communicate with each other and other protection devices. The system have two separate serial ports, which support several communication protocols.

On the whole the system hosts a quite demanding software, which should operate fast and reliably in all circumstances. It was still under development during our project.

of

5.1. **Simulation environment.** The testing was done using a simulation environment, which was chosen originally because of cost reasons. Another obvious reason was that the hardware was not yet completed when the project started.

The simulation environment was ESIM, a program development tool for the simulation and testing of the embedded systems. It's a product of a Finnish company called Prosoft. ESIM is designed to be able to simulate any given embedded system, provided that the software is written in C or C++ language [23]. linked ESIM provides opportunities to monitor program execution. Furthermore it provides simulation functions for the interfaces (ports, buses, user interface), the operating system, registers, tasks, semaphores, etc.

whole

5.2. **Field buses.** The communication between the modules is done via field buses. The testing concentrated on the load testing of the message-processing unit. There were two main field buses in use, CAN (controller area network) and LON (local operating network).

The tested target was the communication unit, which operated under the main CPU. The object software was constructed in such a way that it uses a CAN (controller area network) bus for all data transfer between the main CPU and other electronic modules, including I/O-cards, key panel, and LCD-display. The same processing unit also handles LON bus messages. Communication interface, CAN and LON field buses are also the natural interface to the software. In our test CAN bus data transfer was simulated and tested, while software behaviour was monitored. The main objective was to study if lower priority LON message processing has some effect on the processing of higher priority CAN messages.

CAN is a serial bus which has been developed for the data transfer of advanced real-time and decentralized control system devices, in which for example the sensors and regulating units communicate directly with each other without the help of any intervening control unit. The CAN bus is a so-called multi-master, in other words several nodes can access the channel simultaneously. The CAN protocol is relatively simple which makes the programming of applications fairly easy. The high reliability of the data transfer has also been stated as one of its advantages. The CAN message contains 0 to 8 bytes of information. The data transfer rate can be chosen in the range 125kb-1Mb/s, data transfer distance is dependent on the chosen speed. One CAN bus can contain 2032 objects according to CAN 2.0A specifications [8].

LON is originally a field bus developed mainly for building automation [10]. In the LON bus the processor and memory reside in each control unit, meaning that the net does not need a centralized control. The most important application areas iclude real estate, industrial and process automation, vehicles etc small systems. The devices of different systems can be connected to the same LON bus. The data transfer channel is open and the devices of several different manufacturers can be connected to it.

5.3. **The GA used.** GA simulates the messages that the field bus message handler receives from the I/O-cards and display/interface cards when communicating through the CAN bus. When communicating via the LON bus, it simulates other similar units in the net. Furthermore, it

	Random	*GA*
Count	10,000	10,000
Mean [ms]	155.47	214.73
Std. deviation [ms]	65.52	101.89
Minimum [ms]	19.00	22.00
Maximum [ms]	444.00	576.00
Range [ms]	425.00	554.00
Median [ms]	157.00	194.00

TABLE 1. Descriptive statistics of response times of test cases created by the GA and the pure random method.

simulates the operation of the digital signal processing unit and writes information directly in the databases located in a common memory. A test case record contains among others the delay between two CAN messages, an optional LON load message, send times, data fields, and sender addresses. The fitness value is the time from the sending of the CAN message to the receiving of the CAN answer message.

The first generation in GA is created using the Windows random number generator. The genetic operators used to create new generations were single point and uniform crossovers together with mutation. We re-evaluate the individuals selected from the previous generation, because the system is rather non-deterministic. The most important parameters of the GA used in this study have been: single point and uniform crossover of equal rate, the total crossover rate being 70%, mutation rate 8%, population size 100 of which 30% of the best are selected for the next generation (elitism).

6. RESULTS

In this work both the genetic algorithm and the tested software were run under the ESIM simulator in the same workstation, which was a PC equipped with a Pentium 200 mmx processor and Windows 95 operating system. Observe that the processing times should not be directly compared to those of the target environment, in which the object software runs considerably faster.

In order to compare pure random and a GA based method, 10,000 test cases were generated and tested by both methods. Table I represents the corresponding descriptive statistics. ¿From it we can see that the average response time of test cases created by GA is much longer (38%) than that of the random method. This implies that GA is able to find and favor some test cases that lead to longer response times.

Figure 1 shows the distribution of response times for both methods. As we can see the histograms are quite similar, except that the GA method has an additional peak between 300 and 400 ms. Obviously GA has been able to identify some input parameter combinations resulting to longer response times. In order to find those parameters that cause the difference we calculated the correlations between all input variables and response time. Almost all of these correlations were very small, of magnitude 0.1 or less. The highest correlation (0.45) was found between the response time and in which state of the CAN message processing the LON message was sent.

To demonstrate how strong the effect of LON message sending is on the response time we draw figure 2, where test cases were divided into two groups "no LON messages" and "LON message sent during the test case". It is clear that the additional peak at the longer response time end was almost totally created from test cases, where the LON message was sent. So it is obvious that

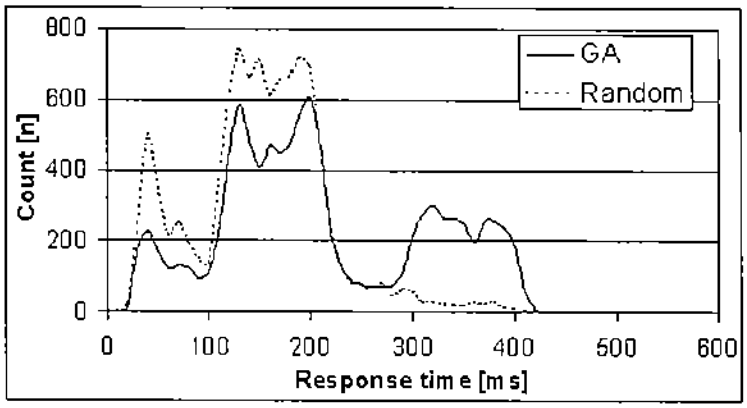


FIGURE 1. Response times distribution for the GA and random methods.

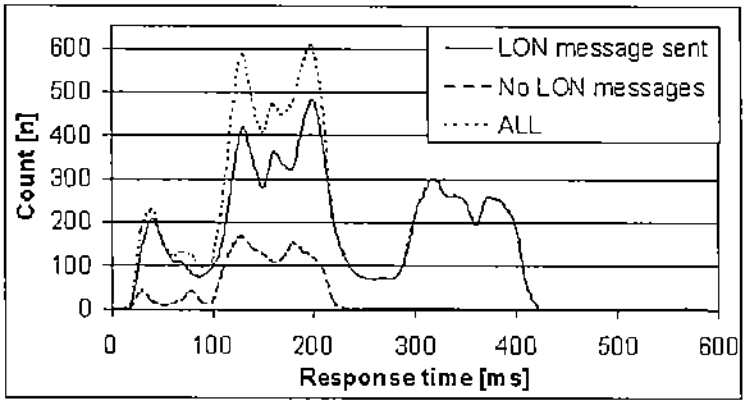


FIGURE 2. Response time dependensy on LON sending.

the GA based method favors the LON message sent during the test case more than the random method. We identify six different states of CAN message processing when a LON message can be sent (figure 3). It is obvious that the longest response times can be expected if the LON message is sent shortly before the CAN message. This is probably due to the fact, that in that case the program starts to process the received LON message, before the higher priority CAN message arrives causing the interrupt handler to start the processing of the CAN message. In figure 3 both the GA and random methods perform similarly in all six classes, while the overall response time is much higher with the GA method.

Furthermore we divided all test cases into the same six classes given in figure 3 and figure 4. From the latter it is obvious that GA learns the problematic test parameters and creates many more test cases for the most problematic class B than the random method. The time segments in the six classes are not equal, which explains that neither method creates evenly distributed test cases.

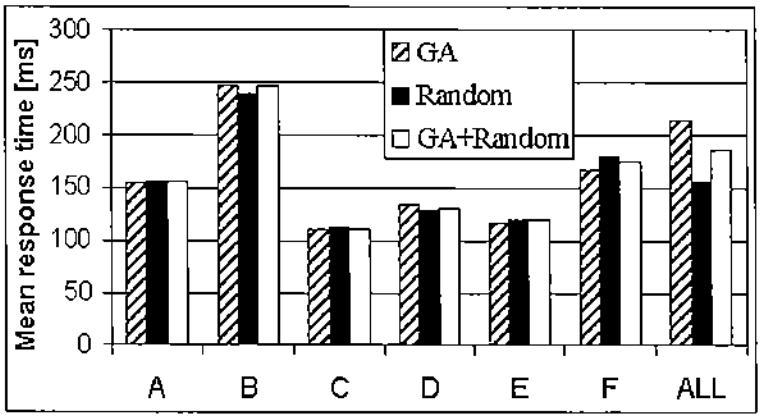


FIGURE 3. The six CAN message processing states, when LON load message can be sent. A=no LON message, B=LON message shortly before the CAN message, C=LON and CAN message simultaneously, D=LON message same time as program processes CAN message, E=LON message simultaneously as tested program starts to generate replay to the CAN message, F=LON message after E, but before replay CAN message comes back from the program.

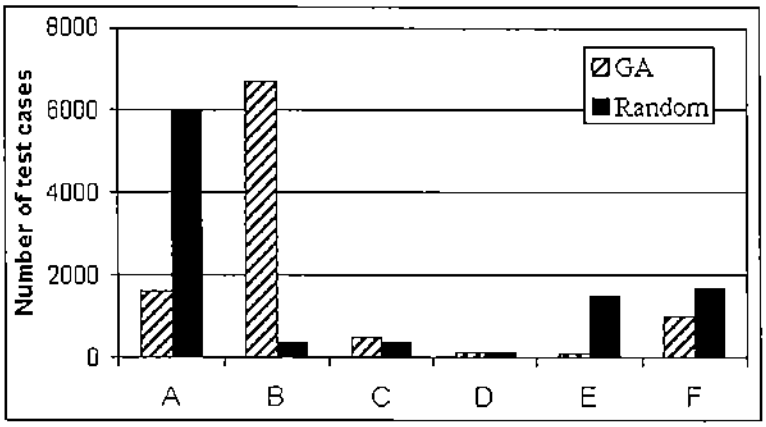


FIGURE 4. Distribution of test cases produced by the GA and random methods. For the notations used see figure 3.

6.1. **Comments and discussion.** It was discovered that at the beginning of the testing the first couple of test cases caused the longest response time. The tested software is a state-machine. Thus we will get rid of this problem if we succeed to move the state-machine between the tests to a more random state.

Because of the non-determinism every test case should be evaluated several times, the average of which could be used as the fitness value for each test case.

The genetic algorithm parameters effect on the test results was also studied briefly, but this optimization problem was not especially sensitive to the GA parameters. This is in good agreement with our and other studies on the sensitivity of GA based optimization [1, 4].

7. Conclusions

This study shows that the genetic algorithm is applicable in pure black box type software testing. It was significantly more efficient in finding the suspect input parameter sets than random testing. When analyzing the results it turned out that the simple statistical analysis combined with some graphics is most useful.[21]

We find the results of this study encouraging and recommend that GA based optimization can be seriously considered when developing software testing automation. However, more research on the possibilities of different kind of program testing with GA should be done in order to evaluate its full potential.

Acknowledgements. This work is a part of a larger project applying GAs in industrial optimization problems and has been supported by the Finnish Technology Development Center (Tekes) and ABB Corporate Research.

Further information on this work can be found in our anonymous ftp server (`ftp.uwasa.fi`) in directory `cs/report99-2` and also raports [5, 6]. The work was based on the thesis done by one of the authors (T. M.) [15] under the guidance of J. A. The project and especially the analysis of results is also reported in [21]. Lilian Grahn is acknowledged for her help with the proofreading of this paper.

Windows NT is a trademark of Microsoft Corp.

References

1. Jarmo T. Alander, *An indexed bibliography of genetic algorithms*, Practical Handbook of Genetic Algorithms: New Frontiers (Lance D. Chambers, ed.), New Frontiers, vol. III, CRC Press, Inc., Boca Raton, FL, 1999, pp. 503–572.
2. ———, *Indexed bibliography of genetic algorithms in computer science*, Report 94-1-CS, University of Vaasa, Department of Information Technology and Production Economics, 1999.
3. ———, *Indexed bibliography of genetic algorithms in electronics and VLSI design and testing*, Report 94-1-VLSI, University of Vaasa, Department of Information Technology and Production Economics, 1999, (`ftp.uwasa.fics/report94-1/gaVLSIbib.ps.Z`).
4. ———, *Population size, building blocks, fitness landscape and genetic algorithm search efficiency in combinatorial optimization: An empirical study*, Practical Handbook of Genetic Algorithms: New Frontiers (Lance D. Chambers, ed.), New Frontiers, vol. III, CRC Press, Inc., Boca Raton, FL, 1999, pp. 459–485.
5. Jarmo T. Alander, Timo Mantere, Ghodrat Moghadampour, and Jukka Matila, *Searching protection relay response time extremes using genetic algorithm-software quality by optimization*, Electric power systems research 46 (1998), 229–233.
6. Jarmo T. Alander, Timo Mantere, and Pekka Turunen, *Genetic algorithm based software testing*, Artificial Neural Nets and Genetic Algorithms, Proceedings of International Conference (ICANNGA97) (Norwich (UK)) (George D. Smith, ed.), Springer-Verlag, Wien, April 1997, pp. 325–328.
7. Antti Auer and Jukka Korhonen, *State testing of embedded software*, EuroSTAR '95, November 1995, Olympia confrence centre, London, England (1995).
8. Bosch, *CAN specification, version 2.0*, Robert Bosch GmbH, Posttach 50, D-7000 Stuttgart 1, 1991.
9. M. Chaufer and T. Mosser, *Testing embedded systems*, Touloise 89: Proceedings of the Second International Workshop on Software Engineering and Its Applications. Touloise, France, 4-8 December 1989. Nanterre, France. (1989).
10. Echelon, *LON talk protocol specification. version 3.0, number 19550*, Echelon Corporation, Palo Alto, CA, 1994.
11. John Holland, *Adaptation in natural and artificial systems*, University of Michigan Press. Reissued by MIT Press, 1992, Ann Arbor, MI, 1975.
12. Hannu Honka, *A simulation-based approach to testing embedded software, vtt publications 124*, VTT, Technical research centre of Finland, Espoo, Finland, 1992.
13. B. F. Jones, D. E. Eyres, and H.-H. Sthamer, *A strategy for using genetic algorithms to automate branch and fault-based testing*, The Computer Journal 41 (1998), no. 2, 98–107.

14. B. F. Jones, H.-H. Sthamer, X. Yang, and D. E. Eyres, *The automatic generation of software test data sets using adaptive search techniques*, Proceedings of the Software Quality Management 3 (Seville, Spain), vol. 2, Computational Mechanics Publications, Ltd., Southhampton, UK, 3.-5. April 1995, pp. 435–444.
15. Timo Mantere, *Automaattinen ohjelmien testaus geneettisten algoritmien avulla [Automatic program testing by optimizing with genetic algorithms]*, University of Vaasa, Vaasa, Finland, 1999.
16. Gary E. McGraw, Christoph C. Michael, and Michael A. Schatz, *Generating software test data by evolution*, Technical Report RSTR-018-97-01, RST Corporation, 1998.
17. G. J. Meyers, *Art of software testing*, John Wiley&Sons, New York, 1979.
18. C. C. Michael, G. E. McGraw, M. A. Schatz, and C. C. Walton, *Genetic algorithms for dynamic test data generation*, Proceedings of the 12th IEEE International Conference Automated Software Engineering (Incline Village, NV (USA)), vol. ?, IEEE Computer Society Press, Los Alamitos, CA, 1.-5. November 1997, pp. 307–308.
19. Christoph C. Michael and Gary E. McGraw, *Opportunism and diversity in automated software test data generation*, Technical Report RSTR-003-97-13, RST Corporation, 1997.
20. Christoph C. Michael, Gary E. McGraw, Michael A. Schatz, and Curtis C. Walton, *Genetic algorithms for dynamic test data generation*, Technical Report RSTR-003-97-11, RST Corporation, 1997.
21. Ghodrat Moghadampour, *Using genetic algorithms to testing a protection relay software - a statistical analysis*, University of Vaasa, Vaasa, Finland, 1999.
22. Min Pei, Erik D. Goodman, Zongyi Gao, and Kaixiang Zhong, *Automated software test data generation using a genetic algorithm*, Technical Report 6/2/94, Beijing University of Aeronautics and Astronautic, 1994.
23. Prosoft, *Esim - testing environment for embedded software, user's guide version 2.1 for Windows NT*, Prosoft Oy, Oulu, Finland, 1995.
24. R. J. Martin R. A. DeMillo, W. M. McCracken and J. F. Passafiume, *Software testing and evaluation*, Benjamin/Cummings Publishing, Menlo Park, California, 1997.
25. Marc Roper, Iain Maclean, Andrew Brooks, James Miller, and Murray Wood, *Genetic algorithms and the automatic generation of test data*, Research report RR/95/195 [EFoCS-19-95], University of Strathclyde, Department of Computer Science, 1995.
26. Alison Lachut Watkins, *The automatic-generation of test data using genetic algorithms*, Proceedings of the 4th Software Quality Conference (Dundee (UK)) (I. M. Marshall, W. B. Samson, and D. G. Edgar-Nevill, eds.), vol. 2, University of Abertay Dundee, Scotland, 4.-5. July 1995, pp. 300–309.
27. S. Xanthakis, C. Ellis, C. Skourlas, A. Le Gall, S. Katsikas, and K. Karapoulios, *Application of genetic algorithms to software testing [Application des algorithmes génétiques au test des logiciels]*, Proceedings of the 5th International Conference on Software Engineering (Toulouse, France), 7.-11. December 1992, pp. 625–636.

Department of Information Technology and Industrial Economics, University of Vaasa, PO Box 700, FIN-65101 Vaasa, Finland

E-mail address: firstname.lastname@uwasa.fi

Automatic Image Generation by Genetic Algorithms for Testing Halftoning Methods

Timo Mantere[*a] and Jarmo T. Alander[**b]

[a]Turku Centre for Computer Science TUCS, Lemminkäisenkatu 14 A,
FIN-20520 Turku, Finland

[b]Dept. of Information Technology and Production Economics, University of Vaasa,
P.O. Box 700, FIN-65101 Vaasa, Finland

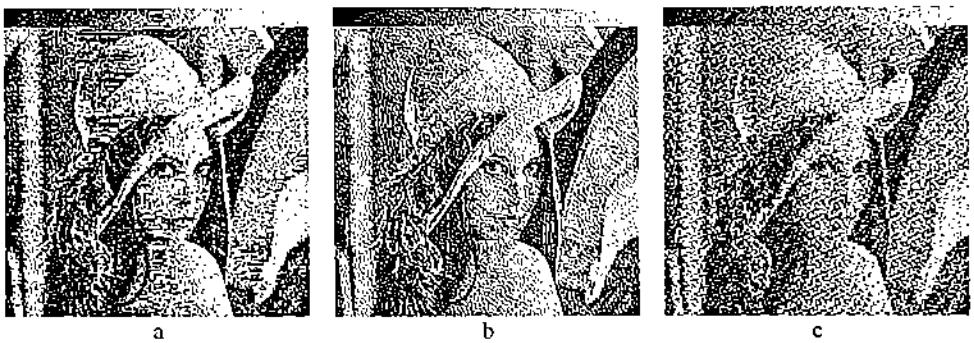
ABSTRACT

Automatic test image generation by genetic algorithms is introduced in this work. In general the proposed method has potential in functional software testing. This study was done by joining two different projects: the first one concentrates on software test data generation by genetic algorithms and the second one studied digital halftoning for an ink jet marking machine also by genetic algorithm optimization. The object software halftones images with different image filters. The goal was to reveal, if genetic algorithm is able to generate images that are difficult for the object software to halftone, in other words to find if some prominent characteristics of the original image disappear or ghost images appear due to the halftoning process. The preliminary results showed that genetic algorithm is able to find images that are considerably changed when halftoned, and thus reveal potential problems with the halftoning method, i.e. essentially tests for errors in the halftoning software.

