MEASURING EVOLUTIONARY TESTABILITY OF REAL-TIME SOFTWARE

Hans-Gerhard Groß

A submission presented in partial fulfilment of the requirements of the University of Glamorgan/Prifysgol Morgannwg for the degree Doctor of Philosophy

June 2000

Abstract

Evolutionary testing is a new testing technique based on the application of evolutionary algorithms. It can be used to automatically generate test cases which satisfy a given test criterion. For the best- and worst-case execution time analysis of real-time systems it can be applied to generate test cases which minimise or maximise the execution time, or violate the timing specification of such a system.

Evolutionary testing regards the testing process entirely as an optimisation problem with the test criterion as the goal for which suitable test cases must be found. As a typical optimisation technique, evolutionary testing cannot guarantee to find test cases which satisfy this test criterion. The only outcome is the time found, but there is no information on how close this time is compared with the actual minimum or maximum time which must be found.

Experiments with the technique have revealed a relationship between the complexity of a test object and the success of the search technique to generate test cases according to best- or worst-case execution times as test target. For very 'simple' test programs, evolutionary testing could easily generate test-cases according to this target. For more complex programs, the optimisation process exhibited significant difficulties to succeed. This thesis discusses and defines attributes of programs which correspond to aspects of complexity. These are high nesting, small input domains, high parameter interaction or interdependence, and the size of a program's input. New complexity measures are introduced which attempt to capture these aspects of evolutionary testability. The validation of the measures through a technique which was introduced by Shepperd and Ince, identifies three core measures which can be used to predict evolutionary testability for dynamic timing analysis. These are measures for nesting, small input domains and the input size. These measures can be used to predict the expected outcome of an evolutionary testing process, so that the technique can be applied to programs where the expected outcome of evolutionary testing is likely to be of high quality. Additional measures could not be identified as being valid maesures, but they can help to indicate evolutionary testability.

Contents

1 Introduction **8**

2 The scope of this thesis **11**

2.1 Problems with existing timing analysis techniques 11

2.2 Overview on evolutionary algorithms 12

2.3 Genetic operators 14

2.4 Dynamic timing analysis with evolutionary algorithms 15

2.5 The problem of evolutionary testing 18

2.6 Software complexity 20

3 Complexity measures for evolutionary testability **22**

3.1 A basic evolutionary testability measure 24

3.2 Experimental setup 26

3.3 Results of evolutionary testing 35

3.4 A measure for parameter interdependence 38

3.5 A measure for small or single-value domains 39

3.6 Discussion of the combined measure 42

3.7 Individual properties for evolutionary testing complexity 46

4 Validation of the individual measures **54**

4.1 Axioms which are fundamental to all measurement 54

4.2 Axioms which are specific for the scale type 61

4.3 Axioms which are specific to the model 62

4.4 Prediction models for evolutionary testability 63

4.5 Composition operations for the valid measures 68

5 Conclusions and further research **71**

List of Figures

1 Hypothetical relation between the complexity of a test object and the outcome of evolutionary testing for B/WCET testing (simplified). The success rate of the testing strategy is expected to decrease with an increase in 'complexity' of the test object. 19
2 Example control-flowgraph of an *insertion sort* algorithm. Nodes are indicated by a circle (procedure node), a double circle (start and stop node), or a bullet (predicate node). Arcs are indicated by an arrow which shows the direction of the control-flow. 25
3 The test objects of table 2 according to the Measure ESS-BAND. Displayed is the performance of ET as average of BCET/WCET for *target* (average of columns WCET and BCET (Avg) under *target* in table 2). The average of B/WCET is chosen as ESS-BAND only indicates the effect of nesting/sequencing on the performance of ET. 37
4 Extension of the measure ESS-BAND. EXT-ESS-BAND captures the effect of parameter interdependence more accurately. This is caused by multiple parameter references in predicates. 41
5 Second Extension of the measure ESS-BAND. EXT2-ESS-BAND additionally captures the effect of small or single input domains. This is based upon the minimum probability with which a branch is taken at a decision. 44
6 Graph of the measure DPC mapped against ET performance (*target* %) according to table 7. 58
7 Graph of the measure DN mapped against ET performance (*target* %) according to table 9. 61

8 The prediction of evolutionary testability through the measure DN, predicted performance (predicted target in %) against the actual performance (target in %) for all test objects , $b_0 = 88.4288$, $b_1 = -1.18214$ (DN), prediction model $f(x) = b_0 + b_1 * x$, correlation coefficient $r = 0.582472$. 64

9 The prediction of evolutionary testability through the measure N_p, predicted performance (predicted target in %) against the actual performance (target in %), $b_0 = 95.6623$, $b_1 = -4.5823$, for the model $f(x) = b_0 + b_1 * x$, correlation coefficient $r = 0.625125$. 65

10 The prediction of evolutionary testability through the combination of the measures DN, DIR, ADIR and DPC, predicted performance (predicted target in %) against the actual performance (target in %). 67

11 The prediction of evolutionary testability through the combination of the measures DN, ADN, DIR, ADIR, DPC and ADPC predicted performance (predicted target in %) against the actual performance (target in %). 68

List of Tables

1 Description of the test objects including input vector size. Their source codes are displayed in the appendix. 33

2 Example test programs with measures number of nodes (N), number of predicate nodes (N_P), Belady's BAND measure, the new ESS-BAND measure and the results of the ET process. Avg is the average of BCET and WCET for the coverage criterion *target*. The results are averaged over 10 repetitive trials. 36

3 Values for the extension of the measure ESS-BAND which captures parameter interdependence (EXT-ESS-BAND). The modules are sorted according to EXT-ESS-BAND. 40

4 Values for the extension of the measure ESS-BAND which captures 'probability complexity' (EXT2-ESS-BAND). The modules are sorted according to EXT2-ESS-BAND. 43

5 Individual measures for evolutionary testability. The modules are ordered according to ET performance (average *target* %). 48

6 Performance of ET (*target* in % for WCET) on a *selection sort* algorithm. The values illustrate the effect of different data types on the outcome of ET. 53

7 Illustration of the effect of the measure DPC on the performance of ET (*target* %). Additional measurable program properties are unchanged (IS=3.0, DIR=1, ADIR=1, DN=6, ADN=2). The corresponding graph is displayed in figure 6. 58

8 The validation of ADPC shows that it is no measure for evolutionary testability. It is apparent that ADPC(P_1) $\geq$ ADPC(P_2) but $P_1 <^{\tau} P_2$. 59

9 Illustration of the effect of the measure DN on the performance of ET (*target* %). This is in combination with the measure DPC for increasing decision nesting of the test program. Additional measurable program properties are unchanged (IS=3.0, ADIR=1). The corresponding graph is displayed in figure 7. 60

10 The validation of ADN shows that it is not fit for indicating evolutionary testability. A scenario may be constructed easily with $\mathrm{ADN}(P_1) \geq \mathrm{ADN}(P_2)$ but $P_1 <^{\tau} P_2$. 60

11 Rank correlation for the individual measures; data taken from table 5. 66

12 Multiple regression for the measures DN, DIR, ADIR and DPC. The graph of the prediction is displayed in figure 10. 66

13 Multiple regression for the measures DN, ADN, DIR, ADIR, DPC and ADPC. The graph of the prediction is displayed in figure 11. . 67

1 Introduction

The operation speed of real-time systems is critical as they must produce results according to a predefined time schedule which is determined by the specification [47] of the system. The knowledge about the best- and worst-case execution time (BCET/WCET) is an additional requirement for the correct operation of real-time software. Assessing that all deadlines during program execution will be met is an essential and difficult task in the development of such systems. B/WCET analysis demands full knowledge about the behaviour of the underlying hardware, the scheduling and timing of the operating system, and the execution time of the real-time software under development.

Software testing is a widely used and accepted technique for verification and validation [44], and it is considered the ultimate review of the specification, design and implementation of a software product [33]. It can be used to generate modes of operation which show that the software is conforming to its specification and to support the confidence in the safe and correct operation of the system.

The process of testing can be regarded as a search or optimisation process. The goal of this process is to find test cases which comply to a specific test criterion. Evolutionary testing (ET) [51] is a new testing technique based upon the application of genetic algorithms (GA) [8], evolution strategies (ES) [42], genetic programming (GP) [22] or simulated annealing [49]. All these methods which are referred to as evolutionary algorithms (EA), are optimisation techniques. For testing the timing behaviour of systems, ET can be used to generate test cases automatically [19, 21, 46, 51] which are most likely to produce the shortest or longest execution time on the test object or violate its timing schedule. Static analysis is the traditional technique for determining the timing behaviour of software [3, 36, 47], although, dynamic techniques are increasingly used to overcome limitations of the static techniques. Evolutionary testing has been successfully applied to supplement or verify existing static analysis techniques [37, 28], and it has already been used for structural testing [46, 48].

As a typical search or optimisation technique, evolutionary algorithms reveal no information on how close the best found solution is compared with the actual optimal solution. This lack of quality assessment might have inhibited a widespread use of evolutionary testing. Experiments with the technique performed for this and previous work [9, 10, 11] revealed a relationship between the success of the search technique to find the optimal (or near optimal) solution and the complexity of the test object.

The work introduced in this thesis concentrates on performance assessment of evolutionary algorithms applied to B/WCET testing. The aim of the project is to

- investigate the effect of complexity on the success of evolutionary testing in finding the B/WCET, and to
- define and validate complexity metrics which can assess and possibly predict the likely success of evolutionary testing.

The emphasis here is on the application to timing analysis of real-time systems. This is relatively easy to perform, as the measure of how well the generated test cases comply with the test objective is straightforward. It is simply the time it takes to execute the test module with the input which is generated by evolutionary testing.

The following chapter (chapter 2) provides all required background information (scope) for this thesis. It discusses the problems of static techniques which have been traditionally used for timing analysis and introduces testing techniques based upon evolutionary algorithms (2.1). All relevant information on evolutionary testing is given (2.2, 2.3), and the fitness function is described (2.4). This chapter concludes with a description of the main problem of assessing the outcome of evolutionary testing (2.5). Chapter 3 introduces and discusses all measures for assessing the outcome of evolutionary testing. These have been developed over the course of this project and are relevant for this thesis. The validation of some of these measures confirms that three measures can serve as valid indicators for the expected success of evolutionary testing (chapter 4). Some of the introduced measures are used to

develop prediction systems for evolutionary testability in section 4.4. General considerations concerning this work and an outlook on further research conclude this thesis in chapter 5.

2 The scope of this thesis

2.1 Problems with existing timing analysis techniques

The traditional technique for analytical determination of best- and worst-case task execution time is to initially determine the required time for executing each basic block (a segment of straight-line code), and then to combine the execution times of individual basic blocks into a cumulative total which represents the execution time for the entire task [30]. This technique is called static timing analysis, since it statically analyses the code of a task for possible execution paths, and models the timing of the code on the target hardware without executing the task [37].

