# Improving Regression Test Code Coverage with Meta-heuristics

*Master Thesis*

*DRAFT – version of August 8, 2008*

Jodi van Oenen

# Improving Regression Test Code Coverage with Meta-heuristics

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Jodi van Oenen
born in Naarden, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

TNO Automotive Safety Solutions
(TASS)
Schoemakerstraat 97
Delft, the Netherlands
www.tass-safe.com

# Improving Regression Test Code Coverage with Meta-heuristics

Author: Jodi van Oenen
Student id: 9616761
Email: J.vanOenen@Student.TUDelft.nl

**Abstract**

Software testing is becoming more and more important because of ever growing and more complicated systems. Software has to be tested, to assure the quality; one of the current test methods is the regression test. A code base evolves as time goes by, because new features are added, bugs are solved, et cetera. Regression testing assesses to which extent changes in the software affect existing functionality. Old test cases are rerun after software updates have been committed, to see whether changes had an effect on the existing code base. All test cases combined cover a part of the code of the software; ideally, all code is tested to assure quality as much as possible.

TASS is a software company in the automotive industry, with a relatively new testing department. TASS would like to improve the code coverage of their test suites, in order to improve the regression testing. This work builds upon our previous literature study [17]; it explored how TASS develops and tests, how software can be tested, and how code coverage can be improved.

A methodology has been devised to generate test data to increase code coverage, utilizing a meta-heuristic: evolutionary testing. The algorithm has proven to be able to increase code coverage to the maximum achievable coverage in all test cases.

Thesis Committee:

| | |
|---|---|
| Chair: | Prof. Dr. Ir. A.J.C. van Gemund, Faculty EEMCS, TU Delft |
| University supervisor: | H.G. Gross, M.Sc., Ph.D., Faculty EEMCS, TU Delft |
| Company supervisor: | M.G.A. Tijssens, M.Sc., Ph.D., TASS |
| Committee Member: | Ir. B.R. Sodoyer, Faculty EEMCS, TU Delft |

# Acknowledgments

This report is the product of nine months of hard work. I have learned so much about software testing, the importance and difficulties of testing, genetic algorithms, evolutionary testing, and so much more. I could not have done all this work alone; therefore I would like to thank a number of people for their support.

First of all I would like to thank my supervisors, Gerd Gross and Martin Tijssens, for their guidance, support, and discussions throughout the entire project. Their knowledge of testing, genetic algorithms and writing a scientific report have greatly improved the quality of my work.

Of course I also want to thank TASS as a company, for giving me the opportunity to do my thesis there. The people from TASS have shown great interest in my work. A few of them have helped me with many things: I have interviewed them, asked them questions about the build and test environment, and much more: Anton (my LaTeX-hero!), Cees, and Jeroen. A special thanks goes to members of the test group, Arjen and Sander, for taking me into their group, helping me with many things, and laughing at me when I had another whacky idea.

I want to thank a number of people on a more personal note for their support. My girlfriend Marlous for just being there, putting up with me, and helping me through stressful times. My friend Vincent for always being there when I needed some distraction. And last, but certainly not least, my family. Without my parents I would not have been here. They have made it possible for me to go to Delft in the first place; it took ten long years (whoops), but I made it. Your support kept me going.

Jodi van Oenen
Delft, the Netherlands
August 8, 2008

# Contents

# List of Tables

# List of Figures

# Listings

# Chapter 1

# Introduction

Modern software programs are becoming increasingly complex, while the market and the users demand more and more quality. To assure a certain level of quality, *testing* is becoming ever more important. A special form of testing is the *regression test*: evolutionary development steps change the code of software, while the behavior of existing functionality must stay the same. In regression tests an input-output combination for the software is verified as correct behavior; every repetition of the test should yield the same result. When the result has changed, *regression* has occurred.

To guarantee the quality of software it is tested. As each line of code can contain a potential bug, it is common sense to attempt to *cover* as much of the code as possible. A measure is the so-called *code coverage*; a percentage indicating what part of the code is covered by one or more tests.

To increase the code coverage, new tests have to be designed. The process of designing tests to increase code coverage can be seen as a search problem: the tester searches for a set of test cases, which increases code coverage as much as possible. A search is a mathematical optimization; an algorithm can optimize towards the optimum, in this case being the maximum achievable code coverage. Search heuristics can be applied can be applied to the problem of finding the right test cases, where 'right' in this case means 'increasing code coverage as much as possible'.

This work takes place at TASS, a software company. This chapter will introduce TASS, and give a general summary of the conclusions from our literature study. The goals for this work will be set by posing a number of problem statements; finally, an outline for the rest of this work will be given.

## 1.1 TASS

TASS is a company in the automotive industry, and writes software for car crash simulations and crash test dummies. This software is called *Madymo*—an acronym for Mathematical Dynamic Models. Madymo is under constant development: new features, bug-fixes, et cetera.

TASS is an acronym for *TNO Automotive Safety Solutions*. TASS has two activities:

consultation and software development. TASS provides software for auto crash design, simulation and virtual testing: Madymo. Madymo development has started in the 1970's by TNO, and since a few years Madymo development is part of the activities of TASS. The software consists of two main components: the solver and the workspace. The solver employs rigid body mechanics and finite element methods for solving Newtonian equations of motion. The Madymo Workspace is built around the solver, and consists of a preprocessor (XMADgic), a postprocessor (MADPost) and several other GUI tools that support safety engineers in their task to design automotive safety systems.

**A Note on TASS Software**

The source of the Madymo solver and workspace applications is strictly confidential. Although we have been able to work with these sources, we can only publish our results; the parts of the source we have used cannot be listed or published.

## 1.2 Summary of Previous Work

In the preceding literature study we have explained a number of theoretical concepts on software development and testing, and TASS' development methods were linked to those concepts.

*Software Engineering* (SE) is much more than just programming a piece of software. It involves a complete life cycle, of which the actual implementation is just a phase, somewhere in the process. Software is a system, of which the structure is captured in an architecture: an abstract view on the system, showing the externally visible properties and the internal relations between components.

Design is documented, preferably in various documents [4]. For example: first a *Requirements Analysis Document*, then a *System Design Document*, and finally an *Object Design Document*. The level of abstraction decreases as the documents are written.

To assure a certain level of quality (*Quality Assurance*, QA), *software testing* is becoming more and more important [3]. Testing has a destructive approach towards software [19]: to ensure quality, first it has to be made certain that the software cannot be broken. Tests can be conducted in a number of ways: white box, black box and grey box; and static or dynamic. Various aspects can be tested, e.g., exploratory testing for bugs, regression testing, or temporal tests can be performed, searching for the *Worst Case Execution Time (WCET)* [18] et cetera.

Testing at TASS is performed by three departments: the solver group, the workspace group, and the testing group. Tests are performed on unit basis, on the GUI, and on exploratory basis. The number of unit tests is being increased to improve the code coverage, i.e., more code has to be tested by the automated regression tests.

To automatically generate test cases aiming at increasing code coverage, several algorithms exist. Meta-heuristics, and especially genetic or evolutionary algorithms have been presented as promising techniques.

## 1.3 Project Boundaries

Increasing the code coverage is one of the goals set by TASS. In order to achieve this new tests will have to be designed, which is a laborious job. This project takes place in the following problem context, with a number of statements.

### 1.3.1 Problem Context

The test department is a young department, and the Madymo code base is old and partially undocumented. Designing tests for covering code of which the function is not known—or only partially known—is hard and very costly. Code has to be analyzed, tests have to be designed, and even then it is unsure whether the designed test is adequate and correct. To eliminate part of the problem, TASS would like to have some form of automated test data generation, making it possible to generate a test or multiple tests at little effort, whilst increasing the code coverage as much as possible.

From the developer's point of view, the automatic generation of test cases immediately raises one critical issue. The number of test cases must be manageable: the developers at TASS do not want to be 'swamped' by a vast amount of test cases.

### 1.3.2 Problem Statement

To improve the coverage of the regression tests, *search-heuristics* are a very promising technique to automate the generation of test data for regression testing at TASS. Generated tests should improve the coverage as much as possible, with the least amount of extra effort as possible. The problem statement can be formulated as:

> *Can regression test code coverage be improved by genetic or evolutionary algorithms, in the context of TASS.*

The TASS context is an important part of the problem context, because of the possible consequences of automatic test case generation: how will this affect the testing and development processes?

The problem statement can be decomposed into a number of smaller problems. This work will focus on proving the concept, which means that a number of problems will not be solved. These will be mentioned for future work. Problems for this work are:

- Code will have to be *instrumented* to be able to receive feedback on achieved coverage. How will this instrumentation be constructed?
- The instrumentation will result in a *fitness* for the current input set. Current works aim at achieving full coverage; how will fitness values be determined when the goal is to achieve *extra* coverage?
- Can TASS benefit from such a method: can it be applied to Madymo Workspace applications?

Problems for future work are:

- The process of instrumentation will have to be automated to make it usable in a production environment. How can this process be automated?
- A genetic or evolutionary algorithm consists of a number of components, all of which can potentially be optimized for the specific problem at hand. What is the optimum composition of such an algorithm?

## 1.4 Outline

This work is organized as follows: Chapter 2 will briefly explore related topics and related work, as preparation for Chapter 3. There evolutionary algorithms will be explained in detail, describing how it can be utilized to generate test data; the chapter concludes with a proposed methodology on how we want to increase code coverage. Chapter 4 will then evaluate the proposed method by describing the performed experiments and their outcomes, as well as found difficulties. Finally Chapter 5 will reflect on the proposed methods with conclusions and future work, as well as indicate usability for TASS.

# Chapter 2

# Related Work

Before our method is introduced, a number of concepts have to be explored. In the literature study we have shortly introduced regression testing, code coverage, and meta-heuristics; this chapter will give a more thorough explanation of these concepts, linking them to each other.

## 2.1 Genetic Algorithms

A genetic algorithm (GA) is a search technique used in the computing field. It is able to generate approximate solutions for search or optimization problems. GA's have many fields of application, ranging from[1] electronic circuit design to game theory, scheduling, and molecular structure optimization.

A GA is a *meta-heuristic*: 'Meta' means *abstract*, and a 'heuristic' is a *search*. Where other (non meta-heuristic) methods operate on the search space itself, meta-heuristics abstract away from this search space. A GA is an abstract search: it abstracts away from the solutions it generates. It works on bit-strings, and the translation to meaningful entities (in the search space) is done outside of the GA.

The approach is very different from other search techniques in how it handles the search space: it attempts to combine different solutions to evolve better ones. It also applies mutations on the solutions to make sure as many solutions as possible are explored, as quickly and efficiently as possible.

For evolutionary strategies the search is solely based on mutation of solutions; genetic algorithms make use of recombination. Both techniques can also be combined [15, 20]: this paper will use the terms evolutionary algorithms and evolutionary testing, but this also contains the genetic algorithm-specific recombination.

Evolutionary and Genetic Algorithms (EA and GA) have a very strong analogy with evolution in nature. A collection of individuals—a population—produces offspring by combining genetic material. This genetic material can also change over time due to spontaneous mutation, thus introducing new genetic material in the population. An evolutionary or ge-

[1] For a longer list of examples, see the Wikipedia article on Genetic Algorithms.

netic algorithm utilizes the same scheme to evolve solutions. Evolution is based on the fitness of an individual, just as in biology; this fitness is calculated by a *fitness function*.

### 2.1.1 Fitness Function

An EA works on bit strings, without knowing what the bit string means. To assess the fitness of each individual a component has to be added to the EA: the fitness function. When an individual is to be assessed, its chromosome is passed to the *fitness function*. In this function the chromosome will be split into separate genes, which are then decoded to meaningful entities. These entities can then be passed to the problem on which the EA operates, resulting in a fitness value. How this value is determined depends on the problem at hand: the value is directly linked to the quality of the assessed individual. Better individuals get a higher fitness value.

## 2.2 Software Testing

Software is becoming ever more complex while at the time the market and users demand an ever higher level of quality. Software testing is a form of QA, and is becoming more important due to the risen importance of *Quality Assurance* (QA). Many aspects of the software can be tested, e.g. usability (functional or black box testing) and running times (temporal testing). GA's can play a role in software testing, e.g., a GA can be used to obtain the worst case execution time of the software under test.

When the source of a program is available, structural testing (also known as *white box testing*) can be performed. The knowledge of the code's internal workings is utilized to design new test cases. For example, a tester can design a test to follow a specific execution path through the software under test. Another approach is to try to test all code in the software or unit under test: maximizing *code coverage*.

*Evolutionary testing* (ET) is the process of testing software with use of an EA. The process which guides the execution of the EA and steers the search for full code coverage requires some modifications of the code, which then serves as both a marking and a steering mechanism.

### 2.2.1 Code Coverage

When a test is run, there is a specific path through the program which is followed. The lines of code which are executed by this test are *covered*. Every line potentially contains bugs; also a sequence of statements with a certain input vector can cause faulty behavior. All tests combined result in a subset of the code which is covered: this coverage is called the *code coverage*, a percentage which expresses the amount of code which has been covered by one or more tests.

In an ideal situation the achieved coverage is 100%; however, this is infeasible because of certain code constructs. Think of code which is never executed, such as dead code or exception handling code. Goals can and are being set by companies—such as TASS—on

what percentage of the code should be covered at a certain point in time, e.g. for a release of the software.

### 2.2.2 Improving Code Coverage

Improving the code coverage is simple, in theory: as more test cases are designed, more code can be covered. For example, new test cases can be generated by randomly generating input parameters for the software under test. This is not an effective way however, because many test cases will follow the same execution path and very specific branches in the software will have a very low probability of being covered. The problem is how to *effectively* increase the code coverage; Feeding random new input to the software under test will most likely not increase coverage efficiently. When a certain part of the code has to be covered, some knowledge of the code is required to be able to design a test which may execute it.

This means that the manual designing of a test is a form of trial-and-error approach, which is expensive in terms of time and (thus) money. This method of testing can be seen as *random testing*: generating 'random' input to form a new test case and analyzing the effects of this test case. This is a form of black box testing: no knowledge of the code is used. White box methods have information about the code. Static (offline) white box methods such as equation solving and symbolic execution can be applied to derive input parameters for the code under test. In our literature study it has been indicated that these methods are hard to apply, because the C++ programming language—which is the language in which the Madymo Workspace applications are written—introduces difficulties: pointers, function pointers, and array subscripts are sometimes only known at runtime.