Keywords: Dithering, genetic algorithms, digital halftoning, image filtering, test image generation, software testing, image comparison

1. INTRODUCTION

There does not seem to be much research in the field of test image evaluation. How to determine a good test image. What are the essential characteristics of a good test image? How to determine that a particular image is good for testing some specific image processing software? More often than not researchers rely on commonly used and very limited test image sets (fig. 1).



a b c

Figure 1: Lena image dithered by
(a) Floyd-Steinberg error diffusion (b) Jarvis-Judge.Ninke (c) Threshold matrix method

* email: tmantere@abo.fi; ** email: jal@uwasa.fi

Part of the SPIE Conference on Intelligent Robots and Computer Vision XIX:
Algorithms, Techniques, and Active Vision • Boston, Massachusetts • November 2000
SPIE Vol. 4197 • 0277-786X/00/$12.00

We encountered this problem, when we wanted to test the image-processing system we implemented for an ink jet marking machine[1, 2]. In our other study[3, 4, 5] we used genetic algorithms (GA) for software testing purposes. In this work we try to combine the knowledge of these two previous studies and use GA for generating test images for halftoning software.

The object software was originally developed with the Khoros[6] image processing system, but later translated into Java. The genetic algorithm (GA) was also written in Java. The advantage of Java is its easy to use image handling procedures, but the speed is not the best possible, however it outperformed most of the functions of Khoros that we needed.

1.1. Genetic algorithms

Genetic algorithms[7] are optimization methods that mimic evolution in nature[8]. They are simplified computational models of evolutionary biology. A GAs form a kind of electronic population, the members of which fight for survival, adapting as well as possible to the environment, which is actually an optimization problem. GAs use genetic operations, such as selection, crossover, and mutation in order to generate solutions that meet the given optimization constrains ever better and better. Surviving and crossbreeding possibilities depend on how well individuals fulfill the target function. The set of the best solutions is usually kept in an array called population. GAs do not require the optimized function to be continuous or derivable, or even be a mathematical formula, which is why they are gaining more and more popularity in practical technical optimization. Today GA methods form a broad spectrum of heuristic optimization methods.

Genetic algorithm were previously adapted to the dithering problem[9, 10]. For further references of GAs in image processing see bibliography[11] or book[12]. Image generation with GA is used at least in[13]. Image generation for algorithm validation is represented in[14]. GAs has previously been adapted to automatic software test data generation in several studies, see[3, 4, 5] and references therein.

1.2. Dithering

Digital halftoning[15], or dithering, is a method used to convert continuous tone images into images with a limited number of tones, usually only two: black and white. The main problem is to do the halftoning, so that the bi-level result image does not contain artifacts, such as moiré, lines or clusters, caused by dot placement[16]. The average density of the halftoned dot pattern should interpolate as precisely the original image pixel values as possible.

Dithering methods include static methods, where each pixel is compared to a threshold value that is obtained f.e. from a threshold matrix, generated randomly or is a static median value. There are also error diffusion methods, such as Floyd-Steinberg and Jarvis-Judge-Ninke used in this paper. In these methods the rounding error of the current pixel is spread into those neighboring pixels, the bi-level value of which is not yet determined.

2. THE PROPOSED METHOD

The proposed method is shown in figure 2. The GA runs as its own program and optimizes parameter vectors which are used by an image producer to create images, which are further sent to the object software, that halftones it and returns the resulting image. The pixelgrapper reads pixels from both the test image and its halftoned transformation image and transmits 8-bit pixel arrays of both images to the fitness function evaluator. The difference between these images is used as the fitness function. GA generates new parameter vectors by using crossover and mutation, favoring those parent chromosomes that previously had gotten a high fitness value.

Test images can be created by two different ways, either using an image bitmap as a genetic algorithm chromosome, which is time consuming and thus restricts one to use only relatively small images. Another way is to optimize parameters, such as place, size and color of elementary graphical objects, like lines, rectangles, circles and letters, together with the background color all encoded as a GA chromosome. Several fitness functions were tested; the average density at the corresponding image areas, edge location comparison, pixel by pixel comparison using low pass filtered images, and tone difference between consecutive pixels.

2.1. Comparing the images

Comparing a dithered image with the original one is obviously a challenging problem. One cannot simply use pixel by pixel comparison, since dithered images usually have only two tones. The minimum difference by that measure would be achieved if every gray tone were rounded to the nearest tone (black or white), which in practice usually results in poor images.

Better image comparison methods have been developed[15, 17]. One alternative is to sum the pixel values from the corresponding areas ($n \times n$ window) over the images to see if the average gray tones have been preserved. With this method one can compare the images directly, and this is one of the methods that we used.

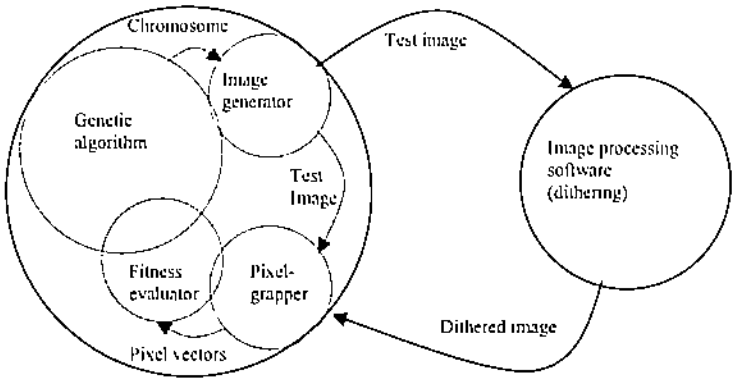


Figure 2: The proposed GA based method to test dithering software system

A set of methods called inverse halftoning[15] has been developed. From these we used the perhaps most common low pass filtering method. In this method images are first low pass filtered and the resulting images are then compared pixel by pixel. The problem with lowpass filtering is that the high frequencies will disappear and the images get a somewhat blurred overall appearance. However, this method is easy to implement and it enables pixel by pixel comparison.

Another comparison method we tried is a line by line comparison of the difference between the current and the previous pixel. This method is introduced in[18]. For example if we take a chess board and the mirror image of it, comparing them pixel by pixel we get the result that the two images are as far from each other as can be. If, on the contrary, the comparison is done by comparing the difference between consecutive pixels in each image, the sum of the difference will be very small. If we use the absolute value of difference between pixels, the two images are identical, i.e. the sum of the difference is zero. This method can be used to compare low pass filtered images, and may be used to some extent with also the original and halftoned images.

Yet another method to find features that have changed is to use edge detection, for example the Prewitt, Roberts or Sobel edge detection filters[19]. Edge detection can be applied to low pass filtered images and to some extent to original images as well.

3. EXPERIMENTAL RESULTS

3.1. Implementation: bitmap

Our first implementation was based on an integer (8-bits) coded GA, where the whole image bitmap was encoded as a GA chromosome. A gene was thus an integer value between 0 and 255 (grayscale values). The length of one GA chromosome was equal to the image size (i.e. 256 with 16×16 image). The uniform crossover[20] was used by choosing randomly a gene from one of the parents. A randomly selected gene was changed by a random grayscale value with the mutation probability 0.01. Other GA parameters: population size 50, elitism 40%, 10,000 individuals evaluated.

3.2. Results

This implementation was found to be very slow. So it was used only for small images, where the optimal solution may still be calculated, just to check that the GA optimization works properly for the problem. The images produced by this method were just shapeless noise.

The simplest halftone filter is to round each grayscale value to the closest bi-level image value, in practice with 8-bits, 0 or 255 (= w), black and white respectively. The maximum error is achieved if each pixel has the grayscale values $\lfloor w/2 \rfloor$ or $\lceil w/2 \rceil$ (rounds to 0 and w). The maximum error, calculated pixel by pixel, for image size 16×16 is then 32,512. GA reached the value 30,135.

Another method for which the maximum error is easily calculated is the threshold matrix filter. If we use a 16×16 matrix, which has exactly one of each threshold values $r_i \in [0, 255]$, the maximum error is reached if every pixel is either $\lfloor r_i \rfloor$ or $\lceil r_i \rceil$. The maximum error is now 48,133. GA reached the value 41,721.

The third filter we tested was the Floyd-Steinberg error diffusion method. In that test the maximum error is more difficult to calculate, since the error is spread to neighboring pixels, but we can assume that it is less than 32,512 because of the diffusion effect. GA reached the value 27,292. With the Jarvis-Judge-Ninke error diffusion method GA reached the value 29,304.

3.3. Implementation: feature vector

In our second implementation we used integer coded GA, where the chromosome consisted of image parameters that define features, such as background color, what objects, which color, what size and where to generate them. This encoding has some similarities with genetic programming principle[21]. The size of the generated image was 256×256, so that the values of most parameters; upper left and lower right x and y coordinates of objects, color gray tones, and ASCII value of letters, would fit 8-bits. However the font used, font style and size had to be scaled. The coding used in the images represented in the result section generated five lines, one rectangle, one circle and two letters into the test image. That coding required the chromosome length 50 bytes (1 for background color, five per each line, rectangle and ellipse, seven for each letter). Population size was 50, elitism 50%, only 550 evaluations (initial population + 20 generations) were made because of the heavy time demand on evaluating dithering result, only uniform crossovers, and mutation probability 2%.

3.4. Results

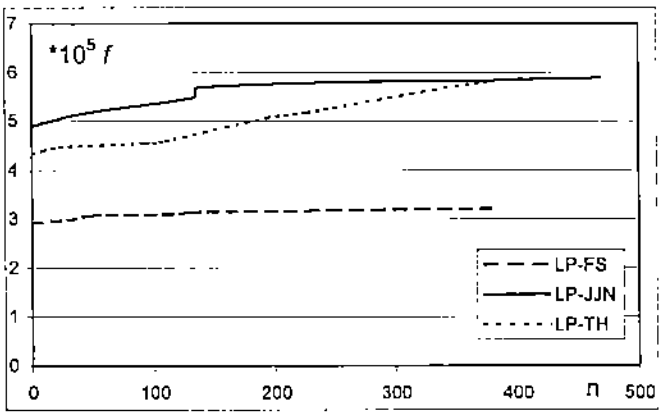


Figure 3: The development of GA fitness f vs. number of iterations with comparison method LP.

All test runs with each dithering method together with each comparison method was repeated at least five times in order to get some certainty that some test repeatable features could be found. The very first observation was that the separate test runs with the given dithering method and image comparison method tend to produce images with nearly the same background tone *b*. Table 1 collects the background colors of the best solution images found in each test run. Table 1 shows the result that with almost all dithering/comparison method pairs the best test image had nearly the same background tone *b* in all separate test runs. Note that with 8-bit representation f.e. $b = 8$ and $b = 246$ are complements of each other and lead to the same minority pixel density. It is obvious that *b* is a significant factor when evaluating halftoning quality. However the comparison method used also clearly had an effect on the result. So it is essential to find a proper comparison method.

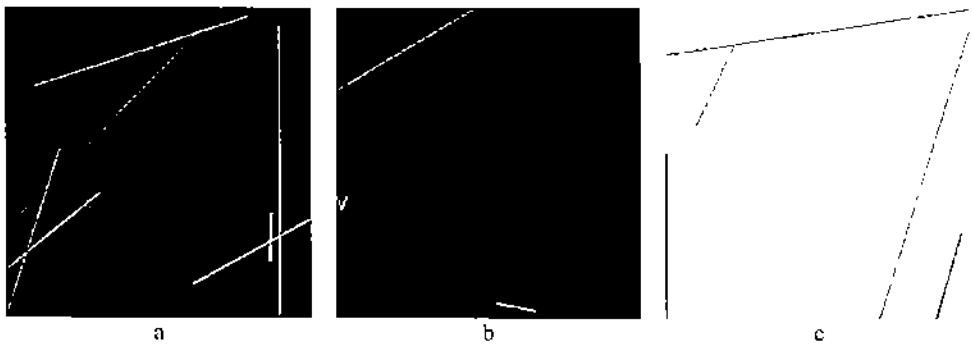
The notations used hereon f.e. `LP-FS` = comparison method `LP` combined with dithering method `FS`.

The optimization result with GA usually develops logarithmically, fig. 3 shows the development of optimization with first method. However with as short optimization runs as in this the development is quite small, only between 2 to 30 % from the best of initial population to the best overall.

Table 1: Background tone *b* of best test images in each run.

Comparison method		Dithering method	Test run				
			1.	2	3.	4.	5.
LP	Low pass filtered images pixel by pixel	FS Floyd-Steinberg error diffusion	8	246	8	13	12
		JJN Jarvis-Judge-Ninke error diffusion	246	246	243	241	246
		TH GA optimized threshold matrix	205	203	206	203	201
LS	Low pass filtered images by the difference between sequential pixels	FS	65	65	65	65	67
		JJN	180	189	182	178	82
		TH	158	155	155	157	159
SW	Sum of tones inside corresponding observation windows	FS	47	44	44	46	209
		JJN	47	212	154	46	157
		TH	153	156	152	153	153
DE	The difference between found edges in originals	FS	158	154	153	152	157
		JJN	139	140	139	137	112
		TH	123	118	113	118	119
DL	The difference between found edges in low pass filtered images	FS	242	241	8	246	245
		JJN	233	232	27	236	234
		TH	201	204	201	207	206

3.4.1. Pixel by pixel comparison after low pass filtering (Method `LP`)



a b c

Figure 4: GA generated test images for Floyd-Steinberg error diffusion filter.

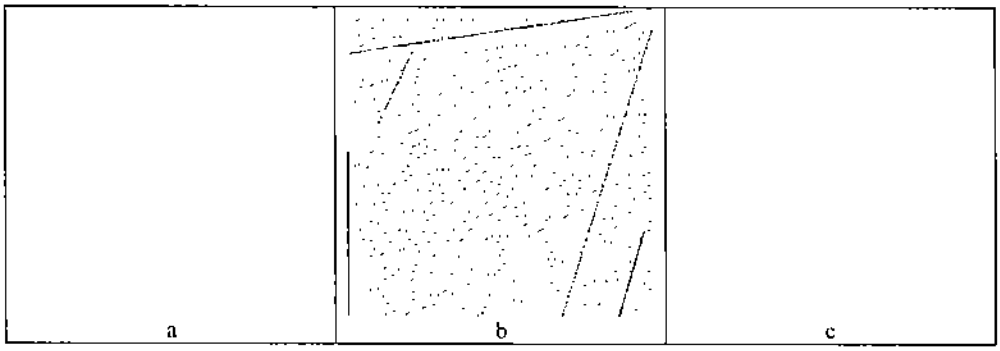


Figure 5: Test image generated by GA for Floyd-Steinberg error diffusion method.
(a) Low pass filtered image 4c. (b) Dithered image 4c. (c) Low pass filtered image b.

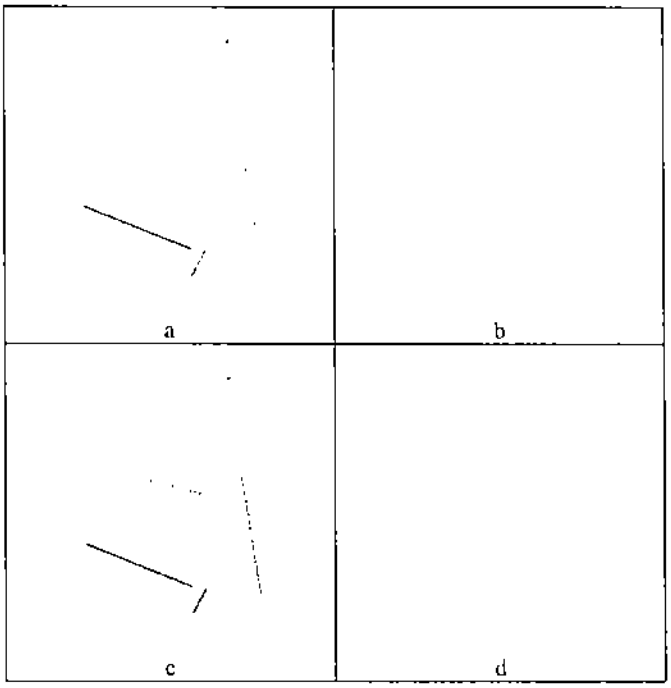


Figure 6: Test image for Jarvis-Judge-Ninke error diffusion filter found by GA.
(a) Optimized gray image, $b = 246$. (b) Low pass filtered image a.
(c) Dithered image a. (d) Low pass filtered image c.

LP-FS. Test runs resulted in an image, where $b \in [8, 13]$ or $b = 246$. It seems that these gray tones gives the kind of minority pixel placement that results in a highly foggy-looking image after Butterworth low pass filtering (see fig. 5c). The minority pixels in the dithered images are also visibly disturbing to the human eye (fig. 5b), since they seem to create new features that are not existent in the original image. The other GA generated features with these images seem surprisingly consistent, in every test run GA created only lines, but not rectangles, ovals and only rarely letters. Even the line placing between separate test runs may look quite alike for the human observer (see fig. 4). Figure 5c shows the grainy result of a

low pass filtered halftone image. This kind of grainy appearance proves to be furthest from the original, when comparing low pass filtered images pixel-wise.

LP-JJN. The results seem quite similar to those at LP-FS. Test runs resulted in an image for which b was $\in[241, 246]$ (the best found solution of each run). With this halftone filter GA also did not generate any ellipses or rectangles, only lines and rarely letters. This might be because all possible objects may be halftoned quite properly, but losing the whole background color creates a great difference.

With this dithering method the difference between the images seems to be composed of the background color totally vanishing (filter does not repeat it, see figure 6c). Also some light lines disappear (see figs. 6a and c) either totally or partly. In some test runs the surviving lines also seemed strengthened, this might be because when the background is not repeated the lines get more dots than they should and they seem darker and thinner in the halftoned image and its low pass filtered version.

As a conclusion the best test image for this method seems to be an image, where b is so small, that it just and just vanishes when halftoned (no minority dots present), and where some lines either disappear or strengthen.

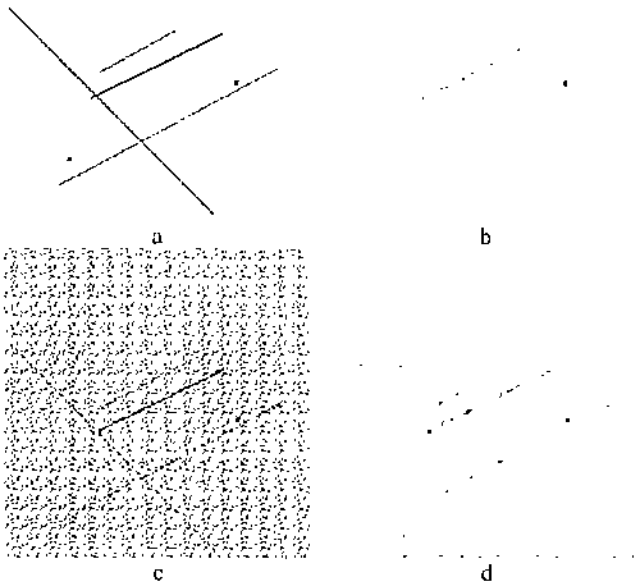


Figure 7: GA generated test image for a GA optimized threshold matrix method.
(a) Optimized image, $b = 205$. (b) Low pass filtered image a.
(c) Dithered image a. (d) Low pass filtered image c.

LP-TH. For this method images having the background gray tone $b \in[201, 206]$ were found to be hard. Other repeating hard features was not recognized. Most likely the threshold matrix generates the most uneven spread of minority pixels with that background color. When we change the matrix, also b changed, remaining quite consistent when using the given matrix, however.

As a conclusion it seems that GA is able to also reveal weak points of the threshold matrix methods. Remember that these matrices were originally optimized with $GA^{1,2}$. Optimization seems to favor round shapes and the results looked fine when

dithering the famous Lena image (Fig. 1e) or a grayscale[2]. When dithering GA generated test images, the uneven spread of threshold values gets painfully visible, however. With a uniform background tone the periodicity of the threshold matrix method is clearly visible. These threshold matrices also have difficulties repeating lines and small objects (see fig. 7e). The main difference between images seems to be caused by a grained background (compare fig. 7b and 7d).

3.4.2. Comparing differences between sequential pixels (Method LS).

LS-FS. This leads to images, for which $b \in [65, 67]$. Other features in common were not recognized, except that quite a few objects were present. The best test image seems to be the one, where all the objects disappear from the image leaving only the background tone $b = 65$ resulting a regular, web-like dot placement (look fig. 8d). By low pass filtering this halftone image we got an image with three tones $b \in \{58, 63, 69\}$. While the original image does not have any difference between sequential pixels, the resulting image always have a small difference. Summing these small differences results in the longest distance between images. Note in fig. 8d that the objects break the perfect web-like structure caused by $b = 65$ and develop extra ghost lines. These lines can also be detected by edge detection (see figure 8f).