Puschner [36] describes the *MARS-C* methodology which, along with extensions and alterations [34, 35, 38] for increased accuracy, implements such a static analysis technique. Here, several criteria are introduced to which software must comply in order to permit the application of static analysis. These are the knowledge about the maximum iterations of loops and the absence of recursion and function variables. Puschner and Vrchoticky [39] outline further problems with static execution time analysis. These are for example the need for information about possible sequences of program actions during execution, such as path information and code annotations. Static timing analysis uses no information about input parameter values, since the source code of a program does not provide these. This is a major disadvantage of this technique. In order to overcome these limitations, additional information must be provided through human interaction. For static analysis it is essential that this auxiliary information is available and complete [30]. This is emphasised by Nilsen and Rygg [30], Park [32], and Puschner and Schedl [38]. Extensive human interaction to provide this missing information makes static B/WCET analysis difficult, expensive and unreliable [37].

These problems and the fact that there is no commercial solution for static B/WCET analysis, and little hope that such a product will be on the market in the foreseeable future [39, 37], has shifted research effort towards dynamic timing

analysis techniques. In contrast to the static approaches, these dynamic approaches only concentrate on the execution of the real-time task under scrutiny and aim to generate modes of operation which minimise or maximise the timing or violate the timing schedule of a program. In this scope, timing analysis can entirely be regarded as an optimisation problem with the minimum and maximum execution times as optimisation criteria.

The application of meta-heuristics to structural test-case generation for software has been subject of several investigations, such as Xanthakis *et al.* [56], Hunt [19], Sthamer [46], Watkins [50], Jones *et al.* [20, 21] and Holmes [17]. More recently, this technique has been extended to dynamic execution time analysis of real-time software. For instance Wegener *et al.* [52], and Wegener and Grochtmann [51] apply genetic algorithms to finding the longest and shortest execution times of real-time modules automatically. Müller and Wegener [28] compare dynamic timing analysis based on meta-heuristics and static analysis, and suggest the dynamic technique as a supplement for the static technique. A similar approach is taken by Puschner [37], where dynamic timing analysis based on genetic algorithms is used to test the outcome of a static timing analysis technique. O'Sullivan *et al.* [31] develop a new stopping criterion for the dynamic testing of software with evolutionary algorithms based on cluster analysis.

Wegener and Grochtmann [51] have introduced the terminology of 'evolutionary testing' for the dynamic and automatic generation of test cases through an evolutionary algorithm.

2.2 Overview on evolutionary algorithms

Evolutionary algorithms are sophisticated search or optimisation techniques, loosely related to the mechanisms of natural evolution which is based upon reproduction and selection. They perform on populations of binary strings (GA) or real numbers (ES) which represent possible solutions to an optimisation or search problem. EA recombine and mutate strings and the resulting new strings are selected as a new

generation according to a fitness function. The operation of evolutionary algorithms can be divided into three steps: Reproduction, mutation and selection.

The parameters of an optimisation problem can be encoded as a (binary) string, the so-called chromosome, with each string consisting of representations of the parameters which are to be optimised. Each set of parameters, the so-called individual, is represented by a different string within a population. During reproduction, pairs of individuals are selected for recombination and some parts of their chromosomes form a new individual. This process is called crossover and is controlled by the recombination operator and the crossover probability (p_c). Individual bits within the new chromosome are then mutated according to the mutation probability (p_m). The resulting new individuals are tested and their fitness evaluated through the fitness function. This function assesses how well each individual set of parameters solves the original problem. Usually, fitter individuals have a higher chance of being selected for recombination. This is controlled by the selection operator. The fittest individuals remain in the population and build the basis for the next generation. This process can be represented by the following pseudo-code (P, P1, P2, P3 are sets of possible solutions):

```
begin ea
  initialise (P);
  while not break_condition do begin
    P1 = selection (P);
    P2 = recombination (P1);
    P3 = mutation (P2);
    P  = fittest (P3,P);
  end
end
```

The process is repeated over many generations until the stopping criterion is satisfied; for example, if a predetermined number of generations does not improve the fitness, or simply if a predetermined number of generations is reached.

The evolutionary process generates new solutions based on information of existing solutions, so that the population is likely to consist of fitter individuals after many generations. These individuals are able to yield better results and represent better solutions [16].

2.3 Genetic operators

A population of individuals can be regarded as a mapping of the problem combination space with each individual occupying a distinct location in this (multidimensional) volume. The number of parameters to be optimised determines the size of the search space. Large and complex search spaces demand extended populations in order to increase the sampling of the overall volume. The **population size** should be large enough to provide a good sampling accuracy of such a search space. Larger populations lead to better ultimate performance because of the larger pool of diverse samples in such a population [8]. However, memory limitations and the time it takes to evolve a population may put an upper bound on the maximum number of individuals.

Crossover and **mutation** control the degree of exploitation and exploration of the search space by a population [16]. Crossover creates new individuals (offspring) by recombining the genetic information contained in two parents' chromosomes. The locations on the chromosome where crossover appears is determined randomly. The number of crossover points defines the disruption of the genetic information in the original individuals. *One-* and *two-point crossover* operators tend to keep offspring similar to their parents [16], whereas *n-point* ($n \gg 2$) and *uniform crossover* generate offspring which can differ from its parents quite considerably. Uniform crossover is the most disruptive crossover operator as it creates on average $L/2$ crossover points for L binary locations in the chromosome. This is for crossover probability $p_c = 0.5$. Traditionally, high disruption was considered detrimental for an evolutionary process [16], although more recent work suggests otherwise. De Jong and Spears point out that uniform crossover is highly successful on insufficiently sampled search spaces

[6], and it performs well regardless of the distribution of important values on the chromosome, since it is unbiased [45]. The degree of disruption can be adjusted by the crossover probability. Recombination alone is insufficient for an evolutionary process as it is only able to produce offspring by exploiting the volume of the search space which is initially represented by the population at startup. Mutation is the only force to direct the search into new areas of the search space. It is usually applied at a low rate, for example $p_m = 0.001$ [16].

The **selection operator** determines the two individuals to be recombined. *Tournament selection* [27] in combination with *rank-based fitness allocation* [55] provides a robust and easy-to-implement selection method. Selection pressure is adjusted by a single parameter, the tournament size. Tournament size = 1 implements random selection for recombination, and values greater than one make the selection increasingly 'elitist'. The chance of an individual being selected for recombination only depends on its position (hierarchy) within the population and not on the fitness. The fitness merely determines the position of the individual. This reduces the probability of generating a 'super-individual' which is then constantly selected for recombination with the more traditional 'roulette-wheel-based' selection procedure [8].

2.4 Dynamic timing analysis with evolutionary algorithms

To analyse the B/WCET of a module for a real-time system, an evolutionary algorithm can be applied as a test-case generator to produce populations of input parameter sets. These input parameters determine the dynamic behaviour of the test object. Each individual's chromosome is equivalent to one set of input parameters, which represents one single process. The **fitness function** executes the test program with the provided set of inputs and measures the execution time [52]. This time is the only information to guide the optimisation process. Individuals which produce longer execution times (or shorter times for BCET) are favoured by the selection of the evolutionary algorithm. The recombination of highly fit individuals

tends to produce highly fit offspring [16] which is allowed to 'spread' through the population. Subsequent generations therefore consist of 'fitter' individuals which produce shorter or longer execution times.

For timing analysis, time is the only objective measure (fitness) for a chromosome's ability to solve the optimisation problem. Program timing can therefore be considered as a primary fitness function for the purpose under consideration. Apart from timing, object code annotations and source code annotations can be used to determine the fitness value for the execution of a program. In the following, these techniques are discussed in more detail:

- **Timing**. Time is provided by the real-time clock on a computer system. Using time as fitness seems to be straightforward at a first glance. Although, this simplicity is traded against non-determinism. The way timing is implemented on most computer systems prohibits accurate measurements. The system's real-time clock generates an interrupt whenever a given time interval has expired. This is typically set to 100 Hz on most PC-based systems and it causes the processor to increment a counter. This requires extra CPU time, and extra time is used for taking the actual measurements (through system calls). Additionally, very small tasks can fall between two timer calls, so that they cannot be measured. Increasing the granularity of the timer does not increase the quality of the outcome. This gain in accuracy of the timer is levelled out by the fact that the processor must respond to more interrupts. Therefore, timing may produce a different outcome for consecutive executions of the same test case, and this is disadvantageous for experimental work.

- **Object code annotation**. Deterministic timing may be achieved through an execution time monitor such as Rational's *Visual Quantify*. This tool implements object code annotation and a 'look-up table' for the processor specification. This contains the number of machine cycles for each machine instruction depending upon the values of its operands. The tool predicts the processor cycles according to these annotations for a given program input. Harmon *et*

al. [14] and Ramsey and Fernandez [40] provide details on how this technique is implemented.

- **Source code annotation**. Another deterministic approach is to instrument the source code with annotations which enable high-level language monitoring. This can be implemented through procedure calls which are inserted into the source code at locations where they correspond to the longest or shortest path. This technique is mainly used for structural testing [17, 46]. It can be applied to B/WCET testing, if the assumption is made that source code annotations can represent the execution time on a distinct section of code. The outcome is not real-time but a more abstract value which indicates whether (to which extent) the execution complies to the target for which the source code has been instrumented. The longest and shortest paths must have been determined in advance. The fitness function only generates information whether these paths have been executed during testing. The way this is implemented is described in later paragraphs (section 3.2). In this work, this method is preferred over the object code insertion technique for reasons which are discussed later (section 3.2).

The work presented in this thesis concentrates entirely on software issues of timing analysis. It must be noted that the effects of the hardware on timing, for example caching and pipelining, are not considered. These hardware technologies are available on all modern computer systems, and they make timing analysis extremely difficult. They have no effect on the results produced for this work through the way the fitness function is implemented with the source code annotations.

The use of annotations as fitness and the produced outcome are practically infeasible to apply to a given real-time system for determining best- or worst-case timing behaviour. The real-time clock is the only device to control timing on an operational system. The advantages of annotations are therefore more of academic nature. Additionally, the tested program is not the same as the original program.

This must be regarded as being critical for testing, since different programs are tested and released.

2.5 The problem of evolutionary testing

Evolutionary testing is a typical search or optimisation technique. Such techniques reveal no information on how close the best solution of the search process is compared with the actual optimal solution – or an acceptable sub-optimal solution – which is to be found. The only available information is the distribution of local optima in the search volume as shown by O'Sullivan *et al.* [31]. This 'reticence' of search techniques makes an assessment of the outcome of evolutionary testing nearly impossible. Previous research has only focused on a comparison of evolutionary testing with existing techniques [28, 51, 52], and only made statements about the quality of evolutionary testing in respect to these existing techniques. No effort has been committed to enable an assessment of evolutionary testing as a 'stand-alone technique'.

In the case of B/WCET testing, the evolutionary optimisation process attempts to find input situations which minimise or maximise the execution time, or violate the timing specification. The result of this search is the found execution time. There is no information on how close this estimate comes to the actual minimum or maximum execution times of the test object. The only available information is that this value lies between the shortest and the longest time, so that equation 1 always holds:

$$BCET_{Actual} \leq BCET_{Found} \leq WCET_{Found} \leq WCET_{Actual} \tag{1}$$

This equation states that the outcome of evolutionary testing tends to underestimate the execution time for WCET, and to overestimate the execution time for BCET. Evolutionary testing can never generate results which are greater or less than the true extreme times, so that it can be seen as an over-optimistic approach. The outcome of a static analysis technique tends to be pessimistic as it approaches the actual extreme execution times from the 'opposite directions'.