Dynamic methods involve running the software with a set of input parameters, and observing the results by some form of feedback from the code. When a specific code sequence—the 'target code'—has to be covered, specific input parameters have to be generated to actually cover that target. This is a search problem: a search has to be performed to find a combination of parameters which cover the target code.

When the test target is full coverage, this approach can be extended to a search for a *set* of test cases which achieves full code coverage. Because search problems can be mathematically solved by an optimization, it is possible to write an algorithm which searches for such a set.

A number of algorithms are used for the generation of test data. For example local search methods such as hill climbing, and meta-heuristics like simulated annealing and genetic (or evolutionary) algorithms. McMinn [15] performs a survey on these algorithms in the context of automatically generating test data to improve code coverage. Sthamer [21] has proven that a GA is capable of generating test data, achieving full or very high code coverage. Sthamer *et al.* in [22] have extended these results to various testing techniques. Wegener *et al.* in [28, 29, 30] confirmed that a GA is a very promising technique to automatically generate test data for full code coverage. Tracey [24] compared random testing, genetic algorithms, simulated annealing, and hill climbing; his work shows that meta-heuristics are the most reliable type of algorithms for this problem—genetic algorithms in particular.

Current work has aimed at achieving full code coverage of a 'new' piece of code. In this

thesis, we are going to use a GA for the generation of extra test cases: a company like TASS already has test suites, partly covering the code. The extra test cases should then *increase* the code coverage.

### 2.2.3 Keeping Test Suites Manageable

Although current work has already researched the general possibilities of utilizing a GA to achieve full coverage, TASS is afraid that such an approach will simply generate too much tests. A GA can generate tests for the code, regardless of what parts of the code have already been covered by current tests.

In an attempt to keep the test suites at a manageable size, TASS would like to be able to extend the current test suites with a limited amount of tests, which increase the code coverage. This project will therefore aim at utilizing a GA to generate extra test cases which increase code coverage as much as possible, and not to generate a set of tests which cover the complete code.

## 2.3 Regression Testing

During the development of a software package, the code changes over time. Methods or entire classes are rewritten, functionality is added or removed, code is optimized, bugs are fixed, et cetera. Although the code changes, the existing functionality often has to stay the same. This can be tested by feeding standardized input to a piece of software and analyzing its output; the output is then compared to a reference output or *test oracle*, which has been verified to be correct. If the output changes this is a potential bug: maybe the change was intended, but it is a signal that a bug may have been introduce. In the latter case *regression* has occurred in the software.

This method of testing is called *regression testing*. For example, the TMap [12, 13] testing methodology—which TASS uses—makes use of regression testing. Regression testing can take place on several levels; it can be used for unit testing as well as integration or GUI testing. Unit tests will result in a certain numerical output or a file, whereas GUI testing checks the interface for certain elements or contents of (output) fields.

Regression testing can be used in an automated testing framework; a process can run all tests and verify all outputs of these tests, resulting in an overall testing report. The process at TASS is very similar: all regression tests are run every night, and code coverage is analyzed. The manager of the TASS test group would like to increase the regression test code coverage. As mentioned in the previous section, we are going to use a GA to achieve this.

# Chapter 3

# Methodology

The previous chapters have introduced the context of this project, and gave an indication where a solution for the automatic generation of test data can be found: genetic algorithms. This chapter will give background information, genetic algorithms will be explained in detail, and how they can be applied to *improve* code coverage: GA's are already being utilized for achieving full code coverage from scratch, but we are interested in generating a manageable amount of tests. We want to take the existing tests and their achieved code coverage, and increase the code coverage.

## 3.1 Genetic and Evolutionary Algorithms

The concepts behind a genetic algorithm [9] will now be explained. What is the terminology, what steps are taken in the algorithm, and most importantly, how such an algorithm can be utilized in a software testing environment.

### 3.1.1 Terminology

As stated in the previous chapter, evolutionary algorithms have a very strong analogy with natural evolution. A possible solution is called an *individual* or *chromosome*, and the components of an individual are referred to as *genes*. A gene can have different values, which are known as *alleles*; the position of a gene in a sequence is called the *locus*. The solution's encoded form is the *genotype*, the decoded form is the *phenotype*.

The encoding is standardized for generic use [7, 21], i.e., it does not matter what the decoded form means, the encoded form is always in the same format: a single string of bits.

Evolutionary algorithms do not start at one single individual, they have an initial *population* of individuals. The algorithm progresses in rounds or *generations*, and individuals can be given a certain *fitness*: an objective, numerical value which makes the individuals comparable.

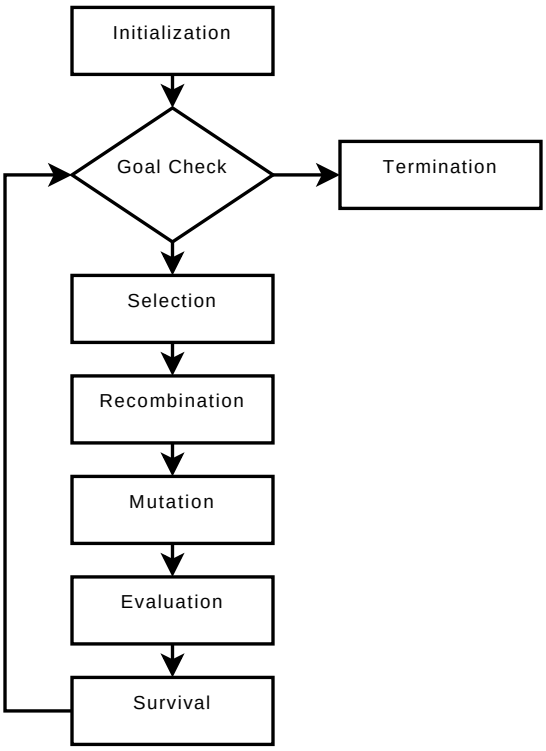



Figure 3.1: Steps in an evolutionary algorithm

### 3.1.2 Fitness Function Design

The fitness function is the link between the EA and the test subject. In this case, the test subject is the software under test. A fitness function has to be devised which calculates a fitness function based on the achieved code coverage. For example, the fitness can be expressed as the percentage of the code which has been covered by an individual; however, this approach leads to a scenario where the EA converges to seeking the longest execution path through the code base. This behavior is undesirable: it means that harder-to-reach code constructs will not be taken into account by the EA.

To make sure all code is covered, the code under test is divided into multiple targets, and all targets must be satisfied (covered) before the EA stops execution: when all targets are met, all code has been covered. How this is implemented is described in Section 3.2.1. The fitness can be determined on a per-target basis, and the EA works through the list of targets.

### 3.1.3 Procedure

An EA converges to a solution in *rounds*, or *generations*. Each round of an evolutionary algorithm (EA) consists of a number of phases [29], as shown in Figure 3.1. These phases all have their specific task:

**Initialization:** The algorithm starts with the generation of an initial population. This can be done at random, or it may be *seeded* by the user (or starting application): preselected individuals are injected into the initial population, to direct the search in a certain direction.

**Goal check:** The goal of the EA is to find a solution for its search problem. In this phase the algorithm checks whether an individual has been generated which satisfies the goal, *or* whether the maximum number of generations has been exceeded. If either of these two conditions are true, the EA goes to the Termination phase; otherwise the next round is started.

**Selection:** If a new round is started, the algorithm makes a selection of individuals from the available parents, which will "breed" the next generation. The selection is based on, for example, the fitness value: a better fitness value implies a larger probability of being selected.

**Recombination:** The selected individuals can now be recombined into new offspring. The genetic material of the parents is copied and recombined according to a chosen strategy (more details in Section 3.3.3) which results in two new individuals. The new individuals will form the offspring population.

**Mutation:** The new offspring is now subjected to a probability of mutation. This usually means flipping one or more bits, or incrementing or decrementing one of the decoded genes. Mutation can introduce new or reintroduce lost genetic material into the search.

**Evaluation:** The population of new individuals—the offspring—has to be evaluated. The evaluation results in a numerical fitness. This means running the software under test with all individuals in the current population, resulting in an objective fitness value for each individual.

**Survival:** This phase is not documented in all literature; however, in [28] the authors describe survival (which they call reinsertion) as the phase where it is determined which individuals are taken from the combined parent and the offspring populations to form the next generation's parent population. In this work the reinsertion phase will be considered part of the cycle.

**Termination:** In this phase the EA is stopped, and the results are published.

After a round the current population is replaced by the new population, being parts of the surviving parents and new offspring, and the cycle starts again. Evolutionary algorithms perform better with larger populations, because this leads to a better sampling accuracy of the search space [6]; however, larger populations lead to longer calculation times and increased memory usage, thus placing certain demands on the used hardware platform [7].

### 3.1.4 The Effects of Evolution

When an optimization problem is solved by an EA, the EA mimics evolution. It generates new individuals based upon the fitness values of the current individuals, and as generations pass the average fitness of the population will get higher: less fit individuals are discarded, and fitter individuals replace current weak individuals. The idea is that at some point in time this evolution will generate an individual with the maximum fitness value: this individual satisfies the goal of the EA.

This is equivalent to biological evolution: there is a population, and based on survival techniques (for example, "recombination of the fittest") the population evolves into a better population.

## 3.2 Evolutionary Testing

Evolutionary Testing (ET) is testing by means of an EA. This approach requires adaptation of both the EA and the software under test; the EA requires feedback from the software under test to determine fitness values for the individuals in its population. This feedback is generated by the software under test by means of extra function calls to standardized functions, which are added to the EA. The added function calls are so-called *instrumentation*. The instrumentation divides the code of the software under test into separate targets, as described above; these targets are called *partial aims*.

### 3.2.1 Partial Aims

An EA evolves as generations pass; the fitness value determines which individuals are allowed to reproduce and which not. To be able to use an EA for reaching full code coverage, the individuals need to be assessed on various criteria. This is because a single individual which covers all code, most likely does not exist.

To automate the search for the test data set, the test is divided into *partial aims*. Each partial aim represents a program structure that requires execution to achieve full coverage, for example a certain statement or branch [28, 22, 29]. An individual objective function is formulated for each partial aim: a separate optimization is performed for each partial aim, seeking for a test set which executes that partial aim. This mechanism of partial aims will be explained in more detail in the section on Code Instrumentation, Section 3.2.2.

### 3.2.2 Code Instrumentation

To guide the search instrumentation is used. Sthamer describes a form of instrumentation in [21]; the principles behind his ideas are used as a basis for this work. We have extended his approach by adapting the instrumentation to achieve *extra* code coverage, with manageability of the amount of tests in mind.

An EA works on a population, which evolves as generations are formed. To be able to compare individuals—being input sets for the software under test—a certain objective value has to be assigned to each individual: the fitness. The EA will perform branch testing because instrumentation is added on a per-branch basis, making it easier to instrument

code by hand. Branch testing implies that all statements in the instrumented branches are covered as well; full branch coverage thus implies full code coverage. There are a few exceptions, however; some expressions in code such as 'lazy and' and 'lazy or' evaluations [8] contain statements which are not always executed. Although this can be solved by using statement testing, this form of testing requires a much more complicated and elaborate form of instrumentation.

For branch instrumentation all branches are numbered and a few lines of instrumentation are added, making it possible to monitor execution. Every newly executed branch is marked as 'executed'. Every branch is a partial aim which has to be covered, and the EA works through all partial aims until either all partial aims are fulfilled or its execution limit has been reached (e.g. a fixed number of generations to be generated). When all partial aims have been satisfied, full branch coverage has been achieved.

Many forms of white box testing make reference to the *Control Flow Graph (CFG)* of the program in question [15]. A CFG describes a program $F$ as a directed graph $G = (N, E, s, e)$, where $N$ is a set of nodes, $E$ is a set of edges, and $s$ and $e$ respectively are the start and exit nodes of $F$. Every node $n \in N$ represents a statement in $F$, and every edge $(n_i, n_j) \in E$ represents the transfer of control from $n_i$ to $n_j$. From $G$ all possible paths through $F$ can be determined, making both statement and branch analysis possible.

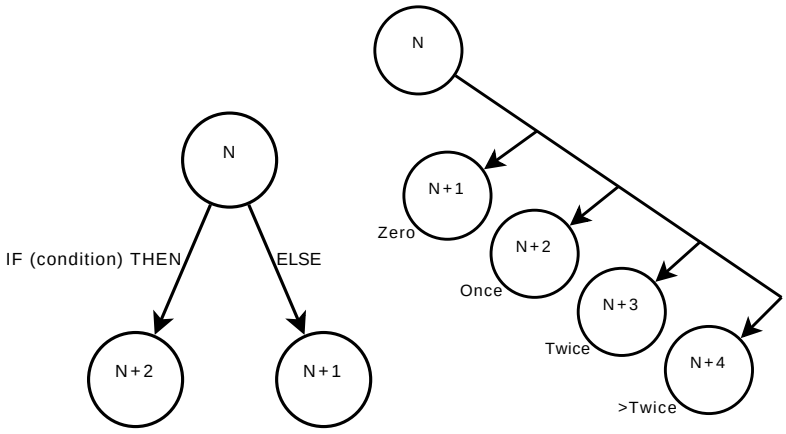
First of all, all individuals in the initial population have to be evaluated, marking a number of branches as 'executed' and producing a first partial aim to actively search for: this is a branch number which is assigned to a global variable, the *LOOKING* node. During the execution of the EA the LOOKING node holds the number of the node the EA currently is searching for—the current partial aim. When offspring is being evaluated, its genetic material is passed to the fitness function. This decodes the gene as appropriate, and starts executing the software under test with the decoded genetic material as input parameters. Execution now flows through the CFG of the software under test, leading to one of three possible scenarios:

1. The execution hits the LOOKING node: the current partial aim is fulfilled. This means that LOOKING has to be updated, and fitness values have to be recalculated. This is because the current fitness values are with respect to the *current* LOOKING node. Updating LOOKING thus implies updating *all* fitness values.