LS-JJN results in images, which do not seem to have any features in common between separate test runs, with the exception of the background tone, which settles to $b = 82$ or $b \in [178, 189]$. The low pass filtering of the halftone image produces an image with a misty background, where gray tones vary in interval $b \pm 6$.

For LS-TH we got test images that did not seem to have any recognizable features in common. The GA optimized threshold matrix was the same as with comparison method LP.

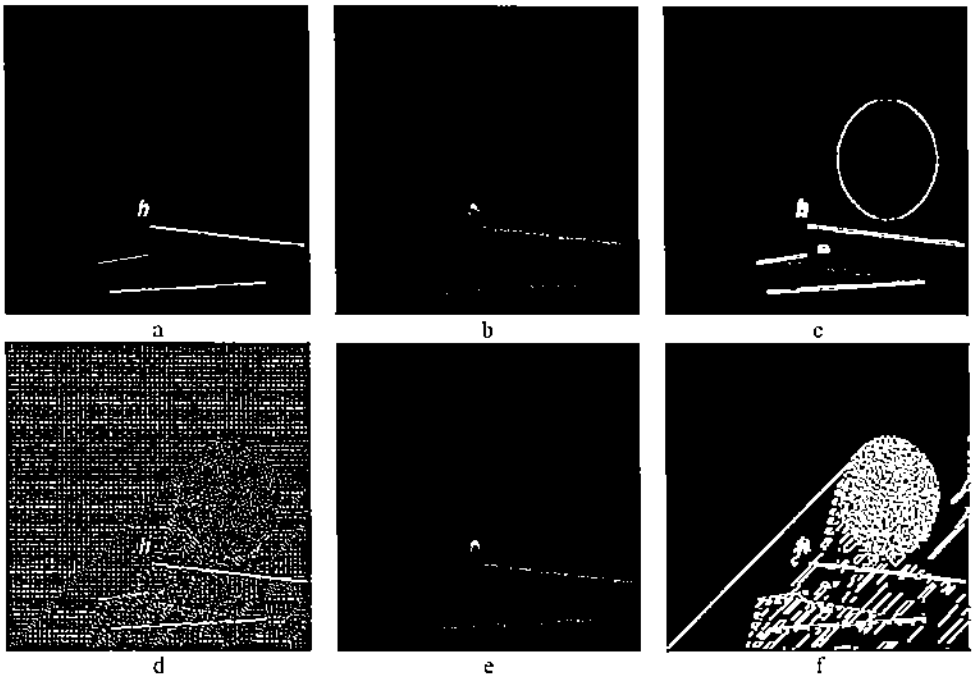


Figure 8 (a) GA generated test image for LS-FS. (b) Low pass filtered a. (c) Image a after edge detection. (d) Floyd-Steinberg dithered a. (e) Low pass filtered d. (f) Image of d after edge detection.

3.4.3. Comparing sum of tones within *n*×*n* windows (Method SW)

The comparison of images was done by calculating the sum of pixel tones inside the corresponding *n*×*n* windows. The window pairs were moved over the images with step *n*/2, and in the next round window size has increased by *n*/2 (see Code extract 1). The total sum of all observation window differences is the final difference figure.

SW-FS. This time the background color $b \in [44, 47]$ or $b = 209$. All the test runs with method A and B resulted in images with quite a few objects, other than lines. For SW-FS a large rectangle was placed in each five images, 3 out of 4 times it had tone 209 (see fig. 9ab). In one case the background/rectangle tones were exactly switched (fig. 9c). Floyd-Steinberg dithering may result in most uneven dot placement in the border of those two tones, which would explain this behavior.

```
diff=0; //variable for difference sum
for(n=2;n<15;n+=n/2) //n = size of observation window
       for(i=0;i<(i.height-n);i+=n/2) //go through the image
              for(j=0;j<(i.width-n);j+=n/2)
                     { // ^ Moves window half of it's width and height
                     xx=0;yy=0; //tone sum variables
                     for(ii=0;ii<n;ii++) //move inside n*n window
                            for(jj=0;jj<n;jj++)
                                   {
                                   xx+=pixoriginal[(i+ii)*i.width+j+jj];
                                   yy+=pixdithered[(i+ii)*i.width+j+jj];
                                   }
                     diff+=Math.abs(xx-yy); //difference sum
                     }
```

Code extract 1: Java code that calculates tone distribution difference between images using *n*×*n* windows.

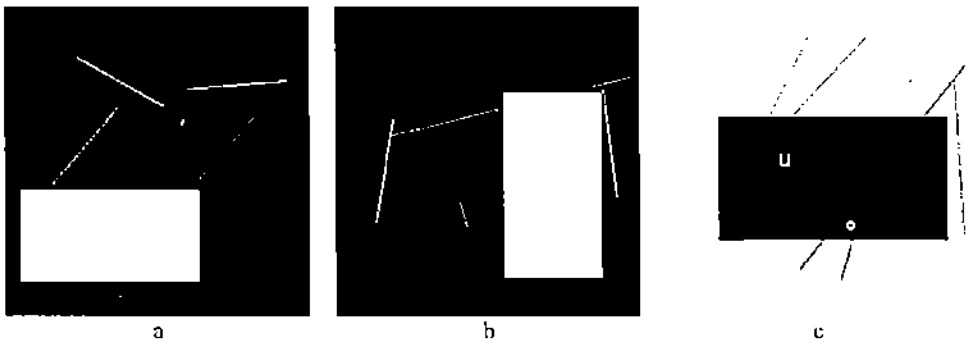


Figure 9: GA generated test images optimized using SW-FS.

For SW-JJN, we did not find any uniformity between test runs, there were larger and smaller ellipses and rectangles present, but they did not seem to follow any pattern, $b \in [46, 212]$. This uniformity might be due to the fact that there are not many prominent features in the SW-JJN pair that GA could adapt to. This might be further because the dithering method has been developed to spread dots as evenly as possible.

With SW-TH test images generated with all runs were quite similar, $b \in [152, 156]$, no other objects than lines and some small letters. This background range is probably the most unevenly repeated with that particular GA optimized threshold matrix when comparing the tones within observation windows.

4.3.4. Comparing images by using edge detection (Method DE)

In this part we generated test images by the difference between found edges as the fitness function. The Sobel edge detector was used to filter edges from the original and dithered images. The Prewitt and Roberts edge detectors were also implemented and tested, but the results are not reported here.

DE-FS generated images with $b \in [152, 158]$, there were not many objects present, only lines and some small letters. With these tones edge detector do not find any real edges, but only noise, which looks like that inside the ball in fig. 8f.

DE-JJN generates images with $b = 112$ or $b \in [137, 140]$, with especially few small objects, short lines, in them. The edge detection from halftone image is also noise.

DE-TH generate images with $b \in [113, 123]$, and also especially few small objects like DE-JJN. The background color seemed to cause several false edges with that particular threshold matrix.

These observations suggest that the Sobel edge detector might not be good as a fitness function when immediately applied to the original and halftoned image. Edge detectors, that are particularly developed for halftoned images might be more suitable.

4.3.5. Edge detection with low pass filtered images (Method DL)

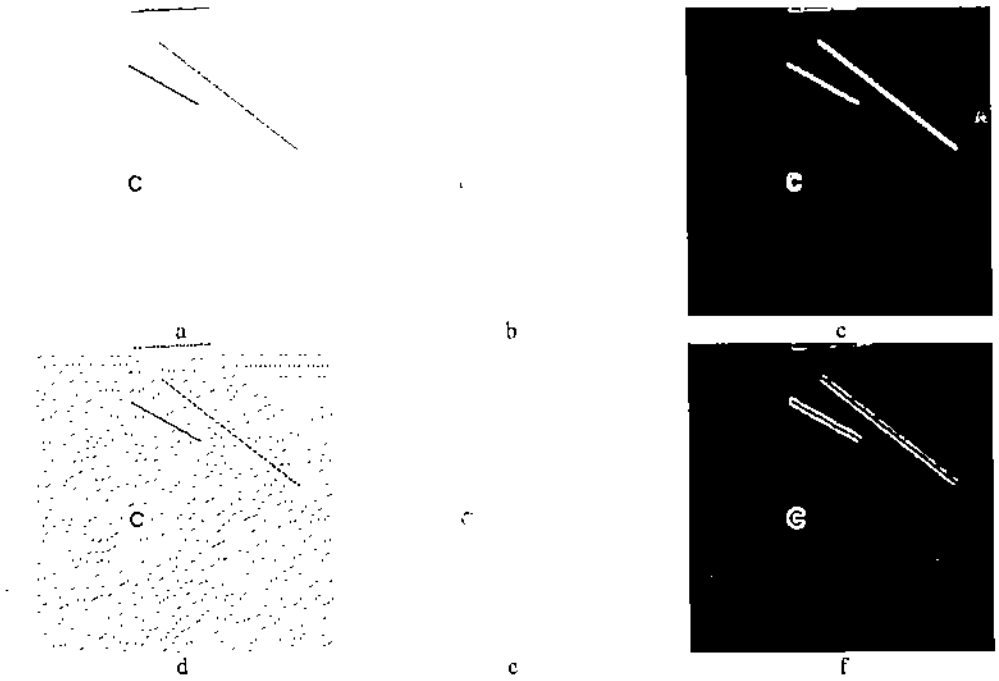


Figure 10: Edge detector as fitness function.
(a) Test image generated by GA. (b) Low pass filtered image a. (c) Edge detection applied to b.
(d) Floyd-Steinberg dithered image a. (e) Low pass filtered image d. (f) Edge detector applied to e.

In this chapter we apply the Sobel edge detection to low pass filtered images. These test runs resulted in fairly similar images when comparing low pass filtered images directly pixel-wise, with the exception of Jarvis-Judge-Ninke dithering.

DL-FS results in images for which $b = 8$ or $b \in [241,246]$, it is similar to LP-FS, but this time there were also some rectangles, ellipses and large letters present. Low pass filtering the dithered image results again in a foggy background that gets even more prominent with edge detection (see fig. 10).

DL-JJN results images for which $b = 27$ or $b \in [232, 236]$, some containing rectangles and ellipses. These background values b produce halftone images the background of which consists of rare minority pixels. Low pass filtering results in the same kind of fogginess as with DL-FS. In addition some light lines are not repeated.

With DL-TH the results were quite similar as in LP-TH. The background tone is $b \in [201, 207]$, no other objects than lines were present. No other recognizable similarities between separate test runs were detected.

4. CONCLUSIONS AND DISCUSSION

The results confirm that GA is capable of generating test images for testing different halftoning methods. Either some features of the original image disappear or some artifacts appear. The changes were perceived either by comparing the original and the dithered image, comparing low pass filtered versions of the images or applying edge detection after low pass filtering. The preliminary results of this work showed that the background tone is by far the most significant factor when testing the dithering methods.

The background color of the generated test images clearly depends on the dithering method used. This leads to the conclusion that problematic images for a given dithering methods have some features in common that GA is able to adapt. Also the fitness function has some influence. This leads to the conclusion that we must certainly do more research on which GA fitness function is most proper in this context. Best fitness function might be some kind of hybrid of the different comparison methods. Ideal fitness function would model human vision as precisely as possible.

Software testing is an important part of software development. Testing is time demanding, and even the partial automation can produce savings. It is not recommended that the software developer does the testing. Automatic test tools may cover some faults that the tester is blind to. In this work the object software was not that complicated, it mainly consisted of well-known image operations, and yet feeding a large number of images did cover faults that only manual testing with a couple of common images did not uncover when we implemented it.

4.1. Future

Image comparison could be enhanced by applying some feature extraction method, like MRDF[22]. Also the possibilities of applying fuzzy logic to image comparison is under research. Statistical analysis of the generated image parameters should be done in order to fully determine possible correlating parameters. So far the observations have been made by observing the images and analyzing only a couple of variables. After a satisfying fitness function has been found, the obvious application of the above testing method is automatic dithering method design. One GA generates halftone filters while the other GA tries to create the hardest test image for each filter. The best filter being the one where the hardest test image is closest to the original after dithering. In general this kind of approach could be used in the design and testing of demanding software.

ACKNOWLEDGEMENTS

Lilian Grahn is acknowledged for her help with the proofreading of this paper.

REFERENCES

1. Jarmo T. Alander, Timo Mantere, and Tero Pyylampi, Threshold matrix generation for digital halftoning by genetic algorithm optimization. In David P. Casasent, editor, *Intelligent Systems and Advanced Manufacturing: Intelligent Robots and Computer Vision XVII: Algorithms, Techniques, and Active Vision, volume SPIE-3522*, Boston, MA, 1.-6. November 1998. SPIE, 1998.
2. Jarmo T. Alander, Timo Mantere, and Tero Pyylampi, Digital halftoning optimization via genetic algorithms for ink jet machine. In B. H. V. Topping, editor, *Developments in Computational Mechanics with High Performance Computing*, CIVIL-COMP Press. Edinburg, UK, p. 211-216, 1999.
3. Jarmo T. Alander, Timo Mantere, Ghodrat Moghadampour and Jukka Matila, Searching protection relay response time extremes using genetic algorithm – software quality by optimization. *Electric Power Systems Research 46*, pp. 229-233, 1998.
4. Jarmo T. Alander, and Timo Mantere, Automatic software testing by genetic algorithm optimization, a case study. In Conor Ryan and Jim Buckley (eds.) *SCASE'99 - Soft Computing Applied to Software Engineering*, 11.-14.4.1999, Limerick, Ireland, pp. 1-10, 1999.
5. Jarmo T. Alander, and Timo Mantere, Genetic algorithms in software testing - experiments with temporal target functions. In *MENDEL2000 6th International Conference on Soft Computing*, June 7-9, 2000, Brno, Cheek Republic, Brno University of Technology, PC-DIR, Brno, pp. 9-14, 2000.
6. Khoral Research Inc., *Digital Image Processing (DIP) with Khoros Pro 2001*, Version 3.0.1.2, March 2000. Khoral Research Inc., Albuquerque, New Mexico. Available via www: < http://www.khoral.com/contrib/contrib/dip2001/index.html >
7. John Holland, *Adaptation in Natural and Artificial Systems*, University of Michigan Press. Ann Arbor, MI, Reissued by The MIT Press, 1992.
8. Charles Darwin, *The Origin of Species: By Means of Natural Selection or The Preservation of Favoured Races in the Struggle for Life*. Oxford University Press, London, A reprint of the 6th edition, 1968.
9. N. Kobayashi, and H. Saito, Halftoning Technique Using Genetic Algorithms. In *Systems and Computers in Japan 27*, pp. 89-97, Sept. 1996.
10. J. Newbern, and V. M. Bove, Jr., Generation of blue noise arrays using genetic algorithm. In *Human Vision and Electronic Imaging II*, B.E. Rogowitz and T.N. Pappas, eds., vol SPIE-3016, pp. 401-450. SPIE, Bellingham, San Jose, CA, 10.-13. Feb. 1997.
11. Jarmo T. Alander, *An Indexed Bibliography of Genetic Algorithms in Optics and Image Processing*. Department of Information Technology and Production Economics, University of Vaasa, Report Series No. 94-1-OPTICS, 2000, Available via ftp:< ftp://garbo.uwasa.fi/cs/report94-1/ gaOPTICSbib.ps.Z >
12. Sankar K. Pal, Arhish Ghosh, and Malay K. Kundu, *Soft Computing for Image Processing*. Physica-Verlag, Heidelberg, New York, 1999.
13. K.Sims, Artificial Evolution for Computer Graphics. *Computer Graphics (Siggraph '91 Proceedings)*, July 1991, pp.319-328.
14. Mario Miyojim, and Heng-Da Cheng, Synthesized images for pattern recognition. In *Pattern Recognition*, Vol. 28, No. 4, pp. 595-610, 1995.
15. Henry R. Kang, *Digital Color Halftoning*. SPIE Optical Engineering Press, Bellingham, Washington, & IEEE Press, New York, 1999.
16. Peter G. J. Barten, *Contrast Sensitivity of the Human Eye and Its Effects on Image Quality*. SPIE Optical Engineering Press, Bellingham, Washington, USA, 1999.
17. Fredrik Nilsson, Objective quality measures for halftoned images. *Optics, Image, Science and Vision*, Volume 16, Number 9, September 1999, pp. 2151-2162.
18. Patrick Eklund, and Fredrik Georgsson, Unraveling the Thrill of Metric Image Spaces. In G. Bertrand, M. Couprie, and L. Perroton (editors), *Discrete Geometry for Computer Imagery*, LNCS 1586, Springer, 1999
19. R.C. Gonzales and R. E. Woods, *Digital Image Processing*. Addison Wesley, 1993.
20. Gilbert Syswerda, Uniform crossover in genetic algorithms. In J. David Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*. George Mason University, June 4-7, 1989. Morgan Kaufmann Publishers, Inc. San Mateo, California.
21. John R. Koza, *Genetic Programming*. Cambridge, MA; The MIT Press, 1992.
22. A. Talukder, and D. Casasent, General methodology for simultaneous representation and discrimination of multiple object classes. *Optical Engineering, Special Issue on Advanced Recognition Techniques 37* (3), pp. 904-913, March 1998.

Testing a Structural Light Vision Software by Genetic Algorithms – Estimating the Worst Case Behavior of Volume Measurement

Timo Mantere[*a,b] and Jarmo T. Alander[**b]

[a]Turku Centre for Computer Science TUCS, Lemminkäisenkatu 14 A, FIN-20520 Turku, Finland

[b]Dept. of Information Technology and Production Economics, University of Vaasa, P. O. Box 700, FIN-65101 Vaasa, Finland

ABSTRACT

In this study we use genetic algorithms to generate test surfaces for a proposed structured light 3-D vision system in order to estimate the worst case behavior of error tolerances. The object software evaluates surface profiles for measuring volumes of small objects attached on surfaces that are highly constrained while somewhat arbitrarily shaped. The test system tries to find, by using genetic algorithm search, the shape that results the highest relative error of volume. The parameters of the object system to be optimized include laser angle, image size, object step size, and the number of scan directions. The preliminary results got seem to indicate that a genetic algorithm based approach is a beneficial aid in optical system design.

Keywords: Genetic algorithms, image processing, 3-D imaging, 3-D metrology, machine vision, simulation, structured light vision.

1. INTRODUCTION

Automatic test surface generation by genetic algorithms[1] (GA) is introduced in this work. In general, the proposed method has potential in functional software testing. The goal was to find out, if genetic algorithm is able to generate surfaces that are difficult for the object software to measure, in other words to find out the accuracy or error tolerances of the object software. Software testing is an important part of software development. It is time consuming, and even a partial automation can produce considerable savings. The benefits of automatic test tools also include that they can be much more objective, than a human testers. In this work the object software consisted of some ray tracing[2] and related trigonometry procedures.

The measuring of small objects fixed on a surface is a difficult problem. Knowing the surface profiles we can interpolate its volume. There are several methods that are developed for imaging based surface metrology, like stereo photography, structured light vision, and shading, reflection, and focus based methods. In this paper, we concentrate on structured light vision[3, 4]. This work was done in one of our research projects, developing a high speed 3-D measurement system for small objects. First we want to confirm that the proposed system meets the accuracy demands. Therefore, we decided to first simulate the system.

Genetic algorithms are optimization methods that are known to be solving quite well many difficult optimization problems; therefore, we decided to use the GA to generate simulated 3-D test surfaces to be measured. We simulated the imaging process by evaluating how light planes would lie on the simulated surfaces (ray tracing) and further generating simulated images from these height curves. The simulated surface/image may not exactly correspond to the real machine vision system, the former being more ideal and exact than the latter.

* email: tmantere@abo.fi; ** jal@uwasa.fi

Intelligent Robots and Computer Vision XX: Algorithms, Techniques, and Active Vision, David P. Casasent, Ernest L. Hall, Editors, Proceedings of SPIE Vol. 4572 (2001)

1.1. Genetic algorithms

Genetic algorithms are optimization methods that mimic evolution in nature[5]. They are simplified computational models of evolutionary biology. The GA forms a kind of electronic population, the members of which fight for survival, adapting as well as possible to the environment, which is actually an optimization problem. The GAs use genetic operations, such as selection, crossover, and mutation in order to generate solutions that meet the given optimization constrains better and better. Survival and crossbreeding probabilities depend on how well individuals fulfill the target function. The set of the best solutions is usually kept in an array called population. The GAs do not require the optimized function to be continuous or derivable, or even be expressed as a mathematical formula, and that is why they gain more and more popularity in practical technical optimization. Today the GA methods form a broad spectrum of heuristic optimization methods.