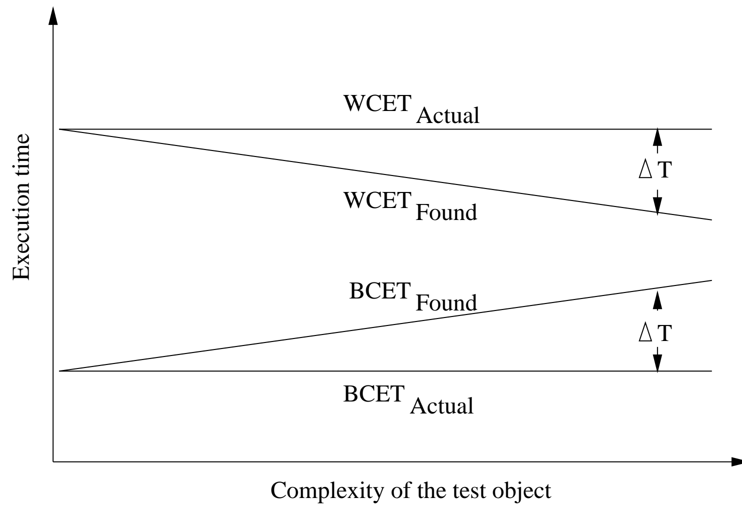

Figure 1: Hypothetical relation between the complexity of a test object and the outcome of evolutionary testing for B/WCET testing (simplified). The success rate of the testing strategy is expected to decrease with an increase in 'complexity' of the test object.

The motivation for this work comes from the need to assess the outcome of evolutionary testing. This would permit its application to programs where its outcome is of high expected quality. For programs, for which this expected quality is low, the more traditional static techniques, or a hybrid of these two techniques can be applied.

Initial experiments with evolutionary testing performed for previous work [9, 10, 11] identified severe difficulties in detecting the best- or worst-case execution times for 'complicated' test programs. The evolutionary approach fails to produce input situations for some of the test objects, which directs the control-flow into the shortest or longest sections of the program. For 'less complicated' programs, evolutionary testing is able to generate input situations according to the test target (B/WCET). This observation seems most trivial, and it is common-sense to assume that the testing effort is increased if the complexity of the test object is increased [3, 24]. This

relationship between the 'complexity of the test object' and the effort of testing can be extended into assuming that the quality of the outcome of evolutionary testing should decrease with increased complexity of the test object. This hypothesis is illustrated in figure 1. It was assumed that:

- The testing process should be able to easily find input parameters according to the extreme execution times for a 'simple' program.
- The outcome would become more and more unreliable, if the complexity of the test object was increased.
- If the properties of a test object which make it difficult for evolutionary testing, to produce meaningful results, could be identified and measured, and
- this measure could be mapped against the performance of the testing technique, then
- it would be feasible to give an estimate on how successfully evolutionary testing would perform on a given test object.

The aim of this thesis is the

- derivation and validation of a complexity measure. This should be able to assess or estimate how successfully evolutionary testing generates test cases according to the test criterion. In order to achieve this,
- the hypothetical relationship between the complexity of a program and the quality of the outcome of evolutionary testing must be investigated, and
- program properties must be identified which inhibit evolutionary testing.

2.6 Software complexity

Software complexity is difficult to define. Many different interpretations can be found in the literature. Definitions range from 'difficulty to maintain, change and

understand software' [58] to 'amount of information which must be understood and processed in order to produce, use, maintain and change software' [23]. Many definitions of software complexity emphasise cognitive complexity which indicates the effort to understand the software [7]. Most software complexity metrics, including the standard metrics of Halstead [12], the measures of Myers [29] and Harrison [15], concentrate on complexity as understandability. The measure of McCabe [24] which was intended as a testability measure, is now mainly used as an overall complexity measure with more emphasis on cognitive aspects. Software testability which can be seen as the degree of difficulty to test software, is one aspect of software complexity, and most definitions of software testability only focus on one of its properties. For example, Bache and Müllerburg [1] use the terminology to calculate the number of required test cases for a test object for satisfying a specific test strategy. They define this as the effort to test software, although it is not the same as 'software testability' [33].

3 Complexity measures for evolutionary testability

A very specific definition of an objective complexity measure, and consequently testability, is required for evolutionary testing. A possible definition of complexity could be 'the difficulty for the testing technique (EA) to generate test cases which satisfy the test criterion'. On a module level, complexity may be seen as the combination of the most important program properties which make it difficult for an evolutionary algorithm to generate input parameters corresponding to the test objective. In this instance, it is finding the best- or worst-case execution time. Evolutionary testability can be defined as the degree of difficulty to successfully apply evolutionary testing to a particular software. Here, successful application means the generation of test cases which result in the longest or shortest execution of the program. In this case, complexity must not be understood from the human perspective but primarily from the perspective of the (automatic) testing methodology which is an optimisation technique.

The execution of a program under test ideally covers every single entry-exit path. This is every path whose execution starts at the start node of a program's flowgraph and stops at its stop node. The result is full path-coverage. The term flowgraph is defined in section 3.1. If all paths can be covered (executed) by ET, then their execution times can be determined. If input for all paths can be easily generated, the EA can optimise the execution times along these paths, then the outcome of ET is of high expected quality.

Full path-coverage is generally impossible to achieve [3]. Therefore, weaker criteria are typically accepted. ET must ensure that all sections of a program are 'inspected'. If a branch is not executed, its execution time cannot be determined. In this case, the outcome of ET is of low expected quality. The overall optimum may be located on this branch, but it is never found. Full branch or statement coverage is therefore a minimum requirement for ET to generate useful outcome. This assumes that every single statement has been executed at least once. Additionally, if the number of iterations in a loop depends upon input, ET must find the highest

possible number of iterations in case of WCET analysis, and the lowest possible number of iterations for BCET. Beizer [3] discusses many more coverage criteria.

The testing process must be able to easily examine all possible paths in order to 'decide' which of them are most promising for the required testing objective. The difficulty of generating input according to this requirement is determined by the decisions in the test program. Here, 'difficult' decisions create serious problems for evolutionary testing. For instance, these can be decisions which create small domains, so that one branch at the decision is only taken with a very low probability. The following list outlines properties of programs which have been identified as creating most problems for an evolutionary testing process to generate test cases according to B/WCET. They can be used to describe the terminology of 'evolutionary testability complexity' [11]:

- *Small input domains or single-value domains.* These are caused by decisions which execute one branch with a very low probability. This restricts the ability of the EA for large search spaces as it is unlikely to generate the required value by chance.
- *Parameter dependent loops.* Loops whose number of iterations depend upon input variables are equivalent to decisions with a *single-value domain.* Here, the EA must generate input which leads to the lowest or highest number of iterations for the loop.
- *High parameter interdependence (data dependence) and/or large input vectors.* High interdependence may either be caused by decisions which require some of the input to be in a specific relation, for example a specified pattern, in order to lead the program flow into a distinct branch, or by calculations on input. In the second case the values of the input variables determine the time it takes to perform the calculation. This depends on the system architecture.
- *Nesting and sequencing.* Combinations of all previous items.

The following section suggests and discusses a basic model for measuring 'evolutionary testing complexity' or evolutionary testability.

3.1 A basic evolutionary testability measure

Determining the nesting or sequencing of a program can be regarded as initial step towards a structural predictive measure of ET performance. A number of structural measures exist in the literature (compare [58]). For example, McCabe's 'cyclomatic complexity' [24] is one of the most known and used complexity measures based on a control-flowgraph [58].

Control-flow measures are typically modelled with a directed graph (control-flowgraph) consisting of a set of nodes and a set of arcs. Each node corresponds to a program statement, and each arc indicates the control-flow from one statement to another. The 'in-degree' of a node is the number of arcs arriving at the node, and the 'out-degree' is the number of arcs that leave the node. The 'stop node' has out-degree zero. Nodes with out-degree equal to one are termed 'procedure nodes', all other nodes are called 'predicate nodes' [7, 58]. Each conditional expression in a program is represented by a separate predicate node. An example flowgraph of a simple *insertion sort* algorithm is displayed in figure 2.

One of the first nesting measures which is based on a control-flowgraph was proposed by Belady and Evangelisti [4, 5, 58]. This measure was initially considered useful for determining evolutionary testability complexity, as a weight for each node in the control-flowgraph is introduced which can be used to specify the complexity of the node more distinctively.

Zuse [58] defines this measure as:

$$\text{BAND} = \sum_{i=1}^{N} L(i) * n(i) \qquad (2)$$

where N is the number of nodes in the flowgraph, $n(i)$ is the weight of node i (typically $n(i) = 1$) and $L(i)$ is the nesting level of node i. The nesting level of a node is defined by Zuse [58]: To the starting node s of the flowgraph, the nesting

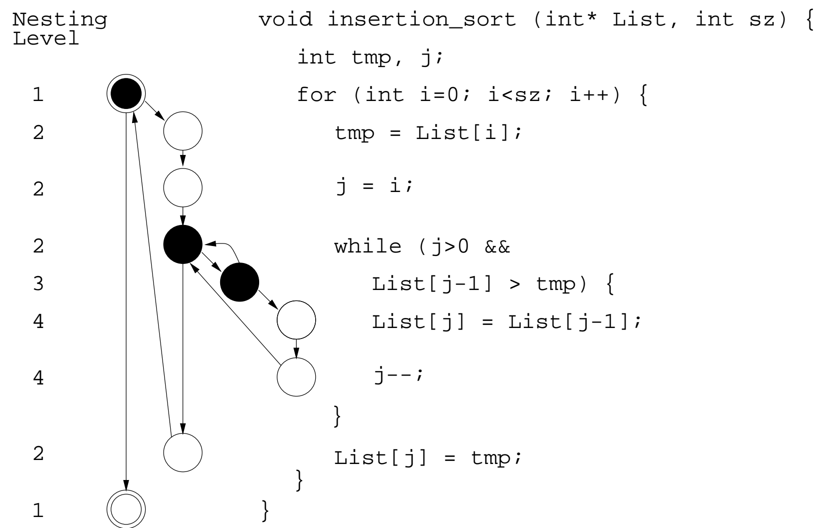

Figure 2: Example control-flowgraph of an *insertion sort* algorithm. Nodes are indicated by a circle (procedure node), a double circle (start and stop node), or a bullet (predicate node). Arcs are indicated by an arrow which shows the direction of the control-flow.

level $L(s) = 1$ is assigned. If a node n_1 is a conditional transfer node (n_1 is a predicate node) and node n_2 can be reached via node n_1, without crossing another predicate node, then the nesting level of n_2 is $L(n_2) = L(n_1) + 1$. If node n_1 is at level $L(n_1)$ and it is simply followed sequentially by node n_2 (n_1 is a procedure node), then the nesting level of node n_2 is $L(n_2) = L(n_1)$. Figure 2 additionally displays each node's nesting level for the example *insertion sort*.