2. The execution misses the LOOKING node, but hits a direct sibling node; for example, take an `if(condition) then N+2 else N+1` block as shown in Figure 3.2a. If `condition` evaluates to true, the program goes from node $N$ to $N+2$, else to node $N+1$. Now, suppose LOOKING is $N+2$, but control flows to $N+1$. Fitness is now set to a high value, and the closer `condition` is to evaluating to true the higher the fitness of the current individual. How exactly the fitness is decided upon will be explained in Section 3.3.1.

3. The execution misses both LOOKING and LOOKING's direct sibling nodes. Fitness is set to a low value.

This scenario works fine for `if-then-else` blocks, `switch-case` blocks, et cetera. Loops form a special case.




(a) IF-statement (b) LOOP-statement

Figure 3.2: Control flow for two different code constructs

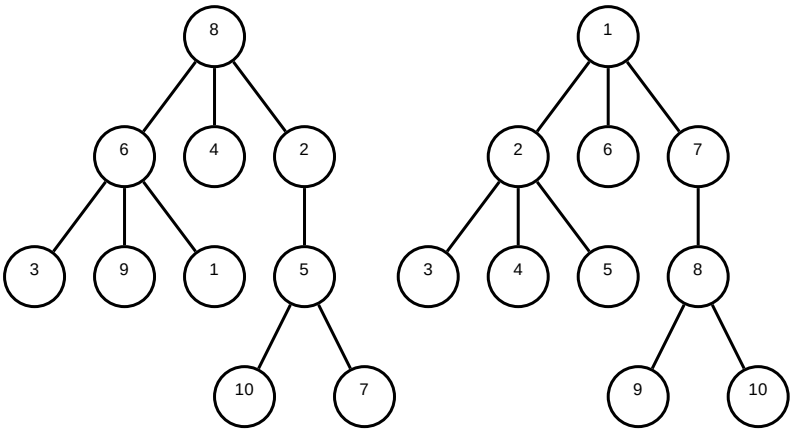
### 3.2.3 Loop Instrumentation

Because instrumentation is standardized to reuse code as much as possible, every loop is treated the same. Looking at a loop in an arbitrary program, it iterates for an unknown number of times: some loops only make one iterations, while others make thousands of iterations. Testing each possible amount of iterations for each loop requires an enormous amount of instrumentation. Therefore the principle of proof by induction is applied: a loop can be iterated zero, one, two, or more than two times [21]. If the software under test runs as expected in these four cases, the code in the loop is considered correct and "bug-free". Each of these cases can be seen as a separate node, which is shown in Figure 3.2b. So when node $N+1$ is the LOOKING node, it is marked as executed when the loop is iterated zero times, and when $N+2$ is the LOOKING node, it is marked when the loop is iterated one time, and so on. The fitness is now determined based on the amount of iterations actually made compared to the amount required by the current LOOKING node. This may seem as a strange course of actions, because this fitness doesn't rate the *meaning* of the current individual, but the actual number of iterations made by the individual. However, this is directly linked to the individual: as the number of iterations change, so does the fitness, and thus the individual. The EA converges to the highest possible fitness, being the required amount of iterations by the current LOOKING node.

After all loop-nodes have been executed, execution of the EA continues as normal; the next node can be both a 'normal' node, as well as a 'loop' node.

### 3.2.4 Branch Numbering

Previously it has been mentioned that branches (nodes) are numbered in the instrumentation. For this work the numbering of the branches is done by hand. This numbering serves two purposes: it makes all branches unique, but it also makes it possible to steer the EA.




(a) Random numbering (b) Depth First numbering

Figure 3.3: Different orderings of nodes in a tree

Suppose the nodes in the CFG are numbered as shown in Figure 3.3a. The EA starts the search by searching for an individual which executes the first node, node 1, which is set as the LOOKING node. Before the EA hits node 1, node 8 and node 6 need to be traversed; the fitness value however, is entirely based on the current LOOKING node, i.e. node 1. Nodes 6 and 8 have no knowledge of node 1, thus resulting in a fitness value zero. This means that the EA needs to evolve towards an individual which executes either node 3, node 9, or node 1. Nodes 3 and 9 are direct siblings and thus have knowledge of node 1, producing a fitness value higher as the predicates for execution of node 1 are closer together. When node 1 is finally executed, the EA sets the LOOKING node to the next node, node 2. This node is in a completely different branch; then node 3, back to the previous branch, et cetera. The EA has to behave as a random algorithm to execute all nodes.

The branch numbering scheme thus has an impact on the performance of the EA. In Figure 3.3b the nodes are numbered according to a depth first search; this scheme results in continuous guidance of the search. Starting the search at node 1, it always hits. Then node 2: execution either hits immediately, or nodes 6 or 7, which have knowledge of node 2—their direct sibling. Nodes 3, 4, and 5 follow the same scheme. Then node 6: it relies in a different branch than the previous node (5), but to execute node 5, node 2 is touched! Node 2 has knowledge of node 6, resulting in a fitness value.

The depth first numbering scheme ensures the calculation of a fitness greater than zero for at least some nodes, at all times. A wrong sequence can lead to random behavior, greatly impacting the overall performance of the EA.

## 3.3 Evolutionary Algorithm Application for Code Coverage

As shown in Figure 3.1, an EA consists of various phases. These phases can all be altered, changing the behavior of the EA. Although the optimization of an EA in the context of this

project is for future work, the EA which has been used is based upon Gross' [7] implementation. This section will show how the EA used for the experiments is set up, and most importantly, how the fitness function is implemented.

### 3.3.1 Fitness Function

An EA is entirely dependent on the effectiveness of the fitness function: a well-constructed fitness function can lead to much faster converging to a solution. This makes the fitness function the most important part of the algorithm, as the ability to steer the execution in an efficient way determines how fast the EA can converge towards generating a test case which covers the target code.

As an example, consider a block of code as shown in Listing 3.1. Lines 3 and 7 are added instrumentation, and LOOKING is set to 1: the true-branch of the IF statement. To reach this node, *a* has to be 0. The `checkBranch` call marks a branch as being executed; its parameter is the executed branch, branch 1 in this example. Branch 1 is a partial aim: for this branch, the EA has to be steered in such a way that input values lead to fulfilling the condition in line 2, `a == 0`. For this purpose, the `lookingBranch` call is added to the *sibling* node(s) of the target node. This function determines the fitness of the current individual, making it comparable to other individuals.

Looking at the `lookingBranch` call on line 7, three parameters are passed: the branch number for which the call is made (branch 1 in this example), and two predicates. These predicates are used for the calculation of the fitness. One of them is the current value, *a*, and the other is the target value, 0.

#### Local Distance

The *branch distance* [28], also known as *local distance* [29], is calculated in order to distinguish different individuals taking the same execution path. If the target node is missed at a certain branching condition, the local distance can be seen as the difference between the current value and the value to satisfy the branching condition. For example, if a branching condition is $x == y$ has to be evaluated as true to reach the target node, the objective function may be defined as $|x-y|$. If an individual obtains the local distance 0, a test datum is found which fulfills the branching condition: *x* and *y* have the same value [29].

Calculation of the local distance (fitness) is always the same: two predicates are used, whether they come from an IF statement, a WHILE loop, or any code construct at all: the fitness is always based on two predicates, say *x* and *y*:

$$F = \frac{1}{(|x-y|+\delta)^2}$$

This fitness function reaches its maximum value when *x* and *y* have the same value. The $\delta$ variable is introduced to make sure there is no division by zero; $\delta$ is an arbitrary small value, such as for example 0.01. The values of *x* and *y* can be used to steer the algorithm. In an `IF`-statement this can be used to indicate the distance to fulfilling the condition; in a loop statement this can be the amount of iterations made and the amount of required iterations (for the current LOOKING node).

```
1  int testMethod(int a) {
2      if(a == 0) {
3          checkBranch(1); // Instrumentation
4          /* regular (target) code */
5          return 1;
6      } else {
7          lookingBranch(1, a, 0); // Instrumentation
8          /* regular code */
9          return 0;
10     }
11 }
```

Listing 3.1: An example piece of code, to illustrate predicates

When the target branch is executed, the LOOKING node is set to the next node to be executed. Fitness values are recalculated to update them with respect to the changed LOOKING node, and the search for the new LOOKING node is started.

### Approximation Level

The fitness function can also be based on the *approximation level (AL)*. This can be combined with the local distance of the previous paragraphs. The AL provides a figure for an individual that gives the number of branching nodes lying between program structures covered by the individual and the desired program structure [29]. According to Wegener *et al.* [28] a group of nodes has a certain approximation level when each node group has the following characteristics:

- It has exactly one entry node
- It only contains nodes that are part of at least one path from the entry node to the target node
- It only has exits that miss the partial aim or lead to the entry node of the following node group
- It is minimal, i.e. further division of the node group is not possible without violating the other characteristics

In [29] the node groups close to the target have lower approximation levels than groups which are further away from the target; in [28] this is turned around, meaning node groups closer to the target have higher approximation levels.

### Hamming Distance

When a condition consists of more than two variables or the variables are not a primitive data type, it is not possible to fit these in the available predicates. The condition in Listing 3.2 on line 2 poses a problem. Three variables have to have the same value to reach the target code, while the `lookingBranch()` method can only take two input parameters. In Listing 3.3 on

```
1 /* regular code */
2 if(a == b && b == c) {
3     /* regular (target) code */
4 }
5 /* regular code */
```

Listing 3.2: A condition with more than two variables

```
1 /* regular code */
2 if(someComplexType.isInState(SOMESTATE)) {
3     /* regular (target) code */
4 }
5 /* regular code */
```

Listing 3.3: A condition with a complex data type

line 2 a complex data type is checked for some state; there is no 'simple' primitive to compare this variable to.

To enable the EA to solve these problems, another mechanism must be used. An alternative for—or addition to—the approximation level is the *Hamming distance* between the two predicates. The hamming distance measures the number of differing bits between two entities. In Listing 3.3 the `someComplexType` variable changes during execution of the software under test; the target code can be reached when the variable is in some state. The current variable can thus be compared to a variable of the same type, in the desired state: when these variables are translated to a string of bits, the target is reached when they are equal. Suppose the variables are 8 bits long. The current variable can be 1101 0011, while the target is (for example) 1100 1011. The hamming distance is the number of differing bits. In this example, this would be 2; when the EA generates an individual for which this distance becomes 0, it is in the desired state and the target code is executed.

Apart from dealing with complex equations and data types, the hamming distance has another advantage. When by mutation one bit is flipped for an individual, the effect on the actual value can be a change by orders of magnitude; the impact on the fitness value for this individual can be substantial. Think of, for example, an unsigned integer for which the most significant bit is flipped. The numerical value changes drastically, while only one single bit is changed. To abstract away from the mathematical meaning of the bit the hamming distance can be deployed here as well: compare the bit strings of the current and the target values, and utilize the hamming distance for the fitness value.

### 3.3.2 Selection Procedure

The selection procedure determines which individuals are selected to produce the next generation. A number of strategies can be applied; the most simple ones are *random selection* and *elite selection*. Random selection randomly selects an individual; elite selection involves picking the individual which has the highest fitness. Random selection is very unpredictable, and elite selection causes the biological equivalent of inbreeding: the same

| Parent Chromosome | $n$ | Offspring |
|---|---|---|
| 1101 0011 0111 1000 | 6 | 1101 0000 1010 1100 |
| 0010 0000 1010 1100 | 6 | 0010 0011 0111 1000 |
| 1101 0011 0111 1000 | 4,10 | 1100 0000 1111 1000 |
| 0010 0000 1010 1100 | 4,10 | 0011 0011 0010 1100 |

Table 3.1: Examples of recombinations

individual is selected for all offspring, causing this one individual to spread its genes through the whole population.

To work around this problem, other techniques have been devised [21]. Gross [7] uses 'tournament' selection: an individual is randomly selected from the population, after which a second individual is randomly selected. The one with the highest fitness wins. This can be done multiple times, or *rounds*; this way elite selection is prevented by introducing a random element, while at the same time the amount of rounds determines how elite the selection is.

### 3.3.3 Recombination Procedures

Recombination of two individuals is done on a genetic level, just as in biology. In an EA this means that recombination (sometimes called *crossover*) takes place on a bit level. An individual's chromosome (bit string) is taken and combined with another individual's chromosome. Recombination can be done at one point, *single crossover*, meaning that the two bit strings are split at a certain bit and genetic material is swapped. The first two rows of Table 3.1 illustrate a single crossover where the chromosomes are split at bit 6 ($n = 6$).

Double crossover can also be applied; the principle is the same as single crossover, but now two bits are selected for splitting the chromosomes. An example can be found in the third and fourth row of Table 3.1. The chromosomes are split and recombined at bits 4 and 10.

The actual positions where splitting takes place can be fixed or randomized, e.g. when single crossover is applied $n$ can be 4 for one set of parents, and 8 for another set of parents. The mechanism of single and double crossover can be extended to triple crossover, quadruple crossover, to $n$-point crossover. A special case is the uniform crossover: for every bit there is a probability $p$ that it will serve as a crossover point.

### 3.3.4 Mutation Procedures

Mutation ensures the introduction of new, or reintroduction of lost genetic material. It involves the flipping of one or more bits of a chromosome; this operation is based on a certain probability that mutation will actually take place. The amount of bits to be flipped differs; Sthamer [21] talks about 1 bit per chromosome on average.

### 3.3.5 Survival Procedures

In the survival phase it is determined which individual make it to the next generation. In nature often the principle of "survival of the fittest" holds; in an EA this yields a chance of ultimately leading to inbreeding. This method of survival, *elite survival*, selects only those individuals which have the highest fitness values—this is done for both parent and offspring populations. There are variations on this scheme, which Sthamer describes in [21].

## 3.4 Proposed Method to Increase TASS' Code Coverage

The experiments which other scientists conducted were fully focused on fully covering a code section. The situation at TASS is different; parts of the code are already covered by one or more tests. TASS is interested in achieving *extra* code coverage, i.e., generating test data which covers the currently uncovered code. Combining current test cases with the newly generated cases should yield full coverage. The reason TASS is interested in generating extra test cases—an EA can generate test cases to cover the entire code base—is that the testers and developers fear that the automatic generation of test data will lead to an enormous amount of new tests. They would like to keep the amount of tests within a reasonable limit; one way of achieving this is to take the current tests, and generate some extra cases. This will both increase the code coverage, while at the same time keeping the amount of tests in TASS' test suites at a manageable size.