1.2. Structured light

Structured light vision[3] is a method where the object, the surface height of which is measured, is lighted in such a way that the sharp light and shadow lines on the surface can be imaged. The sharp light plane on the surface follows the surface profile and forms height curves that can be imaged. In order to measure the whole surface we must scan the whole surface by moving either the light plane or the object using small steps and recording a new image after each step. The height information from these images must be evaluated for the further estimation of volume. For this we need to know several parameters including the height curves, the zero level, camera distance from the zero level, scan directions α_s, laser angle α_L, and the focal length, the actual size that the pixel corresponds to the surface (fig. 1). Usually the measurement system consists of a CCD camera and computer.

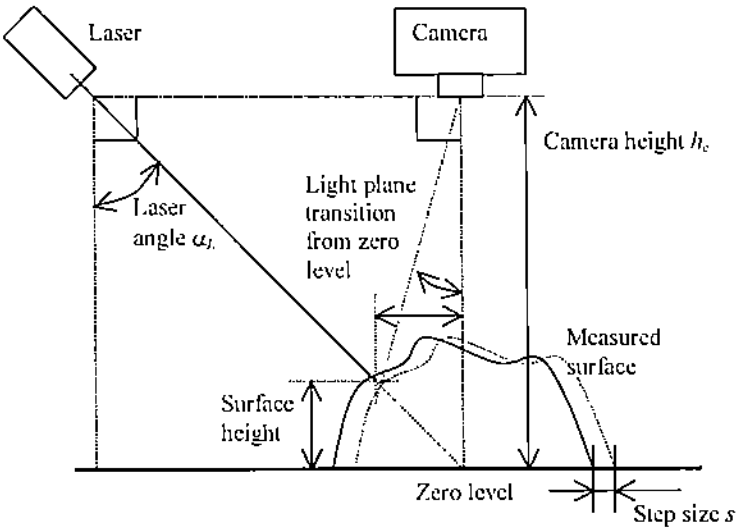


Fig. 1. The structure of the structured light vision measurement system.

The problem with this kind of measurement is that the camera distorts the image towards borders, causing the surface profile and pixel sizes vary spatially. Another problem is that the light plane is not ideally thin, and it may cover several pixels with different tones, when the exact borderline of the light curve is difficult or impossible to define. If the light plane is less than one pixel thick, the max error is equal to the pixel size; the plane is within a pixel. The location cannot be evaluated more accurately.

1.3. Related work

Genetic algorithm has previously been adapted to the surface simulation problem in ref. 6. There are also several studies of shape optimization[7-10] using genetic algorithms. For shape representation using Bezier curves[11] and GAs, see refs. 10,

and 12-13, and references therein. The use of GAs for shape modeling from images is represented in ref. 14, and examples of GAs in optical system design[15, 16]. In addition, the GAs has been largely applied to image processing; see bibliography 17. GAs has previously been applied to automatic data generation for software testing in several studies, see refs. 18-20, and references therein.

2. THE PROPOSED METHOD

The proposed method consists of a GA that optimizes parameter vectors, which are used by a surface generator procedure `SurfCreate` to create 3-D surfaces, from which they are further transformed into surface height curves by procedure `EvalCurve`. The simulated surface height images are then sent to the structured light vision software SLV that evaluates the height information and generates the 3-D model of the surface (fig. 2). The subprogram `Fitness` evaluates the fitness function. It gets as its input the original S and the reconstructed surfaces R, from which it evaluates the difference, i.e. fitness, $f = R - S$. GA generates new parameter vectors by using crossover and mutation, favoring those parent chromosomes that previously have gotten high fitness values.

GA parameters used in these tests were: population size 50, elitism 50%, crossover rate 50%, and mutation probability 3%. New individuals were generated by applying both one point and uniform crossovers between chromosomes with 50/50 ratio; in addition, arithmetic crossover between genes was applied at the rate of 10%. Test runs consisted generations in the range [100, 1000].

Test surfaces can be created in several different ways (see ref. 21). The most natural way may be using surface equation, but when we have many constraints, this approach is less tempting. Other possibilities include splines and Bezier surface representation. Perhaps the simplest alternative would be using a height matrix, this unfortunately enables only very small surfaces in practice. Yet another method is to generate two arrays, horizontal and vertical, the height matrix being product of them, which enables a much larger surface matrix. The obvious drawback is that the line and column values are not independent in this case.

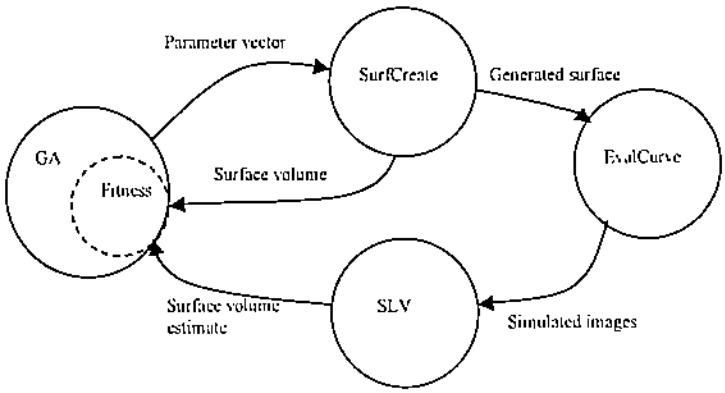


Fig. 2: The structure of the proposed test system.

We run some preliminary tests with the height matrix, but due to slow processing found it to be less practical for the GA optimization. The representation eventually resorted here consisted of two arrays, horizontal and vertical, that are actually control points of Bezier curves, the product of which is the surface. This approach enables continuous surfaces.

2.1 Preliminary tests

In order to see roughly how accurate the measurement system could be, we made preliminary tests with three test surfaces: a cube, hemisphere and pyramid (slope 45°) (fig. 3a-c). The cube was actually half cube, because we wanted the width, length, and height measures to be the same as with the hemisphere and pyramid. These three shapes were selected mainly because of their simplicity. The drawback of this test set is that it consists of only symmetric and convex shapes.

Figure 3d illustrates how light planes are incident on the hemisphere surface (fig. 3b). The corresponding reconstructed hemispherical surface is represented in figure 3e. The measurement error profile is shown in figure 3f. We can see that the measurement error is concentrated in the back area of the surface for the one scan model. This area behind the image is not seen, when scanned from one direction, because its angle is higher than the laser angle, and therefore this invisible mass appears in the error figure. This problem is at least partly avoided when using more than one scan direction.

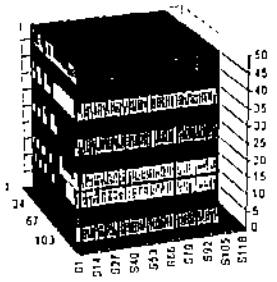


Fig. 3: a) A test cube.

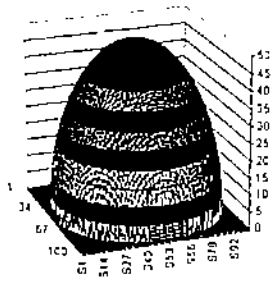


Fig. 3: b) A test hemisphere.

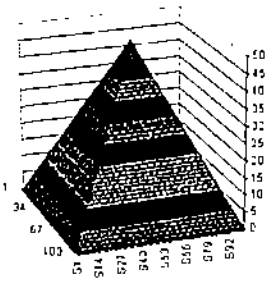


Fig. 3: c) A test pyramid.

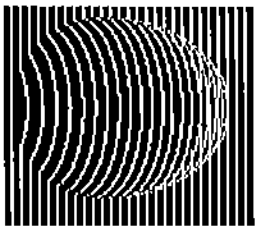


Fig. 3: d) Projection of light planes on hemisphere 3b.

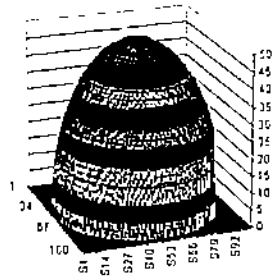


Fig. 3: e) Reconstructed hemisphere from 3d.

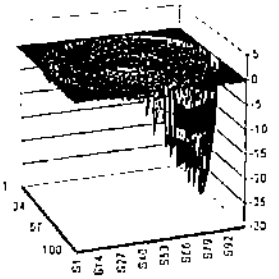


Fig. 3: f) The estimation error, 3e-3b.

Figure 4 shows some selected examples of the preliminary measurements, which were used to determine the parameters for the GA based simulations. Figures 4a-c show how the measurement accuracy behaves with different objects and image sizes 10×10, 50×50 and 100×100 pixels, without any estimation of the hidden surfaces. Figure 4d shows accuracy, when estimating the hidden area to continue with slope α_L, but as it can be seen, it does not perform too well, and we abandoned it. When looking figures 4a-c it seems that the best accuracy is achieved, when $\alpha_L \approx 63°$. Figure 4e shows accuracy as a function of image size in pixels $n \times n$, when $\alpha_L = 63°$. From figures 4a-c we can see that image size 10×10 pixels is excessively inaccurate. 50×50 is much better and 100×100 still better. From figure 4e, we can see that the measurement accuracy is better, when using more pixels. However, the measurement time increases as a square of the number of pixels (fig. 4f). Pixel size 100×100 was selected for GA simulations, since it gave reasonable accuracy and the simulation runs were still reasonably fast: 5000 surfaces with two scans were evaluated in 1h 20min by a 667 MHz Pentium™ computer.

Figure 4g shows how accuracy depends on the light plane step size s. It seems that the shortest possible s corresponds to the pixel size projected on the measured object. However, figure 4g implies that there may be some problems in the software, because the measurement should be more accurate even with little longer step sizes, when we are using these symmetric test surfaces. Figure 4h shows the measurement accuracy, when scanning over the surface from two directions with α_L=90°. When compared to figure 4e, the two directional scan is more accurate, especially when using low laser angle. Increasing the scan directions, however, did not seem to increase the accuracy much.

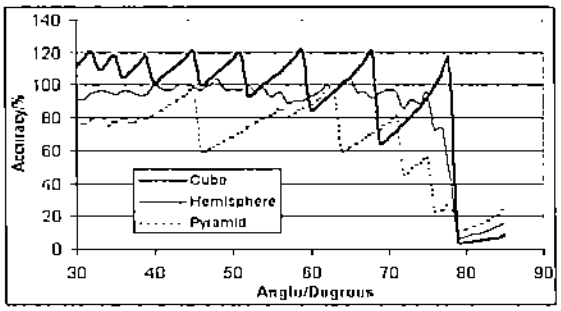


Fig. 4: a) The measurement accuracy as the function of laser angle α_L. Image size 10×10 pixels.

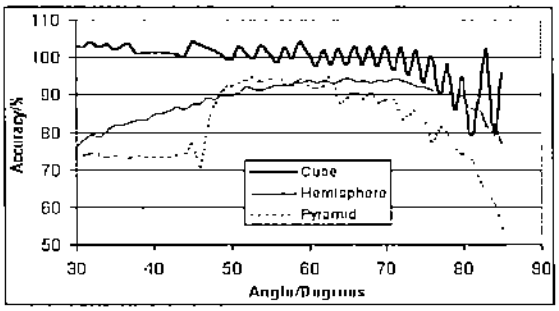


Fig. 4: b) The measurement accuracy as the function of α_L. Image size 50×50 pixels.

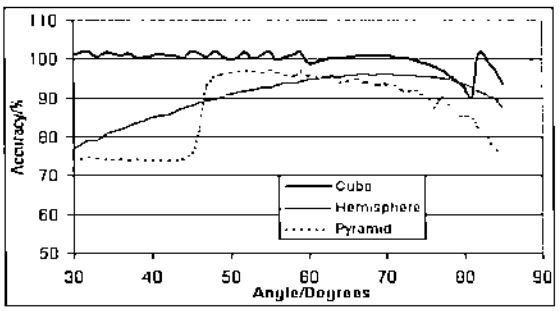


Fig. 4: c) The measurement accuracy as the function of α_L. Image size 100×100 pixels, without hidden area estimation.

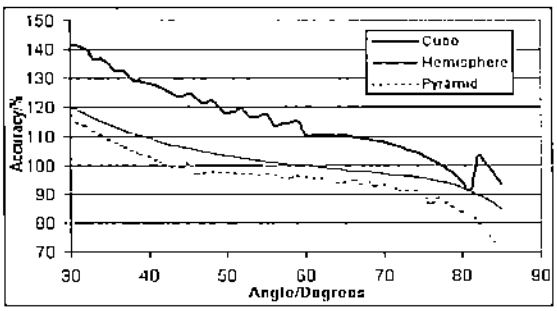


Fig. 4: d) The measurement accuracy as the function of α_L. Image size 100×100 pixels, with hidden area estimation.

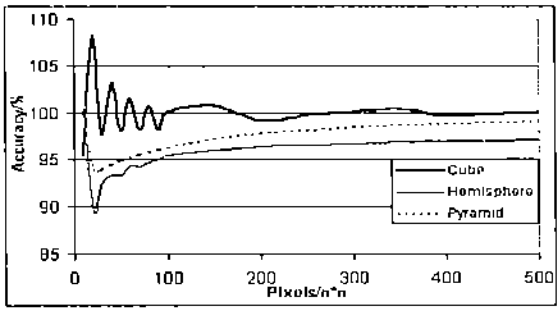


Fig. 4: e) The measurement accuracy as the function of the number of pixels $n \times n$.

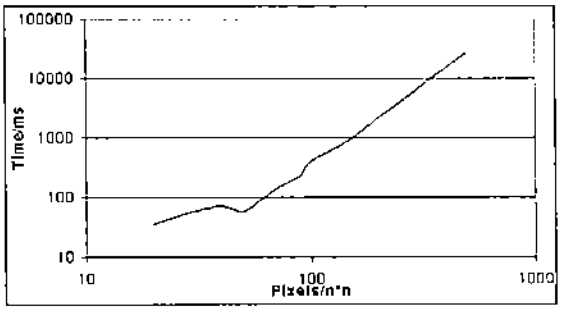


Fig. 4: f) The measurement time as the function of the number of pixels $n \times n$.

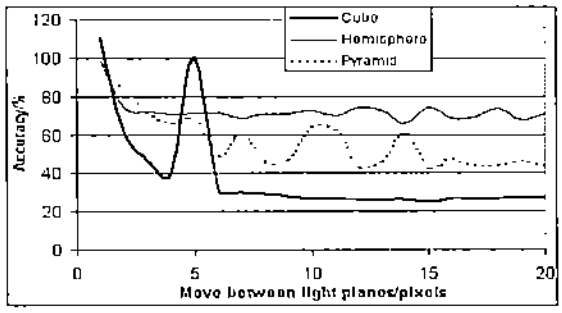


Fig. 4: g) The measurement accuracy as the function of the step size s.

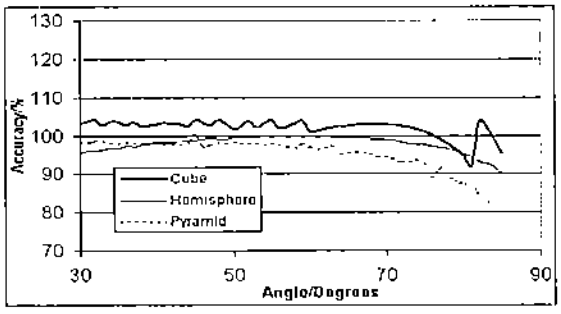


Fig. 4: h) The measurement accuracy as the function of α_L. Image size 100×100 pixels. Two different scans with $\alpha_r = 90°$.

The basic parameters for the GA-based simulations were selected to be: $\alpha_L = 63°$. s corresponds to one pixel, image size 100×100 pixels. Scanning was done using 1-4 directions; with two directional scans with both 90° and 180° angles between scan directions.

3. EXPERIMENTAL RESULTS

3-D free form surfaces were generated as products of 3rd degree Bezier curves. The implementation is a simplified version of the traditional Bezier surfaces and resorted, because we wanted to restrict the number of parameters that GA optimizes, to less than one hundred. In our free form surface model four control points form one curve segment, 11 segments are further attached together to form a composite Bezier curve, in such way that the last control point of each curve segment was also the first control point of the next curve segment. The first and last control points of the composite curve were equal to zero, so that the surface would start and finish at the zero level. Therefore, we needed altogether 32 vertical and 32 horizontal control points. The surface information was implemented as a chromosome consisting of 64 floating point numbers in the range [0.0, 50.0]. We used DeCastaljau[22] algorithm to define $x_i = C_1(i)$ and $y_j = C_2(j)$ values of the current point (i, j), i.e. x_i was calculated from the first composite curve, and y_j from the second, and the value of surface height $z_{i,j} = \sqrt{x_i \times y_j}$.

The object of the first test runs was to find the largest relative error. Unfortunately, the GA then generated almost zero values for the whole surface, and the vision software could not see much difference compared to zero level causing the relative error to be almost 100%. Anyway, this confirms that our GA was capable of finding out some system weaknesses. The tests represented thereon were done in such a way that the volume was normalized to be a constant. Therefore, the absolute and relative errors were proportional.

3.1. Results

Table 1 shows the results of the accuracy measurements. The first column shows that with our test set (cube, hemisphere, pyramid) the worst accuracy was always such that the object was measured too small (negative error). The more scan directions the better the accuracy.

The first GA runs just tried to optimize the absolute relative error without giving any significance to the sign of the error, which lead the optimization always find the largest negative error. However, when we then made new test runs to see what the maximizing of relative error would do, we found out that with almost all cases we can find cases where the proportional error was actually relatively higher on the positive error side. Therefore we decided to run tests, where we minimize and maximize the relative error concurrently. The highest and lowest 25% of GA population survived for the next generation and the middle 50% were replaced by new individuals. This way the population also stays more diverse. We did some comparison runs by only minimizing or maximizing the upper or lower error limits, and it seems that the concurrent optimization finds the limits approximately as effectively.

The accuracy gets also better with the GA generated free form surfaces with more scan directions, in the case of negative error. However, for some reason there seems to be a larger error with 3 and 4 scan directions, in the case of positive error. This implies that the measurement software does not combine the three or four directional height information correctly. This is a point for further software development.

The results imply that the two directional scan is the best overall, and $\alpha_s = 90°$ is better than $\alpha_s = 180°$. For some reason the $\alpha_s = 90°$ two directional scan error limits are strongly skewed towards negative error implying some sort of some systematic error in system. If we could eliminate the error, the system might be quite accurate, since the GA was unable to generate objects that were measured more than [-14.35, …. 2.65]% error. However, for the free form surface model, the accuracy is much lower: free surface matrix with the GA, gave error in the range [-15.6, …, 65.0]%.

Figure 5 shows two examples of GA generated surfaces, the corresponding reconstructed surfaces, and measurement errors. The first three figures (5a-c) show a surface, for which the error is negative. Figure 5c shows the measurement error; the negative areas on the figure are the ones that the measurement system does not see in the original surface. From these figures, it is difficult to tell the difference, but according to the error figure 5c, the error is quite substantial in some parts of the surface; the problem are steep hillsides that are in the shadows.

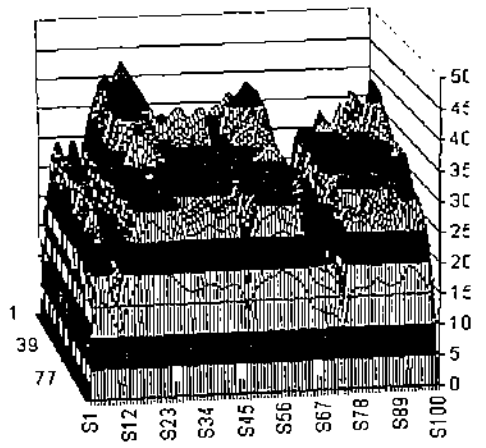


Fig. 5: a) An example of a GA generated 3-D test surface that caused a negative measurement error.

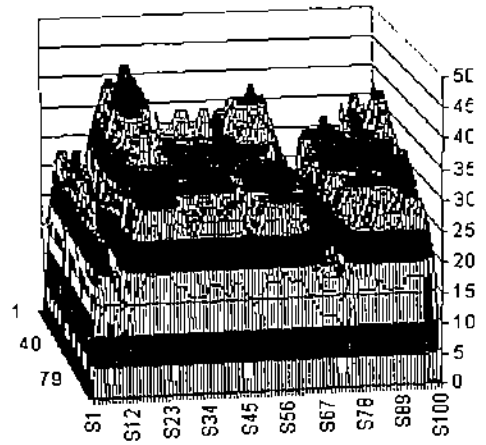


Fig. 5: b) Measured and reconstructed 5a.