The measure BAND captures the total nesting and sequencing of a program. It must be noted that all subsequently introduced measures are new measures which can be considered original contribution to the field of software complexity measures or software testability.

From the measure BAND, the measure Essential BAND (ESS-BAND) is derived as a new measure which only considers the predicate nodes in a flowgraph. This new measure defines the essential nesting – or decision nesting/sequencing – of a

program. It is defined as

$$\text{ESS-BAND} = \sum_{i=1}^{N_P} L(i) * n(i) \tag{3}$$

with N_P as the number of predicate nodes (in branch- or loop-conditions) in the flowgraph, and $n(i) = 1$. This measure can be regarded as a simple basic structural measure which is sufficient to indicate evolutionary testability for many test objects.

ESS-BAND concentrates on the decisions (predicates) in a program and ignores the sequences. Once the program-flow enters a distinct branch in the program, all sequential statements on that branch will be executed. Sequences are also important for timing analysis as the longest path is consisting of sequential statements whose timing must be optimised. This is neglected, in order to keep the proposed model of complexity as simple as possible. The measure ESS-BAND can be seen as an improvement over the original measure BAND for the use with ET, and for the proposed model. A discussion on the measure ESS-BAND is given in section 3.3. This follows the introduction of the experimental setup.

3.2 Experimental setup

The evolutionary algorithm used for all experiments is intentionally very simple. The following pseudo-code represents the used algorithm:

```
ea begin
  create population of size parents + children;
  foreach (parent) do begin
    initialise individual's chromosome randomly;
    initialise source code annotations;
    call test_object with chromosome;
    fitness of the individual = coverage_counter;
  end-foreach
  while (NOT stopping_criterion) do begin
    foreach (parent / 2) do begin
```

```
        select parent_1 from tournament;
        select parent_2 from tournament;
        child_1 = parent_1;
        child_2 = parent_2;
        recombine child_1 and child_2 with uniform crossover;
        mutate child_1;
        mutate child_2;
        for (child_1 and child_2) do begin
          initialise source code annotations;
          call test_object with chromosome;
          fitness of the individual = sum of source code annotations;
        end-for
      end-foreach
      sort population according to fitness;
      determine stopping_criterion;
    end-while
  end-ea
```

There exist specific evolutionary operators for some classes of test objects which can improve the overall outcome. For example, Müller and Wegener [28] use a recombination operator which improves the performance of ET for sorting algorithms. Although, this prohibits a comparison of the performance of the search technique on the different test objects. The following genetic operators were used throughout the experiments:

- Binary string as chromosome [8]. The chromosome is an exact mapping of the memory for each input parameter set. This is the traditional methodology of genetic algorithms as initially introduced by Holland [16].
- Population size equals 40 individuals. The algorithm always keeps the best 40 individuals. As discussed previously in section 2.3, the population size is

a trade-off between sampling accuracy and performance limitations. Deciding upon the population size involves some experimentation with the problem at hand in which the population size is incrementally decreased until the performance of the algorithm deteriorates considerably.

- Tournament selection. The tournament size depends upon the population size and is set to four individuals for this particular instance. Tournament selection is easy to implement and provides the advantage that the 'selection pressure' can be adjusted easily [27]. The chosen value implements selection which is more 'elitist' than random.

- Discrete recombination, uniform crossover, $p_c = 0.5$ This has become a standard for genetic algorithms, and it performs well regardless of the distribution of important features of individuals as previously stated in section 2.3 [45].

- Low constant mutation rate, $p_m = 0.001$. This value has been used successfully on many problems in the literature, e.g. Goldberg [8].

- Rank based fitness [55]. This was preferred over the traditional 'roulette-wheel-based' fitness assignment for reasons which are mentioned in section 2.3 (page 14).

- Random initialisation of the chromosomes [8]. A standard procedure which simply starts the search at arbitrary locations in the search space.

These values remained constant for all experiments. The evolutionary testing process was stopped when after 200 consecutive generations no further improvement in fitness could be observed. This is a simple stopping criterion but it is sufficient for the experiments performed for this work. O'Sullivan *et al.* discuss the properties of more sophisticated stopping criteria for evolutionary testing [31].

The fitness function was implemented by source code annotations as described in section 2.4. In the remainder of this thesis, 'code annotation' is used to mean

'source code annotations'. These code annotations are simple procedure calls which increment coverage counters. They can be used for two different coverage criteria:

- Coverage of the code annotations which is referred to as *target*. This is the total number of code annotation executions compared with the maximum number of code annotation executions for the extreme execution path. The way in which these code annotations are implemented is illustrated below, through the *insertion sort* algorithm which was used before. For worst-case execution time as test criterion

  - 100 % *target* means that all code annotations which result in the worst-case timing path have been executed. This includes multiple executions of the same code annotation.
  - 0 % *target* means that no code annotations which result in the worst-case timing path have been executed.

  For best-case execution time as test criterion

  - 100 % *target* means that all code annotations which result in the best-case timing path have been executed. In this work, the BCET and WCET paths are identified manually. This is described later.
  - 0 % *target* means that none of the code annotations which result in the best-case timing path have been executed.

- Coverage of all the source code statements which is referred to as *coverage*. This is the number of executed source code statements compared with the maximum or minimum number of executed code statements for the worst-case or best-case timing path. For worst-case execution time as test criterion

  - 100 % *coverage* means that all statements which are necessary to produce the worst-case timing path have been covered. This includes multiple executions of the same statement.

- 0 % *coverage* means that none of the statements which are necessary to produce the worst-case timing path have been covered. This can never occur in practice.

For best-case execution time as test criterion

- 100 % *coverage* means that only those statements have been covered which are necessary to produce the best-case timing path,
- 0 % *coverage* means that none of the statements which are necessary to produce the best-case timing path have been covered. This can never occur in practice.

The following source code sections represent a typical test harness for the previously used *insertion sort* example. Typically, each code section with variable coverage is annotated by the 'COVER' procedure. This counts the number of executions of each of these code sections. Their values are stored in a list. The increment can be used according to the coverage criterion. For the coverage of the code annotations (*target*) it equals 1. For the coverage of the source code statements (*coverage*), it is set to the number of source code statements in the section it stands for. In this example, it would be set to 3.

```
const int variable_paths = 1;
int cover [variable_paths];
void COVER (int branch, int increment) {
  cover[branch] += increment;
}
```

There is only one section with variable code coverage contained in this example. It is marked by comments in the source code. The number of executions of all other code sections depends on the number of elements in the list which are to be sorted. Their coverage is always constant. This is determined by the constant *sz* in the example.

```
void insertion_sort (int* List, int sz) {
  int tmp, j;
  for (int i=0; i<sz; i++) {       /* fixed coverage = sz + 1 */
    tmp = List[i];                 /* fixed coverage = sz     */
    j = i;                         /* fixed coverage = sz     */
    while (j>0 && List[j-1]>tmp) { /* variable coverage       */
      COVER (0,1);                 /* code annotation         */
      List[j] = List[j-1];         /* variable coverage       */
      j--;                         /* variable coverage       */
    }
    List[j] = tmp;                 /* fixed coverage = sz     */
  }
}
```

The fitness function resets all counters and executes the test object with the provided input (chromosome). During execution, the executed paths are monitored according to the source code annotations, and the sum of these is eventually returned as fitness value.

```
double fitness_function (unsigned char* chromosome, int sz) {
  int total_paths = 0;
  memset ((char*)&cover,sizeof(int)*variable_paths,0);
  /* test object start */
  insertion_sort ((int*)chromosome, (int)sz/sizeof(int));
  /* test object stop  */
  for (int i=0; i<variable_paths; i++) {
    total_paths += cover[i];
  }
  return (double) total_paths;
}
```

If the implementation of the fitness function is considered, it becomes apparent that the maximum possible values for *target* and *coverage* must have been determined prior to the experimental assessment of the ET performance. This is only feasible through the use of the previously introduced annotation technique. Neither timing nor the use of object code annotations which calculates machine cycles, can be used to determine the best- and worst-case behaviour of the software under consideration. Timing never produces the same outcome for the same test-case, and timing and object code annotations additionally include hardware behaviour which cannot be predicted from the source code of the test program. The use of source code annotations only requires the longest and shortest paths to be determined according to the structure of the test program. This is based upon the simple assumption that more source code statements typically result in longer execution times. This is an over-simplistic view, and it is certainly completely wrong on modern hardware architectures. This model is sufficient, since the measure introduced only assesses the structural complexity of a test object. Another positive effect of source code annotations, which makes this technique so attractive for experimentation, is that the longest and shortest paths can be determined arbitrarily. For ET, it does not matter where the code annotations are placed. It simply assumes that the path of interest is where it finds and executes the code annotations, since this is all the information it is faced with. Therefore, even if the extreme execution paths may not have been determined correctly prior to the experiments, because of the extreme complexity of a test object, the experimental outcome is still valid and useful.

The test objects are modules of different complexity taken from different applications. Table 1 displays their names, functions and input sizes. Their source codes are displayed in the appendix. It is important to note that their input vectors are of similar size, so that the experimental outcome is comparable. The size of the input vector matters for the quality of the outcome of evolutionary testing. It defines the size of the search space as discussed in section 2.3 (page 14). The extent to which

Module	Descriptions	Input (bytes)
delevat	Elevate the degree of a Bézier line interpolation.	28
bhorner	Bézier line interpolation (Horner-scheme-like).	28
bcastel	Bézier line interpolation (De Casteljau).	28
bs1	Bubble sort.	1024
bs2	Bubble sort.	1024
is	Insertion Sort.	1024
polex	Contour plotting, noise filter.	1080
polex1	Contour plotting, redesigned polex.	16
dzz	Contour plotting, noise filter.	1080
dzz1	Contour plotting, redesigned dzz part 1.	1080
dzz2	Contour plotting, redesigned dzz part 2.	1080
dzz3	Contour plotting, redesigned dzz part 3.	1080
exp	Contour plotting, filter.	1080
diff	Robot vision, difference of two picture frames.	1024
sobel	Robot vision, filter.	1025
min	Robot vision, filter.	1024
median	Robot vision, filter.	1024
gstretch	Robot vision, filter.	1024
train	Train control system module.	672
train1	Redesigned train module.	656
dummy1/2	Artificially designed modules.	1080

Table 1: Description of the test objects including input vector size. Their source codes are displayed in the appendix.

the input size of a test object affects the evolutionary search is discussed in section 3.7 (page 49).

Table 2 displays for each test object the number of nodes, number of decision nodes, measure BAND and measure ESS-BAND and the results which evolutionary testing could achieve. The coverage of the source code statements (*coverage* in table 2) is not an essential information, and it is not used for this work. However, it indicates the length of the best- and worst-case paths compared with the total

number of coverable statements in a program. Some of the test objects have extreme execution time paths which are only marginally longer than the average- or best-case times. For these, evolutionary testing is likely to produce high quality outcome in terms of timing. Although, the optimum in terms of the extreme execution path as indicated by the code annotations (*target*), is far from being reached.