Currently the TASS test group analyzes the code coverage by using the profiling options of the GCC compiler, and analyzing the resulting files with LCOV[1]. This can be reversed by indicating what has not yet been covered, and creating the instrumentation the EA needs. The EA can then generate a set of test cases which achieve the wanted full coverage.

The process of analyzing what code has not yet been covered and automatically generating instrumentation for the EA falls outside the scope of this work. This work will focus on proving the concept of achieving *extra* coverage: how should code be instrumented for this goal, and how an EA can be used in an effective way to improve coverage. Instrumentation will be added by hand, as well as starting runs and interpreting results. Future projects can aim at automating these tasks.

[1] LCOV is a graphical front-end for GCC's coverage testing tool GCOV, written by LTP (Linux Test Project). `http://ltp.sourceforge.net/`

# Chapter 4

# Evaluation

The previous chapter has described what the methodology will be to generate test cases. This chapter will describe the experiments performed to evaluate the concept of achieving extra coverage. First, general experiments will be performed to make sure the algorithm is functional, and results from experiments performed by other scientists can be reproduced. Second, the experiments will have the emphasis on the TASS-specific case: whereas the first experiments aim on achieving full coverage without any limitation, tests can now be focused on achieving *extra* coverage. Third, the experiments will focus on the Madymo Workspace code base. This means scaling up to larger pieces of code, with more complicated function arguments. This will give an indication of the usability of the proposed method: will this method of test data generation be usable in a production environment?

The EA will be compared to random testing (RT), for two reasons: RT forms a reference point (it is comparable to the current way of working at TASS), and RT can also show which branches in the software are the hardest to cover.

## 4.1 Initial Experiments: Verifying Evolutionary Testing

First of all a starting point has to be created. This will be based on existing work, with experiments to verify the results. The way of instrumenting code is based on Sthamer [21], and as an initial algorithm the one used by Gross [7] has been handed to the author. The EA is set up to utilize a tournament procedure, by default consisting of three rounds. Recombination is on a uniform basis, as well as mutation. Survival of individual is on an elite basis.

As probabilities for crossover and mutation the value 0.1 is chosen, to achieve a single-point crossover and single bit mutation per byte on average. Both parent and offspring populations are set to size 20, and the algorithm can perform no more than 50 000 generations.

In literature—[22, 29, 28, 30, 21]—a number of ‘standard’ test subjects are used. Two of these test programs will be used to test the algorithm and instrumentation as implemented for our EA: triangle classification, and the `atof` function.

```
1 void checkBranch(int b) {
2     if(!checkedBranches[b]) {
3         checked++;
4     }
5     checkedBranches[b]++;
6 }
```

Listing 4.1: The `checkBranch()` method for RT

### 4.1.1 Triangle Classification

Triangle classification is a small program, without loops. This will be the first test object; its source can be found in Appendix B.1. The program takes three integers as input, which represent the lengths of the three sides of a triangle. The program sorts the sides by length, and then classifies the triangle as either not a triangle, scalene, equiliteral, or isosceles. With RT the isosceles classification is expected to be the hardest branch to cover; this requires the random generation of three exactly identical numbers. The input has been restricted to three `char`'s, making the search space $256^3$ input combinations; using integers or floats took too long to finish a reasonable amount of test runs.

#### Random Testing

First of all, all branches are instrumented for random testing. This is only for branch-marking purposes; if a branch is executed, a counter for that branch is incremented. Every branch is given an extra call to the instrumentation-method, `checkBranch()`, shown in Listing 4.1. On line 2 a check is made whether the current branch has already been covered; if not, the number of checked branches is incremented. Finally—line 5—the counter for the current branch is incremented, to keep track of the number of times the current branch has been executed.

Random input data is generated until all branches have been executed at least once; execution is stopped when full branch coverage has been achieved. On a mathematical note, the chance of randomly picking three identical numbers is $1 \times \frac{1}{256} \times \frac{1}{256} = \frac{1}{65\,536}$; this means that on average there should be 65 536 runs before three identical numbers are randomly chosen in a search space of 256 elements per number.

Running the random test program for a 1 000 times with different seeding for each run gives an average of 65 464 runs before full coverage is achieved.

#### Evolutionary Testing

Now the evolutionary algorithm is put to the test. Again, each branch is instrumented with a call to `checkBranch()`; it is the same as in Listing 4.1, with the addition of some administration for the LOOKING node.

Extra instrumentation is added to actually guide the execution of the EA. This is done by adding calls to the `lookingBranch()` method, which is shown in Listing 4.2. This function is called with three arguments: the node number of the direct sibling, and two

```
1 void lookingBranch(int b, int prd1, int prd2) {
2     if (b == looking) {
3         fitness = (1.0 / pow((abs(prd1 - prd2) + 0.01), 2.0));
4     }
5 }
```

Listing 4.2: The `lookingBranch()` method for the EA

predicates. A fitness is calculated in line 3, but *only* when the sibling of the current node is the LOOKING node. At this point in the project the fitness is stored globally, because every branch calls `lookingBranch()` and only one of them (where `b` equals the LOOKING node) will calculate a fitness. Because the software under test, the triangle classification program, does not contain any loops there is no need for any extra loop instrumentation.

The EA evolves in generations, and each generation requires the evaluation of new offspring. Each evaluation requires the execution of the software under test; for comparison with RT the number of required *executions*—before full coverage is achieved—is counted. When 1 000 runs are performed, and average of 72 931 executions are needed for achieving full coverage. As Table 4.1 shows, evolutionary testing requires about 10% more test cases than random testing.

The EA is actually less efficient than random testing! This contradicts results achieved by Wegener *et al.* in [28] and [29], as their EA outperforms RT. Further investigation of the intermediate output shows that one particular branch is the hardest to cover; the branch where a isosceles triangle is detected, i.e., three identical inputs have to be generated. This branch is in a nested IF statement, which can be seen Figure 4.1. The LOOKING node is node 6. The problem is that the node which will calculate a fitness for node 6, being node 7, is in an ELSE IF construct. So to actually calculate a fitness value, first the condition for node 7 has to be fulfilled. The instrumentation in its simplest form leads to a situation where no fitness values are calculated. The EA will perform recombinations and mutations without guidance of a fitness value: the EA degenerates to a form of random testing. The EA needs to randomly generate (by means of the unguided recombination and mutation) an individual which fulfills the ELSE IF condition; then the EA can calculate a fitness for node 6, and converges to fulfilling its condition in only a few generations.

In an attempt to reproduce and verify the results, the triangle classification experiment was repeated; again 1 000 runs of the EA are performed, with another seeding of the random number generator. This time, the EA on average finished in 36 869 executions. In the first experiment the average was 72 931 runs; a factor two difference! Apparently, 1 000 runs is not enough to obtain a reliable average.Therefore, the experiment was repeated with 10 000 runs. On average, the EA now requires 41 851 executions; each run eventually converged to a solution. Researching the progressing average shows that it takes about 5 000 runs of the EA before the average number of executions stabilizes; this means that the performance of the EA in this form is very unpredictable.

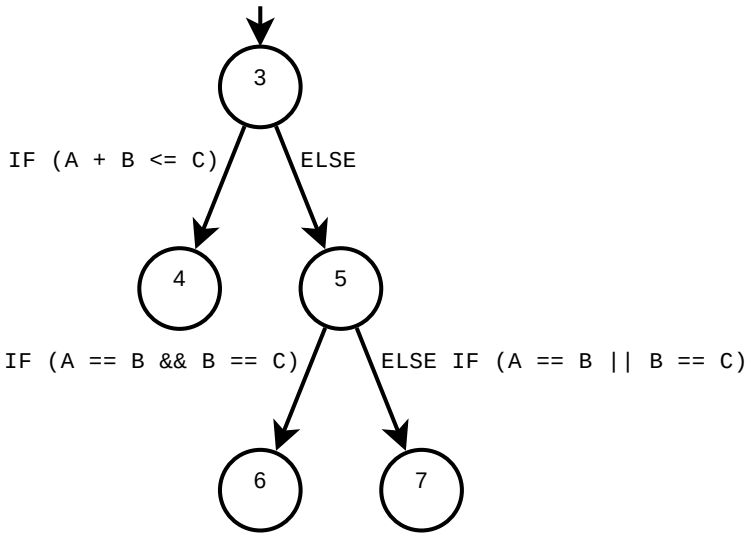



Figure 4.1: Nested IF statements in the triangle classification program

### Adding Extra Instrumentation

Investigating the code reveals a possible cause of the previous, rather strange result: the last branch to cover resides in a nested if statement. As the instrumentation only gives information on direct sibling nodes, the algorithm never receives feedback on certain input! Studying Baresel *et al.* in [2] shows that Baresel has found the same problem; the solution is as simple as it is effective: add extra instrumentation to parenting nodes. A parenting node can 'tell' the EA how to reach nested statements. The same holds for the example in Figure 4.1: a 'virtual' node can be added. Now there is an IF ... ELSE IF construct, which cannot determine fitness for most individuals; if an ELSE block is added, which 'catches' the individuals not fulfilling one of the two conditions, guidance for the EA towards the LOOKING node is provided.

The extra instrumentation is added, and the experiment is repeated: again 1 000 runs are performed, and the average number of executions is measured. This time the EA needs 1 789 executions to achieve full coverage: 36.6 times less than RT. However, again, the same branch as in the previous experiment was the hardest to cover. Investigating the output reveals that almost all effort goes into finding an input for this branch; the rest is covered by the initial input or during the search for this difficult branch.

### Hamming Distance

In an attempt to further improve the performance another speedup is sought. In the previous run the problem was that it was hard to push the three input variables into two predicates for the `lookingBranch()` function. The three variables are combined into one equation in an attempt to objectively compare them. The problem is that when mutation changes a single bit—a small change on chromosome level—the consequence can be that the numer-

| Testing with | Avg. no. of exec's | | | |
|---|---|---|---|---|
| Random Testing | 65 464 | | | |
| Evolutionary Testing | 72 931 | | | |
| Evolutionary Testing w/ extra instr. | | 1 789 | | |
| Evolutionary Testing w/ Hamming dist. | | | 1 127 | |
| Evolutionary Testing w/ Hamming dist. & extra instr. | | | | 395 |
| $RT/ET$ | 0.9 | 36.6 | 58.1 | 165.7 |

Table 4.1: Average number of runs required to achieve full coverage in the triangle classification program

ical change of the predicate is enormous: changing the most significant bit of one of the variables yields a huge change in the predicate.

To abstract away from the *meaning* of a single bit Jones *et al.* [11] utilize the hamming distance: the number of differing bits. After some experiments on paper comparing three `char`'s the following mechanism has been devised. First, the three variables are combined into one bit string:

$$\neg(a \oplus b) \wedge \neg(b \oplus c),$$

where $\neg$ is a logical NOT, $\oplus$ is a logical XOR, and $\wedge$ is a logical AND. This logical equation will produce a string of ones when all three variables are identical. This can be used as the first predicate; the second predicate is a string of ones, of the same length as the input variables—`0xFF` in the `char` case. Now the number of differing bits is used as fitness measure. Because the EA is based on the maximization of the fitness values, the reciprocal of the number of differing bits is used:

$$fitness = \frac{1}{|dist| + \delta}$$

$\delta$ is an arbitrary small value, introduced to prevent division by zero. Swapping a bit now has minor numerical consequences. Also, for comparison reasons, the extra instrumentation is removed. After implementation the EA is ran again for 1 000 times. The results can be seen in Table 4.1: 1 127 executions, which is 58.1 times less than RT. When the extra instrumentation is added again the EA only needs 395 executions on average, which is 165.7 times less than RT.

### Analysis of Initial Experiments

Wegener *et al.* [28] achieved results which are quite different from the results for this work. His EA achieved only 10% less required executions than RT. Wegener also mentions test times varying between 160 and 254 seconds for full coverage, meaning 4–6 seconds per partial aim. Testing time for the experiments for this thesis were, in the Hamming distance case, less then a second for full coverage. In the case with extra instrumentation

the execution took a little longer, with a few spikes to a couple of seconds—again, for full coverage.

It is unknown what sort of hardware has been used for Wegener's experiments. The article containing data on his experiments has been published in 2001. Research on Intel's processor history[1] learns that the Pentium4 was introduced just before that time, running at speeds below 2GHz and missing a lot of enhancements which are present in current processors. The experiments carried out by the author have been run on a Pentium4 running at 3.60GHz—this probably partly explains the difference in running times. This does not, however, explain the difference in achieved speedup. Wegener has based his fitness on evaluating *"individuals according to the branching conditions of the test object"*, just as for the experiments for this work. Wegener also utilizes the approximation level. With this measure as part of the fitness he achieves the same speedup as has been achieved by the author with use of extra instrumentation, as explained in Section 4.1.1; introducing the Hamming distance as part of the fitness function as explained in the previous section yields an enormous increase in performance.

**Analysis of Reliability**

A histogram has been created for the four versions of the EA, keeping track of the amount of needed executions per run, 1 000 runs in total; the results can be seen in Figure 4.2. The initial version performs well: a lot of runs were finished within 10 000 executions. The problem is that sometimes the algorithm hardly converges to an answer; this is shown by the "Too many" column, meaning that it took over 100 000 executions. Of course this pushes the average to a higher number. The initial version is *unpredictable*: most of the times a solution is found rather quick, but not as often as desired. In one case the EA was unable to find a solution at all—50 000 generations, totaling 1 000 040 executions.

Extra instrumentation practically eliminates this problem; in only two occasions the EA took a long time to converge—but converging it did. A solution was found in every run. The hamming distance performed better on average than the extra instrumentation, but as the graph shows this is at the cost of variance in results. *Combining* extra instrumentation and hamming distance seems to be the key. All runs finished within 10 000 executions. To illustrate the power of this combination, Figure 4.3 shows more detail. Note the difference in scale of the X axis! As can be seen, all runs finish within 1 000 executions. It seems that for both figures, a Normal or Poisson distribution can be seen in the data, aside from the "Too many" column. Maybe this information can be used in the future to predict the needed amount of runs; this falls beyond the scope of this work.