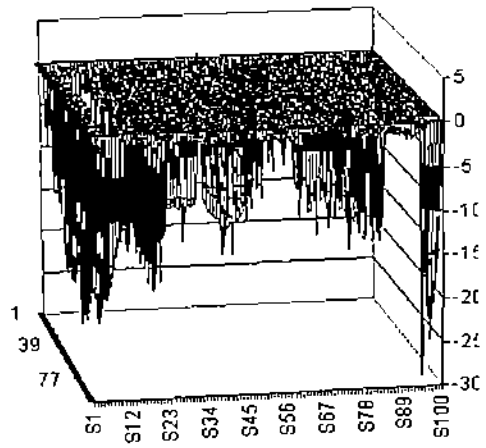


Fig. 5: c) Measurement error between 5a and 5b.

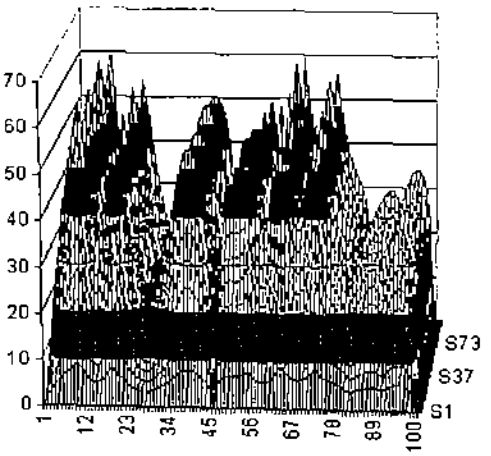


Fig. 5: d) An example of a GA generated 3-D test surface that caused a positive measurement error.

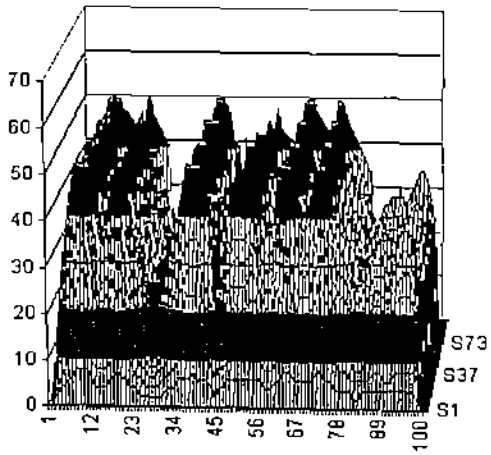


Fig. 5: e) Measured and reconstructed 5d.

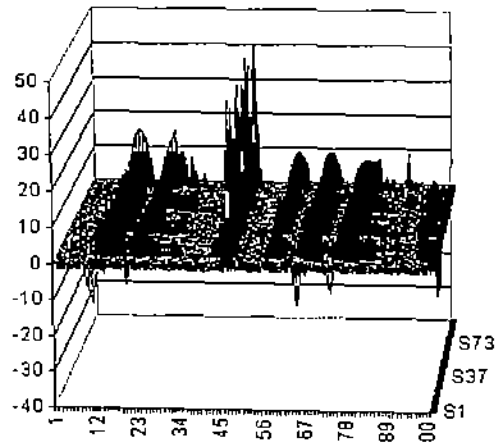


Fig. 5: f) Measurement error between 5d and 5e.

The last three figures (5d-e) show the results of a surface, for which error is positive. The error profile figure (5f) shows as positive those areas that were measured too high and negative those areas that were measured too low. In this series, it is easier to tell the differences between the original and the reconstructed profiles; peaks have been smoothed and correspondingly valleys have been somewhat filled or even disappeared. The error profile discovers also a part of the original surface that the measurement misses, a hill that has been shadowed by larger hills around it.

Table 1: The measurement error tolerances of test objects of figure 3 and GA generated free form surfaces

# scan directions	Worst accuracy with test object set (fig. 3a-e) [%]	Worst accuracy with GA generated surfaces, when error < 0 [%]	Worst accuracy with GA generated surfaces, when error > 0 [%]
1	-5.98	-16.88	34.30
2, $\alpha_s = 90°$	-4.82	-14.35	2.65
2, $\alpha_s = 180°$	-4.82	-18.58	21.83
3	-4.15	-6.31	26.08
4	-4.15	-5.78	29.85

The generated test surfaces were probably more complicated than the ones we were trying to measure with the device. However, especially surface in figure 5a shares many similarities with the real surfaces we were trying to simulate. Taking into consideration that the imaging part of the simulation has been idealized, the more complex surfaces may have properties that balance things. However, reflections and other real world optical problems were not considered, therefore the measurement error limits achieved must be considered as the maximum accuracy limits, rather than minimum or average.

Figure 6 shows an example of how the fitness of the best individuals in the population develops during the test run, and how the diversity of the population fitness behaves. These curves are from optimization run with two scans with α_s=90°. The fitness curves (left vertical axis) show the logarithmic development that is typical for GA. The development virtually ends after 100 generations. The diversity (right vertical axis) of the population is quite well maintained perhaps due to minimizing and maximizing concurrently. A separate maximization and minimization tends to cause the diversity to drop near to zero in 100 generations. Longer test runs up to 1000 generations, were performed, but no significant improvement was detected.

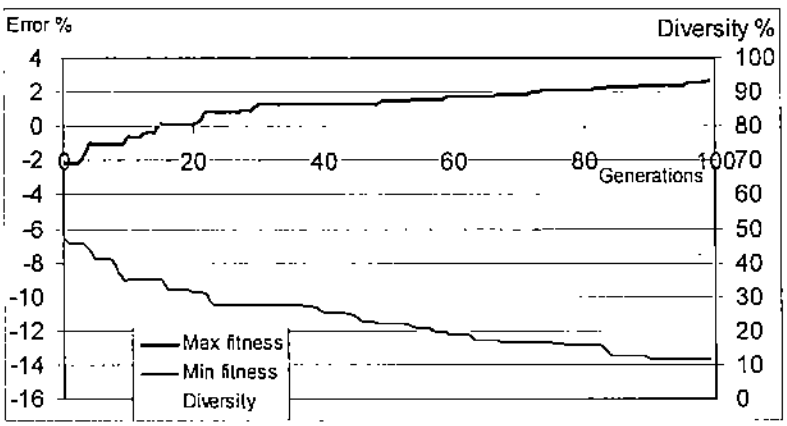


Fig. 6: The development of fitness and population diversity when maximizing the measurement error.

The speed of the object structured light software written in Java and executed in 667 MHz Pentium™ computer was such that analyzing one set of 100×100 pixel surface images took approx. 1/4 s.

4. CONCLUSIONS AND DISCUSSION

The results got in this study confirm that the GAs seem to be capable of generating test surfaces for testing structured light 3-D vision software. This leads to the conclusion that problematic surfaces have some features in common that the GA is able to adapt to. The testing was able to give us some estimation of the software accuracy tolerances. Unfortunately our results are not exactly comparable to any represented in literature, because we wanted to simulate surfaces that somewhat correspond to the ones we try to measure. These test runs also revealed that the tested software did not combine several scans well, because two scan directions lead more accurate results than three or four.

4.1. Future

The implementation was not necessarily the best, and future implementation may use real Bezier surfaces, although it requires more control points and leads lower processing. In future, we might adapt co-evolution[23] to achieve measuring parameters that generates more accurate measurement result. The GA population could contain several species, the height profile species, and species that define laser angle and other possible measurement options. the fitness values for individuals of one species may be defined by the help of other species. The same method could be applied for further software development: there could be species in the GA population that defines rules how the three or four directional height information should be combined. In general, this kind of approach could be used in the design and testing of demanding software.

ACKNOWLEDGEMENTS

Timo Rautakoura is acknowledged for his comments during the work. Pentium is a trademark of Microsoft™ Company.

REFERENCES

1. J. Holland, *Adaptation in Natural and Artificial Systems*, University of Michigan Press. Ann Arbor, MI, Reissued by The MIT Press, 1992.
2. R. Goldstein, and R. Nagel, "3-D visual simulation," *Simulation* 16, pp. 25-31, Jan. 1971.
3. F. DePiero, and M. Trivedi, "3D computer vision using structured light: design, calibration and implementation issues," *Advances in Computers* 43, pp. 243-278, 1996.
4. R. Valkenburg, and A. Melvor, "Accurate 3D measurements using a structured light system," *Image and Vision Computing* 16, pp 99-110, Feb. 1998.
5. C. Darwin, *The Origin of Species: By Means of Natural Selection or The Preservation of Favoured Races in the Struggle for Life*, Oxford University Press, London, A reprint of the 6th edition, 1968.
6. X. Li, T. Kodama, and Y. Uchikawa, "A reconstruction method of surface morphology with genetic algorithms in the scanning electron microscope," *Journal of Electron Microscopy* 49, pp. 599-606, 2000.
7. A. Oyama, S. Obayash, K. Nakahashi, and N. Hirose, "Aerodynamic wing optimization via evolutionary algorithms based on structured coding," *CFD Journal* 8, pp. 570-577, Jan. 2000.
8. R. Mäkinen, J. Periaux, and J. Toivanen, "Multidisciplinary shape optimization in aerodynamics and electromagnetics using genetic algorithms," *International Journal for Numerical Methods in Fluids* 30, pp. 149-159, 1999.
9. M. Peysakhov, V. Galinskaya, and W. Regli, "Using graph-grammars and genetic algorithms to represent and evolve lego assemblies," in *Genetic Algorithms and Evolutionary Computing Conference (GECCO 2000), Las Vegas, NV, Late breaking papers*, pp. 269-275, Morgan Kaufmann Publishers, San Francisco, CA, 2000.
10. J. T. Alander, and J. Lampinen, "Cam shape optimization by genetic algorithm," in D. Quagliarella, J. Périaux, C. Poloni, and G Winter, G., eds., *Genetic Algorithms and Evolution Strategies in Engineering and Computer Science*, pp. 153–174, John Wiley & Sons, Chichester, England, 1997.
11. P. Bezier, *Numerical Control: Mathematics and Applications*, Wiley, 1972.
12. J. Lampinen, "Choosing a shape representation method for optimization of 2D shapes by genetic algorithm," In J. T. Alander, ed., *Proceedings of the Third Nordic Workshop on Genetic Algorithms and their Applications (3NWGA)*, Helsinki, Finland, 18.-22.Aug. 1997, Finnish Artificial Intelligence Society (FAIS), Helsinki, Finland, pp. 305–319, 1997.
13. J. Lampinen, *Cam Shape Optimization by Genetic Algorithms*, Acta Wasaensia, Vaasa, Finland, 1999.

14. S. Kirihara and H. Saito, "Shape modeling from multiple view images using GAs," *ACCJ'98, Lecture Notes in Computer Science* **1352**, pp. 448 – 454, Jan. 1998.
15. N. Evans, and D. Shealy, "Design and optimization of an irradiance profile-shaping system with a genetic algorithm method," *Applied Optics* **37**, pp. 5216-5221, Aug. 1998.
16. I. Ono, Y. Tatsuzawa, S. Kobayashi, and S. Yoshida, "Designing lens systems taking account glass selection by real-coded genetic algorithms," in *Systems, Man, Cybernetics, 1999. IEEE SMC'99 Conference Proceedings, Vol. 3*, IEEE, pp. 592-597, 1999.
17. J. T. Alander, *An Indexed Bibliography of Genetic Algorithms in Optics and Image Processing*. Department of Information Technology and Production Economics, University of Vaasa, Report Series No. 94-1-OPTICS, 2000. Available via ftp:< ftp://garbo.uwasa.fi/cs/report94-1/ gaOPTICSbib.ps.Z >
18. J. T. Alander, T. Mantere, G. Moghadampour and J. Matila, "Searching protection relay response time extremes using genetic algorithm – software quality by optimization," *Electric Power Systems Research* **46**, pp. 229-233, 1998.
19. J. T. Alander, and T. Mantere, "Automatic software testing by genetic algorithm optimization, a case study," in C. Ryan and J. Buckley, eds., *SCASE'99 - Soft Computing Applied to Software Engineering*, Apr. 11-14, 1999, Limerick, Ireland, pp. 1-10, 1999.
20. T. Mantere, and J. T. Alander, "Automatic software testing by genetic algorithms – introduction to method and consideration of possible pitfalls," in R. Matousek and P. Osmera (eds), *MENDEL2001 7th International Conference on Soft Computing*, June 6-8, 2001, Brno, Check Republic, Brno University of Technology, Kuncik, Brno, pp. 19-23, 2001.
21. G. Farin, *Curves and Surfaces for Computer Aided Geometric Design – A Practical Guide, 2nd Edition*, Academic Press, Inc., San Diego, CA, p. 444, 1990.
22. P. DeCastaljau, *Shape Mathematics and CAD*, Kogan Page, London, 1986.
23. J. Koza, "Genetic evolution and co-evolution of computer programs," in Langton et al., eds., *Artificial life II, Proceedings of the Workshop on Artificial Life, Feb. 1990, Santa Fe, NM, Proceedings vol. X, Santa Fe Institute Studies in the Sciences of Complexity*, Addison-Wesley Reading, MA, pp. 603-629, 1992.

Developing and Testing Structural Light Vision Software by Co-Evolutionary Genetic Algorithm

Timo Mantere
University of Vaasa
Department of Engineering
P.O. Box 700, FIN-65101 Vaasa
+358 6 324 8679
timo.mantere@uwasa.fi

Jarmo T. Alander
University of Vaasa
Department of Engineering
P.O. Box 700, FIN-65101 Vaasa
+358 6 324 8444
jarmo.alander@uwasa.fi

Abstract

In this paper we propose an approach to automatically develop and test software by co-evolutionary optimization using genetic algorithms. The idea is to generate both rule based methods for combining scan image data and corresponding simulated test surfaces for a structured light volume measurement system. The goal is to minimize the worst case behavior of error bounds of the volume measurement. One genetic algorithm is used to generate rules to combine scan data that give minimum relative error on the test surface population, which is generated by another genetic algorithm trying to create surfaces giving high measurement error. Thus the surface population defines the fitness of the method population and vice versa. Based on observations of evolution in nature it is believed that it is this kind of co-evolution that leads in the long run to excellent solutions, that would be difficult to find by more traditional genetic algorithm approaches. Indeed, the preliminary results got seem to indicate that co-evolution is beneficial in software development and testing.

Keywords

Co-evolution, genetic algorithms, image processing, 3-D imaging, machine vision, simulation, software engineering, software testing, structured light vision.

1. Introduction

A co-evolutionary optimization based approach for software development and testing is proposed in this work. An example of the approach applied to the development and testing of structured light vision system is given. The goal was to find test surfaces that are most difficult for the object software to measure, in other words to find out the error bounds of the volume measurement. Simultaneously the measurement routine was developed by optimizing its parameters to give the minimum error when applied on the test surfaces.

The problem was to measure the volume of small objects fixed on a planar surface. Knowing the profiles of the objects we can interpolate their volume within certain error bounds. There are several methods that are developed for imaging based surface metrology, like stereo photography, structured light vision, and shading, reflection, and focus based methods. In this paper, we concentrate on structured light vision [9, 25]. This work was done in one of our research projects, developing a high-speed 3-D measurement system for small objects. To confirm that the proposed system meets the accuracy demands we decided to simulate the system first.

Genetic algorithms (GA) [12] are optimization methods that are known to find quite good solutions to many difficult optimization problems; therefore, we decided to use the GA to generate 3-D test surfaces. The imaging process was simulated by evaluating how light planes would lie on the simulated surfaces (ray tracing) [11] and further generating simulated images from these height curves. Though the simulated surface/image does not exactly correspond to the real vision system, it was felt that it is at least a good starting point for testing.

Software testing is an important part of software development. It is time consuming, and even a partial automation can produce considerable savings [21]. The benefits of automatic test tools also include that that they are much more objective than human testers.

2. Genetic Algorithms and Co-evolution

Genetic algorithms are optimization methods that mimic models of evolution in nature [7]. They are simplified computational models of evolutionary biology. The GA forms a kind of electronic population, the members of which fight for survival, adapting as well as possible to the environment, which is actually an optimization problem. The GAs use genetic operations, such as selection, crossover, and mutation in order to generate solutions that meet the given optimization constraints ever better and better. Survival and crossbreeding probabilities depend on how well individuals fulfill the target function. The set of the best solutions is usually kept in an array called population. The GAs does not require the optimized function to be continuous or derivable, or even be expressed as a mathematical formula, and that is why they gain more and more popularity in practical technical optimization. Today the GA methods form a broad spectrum of heuristic optimization methods.

Co-evolutionary computation [6, 14-15] (CEC) generally means that an evolutionary algorithm is composed of several species with different types of individuals, while standard evolutionary algorithm has only single population of individuals. In CEC the genetic operations, crossover and mutations are applied to only

on single species, while selection can be performed among individuals of one or more species. When we deal with an optimization problem, the environmental conditions of which are stochastic or immeasurable, we can try to develop the environmental conditions concurrently with the problem. Trial solutions implied by one species are evaluated in the environment implied by another species. The goal is to accomplish an upward spiral, an arms race, where both species would achieve ever better results.

3. Structured Light Vision

Structured light vision is a method where the object, the surface height of which is measured, is lighted in such a way that sharp lines of light and shadow on the surface can be imaged. In practice, tilted laser illumination is used. The intersection of the light plane and the object forms partial height curves that are recorded by camera and further used to reconstruct the 3-D geometry of the object. In order to measure the whole surface we must use multiple light planes or/and move either the light plane or the object using small steps and recording a new image after each step. The height information from these images must be evaluated for estimation of volume. For this we need to know several parameters including the height curves, the zero level, camera distance from the zero level, scan directions α_i, laser illumination angle α_L, and the focal length, and the actual size that the pixel corresponds to the surface (fig. 1). Usually the measurement system consists on a CCD camera, scanning mechanics and computer.

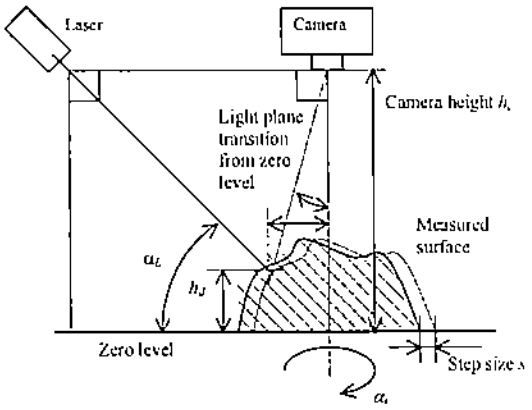


Figure 1. The structure of the structured light vision measurement system, α_L, = illumination angle, α_i = scan direction, h_d directional height of the surface point.

The problem with this kind of measurement is that the camera distorts the image towards borders, causing the surface profile and pixel sizes vary spatially. Another problem is that the light plane is not ideally thin, and it may cover several pixels with different tones, so that the exact borderline of the light curve is difficult or even impossible to define. If the light plane is less than one pixel thick, the maximum error is equal to the pixel size; the plane is within one pixel and the location cannot be evaluated more accurately within one frame.

4. Related Work

Genetic algorithm has previously been adapted to the surface simulation problem in ref. [18]. There are also several studies of shape optimization [2, 19, 23-24] using genetic algorithms. For shape representation using Bezier curves [5] and GAs, see refs. [2, 16-17], and references therein. The use of GAs for shape modeling from images is represented in ref. [13], and examples of GAs in optical system design [10, 22]. In addition, the GAs has been largely applied to image processing; see bibliography [3]. GAs has previously been applied to automatic data generation for software testing in several studies, see refs. [3-4], and references therein.

5. The Proposed Method

The proposed method consists of a co-evolutionary GA that consists of two species: one representing control point vectors, and the other representing measurement rules. The latter includes α_i for each scan direction and the rules how to combine directional height information, when reconstructing the measured surface. The first species are used by a surface generator procedure `SurfCreate` to create 3-D surfaces, from which they are further transformed into surface height curves by procedure `EvalCurve`, by using the α_i from the second species. The simulated surface height images are then sent to the structured light vision software SLV that evaluates the height information and reconstructs the 3-D model of the surface, by using the information how to combine directional height information from the second species.

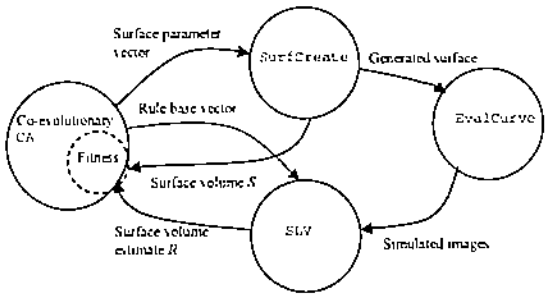


Figure 2. The structure of the proposed test system.