Considering the number of nodes and decisions of the test objects in table 2, it becomes apparent that larger and more complex test programs are under-represented. This is due to the fact that for assessing the behaviour of evolutionary testing, a test object must have been fully analysed and its best- and worst-case paths determined. This was discussed previously. For most of the test programs, the worst- and best case branches are obvious. They mainly contain iterations of decisions leading to a longer and a shorter branch. The difficulty is not determining the longest branch, but determining the possible number of iterative executions for such a branch. Some of the test objects, for example *median*, demand a detailed inspection and many manual executions with different test data to determine the shortest and longest paths. It was always intended to determine the true best- and worst-case paths for all test programs. Although, it cannot be guaranteed that they have been defined accurately for some of the more complex programs. The test objects have been chosen to minimise this source of error. They are quite simple for this reason. Finding and deciding upon the suitability of a test program for the intended purpose was quite difficult. To be suitable for this work, they had to be complex enough to be of interest for the purpose under consideration. Additionally, they had to be simple enough so that they could be analysed manually in order to determine their best- and worst-case paths. For the same reason, no operating system modules such as networking or input/output library functions were used. These modules tend to be of considerable complexity as they make use of low-level library functions whose run-time behaviour is difficult to trace and analyse. Also, concurrency and recursion have not been considered for this reason.

The example test programs are by no means representative for the set of all

possible test modules which could have been considered useful for the experiments performed for this work. They have been chosen because they were available at the time and simple enough to understand. Most of the complexity of the test objects is imposed through a large input vector. In this way, complexity is generated through more iterations, rather than through a complex structure which is difficult to understand.

3.3 Results of evolutionary testing

The modules in table 2 are ordered according to the measure ESS-BAND. The relationship between the measure ESS-BAND and the outcome of ET is depicted in figure 3. Here, the value for average *target* is used, as displayed in table 2. The average *target* of BCET and WCET is used throughout this thesis. The reason for this is that the evolutionary process always generates input parameters according to the two criteria BCET and WCET. Finding the longer paths always involves finding shorter paths as well, and vice versa. There is no difference between finding the BCET and finding the WCET. According to the test target, the EA simply selects longer or shorter times. Whether a path is important for best-case timing or worst-case timing can not be determined a priori by the testing technique. At a decision, both branches must be inspected in order to determine their time. Therefore, evolutionary testability is determined by these two criteria. Furthermore, no additional information which can be connected to either of the two criteria, BCET or WCET, is generated by the fitness function.

The test objects which correspond to ESS-BAND are marked by "$\diamond$". These are the test objects *delevat, bhorner, polex1, bcastel, sobel, diff, bs2, is, bs1, min, median* and *exp*. Most of the 'simple' programs can be identified as 'non-complex' by the measure and for these modules (ESS-BAND ≤ 3), evolutionary testing could always generate input according to B/WCET. Hence the high outcome of *target* > 95 % for both BCET and WCET. The measure applies according to the approximation in figure 3 for the more complex modules (ESS-BAND > 3).

				ESS-	*target %*			*coverage %*	
Module	N	N_P	BAND	BAND	WCET	BCET	Avg	WCET	BCET
delevat	5	1	6	1	100	100	100	100	100
bhorner	6	1	9	1	100	100	100	100	100
polex1	13	1	15	1	100	100	100	100	100
bcastel	7	3	11	3	100	100	100	100	100
sobel	9	2	16	3	97	99	98	99	100
diff	6	2	11	3	96	100	98	98	100
polex	17	3	29	3	0	100	50	80	100
dzz3	5	3	11	6	100	9	55	100	31
bs2	10	3	26	6	96	95	96	98	96
is	9	3	21	6	96	95	96	97	96
bs1	7	3	19	6	95	95	95	97	96
gstretch	14	5	22	7	17	99	58	72	100
dzz1	6	4	14	9	100	28	64	100	91
min	9	4	21	10	62	100	81	97	100
median	15	4	39	10	82	70	76	78	80
dzz2	9	4	25	10	7	100	54	87	100
train1	16	5	48	15	25	100	63	35	100
train	16	5	48	15	5	100	53	17	100
dummy2	10	7	26	16	89	73	81	–	–
dzz	15	7	46	19	35	68	52	73	76
dummy1	13	10	49	30	44	81	63	–	–
exp	17	12	70	42	0	100	50	98	100

Table 2: Example test programs with measures number of nodes (N), number of predicate nodes (N_P), Belady's BAND measure, the new ESS-BAND measure and the results of the ET process. Avg is the average of BCET and WCET for the coverage criterion *target*. The results are averaged over 10 repetitive trials.

However, some of the test programs refused to fit into this schema. These exhibit additional complexity and are marked by '+' in figure 3. A thorough analysis of these programs identified auxiliary properties which are responsible for increased complexity. The program *polex* is such an exception. It violates one of the 'principles of good software design': low coupling [7, 57]. Its input vector consists of 1080 bytes with only 16 bytes actually used. The remaining 1064 bytes have no effect on the

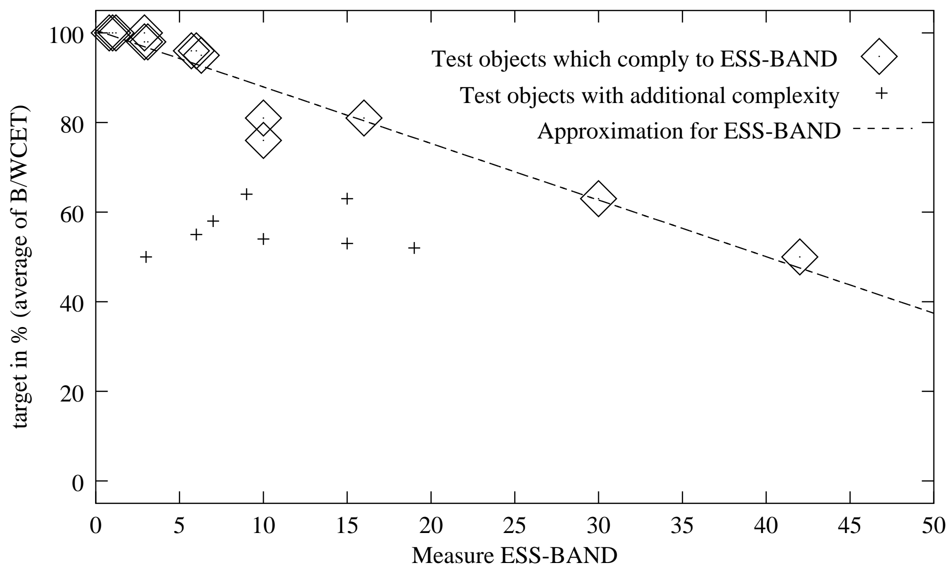

Figure 3: The test objects of table 2 according to the Measure ESS-BAND. Displayed is the performance of ET as average of BCET/WCET for *target* (average of columns WCET and BCET (Avg) under *target* in table 2). The average of B/WCET is chosen as ESS-BAND only indicates the effect of nesting/sequencing on the performance of ET.

testing technique. This creates an insurmountable difficulty for the evolutionary search as the probability of changing one of these 16 bytes is extremely low. There is no information being generated by the fitness function which reveals the most promising path. This results in the low outcome of *target* = 0 % for WCET in table 2. Changing the design and reducing the size of the input vector, and thus the search space, produced an excellent outcome for the new module *polex1* of *target* = 100 % for both test objectives which corresponds to its low structural complexity (ESS-BAND = 1).

The module *dzz* violates another 'principle of good design': high cohesion [7, 57]. By investigating the structure of this module, at least two basic functions which are mutually exclusive, could be identified. This is not primarily a problem of the testing process but of its assessment. Low cohesion typically results in the execution of only one single functionality within a module so that much of its code is not being

executed. For evolutionary testing this suggests that for a high proportion of the module no appropriate input can be generated in order to cover these parts of the program. High cohesion should be enforced if evolutionary testing is to be applied. A new and simpler design of the original module *dzz* resulted in three new units, *dzz1, dzz2* and *dzz3* which performed much better for either BCET or WCET. The fact that the improvement of the new units compared to the original program could only be observed for one of the testing objectives (BCET or WCET) is due to decisions with extremely narrow domains which prohibit some of the branches to be taken. For example, the performance of module *dzz2* is only 7 % for WCET. This is caused by a decision which evaluates to *true* for only 30 out of 2^{16} cases. It is very unlikely for the EA to generate input which follows the longest path as the fitness function generates no information to help finding this. The same applies for generating the BCET for the two modules *dzz1* and *dzz3*.

Additional complexity such as small domains and/or interdependence are also exhibited by the modules *gstretch, train, train1* and *dzz*.

3.4 A measure for parameter interdependence

The measure ESS-BAND is primarily sensitive to nesting. It also captures the effect of parameter interdependence to some extent. Each additional nesting level is created through an additional predicate node which separates the control-flow into two directions. Each additional predicate incorporates input references which must be in a distinct relation in order to execute a branch at that predicate. This creates a filtering effect at each additional predicate node, since the primary purpose of decisions is to reduce the number of possible values for which a distinct action is performed. This filtering effect is typically increased as nesting increases.

Furthermore, interdependence is created by interactions of input values in the decisions. The number of parameter references within the predicate of a decision node determines the degree of interdependence which is generated by that decision node. This number can be used to increment the weight-factor of the measure ESS-

BAND ($n = 1$, equation 3 on page 26). The extended measure EXT-ESS-BAND can then be defined as

$$\text{EXT-ESS-BAND} = \sum_{i=1}^{N_P} L(i) * (1 + \text{IR}(i)) \tag{4}$$

where $\text{IR}(i)$ is the number of input references in the predicate of the decision node i. Input reference is every variable which directly or indirectly depends upon input. This is the case for most variables in a program. A tool which is able to automatically measure this would have to transform the predicate expressions in a decision so that they incorporate all prior calculations which have an effect on the predicates. This is called 'predicate interpretation' [3] and can be performed by static program slicing. Harman [13] defines a slice of a program as follows: A slice of a program according to a slicing criterion (V, n) is a new program which only contains the statements of the original program which can have an effect on all variables V at the location n in the original program [18, 53].

The values for this extended measure for all test objects are displayed in table 3. The corresponding graph is depicted in figure 4.