Jones *et al.* in [11] performed a number of experiments, among which the triangle classification. His results are summarized in Table 4.2. Now he actually used the Hamming distance, and still only managed to achieve a factor nine in efficiency. The only plausible conclusion is that the fitness function construction differs, just as Baresel *et al.* concluded in [2]. Further investigation learns that Jones uses a another variant on the triangle classification program, which also classifies triangles as being right-angled. The program for this work does not do this. Running random tests with this second version learns that the branch

[1] History can be found on Wikipedia, `http://en.wikipedia.org/wiki/Intel`

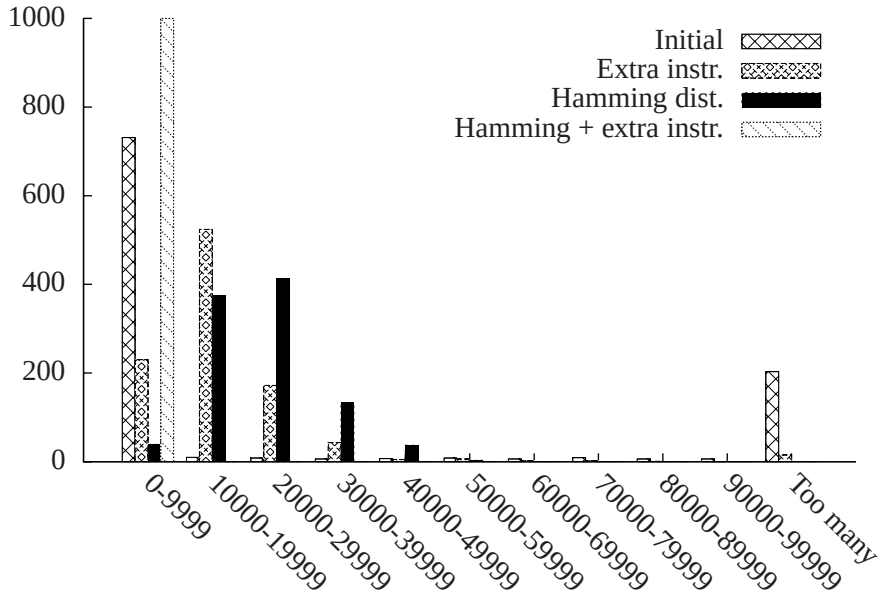



Figure 4.2: The amount of runs per execution categorized, for 1 000 runs

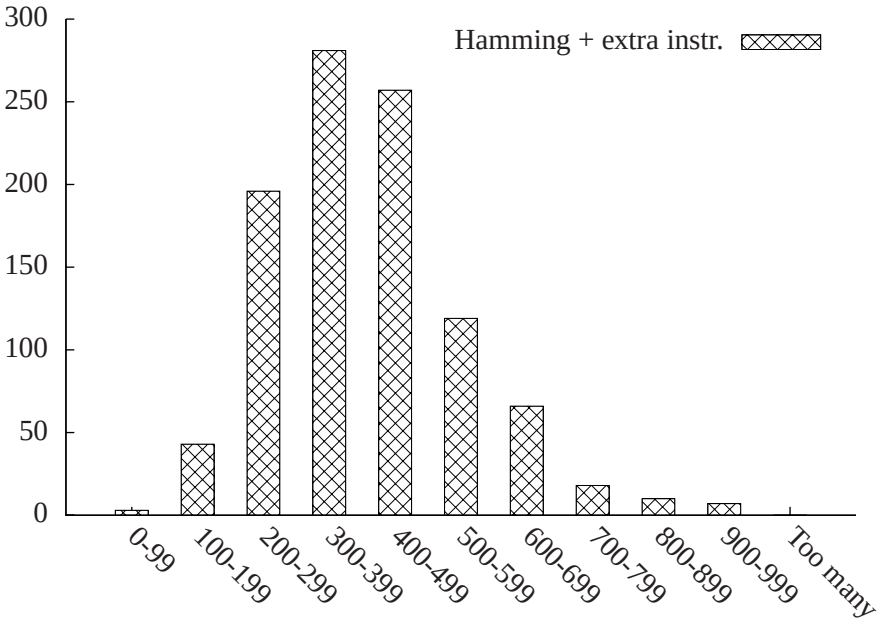



Figure 4.3: Distribution for the run with hamming distance and extra instrumentation

| Testing with | Avg. exec's |
|---|---|
| Random Testing | 163 000 |
| Evolutionary Testing | 18 000 |
| $RT/ET$ | 9 |

Table 4.2: Jones' results on testing with triangle classification

which is hardest to cover—and thus determines the amount of executions—is exactly the same branch as in the first version: the equiliteral triangle case. Wegener tests with both three integer and three float input variables. Even when an integer is only 16 bits long, the average needed amount of randomly generated test cases is $(2^{16}-1)^2 = 4\,294\,836\,225$; Wegener only needed 199 743 test cases. Floats are even longer, thus needing many, *many* more test cases; Wegener only needs 215 834 random test cases in the float scenario. How these results have been achieved is unclear to the author; the differences in achieved performances cannot be explained by the difference in version either.

### 4.1.2 The `atof()` Function

Atof has been taken from the standard C libraries. It is more complex than the triangle classification program, because of the loops it contains, the size of the program, and because of the search space. The source of the atof program can be found in Appendix B.2. Atof takes a string as input, and produces a double as output: it converts the given text—an array of characters—to it's corresponding double. The size of the input string has been limited to 10 characters. This implies a search space of $256^{10}$ input combinations[2], as *all* possible bit-combinations are used as input.

#### Random Testing

Just as with the triangle classification program, atof is tested with random input first. The code is instrumented to collect coverage data, and random data is used as input as long as full coverage has not been achieved. As shown in Figure 3.2b, loops have to be handled in a special way, this time the instrumentation is adapted for the handling of loops.

Performing a single run of the RT algorithm showed that the execution of 1 000 runs would take an unacceptable amount of time. Therefore, for this experiment 100 runs are performed and for every run the amount of executions needed to achieve full branch coverage is counted. The results are a lot worse than expected, as shown in Table 4.3. Random testing needs an enormous amount of test data, and takes a lot of time: about thirteen minutes on average. On average 1 660 936 793 executions were needed before full coverage was achieved. The maximum number of executions was 7 491 066 187, which took more than 90 minutes to complete.

[2] A `char` is one byte long, thus having 256 possible values.

| Testing with | Avg. exec's |
|---|---|
| Random Testing | 1 660 936 793 |
| Evolutionary Testing | 54 339 |
| $RT/ET$ | 30 566 |

Table 4.3: Average number of runs required to achieve full coverage in the `atof()` program

```
1 void Simple_Popul::lookingBranchLoop(int b, int its, int reqIts) {
2     if(b == looking) {
3         fitness = (1.0 / pow((abs(its - reqIts) + 0.01), 2.0));
4     }
5 }
```

Listing 4.3: The `lookingBranchLoop()` method for the EA

**Evolutionary Testing**

In Section 3.2.3 it is stated that the number of iterations made in a certain loop is directly linked to the individual. The newly written function `lookingBranchLoop()` is listed in Listing 4.3. For a certain branch `b` there are `reqIts` loop iterations required; this number is compared to `its`, the actual number of iterations. A fitness is then calculated solely on the number of iterations.

Again 1 000 runs are performed, each run seeking for full coverage. The result is found in Table 4.3: on average, the EA needed 54 339 executions of the software under test; a speedup of 30 566 times less needed executions than RT! Execution time, even the worst case, is in the order of seconds in stead of minutes or even hours in the RT case.

**Analysis**

In the triangle classification program the usage of a Hamming distance greatly improved the performance. For the `atof` program the various predicates all consist of comparing two primitive values; the hamming distance has not been used in the atof case, simply because there was no need to do so. Figure 4.4 shows the distribution of the number of required executions for achieving full coverage. The distribution has a lot of the characteristics of the distribution in the triangle case; the same normal or Poisson distribution. One of the things that stand out is the amount of runs where there were "too many" runs. Looking at the logs reveals that the runs are long, but in the end they do converge. It takes about 400 000 runs to finish the execution of the EA in the most extreme cases.

All in all the EA does its work efficiently; over 70% of the runs finishes within 60 000 executions. A considerable improvement over RT, which is too slow for use in a real-world situation. The `atof()` program is more complex than the triangle classification program, but still fairly simple compared to 'normal' software.

Comparing the achieved results to Wegener's results (from [28]), which have been summarized in Table 4.4, a number of differences stand out. Wegener tests with a ten-

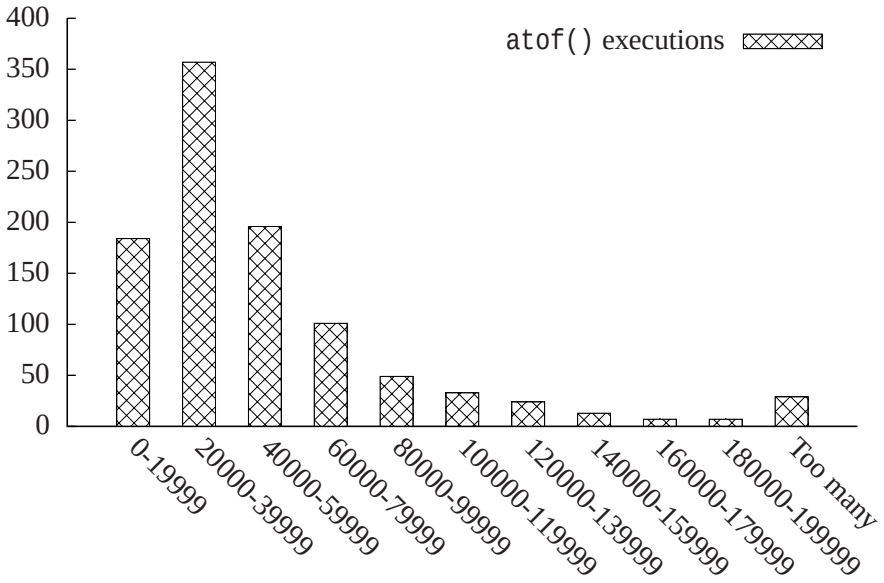



Figure 4.4: The amount of `atof()` runs per execution categorized, for 1 000 runs

| Testing with | Avg. exec's |
|---|---|
| Random Testing | 1 251 038 |
| Evolutionary Testing | 35 263 |
| $RT/ET$ | 35 |

Table 4.4: Average number of runs needed by Wegener for the the `atof()` program

character string, so his search space also is $256^{10}$ individuals. The first major difference is the achieved speedup: 35 versus 30 566 times faster than RT. The second major difference can be found in the number of required random tests. This second difference is hard to explain: the same string length, the same amount of possible characters per position (256), exactly the same as for experiments conducted for this work. The only difference is that Wegener performed all experiments ten times, instead of 100 or 1 000 times; maybe Wegener was 'lucky'. Because `atof` is a standard function, with standardized input and output; a difference in version such as with the triangle classification program, is unlikely.

### 4.1.3 Interim Summary and Conclusions

The initial version—without extra instrumentation and without hamming distance—is too unpredictable to be usable in a real-world situation. Extra instrumentation is required for the EA to behave in a predictable way, and assure converging to a solution in an acceptable

amount of executions.

If more than two predicates are to be compared, the algorithm needs some mechanism to abstract away from the actual meaning of the predicates. Here the hamming distance was introduced, greatly improving both performance and predictability of the EA. Abstraction is used on more levels in the EA; in the handling of loops and dealing with complex data types other forms of abstraction are utilized. It serves as a mechanism to enable objective comparisons between individuals, which are otherwise not comparable in a native way.

When the instrumentation is adapted by adding calls to nodes higher up in the CFG and utilizing the hamming distance, the EA requires 165 times less executions of the software under test than RT in the triangle classification program. Even in a simple program there is an enormous increase in efficiency. In the `atof()` tests the efficiency was increased much more: the EA required over 30 500 times less executions of the software under test before test data were generated. Execution times went from 13 minutes on average for RT to only a few seconds for ET.

Evolutionary testing has been proven efficient, fast, and predictable: our algorithm achieves better results than previous works, while achieving a low variance in required executions. However, the success of the algorithm greatly depends on the instrumentation: how are predicates composed, and sometimes a form of abstraction is needed—for loops and multi-variable predicates. Abstraction is also required for complex data types. Sthamer in [21] explains that also for this purpose the hamming distance can be used; this is supported by Jones *et al* in [11].

## 4.2 Improving Code Coverage with Meta-heuristics

Now the concept of evolutionary testing has been reproduced the research can be focused on achieving *extra* coverage—the main research questions for this project.

The Madymo code base is already partly covered by tests, making it unnecessary to instrument all code. The effectiveness of ET has been proven in the previous sections, when it comes to full instrumentation; now the question becomes how to *partially* instrument the code, without losing predictability, and still reach the coverage targets in an easy way? After all, the instrumentation also ordered the execution of various targets; partial instrumentation removes part of the guidance mechanism of full instrumentation.

### 4.2.1 Sparse Instrumentation

Partial instrumentation in its most simple form means removing all instrumentation except for those branches needing coverage: the untested code. In the previous experiments one branch emerged as the hardest to cover. Let's assume that all code is already covered by tests, and only this branch, this exceptional case, is not tested yet. What will happen when only this branch is instrumented with a `checkBranch()` call, and its direct sibling(s) are instrumented with `lookingBranch()` calls; the code is sparsely instrumented. Although a few calls are needed, this will be treated as a single instrumented branch. The instrumented branch itself is not a nested structure; it evaluates a variable which is depending on the outcome of all sorts of nested structures.

| | Triangle | | `atof()` | |
|---|---|---|---|---|
| Instrumentation | Full | Sparse | Full | Sparse |
| Number of Branches | 7 | 7 | 38 | 38 |
| Instrumented branches | 7 | 1 | 38 | 1 |
| Avg. exec's required | 395 | 442 | 72 931 | 58 395 |

Table 4.5: Comparison of full and sparse instrumentation, applied to the `atof()` program

**Triangle Classification**

For triangle classification a histogram of the achieved results can be seen in Figure 4.5; the distribution is almost the same as with full instrumentation. Note that the instrumentation utilizes the hamming distance, resulting in a solution for every run. The average amount of executions is a little higher, but that is basically the only difference; for triangle classification, sparse instrumentation seems to be effective. Of course this is a very small test program, and additional experiments are needed to show whether other programs give the same result.