The rules are given in form

$$H = \frac{\sum_i (w_i h_{di})}{\sum_i w_i} \qquad (1)$$

where H is the computed height of some surface point, h_{di}s are directional heights scanned from each direction, and weights w_i for the heights. Weights are used so that the heights scanned from different directions are first sorted, and then the first

weight of the array is always multiplied with highest h_{di}, and the other ones correspondingly in the decreasing order. Index i represents the running number of items in the weight array, and h_{di}:s sorted to the decreasing order. This way we can define rules like [1.0, 0, 0, 0] or [0, 1.0, 1.0, 0] that respective mean *"the height is the highest directional height"* and *"the height is the mean of the middle values of directional heights"*.

The rules are floating point numbers, so that we can get any mixture of directional values. The rule base also includes some special rules that cannot directly be expressed as floating point vectors, like *"the height is the lowest directional height that is greater than zero"*, *"the height is the value closest to the average"*, or *"the height is the most frequent value"*. These special rules are forced when extra parameter value is within some predefined interval. The rule base may also contain α_L = [45° ... 85°] for each scan direction.

The fitness function is evaluated by a subprogram called `Fitness`. It gets as its input the original and reconstructed surfaces S and R, from which it evaluates the differences $f_{ij} = R_{ij} - S_{ij}$, where i is the index for surfaces, and j is the index for the rules. The fitness of an item belonging to the surface species is

$$f_S(i) = \max_j(f_{ij},0) - \min_j(f_{ij},0) \qquad (2)$$

and the fitness of an individual item of the method species is

$$f_P(j) = \max_i\left|f_{ij}\right| \qquad (3)$$

Fitness functions define that the fitness value for f_S is the error interval, and for the f_P the fitness value is the absolute value of maximum error. These fitness function definitions were selected, because they seemed to be the most stable and best working of the half a dozen different fitness definitions experimented with. The co-evolutionary method tries to maximize f_S and minimize f_P.

GA parameters used in co-evolutionary rule tuning were: population size 30 in both species, elitism 50%, crossover rate 50%, both one point and uniform crossovers between chromosomes with 50/50 ratio, in addition, arithmetic crossover between genes was applied at the rate of 10%, and mutation probability was 2%. Test runs consisted of 60 generations.

GA parameters used in the verification tests were: population size 100, elitism 50%, crossover rate 50%, and mutation probability 2% New individuals were generated by applying both one point and uniform crossovers between chromosomes with 50/50 ratio; in addition, arithmetic crossover between genes was applied at the rate of 10%. We decided [20] to run tests, where the error bounds were maximized. The highest and lowest fitness quartiles of GA population survived for the next generation and new individuals replaced the middle quartiles. This way the population also stays more diverse. We did some comparison runs by only minimizing or maximizing the upper or lower error bounds, and it seems that the concurrent optimization finds the bounds approximately as effectively. Test runs reported here consisted of 100 generations.

3-D freeform surfaces were generated as products of third degree Bezier curves. The implementation is a simplified version of the traditional Bezier surfaces and resorted, because we wanted to restrict the number of parameters that GA optimizes, to less than one hundred. In our free form surface model four control points form one curve segment, 11 segments are further attached together to form a composite Bezier curve, in such way that the last control point of each curve segment was also the first control point of the next curve segment. The first and last control points of the composite curve were equal to zero, so that the surface would start and end at the zero level. Therefore, we needed altogether 32 vertical and 32 horizontal control points. The surface information was implemented as a chromosome consisting of 64 floating-point numbers in the range [0.0, 50.0]. We used DeCastaljau [8] algorithm to define $x_i = C_1(i)$ and $y_j = C_2(j)$ values of the current point (i, j), i.e. x_i was calculated from the first composite curve, and y_j from the second, and the value of surface height .

The tests represented hereon were done in such a way that the volume was normalized to be a constant. Therefore, the absolute and relative errors were proportional.

When the object is imaged from one direction, the rear hillside is hidden in the shadows, and not measured properly, the same happens to the small peaks behind a high peak. These blind areas can be seen by scanning the object from several directions. However, scanning from different directions produce different height matrices, where in some areas the height values might differ substantially due to the fact that the corresponding area is in the blind zone when viewed from some other direction. We have to apply some rules for defining the height, if scans from different directions lead to different values. The rule could be using the highest, median, mean, etc. value or some combination of them.

This rule base could also be optimized e.g. by genetic algorithm. However, if we optimize the rule base with some static test object set, we cannot be sure that the same rule base produces the best accuracy with other objects to be measured. This is where co-evolution comes in to the picture. We optimize the rule base with GA and at the same time as we are testing the system by trying to find the most difficult object to be measured by GA The aim is that GA optimizes the worst shape for current rule base and at the same time the best rule base for the current test shapes If this kind of co-development is achieved, we could assume that the eventual rule base is satisfactory with any object shapes of the same type.

6. Experimental Results

This work is a continuation to that given in ref [20]. The results in that study showed that the structured light vision software had some problems when combining data from several scan directions.

Figure 3 shows an example of fitness development during the co-evolutionary test run. The fitness values shown are the values of best individual, i.e. for f_S the maximal value and for f_P the minimal value. At the beginning the method population rapidly evolves towards small error bounds. This causes the surface population to evolve slowly towards more difficult test cases, which further causes the fitness of the method population follow quite closely the fitness of the surface population. Obviously the speed of evolution is determined by the speed of the test surface

population. In this case it seems to be much more difficult to create challenging test cases than robust software. The obvious reason is that the surface has much more parameters than the method under development.

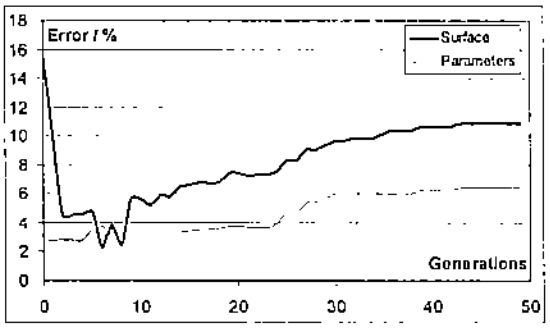


Figure 3. The co-evolutionary development of surface and parameter population fitness functions with four directional scans (4a4+p, see fig. 8).

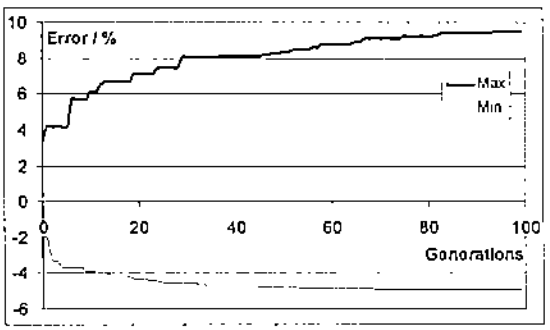


Figure 4. error bounds in the validation test run with same case as in fig. 3 (4a4+p).

Table 1. The relative measurement error. A) and B) are upper and lower error tolerances from the ref. [20], and C) and D) are corresponding error tolerances after co-evolutionary software tuning in this study.

# scan directions	A [%]	B [%]	C [%]	D [%]
1	-16.88	34.30	-5.65	22.66
2, $\alpha_1 = 180°$	-18.58	21.83	-4.66	24.67
2, $\alpha_1 = 90°$	-14.35	2.65	-4.47	11.39
3	-6.31	26.08	-4.66	12.51
4	-5.78	29.85	-5.27	9.51

Figure 4 shows how the upper and lower accuracy error bounds develop in the validation test run, where the GA only tries to find the largest accuracy error by generating test surfaces. The rule base parameters are locked to those found by the co-evolutionary test run.

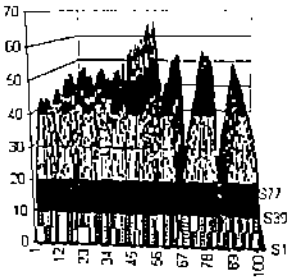


Figure 5. An example of a GA generated 3-D test surface.

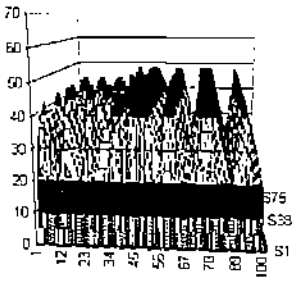


Figure 6. Measured and reconstructed surface of fig. 3.

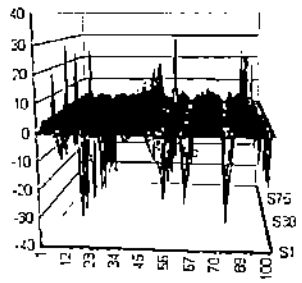


Figure 7. Profile of the measurement error between surfaces shown in figs. 4 and 5.

Table 1 shows the results of the tests in ref. [20]. In the case of negative error (A) the accuracy got better when more scan directions were used. In the case of positive error (B), however, three and four scan directions seem to cause increasing error. This was a point for further software development.

The co-evolutionary method was applied in this study to see if the GA could improve measurement accuracy by optimizing, concurrently with the test surfaces, the rule base of how to combine height data from several directions.

Table 1 shows also the results after tuning the system with the co-evolutionary GA. The accuracy in all cases got better with the negative error (C). In the case of positive error (D), the accuracy get better in all other cases, except in the case of two

scans with 90° angle between the scan directions. In that case the negative error bound has gotten better, but the positive error bound worse, unfortunately, the error bounds interval has widen, but the error bounds were now more centered around the zero level (see fig. 6).

The results imply that now the precision increases with increasing scan directions. This implies that the measurement software has evolved to combine scan data better than the earlier version of our software [20]

In all cases, the error bounds are skewed towards positive error, implying some sort of systematic error. If we could eliminate the error, the system might get even more accurate.

Figures 5-7 shows an example of GA generated surface after co-evolutionary tuning, the corresponding reconstructed surfaces, and the profile of measurement error.

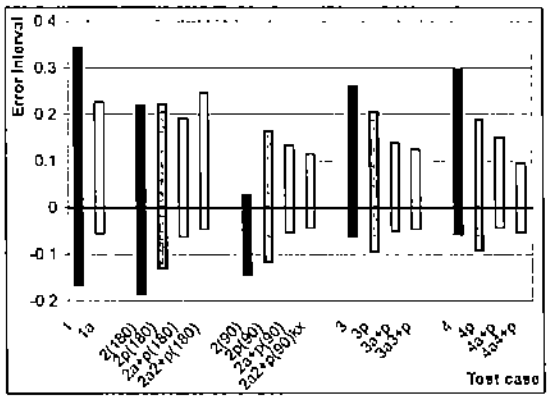


Figure 8. The developments of measurement error tolerance bounds with co-evolutionary tuning for each number of scan directions (1, ..., 4) . Markings of the x-axis in the order: the number of scan directions; what is optimized: p = parameters how to compile directional height information, a=camera angle, a+p = both of the previous with using the same camera angle for each scan directions, an+p as a+p but using different camera angle for each scan direction; the angle between scan directions shown by (α).

Figure 8 shows how the error tolerance bounds of different number of scan directions have developed with co-evolutionary tuning. For one scan direction, for which only the illumination angle could be optimized, the development of bounds were surprisingly high (44.7%), the optimal α_I seems to be about 71.6°.

When we are using more scan directions, we did three tuning tests. In the first test (marked dark gray in fig. 8) we tune only the rule base used to combine the directional information, the overall system (camera angles) were not changed. In the second test (light gray in fig. 8), we tune the camera angle together with the rule base; the same angle was used for each direction. In the third test series (white in fig. 8), we tune the camera angle for each scan direction together with the rule base. The original error bounds taken from ref. [20] are marked with black.

Figure 8 illustrates how the system gets nearly always when the number of system parameters to be tuned is increasing. The only exceptions are the case of two directional scans, with $\alpha_S = 90°$, the first test is worse than the original but increases gradually, and with $\alpha_S = 180°$ the best accuracy was found with the same α_I for each scan direction.

With scans from two directions the co-evolutionary tuning did not gain much more accuracy, while with $\alpha_S = 180°$ the accuracy improved by 27.4%, while with $\alpha_S = 90°$ the improvement was 6.7%. With three and four scan directions the accuracy improved 47% and 58.5% respectively. Not surprisingly after the tuning process the case when scanning from four directions is the most accurate.

The optimal parameter set found case of scanning from four directions were approx. w = [0.44, 0.13, 0.54, 0.69] and α_I = {76.0, 79.5, 72.9, 68.9} degrees

7. Fitness landscape

In order to evaluate if our co-evolutionary approach was beneficial in this case, we decided to analyze the fitness landscape as follows. Starting from the optimized test surface population random mutations were applied to random items of the population while recording the corresponding fitness of the volume measurement method.

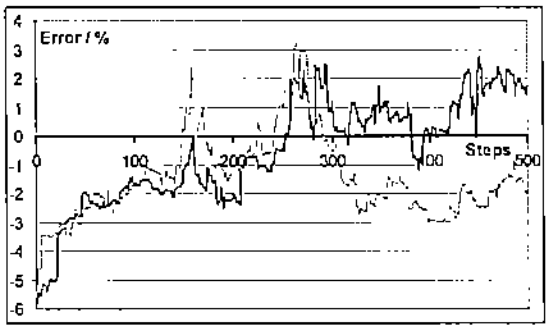


Figure 9. Two random walks in the fitness landscape before co-evolution.

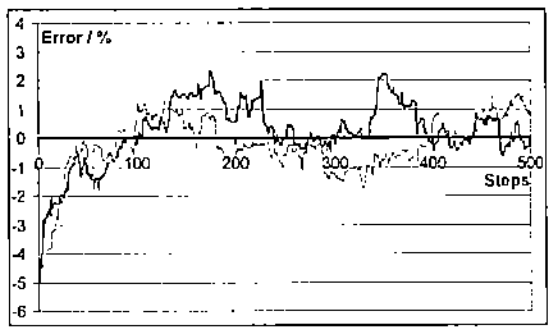


Figure 10. Two random walks in the fitness landscape after co-evolution.

Random walks created by this method are shown in figures 9 and 10. In both figures, the random walk was started from the negative optimum. There are two runs in both figures, just to illustrate that the major patterns of the walks are similar. At first the fitness value roughly exponentially approaches the zero error level starting then randomly vary around it. Before co-evolution the random surfaces cause higher relative error than after co-evolution. Hence co-evolution has made the system more stable. When we run longer random walks the original system causes about twice as high total error, sum of absolute value of relative error, than the co-evolutionarily optimized system.

Figures 9 and 10 also show that the number of random steps needed to escape the difficult test case is higher for the original than for the co-evolutionarily optimized method. This can be explained so that the most difficult test surface is quite exceptional. A short random walk hardly reveals a similar case. However, in the original system it needs more steps to reach optimum from random landscape The optimized system is so well tuned that normal random surfaces causes less error, and the optimal area is closer to the random landscape.

8. Conclusions and Discussion

In this paper we have developed and tested a structural light vision software applying a co-evolutionary method in order to simultaneously develop the measurement system parameters and the corresponding test data.

The results got in this study confirm that the co-evolutionary application of GA seems to be capable of generating test surfaces for testing structured light 3-D vision software, and concurrently finding system rules that lead to better measurement accuracy. This leads to the conclusion that problematic surfaces have some features in common that the GA is able to adapt, and respectively that GA learns how to arrange measurement geometry and process the corresponding scan data.

A preliminary analysis seems to confirm that the tuned system is more capable of handling random surfaces, and there are less space for extreme cases.

The surface model was not necessarily the best, and future implementation may use real Bezier surfaces, although it requires more control points and leads to slower processing.

In general, this kind of co-evolutionary approach could be used in the design and testing of demanding software and measurement systems.

In the future, we intend to do more extensive fitness landscape analysis how the co-evolutionary method effects fitness landscape. We should evaluate if the improvements really are due the using of co-evolution. However, computationally the co-evolutionary tuning is exactly as costly as optimizing the system against some static test surface set. So, it is hard to see what disadvantage it could cause, since static set can not cover all possible cases, and co-evolutionary GA can find the pathological case for bad system parameters during the tuning, which static test set would not be able to do.

9. Acknowledgments

Our thanks to Prof. Jukka Tiusanen for his comments concerning English writing.

10. References

[1] Alander, J.T. An Indexed Bibliography of Genetic Algorithms in Optics and Image Processing. Department of Information Technology and Production Economics, University of Vaasa, Report Series No. 94-1-OPTICS. 2000 Available via ftp:< ftp://ftp uwasa.fi/cs/report94-1/gaOPTICSbib.ps.Z >

[2] Alander, J.T., and Lampinen, J. Cam shape optimization by genetic algorithm. In D. Quagliarella, J. Périaux, C. Poloni, and G Winter, G (eds.) Genetic Algorithms and Evolution Strategies in Engineering and Computer Science, pp. 153–174, John Wiley & Sons, Chichester, England, 1997.

[3] Alander, J.T., Mantere, T., Moghadampour G , and Matila, J. Searching protection relay response time extremes using genetic algorithm – software quality by optimization. In Electric Power Systems Research 46, pp. 229-233, 1998.

[4] Alander, J.T.,and Mantere, T. Automatic software testing by genetic algorithm optimization, a case study. In C. Ryan and J. Buckley (eds.). SCASE'99 - Soft Computing Applied to Software Engineering, April 11-14, 1999, Limerick, Ireland, pp. 1-9, 1999.

[5] Bezier, P. Numerical Control: Mathematics and Applications. Wiley, 1972.

[6] Brodie III, E., and Brodie Jr., E. Predator-prey arms races. In BioScience 49, pp. 557-568, July 1999.

[7] Darwin, C. The Origin of Species: By Means of Natural Selection or The Preservation of Favoured Races in the Struggle for Life. Oxford University Press, London, A reprint of the 6th edition, 1968.

[8] DeCastaljau, P. Shape Mathematics and CAD. Kogan Page. London, 1986.

[9] DePiero, F., and Trivedi, M. 3D computer vision using structured light: design, calibration and implementation issues. In Advances in Computers 43, pp. 243-278, 1996

[10] Evans, N., and Shealy, D. Design and optimization of an irradiance profile-shaping system with a genetic algorithm method. In Applied Optics 37, pp. 5216-5221, Aug 1998.

[11] Goldstein, R., and Nagel, R. 3-D visual simulation. In Simulation 16, pp. 25-31, Jan. 1971.

[12] Holland, J. Adaptation in Natural and Artificial Systems. University of Michigan Press, Ann Arbor, MI, Reissued by The MIT Press, 1992.

[13] Kirihara, S., and Saito, H. Shape modeling from multiple view images using GAs. In ACCV'98, Lecture Notes in Computer Science 1352, pp. 448 – 454, Jan 1998.

[14] Kojima, F., and Kubota, N. Electromagnetic inverse analysis using coevolutionary algorithm and its application to crack profiles identification. In R. Matousek and P. Osmera (eds.). MENDEL2001 7th International Conference on Soft Computing, June 6-8, 2001, Brno, Check Republic, Brno University of Technology, Kunelk, Brno, pp. 75-80, 2001.

[15] Koza, J. Genetic evolution and co-evolution of computer programs. In Langton et al. (eds.). Artificial life II,

Proceedings of the Workshop on Artificial Life, Feb 1990, Santa Fe, NM, Vol. X, Santa Fe Institute Studies in the Sciences of Complexity, Addison-Wesley Reading, MA, pp. 603-629, 1992.

[16] Lampinen, J. Cam Shape Optimization by Genetic Algorithms. Acta Wasaensia, Vaasa, Finland, 1999.

[17] Lampinen, J. Choosing a shape representation method for optimization of 2D shapes by genetic algorithm. In J. T. Alander (ed.). Proceedings of the Third Nordic Workshop on Genetic Algorithms and their Applications (3NWGA), Helsinki, Finland, 18.-22.Aug. 1997, Finnish Artificial Intelligence Society (FAIS), Helsinki, Finland, pp. 305–319, 1997.

[18] Li, X., Kodama, T., and Uchikawa, Y. A reconstruction method of surface morphology with genetic algorithms in the scanning electron microscope. In Journal of Electron Microscopy 49, pp. 599-606, 2000.

[19] Makinen, R., Periaux, J., and Toivanen, J. Multidisciplinary shape optimization in aerodynamics and electromagnetics using genetic algorithms. In International Journal for Numerical Methods in Fluids 30, pp. 149-159, 1999.