3.5 A measure for small or single-value domains

In the case of B/WCET analysis, the fitness function of the EA only generates information when code is executed. In order to generate information about a branch behind a decision, this decision must switch the control-flow into this particular branch. Decisions which lead to one branch only being taken with a very low probability, have considerable impact on the performance of evolutionary testing. Such decisions can generate very small domains which increase the filtering effect in a program. In extreme cases, low probability is the single most problematic property of a test object to inhibit evolutionary testing. The following source code illustrates this complexity:

```
for (int i=0; ... ) {
  if (input_list[i] == constant) {
```

Module	ESS-BAND	EXT-ESS-BAND	*target* % WCET	BCET	Average B/WCET
delevat	1	1	100	100	100
bhorner	1	1	100	100	100
polex1	1	1	100	100	100
bcastel	3	3	100	100	100
polex	3	6	0	100	50
sobel	3	7	97	99	98
diff	3	7	96	100	98
dzz3	6	12	100	9	55
bs2	6	12	96	95	96
is	6	12	96	95	96
bs1	6	12	95	95	95
dzz1	9	15	100	28	64
gstretch	7	17	17	99	58
min	10	18	62	100	81
median	10	18	82	70	76
dummy2	16	26	89	73	81
dzz2	10	28	7	100	54
dummy1	30	30	44	81	63
train1	15	31	25	100	63
train	15	31	5	100	53
exp	42	42	0	100	50
dzz	19	49	35	68	52

Table 3: Values for the extension of the measure ESS-BAND which captures parameter interdependence (EXT-ESS-BAND). The modules are sorted according to EXT-ESS-BAND.

```
        /* long section of code */
        ...
    }
}
```

In this example, the longest execution time is found if each position in *input_list* is equal to the *constant*. If the *input_list* is a list of 16 bit integers, then the probability for each of these relations to be found is $\frac{1}{2^{16}} = 0.0000153$. There is hardly any chance that the program-flow will enter the code section behind this decision for

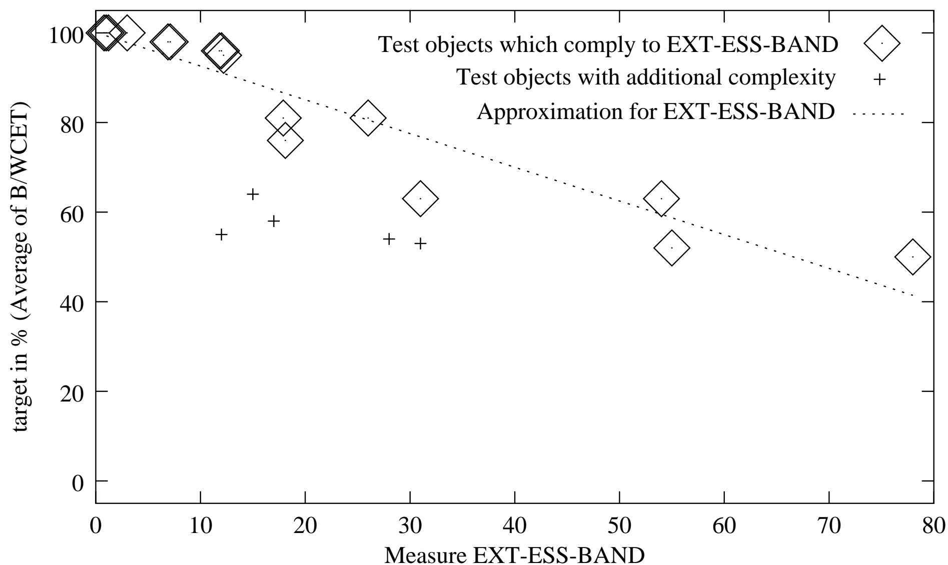

Figure 4: Extension of the measure ESS-BAND. EXT-ESS-BAND captures the effect of parameter interdependence more accurately. This is caused by multiple parameter references in predicates.

many positions in the list.

For simple predicates, the probability can be calculated as in the example. For more complex predicates determining the probability is not trivial. Here, a static program slice containing the decision and all statements which affect this decision can be executed many times with random test data in order to measure the probabilities. The second extension of the measure ESS-BAND can be defined as:

$$\text{EXT2-ESS-BAND} = \sum_{i=1}^{N_P} L(i) * (1 + \text{IR}(i) + C_p(i)) \quad (5)$$

$$C_p(i) = -\log(min(p_{true}(i), p_{false}(i))) \quad + \log(0.5)$$

where p_{true} is the probability that the *true*-branch will be executed, and p_{false} the probability for the *false*-branch. Hence, $p_{true} + p_{false} = 1$. The increment C_p in equation 5 represents the total complexity of the probability with which one branch is taken at a decision. For the previously used example this is $C_p = -log\frac{1}{2^{16}} + \log 0.5 = 4.514$. The expression $min(p_{true}, p_{false})$ determines the minimum probability of executing the *true*- or *false*-branch at the decision un-

der consideration. The minimum is used as both branches at a decision must have been executed in order to determine the total minimum or maximum time. The minimum probability represents the difficulty of finding parameters for the more difficult branch. It is important to note that this probability must be measured or calculated for a complete decision in order to assess its real impact on evolutionary testing. However, the measure ESS-BAND is based upon predicate nodes (N_P), and several predicate nodes can be incorporated into one decision. This is a semantic problem which must be addressed. For the results displayed in table 4, the weight for each decision node which belongs to the decision under consideration, was incremented by the full value for C_p. The use of the logarithm decreases the impact of a very low probability on the overall measure (through the scaling property of the logarithm). The reciprocal ensures that high complexity ($\equiv$ low probability) is expressed by a high value for the measure. The expression log(0.5) eliminates the impact of a decision on the total measure ($C_p = 0$) for equal probability on the two branches ($p_{true} = p_{false} = 0.5$). This seems to be a reasonable approach as for equal probability at a decision either of the branches can be reached equally easily, so that no additional complexity through narrow domains is generated by that decision.

The values for the second extension of the measure ESS-BAND are displayed in table 4 and their graph is depicted in figure 5. In this graph, the measure EXT2-ESS-BAND is displayed on a logarithmic scale as the values for the more complex programs become very large and 'spread' over a wide range.

3.6 Discussion of the combined measure

Each of the introduced measures seems to be able to reveal information about the property of a test program for which the measure had been developed, as indicated through tables 2, 3 and 4. They seem to be able to indicate the difficulty of evolutionary testing or evolutionary testability. This becomes apparent when properties of test programs are changed and the effect on evolutionary testing is assessed. The two test programs *dummy1* and *dummy2* were developed in order to check whether

Module	ESS-BAND	EXT-ESS-BAND	EXT2-ESS-BAND	*target* %		
				WCET	BCET	Average B/WCET
delevat	1	1	1	100	100	100
bhorner	1	1	1	100	100	100
polex1	1	1	1	100	100	100
bcastel	3	3	3	100	100	100
sobel	3	7	7	97	99	98
diff	3	7	7	96	100	98
bs2	6	12	12	96	95	96
is	6	12	12	96	95	96
bs1	6	12	12	95	95	95
polex	3	6	17.2	0	100	50
min	10	18	18	62	100	81
median	10	18	18	82	70	76
gstretch	7	17	19.1	17	99	58
dzz3	6	12	19.8	100	9	55
dummy2	16	26	26	89	73	81
dzz1	9	15	27.6	100	28	64
train1	15	31	34.6	25	100	63
train	15	31	39.4	5	100	53
dzz2	10	28	42.9	7	100	54
dummy1	30	30	60.2	44	81	63
dzz	19	49	81.3	35	68	52
exp	42	42	132	0	100	50

Table 4: Values for the extension of the measure ESS-BAND which captures 'probability complexity' (EXT2-ESS-BAND). The modules are sorted according to EXT2-ESS-BAND.

the perception of complexity for evolutionary testing would hold. They only consist of predicate nodes with no particular functionality. The ease with which these could be developed, and the accuracy with which their measured properties (ESS-BAND, IR, C_p) complied to the performance of evolutionary testing, confirmed the assumption that the three measures can assess the impact of complexity on ET to a certain extent. ESS-BAND provides information about the nesting or sequencing of a program. IR additionally indicates the parameter interdependence at a decision. C_p

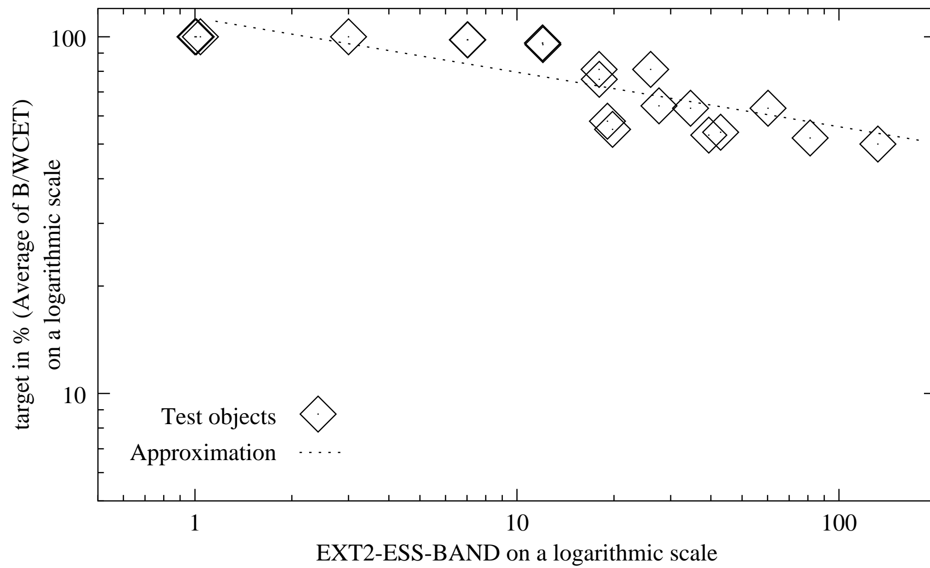

Figure 5: Second Extension of the measure ESS-BAND. EXT2-ESS-BAND additionally captures the effect of small or single input domains. This is based upon the minimum probability with which a branch is taken at a decision.

measures the probability of branches being executed. All of these measures seem to have an effect on evolutionary testability. The results of all experiments support the hypothetical relation between the complexity of a test object and its evolutionary testability. The approximations in the previously depicted graphs are decreasing with an increase in complexity as assessed through the introduced measures.

The results indicate that each further improvement in accuracy for one particular test object may result in a loss of accuracy for another test object. This becomes apparent if the sorting of the modules according to the additionally introduced partial measures in tables 2 to 4 is considered. The ranking of the test objects according to each individually introduced additional measure is different in each table. This may be caused through the fact that

- the impact of each individually introduced measure on the overall performance of ET is not known. Apparently, it cannot be sufficiently described through the introduced model of complexity; and

- the combination of the three individual measures into one single measure as previously suggested, is not entirely evident. The results for the combined measure suggest low accuracy in predicting the evolutionary testability through the proposed model. Although, a trend can be observed which indicates a relation.

The question must be raised whether the previously introduced combined measure is a sensible measure at all. The overall measure EXT2-ESS-BAND is clearly not a measure in the sense of measurement theory as it captures no specific attribute of complexity [7, 59]. The notion of evolutionary testability as an aspect of complexity must be defined much more thoroughly and precisely in order to achieve this. The measure has no meaning, so that it could be interpreted sensibly. This relates to the problem of meaningfulness and scale types of measures as addressed by Fenton [7] and Zuse [58, 59]. It is an assembly of attributes of test programs which have been identified (empirically) as being important or expressive for evolutionary testability without having investigated their internal interdependence and their relations to other program properties. It is appealing to have one single overall measure of evolutionary testability, but it is arguable whether it is feasible or sensible to derive such a measure. Some researchers have criticised attempts to derive single complexity measures [2, 7], although, this depends on the aim of measurement. Clearly, the derivation of a complete measure of complexity is not feasible if it is supposed to indicate every single possible aspect of complexity people can think of. Though, it might be possible, to propose a measure for a clearly defined aim such as evolutionary testability as an aspect of complexity. A 'dumb' search algorithm is likely to be more objective in 'experiencing' complexity than a human observer, who may have constantly changing viewpoints in terms of complexity. This could be the justification to believe that such a general measure for evolutionary testability might exist.