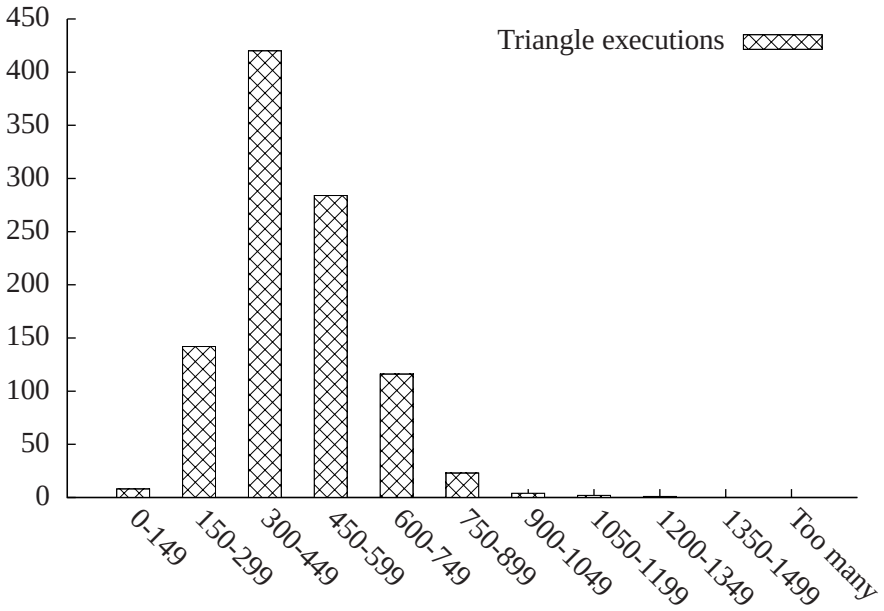



Figure 4.5: The amount of triangle runs per execution with sparse instrumentation

**The `atof()` program**

The same experiment as in the previous section has been repeated for the `atof()` program. From the logs of the initial experiments it became clear that two branches are the hardest to

cover; these two branches are instrumented, and the EA is run. The results are summarized in Table 4.5. Remember that this program is much more complex than triangle classification. Again, these numbers are based on 1 000 runs of the algorithm. Surprisingly, the sparse instrumentation does a fairly good job. Of course it performs much worse on a per-branch basis, but considering how difficult it is to actually cover this branch it is remarkable how fast the EA converges.

However, as Figure 4.6 shows, the variation is enormous. It shows the same distribution as before, but note that a lot of runs ended with "Too many" executions—73 runs to be precise. In 10 of these 73 cases, the EA is unable altogether to come to a solution in 50 000 generations or approximately 1 000 000 executions. This is where the sparse instrumentation seems to fail: on average it performs much better than expected, but at the cost of predictability of the EA.

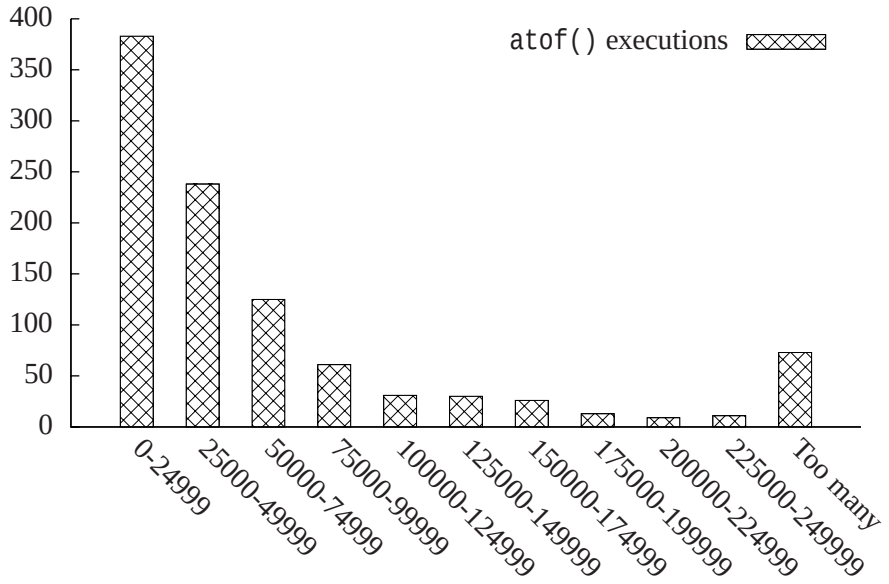



Figure 4.6: The amount of `atof()` runs per execution with sparse instrumentation

### 4.2.2 Steering the EA Towards Target Nodes

With the unpredictability of sparse instrumentation from the previous sections in mind, the next step is to try to give hints to the EA: i.e., to steer the algorithm towards the right branch. In the `atof` case this can be accomplished by adding instrumentation to the branches where the tested value is being 'built'. However, fitness values are determined for the current LOOKING node in a way that there is always only one fitness determination per invocation of the software under test for every target node; even for the extra instrumentation as

described in Section 4.1.1 the determination of a fitness value is unique. Adding instrumentation to steer the EA involves multiple fitness determinations for the same target node: in the execution tree certain branches have to be taken to eventually reach the target node. For each of these nodes where execution can miss the target node extra instrumentation is added.

When the software under test is called with a certain individual, the LOOKING node can only be reached when a certain path through the CFG is taken. For example: when the LOOKING node is in a nested IF statement, the condition for the first IF statement has to be a certain result. Here instrumentation can be added, so the population will evolve to taking the desired branch, meaning the fitness of the population gets higher and higher. When this branch is taken—this subgoal has been met—execution flows to the next node. Execution now splits again, at this node; fitness has to be determined in another way, resulting in new fitness values which can be significantly lower than fitness values achieved by other individuals which missed the first target branch. This means that although execution is closer to the target node but the fitness is lowered, resulting in a lower chance of survival!

The approximation level as explained in Section 3.3.1 seems like a solution. The fitness determination can be preserved as it is, and the approximation level can be added. The approximation level (AL) has precedence over the fitness value, i.e., the AL becomes the most important variable in the sorting and fitness determination algorithm. A smaller AL means that execution of the current individual achieved execution of code closer to the LOOKING node; a higher fitness means that the individual is closer to achieving execution of code closer to the LOOKING node, i.e. reaching a branch with a lower AL.

Calculating a fitness now depends on a few parameters. First of all, most important, the branch parameter (*b*, see Listing 4.2 on page 23) is compared to the current LOOKING node. If these match, the next parameter to be checked is the AL. This is a new parameter; if the AL is larger than the AL already achieved by this individual, the fitness will not be updated. If the AL is smaller than the already achieved AL, this new AL will be the new standard. In the latter case, as well as when the AL is equal to the already achieved AL, the fitness will be updated. No conflict will arise from this operation, because the combination of AL and branch is unique! This way all individuals will be assigned an AL and a fitness value, making them comparable again on two criteria, of which the AL is the most important.

This mechanism is implemented in the EA, and the test programs are re-instrumented with the necessary approximation levels. The EA now seeks coverage for specific targets, with partial instrumentation, based on fitness and approximation level.

#### Triangle Classification

The triangle classification is a small program, and testing such a mechanism as AL on such a simple program might not lead to drastic results—if any at all. The branch which is the hardest to reach is set as only target, and instrumentation in prior nodes is added with appropriate AL's. The average amount of executions for 1 000 runs is 304. Looking at Table 4.5, it can be concluded that the AL yields an improvement of 31 percent less executions needed. As Figure 4.7 shows the distribution of the runs is very clean: there no

runs with “too many” executions.

#### The `atof()` Program

Now the more complicated `atof()` program is tested. This is a much more complicated program, and significant results can be expected. The same two hard to reach branches are instrumented, and extra instrumentation with AL information is added. Again, remember Table 4.5; 58 395 executions were needed on average to achieve full coverage with sparse instrumentation. Now, when 1 000 runs are performed with the AL added to the EA, on average 10 807 executions are needed. An increase in performance of 81 percent less needed executions!

The distributions of the needed executions is shown in Figure 4.8. It might look like a lot of the runs required “too many” executions. To be precise, 59 runs fall in this category; this may seem as a lot, but all 59 runs did converge to a solution within the set limit of 50 000 generations (approximately 1 000 000 executions). The worst case required 384 680 executions; a little over 33 percent of the limit.

### 4.2.3 Interim Summary and Conclusions

Already the usefulness of an EA has been proven in this chapter. This section studied the performance of ET in an environment where only certain branches needed to be covered. Parts of the code are already covered by tests, and only the rest of the code needs covering. An attempt was made with ‘sparse’ instrumentation, i.e., only the branch to cover was instrumented. The results were encouraging, but performance was unpredictable.

Adding another mechanism to the EA, the so-called approximation level (AL), should improve the performance. The AL steers the execution of the EA towards the target node(s) by adding information on how far away execution is from those target branches. This yielded remarkable improvement. Where sparse instrumentation needed about the same amount of executions as in the fully instrumented examples but introduced great variance in results, the AL leads to much faster convergence.

The AL has proven to be very useful for creating test data for only parts of the code. In the triangle classification program a solution was found in 304 executions; a 31% improvement over sparse instrumentation. In the `atof()` case a solution was found in 10 807 executions, which was an 81% improvement over sparse instrumentation.

The conclusion at this point, is that an EA could very well lead to improving the code coverage of an application and that addition of an AL is of great help to reduce variance and increase convergence speed. The next step is putting the EA to the test in a real-life environment: applying it to parts of the code base of the Madymo Workspace.

## 4.3 Applying the Evolutionary Algorithm to More Complex Code

In the previous sections the concept of improving code coverage with the use of an EA has been proven on programs, which are being used as examples throughout the academic

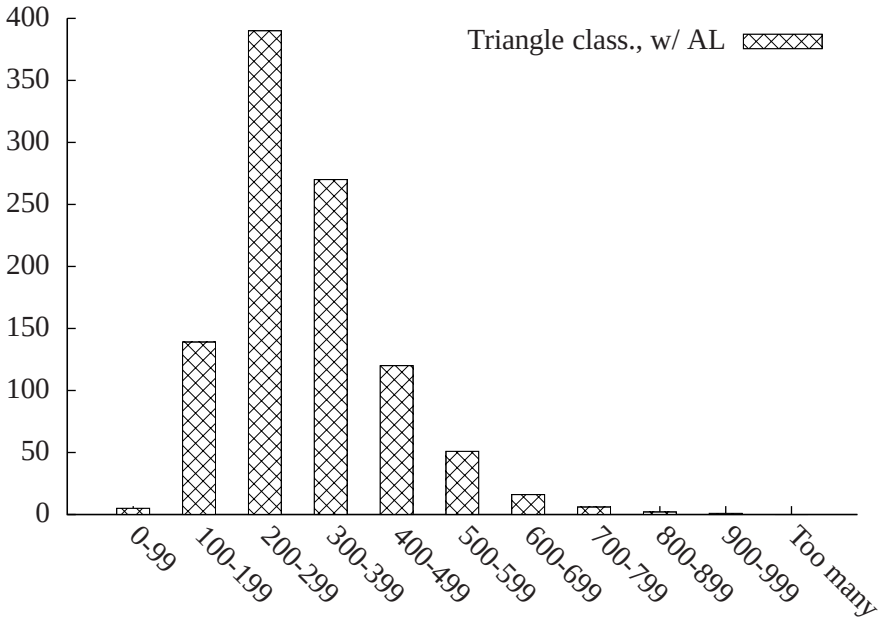



Figure 4.7: The amount of triangle classification runs with approximation level

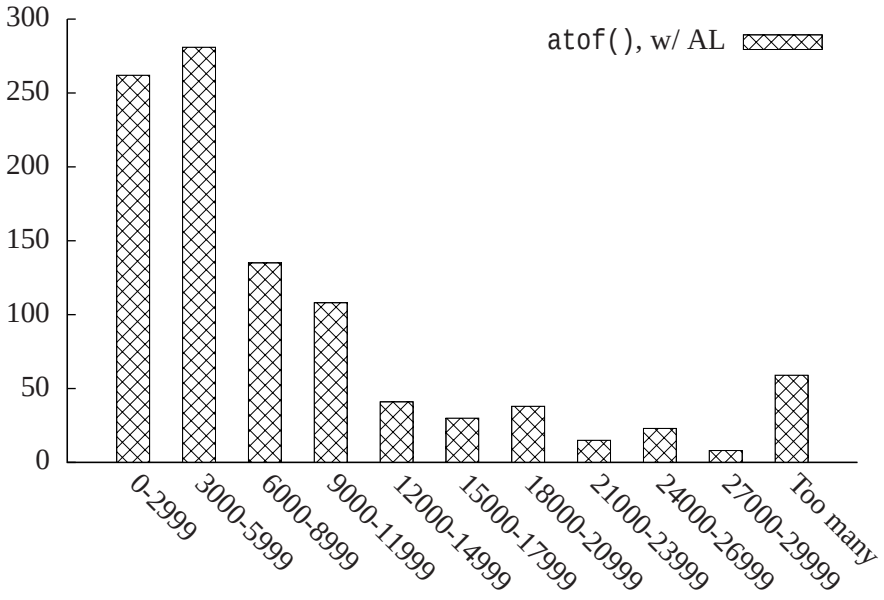



Figure 4.8: The amount of `atof()` runs with approximation level

world. The next challenge is applying the same algorithm and methodology to a more complex and real-world situation: the Madymo Workspace applications.