[20] Mantere, T., and Alander, J.T. Testing Structural Light Vision Software by Genetic Algorithms - Estimating the Worst Case Behaviour of Volume Measurement. In D. Casasent, and E. Hall (eds.). Intelligent Robots and Computer Vision XX: Algorithms, Techniques, and Active Vision, volume SPIE-4572, Newton, MA, October 29-31, 2001, SPIE, Bellingham, Washington, USA, pp 466-475, 2001.

[21] Norman, S. Software Testing Tools. Ovum Ltd. London, 1993.

[22] Ono, I., Tatsuzawa, Y., Kobayashi, S., and Yoshida, S. Designing lens systems taking account glass selection by real-coded genetic algorithms. In Systems, Man, Cybernetics, 1999. IEEE SMC'99 Conference Proceedings, Vol. 3, IEEE, pp. 592-597, 1999.

[23] Oyama, A., Obayash, S., Nakahashi, K., and Hirose, N Aerodynamic wing optimization via evolutionary algorithms based on structured coding. In CFD Journal 8, pp 570-577, Jan. 2000.

[24] Peysakhov, M., Galinskaya, V., and Regli, W. Using graph-grammars and genetic algorithms to represent and evolve lego assemblies. In Genetic Algorithms and Evolutionary Computing Conference (GECCO 2000), Las Vegas, NV, Late breaking papers, pp. 269-275, Morgan Kaufmann Publishers, San Francisco, CA, 2000.

[25] Valkenburg, R., and Melvor, A. Accurate 3D measurements using a structured light system. In Image and Vision Computing 16, pp 99-110, Feb. 1998.

Testing Digital Halftoning Software by Generating Test Images and Filters Co-Evolutionarily

TIMO MANTERE[1] and JARMO T. ALANDER

Department of Electrical Engineering and Industrial Management, University of Vaasa, P.O. Box 700, FIN-65101 Vaasa, Finland

In this paper we evaluate the potential of using the co-evolutionary optimization method to automatically and concurrently generate halftoning filters and their test images. One genetic algorithm generates halftone filters for an image processing software used for digital halftoning, while another genetic algorithm tries to create the hardest test image for each filter. The best filter being the one for which the hardest test image, when dithered, differs least from the original. An image population defines the fitness of halftoning filters and vice versa.

Keywords: Co-evolution, digital halftoning, genetic algorithms, software testing, test image generation.

1 Introduction

There does not seem to be much systematic research in the field of test image evaluation. What are the essential characteristics of a good test image? How does one determine what is a good test image for testing the properties of chosen image processing algorithm. How does one determine that a chosen test image is good for testing the quality of your specific image processing software? What is the benchmark test image set you should use? Much more often than not researchers rely on the most commonly used[1] and unfortunately very limited test image sets. Some areas of research like pattern recognition and computer vision may use some image set[2] that is more carefully chosen. However, in research databases[3-5] there seem to be no references to research papers about good test images or their desirable properties.

This work was not primarily done in order to answer the above questions about test images. The main purpose was to do experiments with computer generated test images, and see whether or not they reveal some weaknesses in the image processing software that standard test images do not necessarily do. This work is also a continuation of our software testing research[6-7], where we previously[7] applied co-evolution to optimize software parameters simultaneously with the testing. The analogy to this study is that here we develop the image filters used in the halftoning software simultaneously, when we simultaneously test its quality with the generated test images.

The reason for choosing this approach was that we previously studied the halftone filter design with genetic algorithms[8] (GA) and in a later study[9] the test image generation with GAs for testing these halftoning filters. A natural extension is to study the optimization of these two contradictory elements together. Our preliminary results[9] showed that genetic algorithms are indeed able to find images that are considerably distorted when halftoned, and thus reveal potential problematic image features that the halftoning software is not able to reproduce satisfactorily.

[1] E-mail: timo.mantere@uwasa.fi, phone: +358 40 520 3022, fax: +358 5 621 2899

1.1 Genetic algorithms and co-evolution

Genetic algorithms[10] are optimization methods that mimic evolution in nature. They are simplified computational models of evolutionary biology. A GA forms a kind of simulated population, the members of which fight for survival, adapting as well as possible to the environment, which is actually an optimization problem. GAs use genetic operations, such as selection, crossover, and mutation in order to generate solution trials that meet the given optimization constraints ever better and better. Surviving and crossbreeding possibilities depend on how well individuals fulfill the target function. The set of the best trials is usually kept in an array called population. GAs do not require the optimized function to be continuous or derivable, or even be expressed as a mathematical formula, and that is perhaps the most important factor why they are gaining more popularity in practical technical optimization. Today GA methods form a broad spectrum of heuristic optimization methods under intense study[11].

Co-evolutionary computation[12-13] (CEC) generally means that an evolutionary algorithm is composed of several species with different types of individuals, while a standard evolutionary algorithm has only one single population of individuals. In CEC the genetic operations, crossover and mutations are applied to only a single species, while selection can be performed among individuals of one or more species. When we deal with an optimization problem, we can try to develop the environmental conditions concurrently with the problem. These environmental conditions may be stochastic or immeasurable. Trial solutions implied by one species are evaluated in the environment implied by another species. The goal is to accomplish an upward spiral, an arms race, where both species would achieve ever better results.

1.2 Dithering

Digital halftoning[14], or dithering, is a method used to convert continuous tone images into images with a limited number of tones, usually only two: black and white. The main problem is to halftone so that the bi-level output image does not contain prominent features, such as alias, moiré, lines or clusters, caused by dot placement[15]. The average density of the halftoned dot pattern should interpolate as precisely as possible to the original tones. Dithering methods contain frequency modulated versions, where the pixel size is static and the distances between these pixels vary, and amplitude modulated, where the distance between dots are static, but the size of these dots varies. This study concentrates on frequency modulated halftoning methods only. The halftoning methods used here were error diffusion and threshold with 16×16 element threshold matrices.

1.3 Comparing the images

To compare a dithered image with the original one is obviously a challenging problem. If the images are not compared properly, the evaluated difference between the images may be greatly biased by the comparison method used. One cannot simply use pixel-by-pixel comparison, since dithered images usually have only two tones. The minimum difference by that measure would be achieved if every gray tone were rounded to the nearest tone (black or white), which unfortunately usually results in poor images. Better image comparison methods have been developed[14, 16-18], including a set of methods called inverse halftoning[14]. From these the most

common is perhaps low pass filtering, in which images are first low pass filtered and then the resulting images are compared pixel by pixel. The problem with low pass filtering is that the high frequencies will disappear and the images get a somewhat blurred overall appearance. However, this method is easy to implement and it enables pixel-by-pixel comparison. The low pass filter model based on the human eye modulation transform function (MTF) was considered the best method for finding optimum halftone patterns by its authors[19].

Several fitness functions *i.e.* image comparison methods were tested[9]. In this study we only used the low pass filtering with MTF[19] and the co-evolutionary approach to define proportional fitness values for each individual. Image comparison after low pass filtering was done by the quality metrics[14, 20-21]: PAE, MAE, MSE, RMS, NRMSE, and PSNR (Peak Absolute Error, Mean Absolute Error, Mean Squared Error, Root Mean Squared Error, Normalized Root Mean Squared Error, Peak Signal to Noise Ratio), that are commonly used in image compression research. The goal is that with the help of co-evolution we would eventually be able to automatically generate an appropriate image comparison method, too.

1.4 Related work

Co-evolution is mostly applied in game playing research, but it is now being applied more in technical research as well. However, from the research indexes[3-4] we did not find any studies where image filters and test images would have been developed together. Only one[22] image processing study by Goulermas and Liatsis that applies co-evolution was found, it uses co-evolution for feature-based matching of edges in stereo imaginary. It applies parallel GA, where each individual GA aims local optimum, while information exchange between neighboring GAs are applied to achieve symbiotic co-evolution towards a global optimum. They suggest that searching global and local level optimums concurrently has advantages compared to other approaches.

Miyojim and Cheng[23] propose that test images with characteristics close enough to reality should be generated with a computer and applied to testing pattern recognition algorithms. We agree with their statement "researchers often reuse the same few available test images, which may compromise the thoroughness of the investigation". They also conclude that the proposed approach can be very useful for the development of computer vision, image processing and pattern recognition algorithms.

Forbes and Draper[24] have used synthetic images for evaluating edge-detection algorithms. Their motivation was that "most of the evaluation techniques use only a few test images, leaving open the question of how broadly their results can be interpreted". Their results show that the rank of the goodness of edge detectors depends on the images used. They think that the average best edge detector can be found if large number of images, representing each possible property of the scene, is tested. This can be best achieved by using synthetic imagery.

Kocsis *et al.*[25] use computer generated test images for assessing image quality changes in the compression-decompression process of lossy JPEG images. They used simulated images with characteristics similar to radiographic images. Their motivation seems to be based on the assessment that it is easy to produce synthetic images with large quantities and no expert observers are needed. However, none of those studies[23-25] used evolutionary methods to create test images.

Sims[26] have applied evolutionary algorithms for generating impressive computer generated art, but has not really applied them for any testing purposes. Genetic algorithms have previously been adapted to generating halftoning filters[18, 27-28], the usual conclusion has been that GA generated threshold matrices produce more rounded dot patterns than the usually used ordered threshold matrix.

For further references of GAs in image processing, see bibliography 29 or book 30. There is a relatively large number of studies where GAs have been applied to generation of automatic software test data for assessing software correctness or quality, see[6-7] and references therein, but we have not seen other studies than ours[7] that have tried to develop software parameters simultaneously with the testing by the help co-evolutionary optimization.

2 The proposed method

Halftoning was selected to be the test problem for our co-evolutionary testing of image filters and test images. This selection was done based on our previous experiences with them[8-9]. However, there is no a priori reason why this testing method could not be applied to other similar problems in image processing research.

The GA population contains two species: a parameter vector, which is used by an image generator to create test images and a halftone filter vector, which can further be transformed either into a threshold matrix or error diffusion coefficients. The whole population of one species is used to determine the fitness of the individual of the other species.

The fitness, f_i for a test image $i \in [1, n]$ is given by (1) and the fitness, g_j for a halftone filter $j \in [1, m]$ is given by (2), where $p \in [1, h \times w]$ is the index for image pixels, h = image height, w = image width, I_i is the low pass filtered original test image i, and $\dot{I}_{i,j}$ is the corresponding halftoned image i, filtered by the j:th filter.

$$f_i = \sum_{j=1}^{m} F_{i,j} \quad (1) \qquad g_j = \sum_{i=1}^{n} F_{i,j} \quad (2) \qquad F_{i,j} = \sum_{p=1}^{h \times w} \left| I_i(p) - \dot{I}_{i,j}(p) \right| \quad (3)$$

The best test image i is the one that is "hardest" to halftone by the population of halftone filters, thus we try to maximize the values gotten by equation (1), while the best halftone filter j is the one that has the lowest value for equation (2). Both these formulae use (3) to define the difference with one image i or filter j.

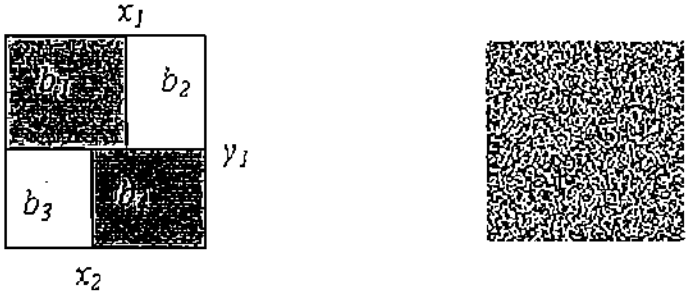


Fig. 1. a) Background segments. b) Chaotic data to be added.

Genetic algorithm generates new test images and halftone filters by using crossover and mutation, favoring those parent chromosomes that previously have gotten the best fitness values.

Test images were combined from such parameters, as position, size and color of elementary graphical objects, which include lines, rectangles, circles, ASCII characters, and background, all their data was encoded to a GA chromosome consisting of 79 integers.

The first seven parameters were for background, three of them (x_1, x_2, and y_1 are shown in Figure 1a) break the background into four segments and the rest (b_1, b_2, b_3, and b_4) determine the tone of these segments. The next 70 parameters were further divided into 10 groups of 7 parameters, each group defining one elementary image pattern as follows:

1. Pattern type (line, rectangle, oval, ASCII character), for characters also the font style.
2. Tone.
3-4. x and y coordinates of the reference point.
5-6. Length in x and y coordinate directions or character font size and type.
7. Character value or void (only printable ASCII characters were used).

All patterns are opaque and may cover patterns created earlier; background is created first and then the other patterns on top of it. The resulting images mostly lack fine details *i.e.* have low gradient value. Real images tend to have more variation between neighboring pixels. This in mind we added noise to the images (see fig. 1b) with the Verhulst[31] logistic equation:

$$x_{n+1}=\mu \times x_n \times (1-x_n) \tag{4}$$

This kind of chaotic data rather than pure white noise was used in order to generate a potentially rich set of repeatable patterns. The last two parameters of the chromosome consist of a 16-bit value μ that was further scaled to be a decimal number in the range [2, 4]. The optimization process usually favored such parameters that generated streaked patterns rather than white noise (fig. 1b). The size of the generated image was 256 × 256 pixels, so that the values of most parameters would fit into eight bits. Some intermediate test runs were run by using $n \times n$ images, where $n \in \{32, 64, 100\}$, in order to speed up the testing.

The reason for generating relatively simple test images is that we think that the generated test images should be easy to interpret. If we use complex images it is hard to analyze the results or see what characteristics actually cause the perceptible differences. By using simple image primitives we wanted to make it more likely that we can find the primary cause for the differences in the operation of filters.

Two different halftoning filter models were used: threshold matrices and error diffusion coefficients[14]. Threshold matrices of size 16 ×16 elements were used containing all possible integer threshold values in the range [0, 255]. When we optimized threshold matrices, we used a special permutation array in order to be able to utilize genetic operations, such as crossovers and mutations without losing any of threshold values (see ref. 8). When optimizing error diffusion coefficients we use the error diffusion matrix (fig. 2) where the weight coefficients w_i are optimized by a floating point coded GA that can have values in the range [0, 1]. These weights are further normalized by dividing by the sum of all weights $\sum_{i}^{24} w_i$.

The initial population of test images is generated randomly, while the initial halftone filter population is generated either randomly or filled by using given solutions. With the error diffusion method the given solutions include: Floyd-Steinberg, Jarvis-Judice-Ninke, Stucki, Fan, Shiau-Fan, Park-Kang-King, Stevenson-Arce, Eschbach, Wong-Allebach, and two weight error diffusion coefficients (ref. 14, chapter 16). With the threshold matrix method the given solutions include: ordered threshold matrix[14], mirror and rotated versions of it, and also the threshold matrices we previously[8] generated by GA.

$$\begin{array}{ccccccc} & & & \bullet & w_1 & w_2 & w_3 \\ w_4 & w_5 & w_6 & w_7 & w_8 & w_9 & w_{10} \\ w_{11} & w_{12} & w_{13} & w_{14} & w_{15} & w_{16} & w_{17} \\ w_{18} & w_{19} & w_{20} & w_{21} & w_{22} & w_{23} & w_{24} \end{array}$$

Fig. 2. Error diffusion matrix. Dot (•) represents the pixel to be evaluated, and w_i:s the pixels, where the rounding error is spread.

Table 1. The basic GA parameters used.

GA parameter	Test image	Error diffusion halftone filter
Population size	30	30
Elitism	50%	50%
Crossover rate	50%	50%
Crossover type	Uniform, single point, and arithmetic	Uniform, single point and arithmetic
Mutation rate	2%	2%
Generations	100-500	100-500
Length of chromosome	79 integer numbers	24 floating point numbers

Table 1 gives the basic GA parameters used. The length of the test runs ranged from 100 to 500 generations, but those reported here are from 100 generation long test runs. The length of the parameter array of the error diffusion filter was 24 floating-point numbers in the range [0, 1]. The size of the permutation array of the threshold matrix filters was 256. Single point and uniform crossovers were used; in addition, arithmetic crossover was applied at a rate of 10%. The mutation probability was 2%.

Every image is evaluated against all filters; this means that the time demand will increase as the square of the population size. It also means that very dynamic populations may cause fluctuation, therefore the high tendency to elitism was selected to bring more stability to the populations. The small population size was resorted to because of the high time demand for running these tests. However, we still wanted to use a relatively large number of generations, because of the slow processing that we observed in a previous experience with threshold matrix optimization[8]. There we needed a rather high number of generations until any decent results were attained.

3 Experimental results

The results were generated by running five test runs with two different dithering methods, threshold matrix and error diffusion, and two different ways to initialize the population, random and given. The primary goal was not to find the best possible results after several test runs, but rather to see whether or not the results have some similarities in common between test runs, *i.e.* to test the stability of the proposed method.

The results with the threshold matrix showed that they are difficult to optimize. To exceed the performance of the well-known ordered threshold matrix is difficult, if not impossible. Good error diffusion coefficients are easier to generate. The results with random initial populations did not evolve too well; many generations were required in order to reach a reasonable fitness and the performance of the reference halftone filters of our experiments was not reached. Therefore,

the following results contain only data generated by the error diffusion method and using a given initial population.

The results represented here were generated by using the peak average error[20] (PAE) quality metric (5) after low pass filtering the images with MTF[19], with the parameters: size 16×16, viewing distance 256 mm, and print quality 300 dpi. PAE was chosen because we wanted to find maximum errors.

$$PAE = \underset{i,j}{MAX}\left|I_{i,j}(p) - \dot{I}_{i,j}(p)\right| \tag{5}$$

However the other metrics mentioned in 1.4 were used in other test runs, and the main results with them were not significantly different.

3.1 Results with error diffusion

An example of a typical fitness curve is shown in Figure 3, for the case when optimizing test images and error diffusion filters are done concurrently, and the initial population is given. Despite the fact that we are trying to minimize the filter fitness values while maximizing the test image fitness values, both of them tend to increase during optimization. The initial filter population contains some random filters that cause the image fitness to be high at the beginning, until these bad filters are eliminated from the population.

The fitness of test images show large fluctuations; extremely unfit image filter(s) born in the filter population was the reason for them. Most of the filters in the population are relatively well behaving after a few generations. If the genetic operations create some totally unfit filters in that state, it causes the fitness values in the image population grow rapidly. These bad filters and their possible offspring are killed relatively soon and replaced by new better individuals, and the fitness values in the test image population decrease back to the longer time span trend level. The diversity (right axis in fig. 3) of the population seems to decrease quite rapidly, and both species therefore become nearly, but not totally, uniform.

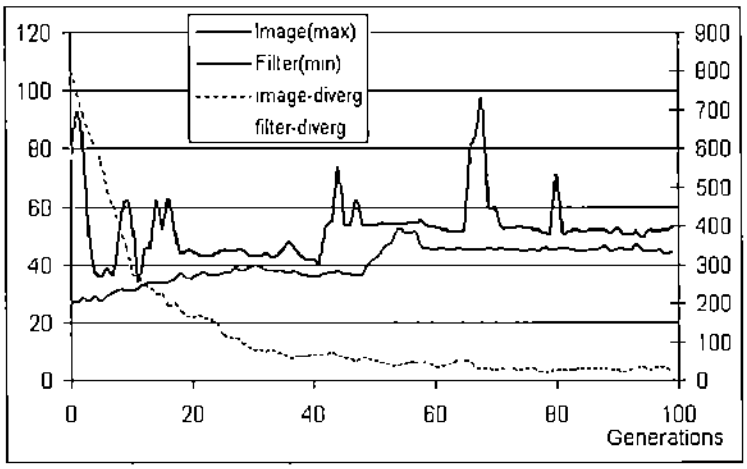


Fig. 3. The development of the best image and filter fitness values as a function of generations. The initial population consists of given error diffusion matrices. The unit of X-axis is the average *PAE* value; the unit of Y-axis is the sum of divergence between the individuals in the population.

For comparison, if test image or filter generation is done without co-evolution, we usually see graphs shown in Figure 4. Test image optimization (4a) shows logarithmic growth that is typical for the GA optimization. However, the filter fitness somewhat changes too, because its fitness is linked to the static image set in the same way as in our co-evolutionary optimization. The filter optimization fitness curve (4b) shows a slow downward logarithmic tendency. However, the original given filter population is so good, that there seems to be relatively limited possibilities for any improvement.

The image set is now static, however, the fitness values of the images still show large fluctuations. The large peaks in the Figure 4b are caused by the same phenomena as in fig. 3. A bad filter created in the filter population causes a high fitness value for some test images. This happens because the image fitness's in this experiment depend on filter fitness's in the same way as in the co-evolutionary experiment. Computationally the co-evolutionary optimization is about as expensive as optimizing test images against a static filter set, since it requires an equal number of fitness evaluations.