The measure as it stands, can only be used for single programs (modules). Measurements during the integration of a system cannot be performed with this measure. It is simply too awkward to handle, as there are no composition operations defined

which could be used to combine measures of single modules into a measure for a system in any meaningful way.

3.7 Individual properties for evolutionary testing complexity

It was demonstrated in the previous section that it is difficult to derive one single accurate and usable measure of 'evolutionary testability' as one aspect of complexity. Complexity is a rather metaphysical attribute which cannot be captured easily as a whole. Therefore, the emphasis of this work is shifted towards the individual properties which determine evolutionary testability.

Nesting and sequencing. As a nesting indicator ESS-BAND is considered useful as it calculates the sum of the nesting levels of all predicate nodes. However, the measure cannot discriminate between a deeply nested but short program and a program which is long but only incorporates sequences of predicate nodes which occur on low nesting levels.

For this, a new measure ADN (average decision nesting) is introduced

$$\mathrm{ADN} = \begin{cases} \frac{\text{ESS-BAND}}{\mathrm{N}_P} & \text{if } \mathrm{N}_P > 0 \\ 0 & \text{if } \mathrm{N}_P = 0 \end{cases} \tag{6}$$

which calculates the average nesting level of all predicate nodes in the program. This measure indicates the distribution of predicate nodes over all nesting levels. As discussed in chapter 4, it has no direct effect on evolutionary testability. Though, it provides useful information, and it is used to define the concatenation operator for the measure ESS-BAND (see section 4.5). ADN = 0 means that the program consists only of sequential code (no decisions and no nesting which is the same as $\mathrm{N}_P = 0$). In this case, there is no structural complexity caused through decisions, so that evolutionary testing is likely to produce meaningful results, in terms of the structure of the program. It must be noted that this might not be the case for timing analysis on programs which perform many calculations on input variables.

ADN $\gg 1$ indicates that the program is heavily nested. This measure is based on a program's control flowgraph. On its own, ADN is not able to distinguish between long and short programs. Therefore, ESS-BAND, which can be renamed into DN (decision nesting) for consistency, and ADN belong together.

Parameter interdependence. Interdependence is created through comparisons of values of parameter references in decisions. The number of input references in a decision determines the degree of interaction in that decision. The total number of input references in decisions (decision input references, DIR) can be calculated for a module

$$\mathrm{DIR} = \sum_{i=1}^{\mathrm{N}_{P,IR}} \mathrm{IR}(i) \tag{7}$$

where $N_{P,IR}$ is the total number of predicate nodes with input references (typically all decisions except static loops). $\mathrm{IR}(i)$ is the number of input references in each decision i. This is the same as the second extension of the measure ESS-BAND. The measure DIR does not distinguish between long and short programs. There is no indicator whether a program consists of many decisions with few references – in this case interdependence is not as critical for evolutionary testability – or whether it consists of only a few decisions which contain many input references. This can be captured by the average number of input references in decisions (ADIR). It simply indicates the distribution of parameter references over the predicate nodes. This can be calculated as

$$\mathrm{ADIR} = \begin{cases} \frac{\mathrm{DIR}}{\mathrm{N}_{P,IR}} & \text{if } \mathrm{N}_{P,IR} > 0 \\ 0 & \text{if } \mathrm{N}_{P,IR} = 0 \end{cases} \tag{8}$$

A value ADIR $= 1$ indicates that each decision node ($N_{P,IR}$) depends on one input parameter reference on average, and a value ADIR > 1 indicates that there must be decision nodes with more than one parameter reference. These generate higher interdependence and decrease evolutionary testability.

Module	DN	ADN	DIR	ADIR	DPC	ADPC	IS	average *target* %
delevat	1	1	0	0	0	0	1.5	100
bhorner	1	1	0	0	0	0	1.5	100
polex1	1	1	1	1	4.5	4.5	1.3	100
bcastel	3	1	0	0	0	0	1.5	100
sobel	3	1.5	2	2	0	0	3.0	98
diff	3	1.5	2	2	0	0	3.0	98
bs2	6	2	2	2	0	0	3.0	96
is	6	2	2	2	0	0	3.0	96
bs1	6	2	2	2	0	0	3.0	95
dummy2	16	2.3	3	1	0	0	3.0	81
min	10	2.5	2	2	0	0	3.0	81
median	10	2.5	2	2	0	0	3.0	76
dzz1	9	2.3	2	1	1.8	1.8	3.0	64
dummy1	30	3	6	1	5.4	0.9	3.0	63
train1	15	3	4	1.3	1.8	0.5	2.3	63
gstretch	7	1.4	6	2	2.1	0.7	3.0	58
dzz3	6	2	2	2	2.6	2.6	3.0	55
dzz2	10	2.5	5	2.5	2.7	2.7	3.0	54
train	15	3	4	1.3	4.2	1.4	2.3	53
dzz	19	2.7	6	1.2	7.4	2.5	3.0	52
polex	3	1	3	1	8.7	2.9	3.0	50
exp	42	3.5	8	1	9	1.1	3.0	50

Table 5: Individual measures for evolutionary testability. The modules are ordered according to ET performance (average *target* %).

Small or single-value domains. Complexity which is generated through small domains in decisions can be measured by DPC ('decision-probability-complexity') which is calculated as the sum of all individual 'decision-probability-complexities' in decisions with input references (D_{IR}):

$$\mathrm{DPC} = \sum_{i=1}^{\mathrm{D}_{IR}} -\log(min(p_{true}(i), p_{false}(i))) \quad + \log(0.5) \tag{9}$$

This is the sum of all C_p as introduced in equation 5 on page 41, although equation 9 is now correct compared to equation 5 as the probabilities must be calculated or measured for a complete decision (D_{IR}) and not only for one predicate node ($\mathrm{N}_{P,IR}$) as proposed in equation 5. It only makes sense to consider complete decisions (compound predicates) in terms of probability. This measure can be retrieved from the source code.

The sum of all decision complexities makes no statement about individual decisions in a test object. In order to have an indicator for each decision, the average DPC (average decision probability complexity) can be calculated as

$$\mathrm{ADPC} = \begin{cases} \frac{\mathrm{DPC}}{\mathrm{D}_{IR}} & \text{if D} > 0 \\ 0 & \text{if D} = 0 \end{cases} \tag{10}$$

The Importance of the input size for evolutionary testing. The size of the input vector of a test object is important for the performance of evolutionary testing. This has been neglected as its effect on the performance of the search technique can be comprehended. So far, the complexity of the search space and not its size has been scrutinized. An increased combination space generally leads to decreased evolutionary testing performance, since it determines the number of possible input combinations. In this case, more possible combinations must be considered in order to achieve a constant mapping of the combination volume. However, sheer size of the search space on its own does not affect the search as exceptionally as one would expect. The testing process takes merely longer to 'evolve' optimal solutions. It is the size of the input in combination with other types of complexity, the complexity

of the search space, which makes the outcome of evolutionary testing unreliable. A measure IS (input size) can be calculated as follows:

$$\text{IS} = \log(\text{InputSize}_{Bytes} + 1) \tag{11}$$

where InputSize_{Bytes} is the size of the input vector in bytes. The logarithm is used as small alterations in the input size do not affect the outcome of ET drastically. IS = 0 indicates that there is no input for the module under consideration, so that the application of ET makes no sense in the way it has been introduced in this work.

Results of evolutionary testing. The values for the previously introduced individual measures for all test objects are summarized in table 5. It becomes apparent that the values of these measures increase in the lower half of the table. There, they correspond to the low performance of the testing technique. This table shows that no individual measure on its own is able to indicate ET performance. They all concentrate on one single aspect, some of which are quite outstanding, for example the values for DPC for the modules *dzz*, *polex* and *exp*. In these cases, they must be the single most important properties to cause the poor performance of evolutionary testing.

The results in tables 2 to 5 seem to confirm the hypothetical relation between complexity and success rate of the testing strategy (figure 1). Although it becomes apparent that not every aspect of complexity is displayed in table 5. Additionally, the performance of evolutionary testing seems not to decrease evenly as assumed and displayed in figure 1. The results for BCET and WCET in table 2 suggest that only one criterion decreases, if complexity increases, which is either BCET or WCET. This is an unexpected finding which might be caused by the small size and low complexity of the test objects. This needs further verification, for instance through the use of more complicated test objects.

The role of algorithmic parameter dependence. 'Input parameter interdependence' has been identified as a main aspect of evolutionary testability complexity.

It is represented through the input references which are used within predicate nodes (measure DIR). Here, distinct values must be generated in order to satisfy the constraints which are imposed by these relations. The complexity which is generated through the probabilities in decisions (measure DPC) is another form of interdependence. Here, the main aspect is the generation of input according to small domains. A distinct property of test programs which has not been considered is 'algorithmic interdependence' or 'algorithmic or combinatorial complexity'. It is exhibited through the change of input by the algorithm according to distinct values in that input. It can be typically found in recursive or iterative algorithms. Since it is a very subtle form of interdependence which was difficult to capture, it became only apparent during later stages of this project. It was exhibited by obviously simple test objects.

An example for this is the procedure *gstretch*. This module determines the minimum and maximum in a list of grey scale values for pixels, and stretches the range of these values accordingly. The testing technique performs poorly on this object (only 17% *target* coverage for WCET, table 2). The reason for this is that action is only performed if the algorithm encounters a new minimum or maximum value in the pixel list. If the maximum or minimum value is found at a very early stage of that process, then no additional action is performed. The longest path through the program is only found if ET generates a reversely ordered list for the pixel values. This is an extremely unlikely outcome. Through the knowledge of the existence of these interactions, this type of complexity is now easy to observe. Though, the development of a technique which can measure this automatically is a difficult task. It is not clear how to assess this type of complexity.

The Importance of the input type for evolutionary testing. The data type of the input vector of a test object has not been scrutinized in the preceding sections, though it has an extraordinary effect on the performance of evolutionary testing. This is mainly exhibited in combination with other types of complexity. For

example, if a test object is distinguished by small domains which is measured by DPC, then it is clearly disadvantageous to have 'large' data types. Here, the adjective 'large' must be seen in terms of the number of possible combinations which can be represented by a data type. The more combinations a data type is able to represent, the lower is the probability that few distinct values out of this huge number are found by ET. This questions the appropriateness of the fitness function and it is discussed in chapter 5.