### 4.3.1 Workspace Applications

The Madymo Workspace consists of a number of programs, all of them having a large code base of multiple KLOC[3] each. Much of the code is dedicated to the graphical user interface (GUI) of the separate applications, which takes events from the GUI as input. Because the EA starts at random input it is infeasible that the necessary events will be generated. They also have to be generated in the right order; therefore the GUI code will not be part of the experiment. A more interesting part of the code base is the *pkg*, or "package" library: a collection of service routines, used by all Workspace Applications, with a large range of functionality. The amount of GUI code is limited, making the pkg library an interesting piece of code to instrument and test with the EA.

Together with TASS' main Software Architect and later also with one of the Senior Developers the library has been reviewed, looking for interesting pieces of code to test. The findings were combined with a study of the generated coverage reports—generated by a member of the Test Group—to determine where *improvement* of the code coverage is wanted.

Looking at the code, a number of differences with respect to the previous experiments immediately stand out:

- Opposed to the initial experiments, all applications and libraries are written in an object oriented language (C++), as opposed to the previous programs, which were written in a procedural language (C);
- Parts of the code are templated[4];
- Conditions in IF statements, WHILE statements, etc, are constructed very differently than those in former experiments;
- There are many, many dependencies which need solving.

These points will be explained in more detail below.

#### Another Programming Language

The C++ language is an object oriented (OO) language; the code of the Workspace uses a large amount of classes, introducing methods and attributes per class. Where the sequence of execution in C programs can be determined with relative ease, the order of execution in a C++ program cannot be guaranteed anymore. As Section 3.2.2 explained the order in which all instrumented branches are numbered has an influence on the execution of the EA. The C++ language thus introduces a challenge.

[3] KLOC: common term for 1000 Lines Of Code

[4] Templates are a form of *Generic Programming*; more information can be found in Wikipedia's article on generic programming.

```
1 if( msgQueue.isFull() ) {
2     // Target code
3 }
```

Listing 4.4: A segment of code with an example of a problematic statement

#### Generics

One of the features of C++ is the availability of generics. In the source of the Workspace this mechanism is often used, introducing a layer of abstraction in the code. Segments of code are written for families of classes, and are thus executed by various data types, of varying sizes. This will make instrumentation of such code segments more challenging than 'normal' code. This also has effects on the initialization of the EA. The length of the chromosomes is fixed; as data types can vary with generics the actual size of data types is only known at runtime.

#### Conditions

In the initial experiments, all conditions were on a mathematical basis. Numbers were compared for equality, one being greater than the sum of two others, et cetera. These mathematical comparisons are very rare in the Workspace code. Many of the conditions are a method call on an instance of an object, resulting in a boolean value, as can be seen in Listing 4.4.

Cheon and Kim [5] propose to just move the code from such a method call to the calling condition. This can only be done in the most simple cases, and in general will not solve the problem. Baresel and Sthamer [1] show that in certain circumstances it is possible to steer the execution in such a way that related variables are set to a required value, so eventually the target can be executed.

#### Dependencies

Opposed to the prior experiments which mainly uses primitives as input, intermediate results, and output, the software in the package library hardly uses primitives at all. Parameters are instances of classes, intermediate results are method calls, which in their turn depend on other variables—primitives or complex data types—making it very hard to control from the EA point of view.

### 4.3.2 Program States

Related to the dependencies mentioned in the previous section, is the state of an object. Global variables can indicate the state of an instance of a class, and methods produce a certain result depending on the current state. Multiple calls on an instance can alter these variables, thus requiring an unknown number of method calls before the required state is achieved. These global variables are often not part of the input parameters, making it impossible for an EA to optimize them: a certain state is required to achieve the desired result,

but the parameters to reach this state are not under direct control. McMinn and Holcomb in [16] acknowledge this problem, and conclude that an EA may not be able to solve this. They introduce a completely different sort of EA to solve this problem.

Wappler and Wegener in [26] argue that the test framework should prepare the environment by a sequence of method calls $S =< m_1, m_2, \ldots, m_n >$, where $m_1, m_2, \ldots$ are the methods which prepare the software under test for execution of the actual method under test, which is $m_n$. The preparation is guided by the tester; parameters are manually chosen and set to a certain state by issuing the needed method calls. The collection of all the classes from which instances can be required is called the *test cluster* [25].

### 4.3.3 Using the EA in an OO Environment

To be able to use the developed EA in an OO environment a number of steps must be taken. Because of the structure of a class, a unit test in an OO environment is the testing of a method. This method cannot be written in a template, because the chromosome size must be fixed. Objects needed to be able to test the method must be created in advance, and to put these objects in their desired/required states a number of method calls must be performed. Because in the method under test conditions often depend on a certain state of the object (or attribute objects) the tester has to prepare the environment in great detail.

This preparation means that a lot of time has to be invested in the actual instrumentation of the code, setting up the environment in which the class under test is operating, analyzing code to discover which objects are required by the environment to enable execution of the target code, and in which state the objects in the environment should be put. Considering the size of a method, the usability of this approach is questionable: a lot of time has to be invested in analysis of the method under test, as well as the entire test cluster. Then, when the required states are determined, the EA hardly has to do any work: the tester has created a scenario which enables the EA to touch the target code.

On top of the time investment needed to be able to use the EA in its current form, the tester also needs to know the function of all code: he has to know the environment, how classes are organized and how certain required states can be reached. This in-depth knowledge can only exist when all code and requirements are documented in detail. In [17] it was concluded for the TASS case that much knowledge is tacid. A situation not specific to TASS: often this level of detailed knowledge only exists in the minds of the programmers.

In literature these problems are first acknowledged by Baresel and Sthamer [1] and Tonella [23], and confirmed by various others [26, 25, 5, 27, 16]: an EA as it has been developed for this work has been proven effective for a procedural environment. Modern programs are often written in an OO programming language, which introduces complexities that cannot be solved in a simple way. Tonella and other researchers use a form of genetic programming [14], which is outside the scope of this work.

## 4.4 Madymo Solver

To test the EA in a procedural industry standard piece of code, an attempt is made to use ET in the mathematical solver of Madymo. The solver has been written in Fortran, a procedural

language, and the EA should thus be able to generate test data for the solver. Instrumentation is to be added, and an attempt is made to generate test data for the solver.

### 4.4.1 Instrumenting the Solver

Instrumenting the solver introduces two difficulties. First, the solver is written in Fortran and the EA has been written in C++. Second, the solver highly depends on input gathered from XML files (so-called *input decks*).

#### Combining Fortran with C++

To be able to communicate between the solver and the EA a new layer is to be added to the EA. This 'wrapper' is a C (procedural) interface to the EA. This way the solver can make calls to the EA to give feedback on achieved coverage, enabling it to assign a fitness to individuals.

#### Input Parameters

The methods (or functions) in the solver take 20 to 25 parameters. In Fortran functions cannot explicitly return a value: this is done through the parameter list. Part of the parameters thus serve as return variables.

Another part of the parameter list are data arrays, such as elements or forces on elements, which come directly from an input deck or are calculated on a high level of the solver. As the EA cannot generate entire *meaningful* XML files, this is a challenge. The solver is highly dependant on input decks. Therefore the choice is made to alter the EA: instead of running the EA as main program—which makes a call to a piece of code in the assigned fitness function, the entire solver is compiled. In the solver a call is added to a function in the wrapper of the EA, which prepares the EA. The method under test is separated, and the fitness function makes a call to the separated function. This way the amount of input parameters can be controlled, and necessary elements are provided by a simple input deck.

The solver can now be run with the input deck, and execution is intercepted at the method under test. The EA than takes over, generating test data which can be added to a unit test. When the EA has finished a call is made to the solver, requesting a shutdown of the solver-run. Execution is then ended.

### 4.4.2 Generating Test Data

A procedure in the solver has been selected by the manager of the test group. It is a procedure from the lowest layer of the solver, where the actual calculations are performed. It represents a material type, and calculates stress increments using an orthotropic, elastic damage model. Of the parameter list, only one parameter actually influenced the execution flow. Although more parameters would have been more challenging, this parameter is one of double precision: an 8 byte long number, consisting of a significand and a mantissa. It

can be considered a complex datatype. At some point in the procedure this parameter is tested in an `IF...ELSE` block for a zero value; this branch is instrumented.

When 1 000 runs of the EA are performed a solution is found in 70 929 executions on average. In eight runs the EA is unable to generate a solution. The generated solutions is always the same: *NaN*[5]. Studying standard IEEE 754 [10] learns that the bit pattern for NaN is a zero significand and a non-zero mantissa. The EA generates NaN because the mantissa is converging to the biggest negative number possible—being closer to zero—and the significand converges to zero. When the significand reaches zero, NaN is generated. It is a special case, but fulfills the condition. It has been checked that both the EA and the solver interpret NaN the same.

This proves (although for a simple case) that the concept works in a real-world, non-C, live environment.

## 4.5 Method Evaluation

It has been confirmed that an EA can be used as a means to automatically generate test cases for achieving full code coverage. Using optimizations as the addition of extra instrumentation calls to steer the EA towards the target node, as well as utilizing the hamming distance as comparing mechanism for individuals greatly improved the performance of the EA.

When an unmodified EA is used for achieving *extra* coverage, the results become very unpredictable. The EA cannot cope with the removal of instrumentation in the parts of the already covered code: it either fails to find a solution at all, and when a solution is found the amount of executions needed greatly varies. By adding extra instrumentation calls to the software under test the EA receives extra guidance; however this introduces ambiguity of fitness values. The fitness is determined multiple times per individual, because multiple branches have to be taken to reach the target code. When an extra measurement is introduced, the *approximation level* or AL, this problem is solved. The AL enables the EA to distinguish between reached branches, thus rewarding individuals getting closer to the target nodes with a higher chance of survival. The AL becomes the most important metric in the EA; the numerical fitness value is still used to compare individuals achieving the same AL. This ensures unique AL/fitness combinations for each individual, providing a way of adding instrumentation without losing efficiency of guidance. Experiments proved the expected effect: partial instrumentation with the use of AL's leads to fast results, with a much larger probability of converging to a solution.

### 4.5.1 Applying the Method

When an attempt was made to apply the method to the Madymo Workspace code, research showed that the EA as it was developed for this work is not usable in an *object oriented* (OO) environment. An OO environment, compared to a procedural environment as used for the experiments, introduces complexities and dependencies which cannot be solved without introducing much extra work—which needs to be done by hand. This extra work also

[5] NaN is a special numerical value; it is an acronym for Not a Number.

requires in-depth knowledge of the code. These issues lead to the conclusion that although the EA and the principle of partial instrumentation give promising results, it needs another way of chromosome composition and how the EA deals with chromosomes when it is used in an OO environment.

To be able to test the EA in a real-world, live environment, options for applying the EA to the Madymo Solver have been explored. A procedure in the solver has been instrumented to achieve code coverage in a specific branch, and the solver has been put on top of the EA, making the solver the 'leading' application. Creating a C wrapper around the EA makes it possible to use the EA in a non-C environment: the EA generates test data, covering the instrumented branch.

### 4.5.2 Usability

Although the concept of partial instrumentation in a procedural has been proven successful in this work, there are drawbacks. During our experiments it became clear that the algorithm in its current form is not usable in an OO environment: it can only operate on software written in a procedural language. Generics, complex conditions, the state problem and the complex dependencies pose a challenge too great for our EA. Genetic programming can—most likely—solve the problems we encountered.

Second, as already indicated in Section 1.3.2, adding the instrumentation calls has been done by hand. This is a laborious work, and hampers using the EA in its current form. If this process is automated—which should be possible, there are tools to automate this—ET could be of great help for TASS. An EA is very well able to generate extra cases, as an addition to existing test cases.

This project aimed at proving the concept of automatically generating test data in an environment where parts of the code are already covered by existing test suites. This concept has been proven to be very successful; although more work is needed—as will be explained in the Future Work section (Section 5.2)—before such a method can be actually implemented in a live environment, our results are promising enough to keep researching this approach.

# Chapter 5

# Summary and Future Work

TASS wanted to increase their code coverage by means of some automated search for test data, without increasing test suite size too much: the amount of tests should be limited. EA's have been used for achieving full test code coverage, but no attention was paid to *how* this was achieved. Especially TASS testers and developers indicated that achieving full code coverage is important, but the price of an unmanageable amount of tests is too high.

This work has successfully proven the possibilities when utilizing an EA in an environment where parts of the code are already covered by one or more tests. An EA has been adapted to be able to cope with the new situation in which it needed to operate, and test cases have been selected.

This chapter will reflect on the performed experiments, and what the results mean for the concept of ET in an already partly covered code base. Also it will try to indicate what future work must be done to make this method a success.

## 5.1 Conclusions

Other works have been investigated, which indicated that an EA is capable of automatically generating test data for achieving full code coverage. Before we extended this approach to achieving *extra* code coverage, an EA had to be designed and verified to work correct. The concept of ET has been confirmed, looking for full code coverage. We have adapted the form of instrumentation by strategically adding extra lines of instrumentation, and introducing the hamming distance. This combination led to an enormous increase in efficiency, outperforming previous works; this also led to a decrease in variance of length (in terms of required executions) of the required run for achieving full coverage. Our EA gives fast results, with great reliability.

When these results are extended to a new concept, being *increasing* code coverage, it has been shown that instrumenting only those branches which need covering is not enough: although the EA is able to often converge to a solution, the results are very unpredictable. There is great variance in the required amount of executions, and in some cases the EA was unable to generate a solution at all within the set limit of 50 000 generations, or approximately 1 000 000 executions. Simply adding extra instrumentation calls to guide the

execution introduced ambiguous fitness values. To solve this the so-called approximation level was added as an extra measure of fitness. This improved the performance of the EA: the average amount of required executions was lowered, and the amount of variance in the results was reduced. This proved the concept that an EA can generate test data to increase code coverage.

After initial experiments, this concept has been applied to an industry standard piece of code: a part of the Madymo Solver, written in Fortran. Although this introduced two new challenges (another programming language and the dependency on XML input decks), the EA was able to generate test data for a test case.

The problem statement for this work was: *Can regression test code coverage be improved by genetic or evolutionary algorithms, in the context of TASS*. The experiments led to the conclusion that ET can be of great value for TASS: our method has the potential to make the process of test-designing shorter and more easy. Tests can automatically be generated to increase code coverage, and only the input-output combinations have to be verified. When this is done, a number of new regression tests has been created. This reduces the needed effort (human labor) in the designing of tests, and can thus save money and time. Also the fact that our approach enables the tester to indicate for what part of the code test data have to be generated is of great value: this can keep the number of test cases limited to a manageable amount. This also enables the tester to incrementally increase code coverage.