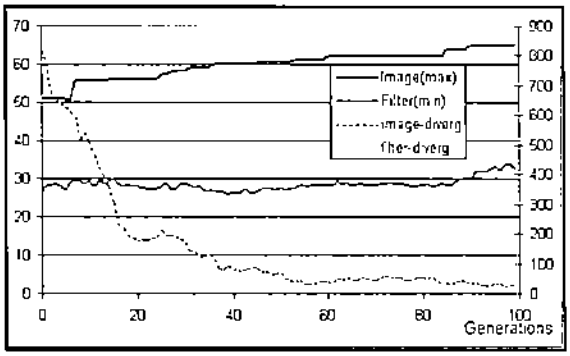


Fig. 4a. The development of the best test image fitness as a function of generations, without co-evolution.

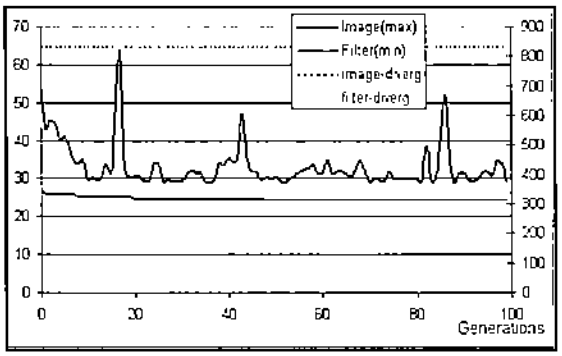


Fig. 4b. The development of the best filter fitness as a function of generations, without co-evolution.

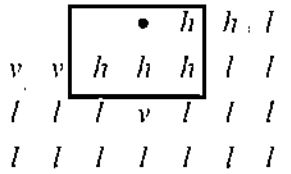
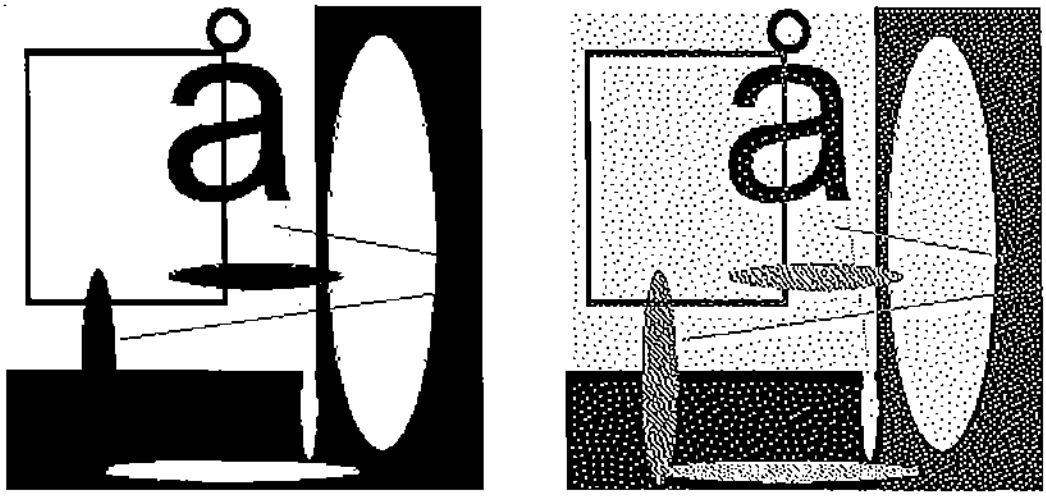


Fig. 5. The outline of the contents of the generated error diffusion matrices. *h*=high value, *l*=low value, *v*=varying value.

The results indicate that after each test run, some coefficients of the error diffusion filter always have proportionally higher values (see fig. 5), whereas some others always have lower values, while still some others vary from small to high. The resulting filters usually have non-zero coefficients only in the immediate surroundings (shown □ in fig. 5) of the pixel to be processed, or less frequently further away (shown ▯ fig. 5) from it. However, the resulting best coefficients were never exactly equal to any of the initial error diffusion matrices. Thus the system is able to generate coefficients that it regards better for halftoning the generated test images than those initial filters taken from book 14.

Figure 6 shows a typical test image generated and the corresponding halftoned image. The test images had usually more objects and borders between them when compared to the results

given in ref. 4. The reason for this difference is likely the fact that the error diffusion method usually responds to gray tone changes after a short delay, and therefore the largest differences between images are caused by high contrast changes.



a) b)

Fig. 6. An example of a test image created when the initial filter population is filled with given error diffusion coefficients. a) A generated test image. b) Image a dithered with a generated error diffusion matrix.

3.2 Discussion

If we optimize the filter against some static test image set, there is no guarantee that the optimized filter is good for filtering other images. The test set could be somewhat limited, and problematic images for the resulted filter might have features that were not present in the test image set. The corresponding phenomena might occur, if we optimize test images against some static filter set. Usually the benchmark test set is planned so that all possibilities are taken into account. However, it seems that there is no benchmark test image set for image processing. At least not anything that is a result of comprehensive study instead of common practice.

The possible advantage of using co-evolution is considered to be achieved by optimizing image filters and test images against each other, i.e. against dynamic and evolving benchmark set. This way both test sets can evolve and discover weaknesses in the other. However, it seems that during the optimization these two populations reach some balanced situation. The question is, are both sets in optimal state in this balanced situation, or is this balanced situation some compromise where both sets are far from optimum.

We performed experiments without co-evolution, and came to the conclusion that the optimal filter is different for different image types. When we optimized filters against the standard test image set it works well with them, but poorly with our GA generated test images, and *vice versa*. The same observation was done considering test images: the generated test images got higher fitness values against the well-known error diffusion coefficient than standard test images.

These findings seem to suggest that the balanced situation after co-evolution is some kind of compromise, where either set cannot improve much. The optimization starts to circle around, the changes in one species are forced back by the other species. However, if we freeze the one

species in this balanced set and only optimize the other, the species that is still able to evolve changes a little bit, but not much. Thus, a little bit better filters for that frozen population of test images, or *vice versa* can be generated starting from that balanced situation. The conclusion is that the balanced situation does not represent the best filters for the subset of natural images, nor the subset of GA generated images. Neither does it hold the subset of best filters for either image subset.

However, the balanced situation represents the compromise between the hardest test image and the best image filter in the subsets that our coding of test images and filters enables from all the possible images or filters. Therefore these results are mainly important for developing filters and test images to be used with some predefined type of images. If we know the image types our image processing software is to be used with, and the filter types we can implement, and are able to code these image and filter subsets as GA chromosomes, then this method is applicable to evolve filters and their test images inside these subsets.

3.3 Error seeding

The main goal of this research was to apply the proposed method to find weaknesses in image processing software by generating test images. Error seeding is a well-known method in software testing. It means that some specific errors are seeded deliberately into the software before tests. The goal is to find these seeded errors with the applied testing method. The other goal of this approach is to statistically estimate the amounts of real errors in the software by measuring how many of the seeded errors were found. In order to be able to do that the seeded errors should be realistic and similar to those expected to be present in the tested software.

In order to see if our co-evolutionary method applied to this kind of software testing is able to reveal faults, we inserted several faults to our halftoning software. These seeded errors were such that they caused distortions during the image processing, if some GA generated image or filter parameter was within some predefined range. The only feedback that GA gets from the halftone processing is the image similarity value. These errors were not observed from any of the parameter values, but only from the image similarity values.

In total 30 different errors were inserted, 10 of them were caused by image parameters, 10 by filter parameters, and 10 by a combination of image and filter parameters present concurrently, when halftoning one test image against one halftoning filter. These errors include e.g. using slightly different low pass filter for images to be compared. The amount of difference (e.g. difference in n and d_0 values of the low pass filter model[19, 32]) was dependent on other parameters than the one that causes this error situation. Other errors include shifting image pixels to a given direction or changing gray tones in part of the image. The size of the shift or area and the amount of tone difference was dependent on other image or filter parameters than the one that triggered the error.

The frequency of randomly encountering inserted erroneous parameter values was relatively high, from 2 up to 100 times in one test run of the GA. The reason to use errors that had a high probability of being found was that GA is not capable of finding any single error with any higher probability than random search. The power of GA is to combine the problematic parameter combinations and amplify the severity of detected error. Therefore the probability of finding error was high, but the severity of random error was not so high. The severity of error was scalable by discovering and changing the other affecting parameters.

The errors were designed so that just finding them does not cause maximal distortion. However, the amount of error is dependent on several parameters, so the GA can learn those parameters and amplify the amount and scope of these distortions as much as possible.

According to our original expectations distortion in the halftoned image is the error type that the co-evolution method should reveal. If some image characteristics noticeably change in the halftoning process, it suggests that the tested image-processing algorithm has weak points.

The results get with the error seeding tests were encouraging; all the seeded errors were found at least in some of the test series. The most frequently found error was the eventually the worst case over half of the test runs. Usually, the severity of the error was optimized by the GA to be nearly maximally high. The fact that so many test runs catch and grip the error cases underlined the power of the proposed approach. After these error-seeding tests we could be more convinced about the proposed method and its capability to find the weaknesses of the tested real image processing system.

3.4 Fitness landscape

It is obvious that the fitness landscapes have a profound effect on the optimization efficiency. In order to evaluate if our co-evolutionary approach is beneficial with this respect, we did some analysis of the fitness landscape. Random walks were performed by starting from the optimized test image, keeping the optimized error diffusion filter static, and doing small random mutations to the test image by changing one of the 79 image parameters either to totally random value, or max. 10% from the current value. Correspondingly random walks were performed starting from the optimized error diffusion filter, by keeping the optimized test image static, and then performing similar random mutations to the image filter. The purpose of these random walks was to reveal the amount of steps needed to move from the optimal hill down to a more normal fitness landscape.

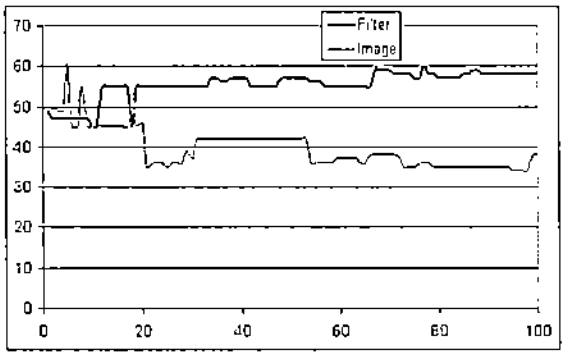


Fig. 7a. Random walks in the fitness landscape, either test image or error diffusion filter is changed.

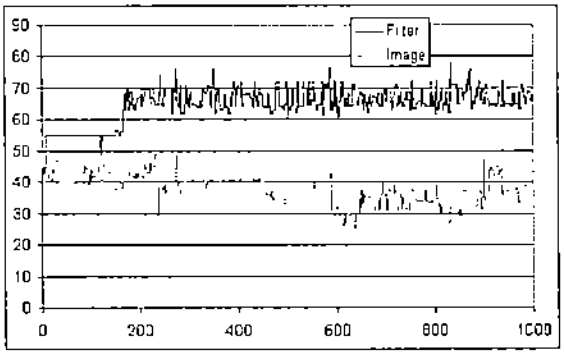


Fig. 7b. Longer random walks in the fitness landscape.

The random walks revealed that only a couple of random mutations were needed to move the optimized test image out of the optimal area. Also the filter fitness rapidly changes to worse, but it still requires more steps in order to reach the totally random landscape.

Figure 7 represents random walks in the fitness landscape. The image fitness starts to decrease rapidly when moving out from optimum, however, a longer random walk shows that every now and then there appears test images of which fitness is almost as high as that of the optimized one. The filter fitness value seems to increase quite far from the optimum, when walked further. A strange phenomenon is that both image and filter values seemed to get better fitness than optimized at the beginning of the shorter random walk. This implies that neither test image nor filter has reached the optimum in the co-evolutionary search, because the random mutation was able to improve both after a couple of random mutations, before the random walk departed from the optimal area. This further shows that the balanced situation reached in the co-evolutionary optimization is not a global optimum for both species.

4 Conclusions

The experiments done seem to indicate that the co-evolutionary method is able on the one hand to detect potential errors in the tested system and on the other hand it can evolve a better methods accordingly. However, in this study the steady-state situation after co-evolution was not a global optimum for the subset of images or filters. The steady state represents a compromise between the subsets of representable test images and filters. Thus, this method can be applied by defining accurately the image and filter type used and generating compromise filters for that image subset. More experiments with redefined image representation model are needed.

In this case the potentially erroneous or at least low quality filters are replaced by better ones while co-evolution creates more challenging test images for these new filters. The eventually the best filter is better than any of the well-known error diffusion filters at least for the test images generated. This can be stated since the optimal filter was never any of the good filters given in the initial population. It seems that co-evolution generates hard test images for error diffusion and then coefficients that halftone them better than those represented in the literature. If one of the known error diffusion filters would be the best possible for the subset, that image model representation allows, it would have been the final solution for some or all of the optimization runs.

4.1 Discussion

The degeneration, i.e. the premature convergence, of the population may be a problem in this kind of co-evolving system. If the test image population degenerates, the filters may turn out to be good only for that type of image and vice versa, if the filter population is too degenerated the test images may be challenging for only that filter type. It is thus of primary importance to take note the diversity of the co-evolving populations. This also applies to ordinary GAs: diversity loss is a sign of a potential premature convergence. Some problems that we had with population size and elitism in our GA implementation might be less restricting when using steady-state GA. The tests whether steady-state CEC would be a better approach for this problem will be done.

This work is not necessarily made in order to put Lena and other standard test images into retirement, but in order to find methods to test image-processing systems with more challenging case sensitive test images. These images may reveal some pitfalls in the systems under test that usual test images do not. This kind of co-evolution could be applied for developing also other image processing features.

One possibility is that in the future we apply parallel computing in order to enable a larger population size. We have considered evaluation against some subset, but that is also left for future work. We are also considering the use of genetic programming to generate test images, and using more realistic image primitives in the image generation process. Multiobjective Pareto optimization by GA might be applied to this problem in the future.

References

1. Standard test images <http://jiu.sourceforge.net/testimages/index.html>
2. Computer vision test images <http://www-2.cs.cmu.edu/~cil/v-images.html>
3. IEEE Xplore <http://ieeexplore.ieee.org/Xplore/DynWel.jsp>
4. Science direct <http://www.sciencedirect.com/>
5. NEC Research Institute ResearchIndex <http://citeseer.nj.nec.com/>
6. J. T. Alander, T. Mantere, G. Moghadampour and J. Matila, Searching protection relay response time extremes using genetic algorithm – software quality by optimization, *Electric Power Systems Research* **46**, pp. 229-233, 1998.
7. Mantere, T., and J. T. Alander. Developing and testing structural light vision software by co-evolutionary genetic algorithm. In *QSSE 2002 The Proceedings of the Second ASERC Workshop on Quantative and Soft Computing based Software Engineering*, 18-20. Feb. 2002, Banff, Alberta, Canada. Alberta Software Engineering Research Consortium (ASERC) and the Department of Electrical and Computer Engineering, University of Alberta, pp. 31-37, 2002.
8. J. T. Alander, T. Mantere, and T. Pyylampi, Threshold matrix generation for digital halftoning by genetic algorithm optimization, in D. Casasent, ed., *Intelligent Systems and Advanced Manufacturing: Intelligent Robots and Computer Vision XVII: Algorithms, Techniques, and Active Vision, Vol. SPIE-3522*, Boston, MA, 1-6. Nov. 1998. SPIE, pp. 204-212, 1998.
9. T. Mantere, and J. T. Alander, Automatic test image generation by genetic algorithms for testing halftoning methods, in D. Casasent, ed., *Intelligent Systems and Advanced Manufacturing: Intelligent Robots and Computer Vision XIX: Algorithms, Techniques, and Active Vision, volume SPIE-4197*, Boston, MA, 5.-8. Nov. 2000, SPIE, Bellingham, Washington, pp. 297-308, 2000.
10. J. Holland, *Adaptation in Natural and Artificial Systems*, University of Michigan Press, Ann Arbor, MI, Reissued by The MIT Press, 1992.
11. J. Koza, Genetic evolution and co-evolution of computer programs, in Langton et al., eds., *Artificial Life II, Proceedings of the Workshop on Artificial Life, Feb. 1990, Santa Fe, NM, vol. X*, Santa Fe Institute Studies in the Sciences of Complexity, Addison-Wesley Reading, MA, pp. 603-629, 1992.
12. M. Michell, *An Introduction to Genetic Algorithms*, MIT Press, 1998.
13. E. Brodie III, and E. Brodie Jr., Predator-prey arms races, *BioScience* **49**, pp. 557-568, July 1999.
14. H. R. Kang, *Digital Color Halftoning*, SPIE Optical Engineering Press, Bellingham, Washington, & IEEE Press, New York, 1999.
15. P. G. J. Barten, *Contrast Sensitivity of the Human Eye and Its Effects on Image Quality*, SPIE Optical Engineering Press, Bellingham, Washington, USA, 1999.
16. F. Nilsson, Objective quality measures for halftoned images, *Optics, Image, Science and Vision* **16**, pp. 2151-2162, Sep. 1999.
17. P. Eklund, and F. Georgsson, Unraveling the thrill of metric image spaces, in G. Bertrand, M. Couprie, and L. Perroton, eds., *Discrete Geometry for Computer Imagery*, LNCS 1586, Springer, 1999.
18. H. Aguirre, K. Tanaka, T. Sugimura, and S. Oshita, Halftone Image Generation with Improved Multiobjective Genetic Algorithm, In Zitzler, Deb, Thiele, Coello, and Corne, eds, *First International Conference on Evolutionary Multi-Criterion Optimization*, Springer-Verlag. Lecture Notes in Computer Science No. 1993, pp 501-515, 2001.

19. J. Sullivan, L. Ray and R. Miller, Design of minimum visual modulation halftone patterns, *IEEE Transactions on Systems, Man, and Cybernetics* **21**, Jan./Feb. 1991.
20. A. K. Chan, *Design of a Wavelet-Based Software System for Telemedicine*, Texas A&M University, 2000. Available via www: < http://ee.tamu.edu/~akchan/Demo/Telemed.pdf >
21. S. Welstead, *Fractal and Wavelet Image Compression Techniques*, SPIE Press, Bellingham, WA, 1999.
22. J. Y. Goulermas, and P. Liatsis, Feature-based stereo matching via coevolution of epipolar subprograms, in *7th International Conference on Image Processing and its Applications* (Conf. Publ. No. 465) , Vol. 1, pp. 23-27, 1999.
23. M. Miyojim, and H. Cheng, Synthesized images for pattern recognition, *Pattern Recognition* **28**, pp. 595-610, 1995.
24. L. A. Forbes, and B. A. Draper, Inconsistencies in edge detector evaluation, in *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition, Vol 2.*, pp. 398-404, 2000.
25. O. Kocsis, L. Costaridou, G. Mandellos, D. Lymberopoulos, and G. Panayiotakis, Compression assessment based on medical image quality concepts using computer-generated test images, in *Computer Methods and Programs in Biomedicine*, article in press, 16. Sept. 2002, 11 pages.
26. K. Sims, Artificial evolution for computer graphics, *Computer Graphics (Siggraph '91 Proceedings)*, pp. 319-328, July 1991.
27. N. Kobayashi, and H. Saito, Halftoning technique using genetic algorithms, *Systems and Computers in Japan* **27**, pp. 89-97, Sep. 1996.
28. J. Newbern, and V. M. Bove, Jr., Generation of blue noise arrays using genetic algorithm, in B. E. Rogowitz and T. N. Pappas, eds., *Human Vision and Electronic Imaging II*, Vol. SPIE-3016, pp. 401-450, SPIE, Bellingham, San Jose, CA, 10.-13. Feb. 1997.
29. J. T. Alander, *An Indexed Bibliography of Genetic Algorithms in Optics and Image Processing*, Department of Information Technology and Production Economics. University of Vaasa. Report Series No. 94-1-OPTICS, 2000. Available via ftp: <ftp://garbo.uwasa.fi/cs/report94-1/gaOPTICSbib.ps.Z >
30. S. Pal, A. Ghosh, and M. Kundu, *Soft Computing for Image Processing*, Physica-Verlag, Heidelberg, New York, 1999.
31. P. S. Addison, *Fractals and Chaos: An Illustrated Course*, Bristol, Philadelphia, Institute of Physics Publishing, 1997.
32. R. C. Gonzales, and R. E. Woods, *Digital Image Processing, 2nd edition*, Addison-Wesley Publishing Company, New York, 1992.