## 5.2 Future work

In the course of this project two issues became clear: the instrumentation introduced a bottleneck, and the EA in its current form is not usable in an OO environment.

In the best-case scenario this method is applied by running the code of the software under test with its current test suite, through some framework; this framework analyzes current code coverage, adds instrumentation where needed, and starts the EA. The manual instrumenting of the code, as has been done for this project, requires knowledge of programming languages and—to some extent—what the functionality of the code under test is. To make this method usable by testers, the instrumenting process should be automated.

Second, when this project was started, it aimed at generating test data for the Madymo Workspace applications. When an attempt was made to instrument a part of the Workspace code it became clear that an OO environment introduced a number of problems: the dependencies on states, boolean result values, uncertainty of execution order, and generics. The EA in its current form cannot cope with these issues. Studying literature learns that this problem has been acknowledged by Tonella in [23]. His approach for solving the issues at hand is one of genetic programming, which fell outside the scope of this work. The results achieved by Tonella look promising, and extending this work to an OO environment with genetic programming as basis is a logical step: modern software is increasingly often written in an OO language.

The automatic instrumenting option in the GNU Compiler Collection can serve as a basis for the framework, which has to be developed. This framework can have a piece of code as input, with the current test suites. It can then automatically analyze the current

coverage, and instrument the parts which are not covered yet. Next it can start a run of an EA, and delivers new test data as output.

## 5.3 Incorporating Evolutionary Testing at TASS

Although this work produced very promising results, it is unlikely that the algorithm in its current form is usable for TASS testers. We have given a proof of concept indicating that an EA can increase code coverage, making ET a promising technique for TASS; however, the issues mentioned under Future Work have to be solved before a 'normal' tester can benefit from ET. A tester does not want to write code (instrumentation), or maybe even has no knowledge of programming languages. When our concept is placed inside a framework, adapted to an OO environment, with an automatic instrumenter, a considerable amount of effort and money can be saved. At the same time the code coverage can be raised to a theoretical 100%, making it possible for TASS to design tests much more efficient without operations being too difficult or technical for testers.

# Bibliography


[1] A. Baresel and H. Sthamer. Evolutionary testing of flag conditions. In *In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2003)*. Springer-Verlag, 2003.

[2] A. Baresel, H. Sthamer, and M. Schmidt. Fitness function design to improve evolutionary structural testing. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002)*, pages 1329–1336. Morgan Kaufmann Publishers Inc., 2002.

[3] B. Beizer. Black-box testing – techniques for functional testing of software and systems. *IEEE Software*, 13(5), 1996.

[4] B. Bruegge and A. A. Dutoit. *Object-Oriented Software Engineering; Conquering Complex and Changing Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.

[5] Y. Cheon and M. Kim. A specification-based fitness function for evolutionary testing of object-oriented programs. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1953–1954, New York, NY, USA, 2006. ACM.

[6] D.E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., 1989.

[7] H.-G. Gross. *Measuring Evolutionary Testability of Real-Time Software*. PhD thesis, University of Glamorgan, Pontyprid, Wales, 2000.

[8] D. Grune, H. E. Bal, C. J. H. Jacobs, and K. Langendoen. *Modern Compiler Design*. John Wiley, 2002.

[9] J. H. Holland. *Adaptation in natural and artificial systems*. MIT Press, 1975.

[10] IEEE. IEEE standard for binary floating-point arithmetic. *ANSI/IEEE Std 754-1985*, 1985.

[11] B. F. Jones, H. H. Sthamer, and D. E. Eyres. Automatic structural testing using genetic algorithms. *The Software Engineering Journal*, 1996.

[12] T. Koomen, L. van der Aalst, B. Broekman, and M. Vroon. *TMap Next*. Uitgeverij Tutein Nolthenius, 2006. Dutch.

[13] T. Koomen and R. Baarda. *TMap Test Topics*. Uitgeverij Tutein Nolthenius, 2004. Dutch.

[14] John R. Koza. Genetic programming: a paradigm for genetically breeding populations of computer programs to solve problems. Technical report, Stanford University, Stanford, CA, USA, 1990.

[15] P. McMinn. Search-based software test data generation: A survey. *Journal on Software Testing Verification and Reliability*, 14(2):105–156, 2004.

[16] P. McMinn and M. Holcombe. Evolutionary testing of state-based programs. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1013–1020, New York, NY, USA, 2005. ACM.

[17] J. van Oenen. *Improving Regression Test Code Coverage with Meta-heuristics – Literature Study*. TASS & TU Delft, 2008.

[18] P. Puschner and A. Burns. A review of worst-case execution-time analysis. *Journal of Real-Time Systems*, 18(2/3):115–128, 2000.

[19] M. Pyhäjärvi, K. Rautiainen, and J. Itkonen. Increasing understanding of the modern testing perspective in software development projects. *System Sciences, 2003. Proceedings of the 36th Annual Hawaii International Conference on*, pages 10 pp.–, 2003.

[20] G. R. Raidl. Evolutionary computation: An overview and recent trends. *ÖGAI Journal*, 2005.

[21] H. Sthamer. *The Automatic Generation of Software Test Data Using Genetic Algorithms*. PhD thesis, University of Glamorgan, Pontyprid, Wales, 1995.

[22] H. Sthamer, J. Wegener, and A. Baresel. Using evolutionary testing to improve efficiency and quality in software testing. In *Proceedings of the 2nd Asia-Pacific Conference on Software Testing Analysis & Review (AsiaSTAR2002)*, 2002.

[23] P. Tonella. Evolutionary testing of classes. *SIGSOFT Softw. Eng. Notes*, 29(4):119–128, 2004.

[24] Nigel James Tracey. *A search-based automated test-data generation framework for safety-critical software*. PhD thesis, University of York, 2000.

[25] S. Wappler and F. Lammermann. Using evolutionary algorithms for the unit testing of object-oriented software. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1053–1060, New York, NY, USA, 2005. ACM.

[26] S. Wappler and J. Wegener. Evolutionary unit testing of object-oriented software using A hybrid evolutionary algorithm. In *Proceedings of the 2006 IEEE Congress on Evolutionary Computation*, pages 3193–3200, Vancouver, 6-21 July 2006. IEEE Press.

[27] S. Wappler and J. Wegener. Evolutionary unit testing of object-oriented software using strongly-typed genetic programming. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1925–1932, New York, NY, USA, 2006. ACM.

[28] J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology, Special Issue devoted to the Application of Metaheuristic Algorithms to Problems in Software Engineering*, 2001.

[29] J. Wegener, K. Buhr, and H. Pohlheim. Automatic test data generation for structural testing of embedded software systems by evolutionary testing. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002)*. Morgan Kaufmann Publishers Inc., 2002.

[30] J. Wegener, H. Sthamer, and A. Baresel. Application fields for evolutionary testing. In *Proceedings of the 9th European International Conference on Software Testing Analysis & Review (EuroSTAR 2001)*, 2001.

# Appendix A

# Glossary

In this appendix an overview is given of used abbreviations.

| | |
|---|---|
| **AL** | Approximation Level |
| **CFG** | Control Flow Graph |
| **EA** | Evolutionary Algorithm |
| **ET** | Evolutionary Testing |
| **GA** | Genetic Algorithm |
| **GUI** | Graphical User Interface |
| **KLOC** | 1000 lines of code |
| **OO** | Object Oriented |
| **QA** | Quality Assurance |
| **SE** | Software Engineering |

# Appendix B

# Program Sources

Listing B.1: Source of Triangle classification program

```
1  #include <iostream>
2  #include "triangle.h"
3
4  using namespace std;
5
6  int tri_type(char a, char b, char c) {
7      int ttype;
8
9      if(a > b) {
10         int t = a;
11         a = b;
12         b = t;
13     }
14
15     if(a > c) {
16         int t = a;
17         a = c;
18         c = t;
19     }
20
21     if(b > c) {
22         int t = b;
23         b = c;
24         c = t;
25     }
26
27     if(a + b <= c) {
28         ttype = NOT_A_TRIANGLE;
29     } else {
30         ttype = SCALENE;
31
32         if(a == b && b == c) {
33             ttype = EQUILITERAL;
34         } else if (a == b || b == c) {
35             ttype = ISOSCELES;
36         }
```

```
37      }
38
39      return ttype;
40  }
```

Listing B.2: Source of `atof()`

```
#include <errno.h>
#include <ctype.h>
#include <math.h>
#include <float.h>
#include <stdlib.h>

double atof(const char *str) {
    return strtod(str, NULL);
}

double strtod(const char *str, char **endptr) {
    double number;
    int exponent;
    int negative;
    char *p = (char *) str;
    double p10;
    int n;
    int num_digits;
    int num_decimals;

    // Skip leading whitespace
    while (isspace(*p)) p++;

    // Handle optional sign
    negative = 0;
    switch (*p) {
        case '-': negative = 1;
        case '+': p++;
    }

    number = 0.;
    exponent = 0;
    num_digits = 0;
    num_decimals = 0;

    // Process string of digits
    while (isdigit(*p)) {
        number = number * 10. + (*p - '0');
        p++;
        num_digits++;
    }

    // Process decimal part
    if (*p == '.') {
        p++;

        while (isdigit(*p)) {
            number = number * 10. + (*p - '0');
            p++;
            num_digits++;
            num_decimals++;
        }
        exponent -= num_decimals;
```

```
54      }
55
56      if (num_digits == 0) {
57          errno = ERANGE;
58          return 0.0;
59      }
60
61      // Correct for sign
62      if (negative) number = -number;
63
64      // Process an exponent string
65      if (*p == 'e' || *p == 'E') {
66          // Handle optional sign
67          negative = 0;
68          switch(*++p) {
69              case '-': negative = 1;
70              case '+': p++;
71          }
72
73          // Process string of digits
74          n = 0;
75          while (isdigit(*p)) {
76              n = n * 10 + (*p - '0');
77              p++;
78          }
79
80          if (negative)
81              exponent -= n;
82          else
83              exponent += n;
84      }
85
86      if (exponent < DBL_MIN_EXP  || exponent > DBL_MAX_EXP) {
87          errno = ERANGE;
88          return HUGE_VAL;
89      }
90
91      // Scale the result
92      p10 = 10.;
93      n = exponent;
94      if (n < 0) n = -n;
95      while (n) {
96          if (n & 1) {
97              if (exponent < 0)
98                  number /= p10;
99              else
100                 number *= p10;
101         }
102         n >>= 1;
103         p10 *= p10;
104     }
105
106     if (number == HUGE_VAL) errno = ERANGE;
107
```

```
108     if (endptr) *endptr = p;
109
110     return number;
111 }
```

# Appendix C

# Revision History

**April 10, 2008:** initial version

**June 5, 2008:** first revision, many updates

**July 9, 2008:** second revision, many updates:

- The “Preface” section has been renamed to “Acknowledgments”, and the section has been written
- Abstract has been written
- Chapter 1:
  - Rewritten introduction of chapter
  - TASS section: added ‘acronym’ word, moved meaning of Madymo
  - Summary section: added meaning of WCET
  - Research Questions section has been renamed to Problem Statements
  - Problem Context section: corrected spelling error
  - Problem Statements section: reformulated statements, separated future work statements
- Chapter 2:
  - Chapter renamed to Related Work
  - Almost fully rewritten chapter. The chapter is much shorter now, and only touches subjects with direct relevance to my research.
  - Section on Evolutionary and Genetic algorithms has been preserved.
- Chapter 3:
  - Chapter has been renamed to Methodology
  - Corrected some spelling errors
  - Section introduction of ET has been changed
  - Section on Optimizing an EA has been renamed to EA Parameters; also this section has been restructured, and parts have been rewritten. The section

now focuses on parameters of the used EA, instead of exploring all options for each phase.
  - The Interim Conclusion section has been replaced by a Proposed Method section. This indicates the method when all research has been done, and what part this project will cover.
- Chapter 4:
  - Chapter has been renamed to Evaluation
  - Corrected some spelling errors
  - Rewritten start of chapter intro
  - Old Section 4.1.1 has been restructured and merged with Section 4.1 (current version).
  - Section 4.1.1 has been split into an intro and a RT subsubsection.
  - The "First Performance" subsubsection has been renamed to Evolutionary Testing
  - Subsubsection "Improving the Performance" has been renamed to "Adding Extra Instrumentation"
  - Last paragraph of the "Hamming Distance" subsubsection has been moved to the "Analysis" subsubsection
  - The "Analysis" subsubsection has been split into "Analysis" and "Analysis of Performance"
  - Section 4.3.2 - remarks added on genetic programming
  - A new Section 4.4 has been written, "Madymo Solver"
  - Interim Conclusion section (old 4.4, now 4.5) has been renamed to Method evaluation. Extra text has been added
- The proposed chapter on OO code has been removed. Extra experiments (on the fortran solver) are described in section 4.4
- Chapter 5 has been written

**July 18, 2008:** third revision, based on partial feedback from Gerd:

- Corrected various spelling errors
- Results added to Abstract
- Some rewriting on the introduction of the Introduction chapter
- Section 1.3 has been renamed to Project Boundaries
- Section 1.3.2 – The second problem statement has been extended, and a third, TASS-specific statement has been added.
- Chapter 2: reversed the order of paragraphs, which means major rewriting
- The old Section 2.3.1 has been moved to the Methodology chapter, as general introduction for GA's (new Section 3.1)

- Old Section 2.4 on TASS Testing methods has been removed. The information is found in various other sections.
- Start of Chapter 3 has been restructured
- Section 3.1 – Background Information has been removed. The information on CFG's has been moved to the Code Instrumentation section, (3.2.1).
- Some implementation details have been removed from the Terminology section (former 3.2.1, now 3.1.2).
- Section 3.1.3 – Fitness Funtion: section has been added
- Section 3.1.4 – Procedure: clarified some details of the phases, and added reference on population size
- Section 3.1.5 – The Effects of Evolution: section has been added

**August 8, 2008:** Many small updates throughout entire thesis