However, 'large' data types are advantageous in combination with 'algorithmic or combinatorial complexity'. A *selection sort* algorithm can serve as an example for this. If a list of 256 8-bit integers is to be sorted, then there is only one single combination which generates WCET behaviour: the maximum time is generated through a reversely sorted list with each integer on its exact predetermined position. Evolutionary testing must generate this single solution in order to find the maximum time. For a list of 128 16-bit integers, this is far easier to generate as there are many global maxima which result in the same execution time. The outcome for a *selection sort* is much worse than for a *bubble sort*, for example. This is due to the extreme 'combinatorial complexity' of this module, which is similar to that exhibited by the module *gstretch*. The sorting algorithm always selects the next minimum for each iteration. Table 6 displays the outcome of ET for such a test object. The table shows the consistent decrease in performance for each distinct data type corresponding to an increase in the number of values which are to be sorted (columns in table 6). Keeping the number of values constant, does not affect the outcome for the different data types through an extended search space (lines in table 6). If the size of the search space is constant (in this case 2^{2048} combinations), then the performance is increased for 'larger' data types. This is illustrated by the bold-faced diagonal entries in table 6. This effect is entirely caused through the increase in global optima in the search volume corresponding to an increase in the possible combinations which are represented by that data type.

It is not clear, how to assess the effect of the used data type on the performance

Input size	8-bit char	16-bit short	32-bit int
64 values	37.5 %	36.8 %	**36.6 %**
128 values	19.7 %	**19.4 %**	19.3 %
256 values	**9.3**%	10.1 %	10.8 %

Table 6: Performance of ET (*target* in % for WCET) on a *selection sort* algorithm. The values illustrate the effect of different data types on the outcome of ET.

of evolutionary testing. It is easy to see how this attribute interacts with other properties, but it is rather difficult to incorporate this into the previously introduced model of evolutionary testability measures.

A validation of how suitable the introduced measures are for predicting evolutionary testability is given in section 4.4.

4 Validation of the individual measures

Much work has been committed over years into the development of software measures, but amazingly little effort has been put into showing that the measures are useful or actually measure what they claim. This lack of validation is a major problem in the software engineering community and has led to many measures being used inappropriately or inhibited their application entirely [7, 59]. The need for validation has been recognised for long and some researchers have attempted to develop methodologies for software measurement validation [26, 41, 43, 54, 59].

One such methodology which is based upon the Goal-Question-Metric paradigm by Basili and Rombach [2], and an axiomatic approach such as Weyuker's desirable properties of software metrics [54] is introduced by Shepperd and Ince [43]. Their approach is based upon tailored axioms to which the measures must comply. These are

- axioms which are fundamental to all measurement,
- axioms which are necessary for the type of scale,
- axioms which are specific to the model of the measure.

The first and most fundamental stage of measure validation is to determine the purpose of the measure (problem identification, [43]). This has been sufficiently demonstrated in the preceding sections, as well as the application area and scope for which the measures are intended (informal model). The actual validation for the new measures is demonstrated in the following sections. All axioms of these sections are taken from Shepperd and Ince [43], and from Weyuker [54]. The introduced measures must be checked against these criteria in order to determine their validity. Only the individual measures are validated and not the combined measure. The validation follows the strategy which is proposed by Shepperd and Ince [43].

4.1 Axioms which are fundamental to all measurement

Axiom 1 *It must be possible, even if not formally, to describe the rules governing the measurement.*

Since all the introduced individual measures simply count the occurrences of distinct properties of test programs, their rules are quite simple. They have been introduced (not entirely formally) in the preceding chapter.

Axiom 2 *The measure must generate at least two equivalence classes.*

This simply claims that measures must be able to discriminate in order to be useful [43, 54].

Axiom 3 *An equality relation is required.*

In the same way as it is undesirable to have all objects being assigned to one single class by a measure (axiom 2), it is not useful to have each object assigned to a separate class [43, 54].

Axiom 4 *There must exist two or more objects which will be assigned to the same equivalence class.*

This requirement states that a measure should not generate a unique value for each unique test object, so that for an infinite number of test objects there exists an infinite number of measured values. This is clearly not a useful property of a software measure [43]. It can be seen as a strengthening of axiom 3. Axioms 2, 3 and 4 apply to all introduced measures as can be seen from the results in table 5.

Axiom 5 *The measure must not produce anomalies; it must preserve empirical orderings (Representation Theorem).*

This is concerned with the question whether a particular empirical system has a representation in a numerical relation system [7].

Axiom 6 *The uniqueness theorem must hold for all permissible transformations for the particular scale type.*

This imposes limitations on the mathematical manipulations that can be performed on the numbers arising as scale values [59].

The representation and uniqueness theorems are necessary in order to determine the scale on which the measurement is based. This contains the conditions for

the homomorphism (mapping from the empirical world into numbers) [59]. For the introduced measures it might be tempting to assume a ratio scale as they are simple counts of items based on real numbers, and to manipulate the retrieved values in a familiar mathematical way. This is a critical approach, since the scale type on which the measurement is based must initially be evaluated. Scale types are ordered according to their admissible transformations.

The nominal scale simply assumes that objects can be classified in an arbitrary way [7]. For the empirical relational system $(\mathbf{P}, \approx^\tau)$ with $\mathbf{P}$ as a set of programs and $\approx^\tau$ as an empirical equivalence relation for evolutionary testability, the nominal scale can be formally defined as $((\mathbf{P}, \approx^\tau), (\mathbb{R}, =), \mu)$ with the measure $\mu : \mathbf{P} \to \mathbb{R}$ as adapted from Zuse [59]. This is trivial for the introduced measures as each measure can be associated with a distinct class (measurement value) as displayed in table 5.

The ordinal scale introduces the notion of order. Any homomorphic mapping must preserve the order of the measure. This scale is based upon the relational system $((\mathbf{P}, \gtrsim^\tau), (\mathbb{R}, \geq), \mu)$ with $\gtrsim^\tau$ as the empirical relation "equally or more difficult to test (for evolutionary testing)". The mapping $\mu : \mathbf{P} \to \mathbb{R}$ must be monotonic [7], so that on the ordinal scale, the following expressions must always be true for programs $P_1, P_2 \in \mathbf{P}$:

$$\mathrm{IS}(P_1) \geq \mathrm{IS}(P_2) \iff P_1 \gtrsim^\tau P_2 \tag{12}$$

$$\mathrm{DPC}(P_1) \geq \mathrm{DPC}(P_2) \iff P_1 \gtrsim^\tau P_2 \tag{13}$$

$$\mathrm{ADPC}(P_1) \geq \mathrm{ADPC}(P_2) \iff P_1 \gtrsim^\tau P_2 \tag{14}$$

$$\mathrm{DIR}(P_1) \geq \mathrm{DIR}(P_2) \iff P_1 \gtrsim^\tau P_2 \tag{15}$$

$$\mathrm{ADIR}(P_1) \geq \mathrm{ADIR}(P_2) \iff P_1 \gtrsim^\tau P_2 \tag{16}$$

$$\mathrm{DN}(P_1) \geq \mathrm{DN}(P_2) \iff P_1 \gtrsim^\tau P_2 \tag{17}$$

$$\mathrm{ADN}(P_1) \geq \mathrm{ADN}(P_2) \iff P_1 \gtrsim^\tau P_2 \tag{18}$$

However, these can not be shown to be valid for the overall results displayed in table 5 as these results depend on the interaction of all program attributes. In order to discuss their validity, each individual property of a test program must be

changed separately (and all others kept constant) and the effect on the outcome of evolutionary testing assessed.

The effect of the input size (**IS**) on the performance of evolutionary testing is very subtle, as mentioned in section 3.7. For programs which exhibit no observable 'combination space complexity', changing the size of the input space may not have an observable effect on the performance in terms of the quality of the produced outcome. For most cases, the only observable effect is that it will take longer to produce this outcome. Given the same time to evolve, a larger input space will certainly always result in a decreased performance of ET. It could never be observed that the performance increased through an increase of the input size. The statement of equation 12 can therefore be considered valid.

The effect of the measure **DPC** on the performance of ET can be observed most easily. On a test object for which the effects of all other attributes of complexity have been fixed, the performance is decreased quite considerably as DPC is increased. This is illustrated by the values for WCET in table 7 and the corresponding graph in figure 6. The same tendency can be observed for the measure **ADPC** as long as the denominator (D_{IR}, equation 10) is constant and only DPC is changed. In this case, equation 14 is trivially valid. However, ADPC as individual measure, does not comply with the relation in equation 10, as two programs P_1 and P_2 can be easily constructed where ADPC(P_1) $\geq$ ADPC(P_2) but $P_1 <^{\tau} P_2$. This is illustrated in table 8. ADPC can therefore not be used as an individual measure of evolutionary testability. It is rather to be regarded as an extension to DPC which is able to reveal more information about the distribution of complex decisions over the whole program.

The idea behind the measure **DIR** is that decisions with more input references in their predicates create higher data dependence and are therefore responsible for a higher filtering effect. Consequently, predicates with more input references should reduce evolutionary testability to a greater extent than predicates with less input references. This assumption seems not to hold. A test object with a very low

	target %		
DPC	WCET	BCET	Average B/WCET
0.0	98.8	100	99.4
0.1	95.2	100	97.6
0.2	91.6	100	95.8
0.3	87.6	100	93.8
0.5	77.2	100	88.6
0.8	63.6	100	81.8
1.2	48.2	100	74.1

Table 7: Illustration of the effect of the measure DPC on the performance of ET (*target* %). Additional measurable program properties are unchanged (IS=3.0, DIR=1, ADIR=1, DN=6, ADN=2). The corresponding graph is displayed in figure 6.

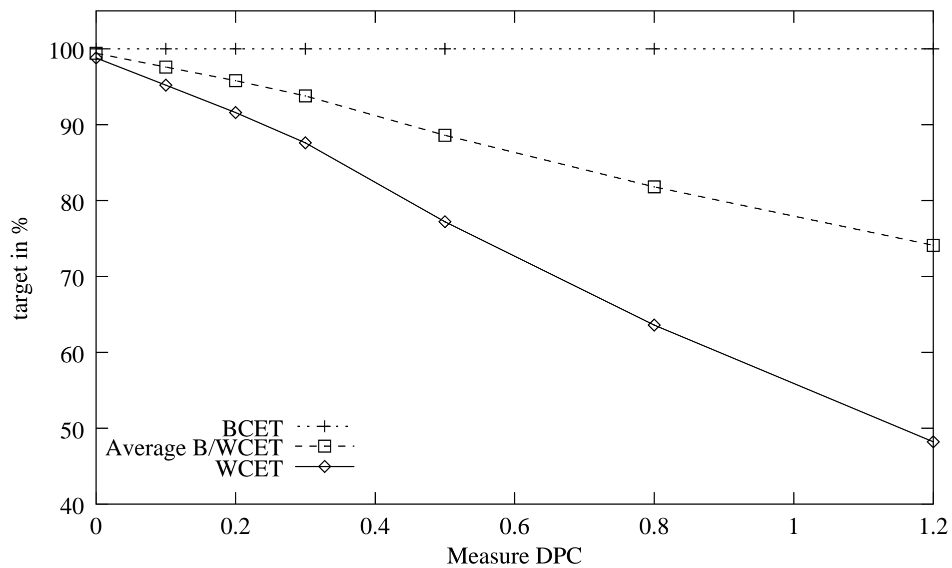

Figure 6: Graph of the measure DPC mapped against ET performance (*target* %) according to table 7.

	DPC(P$_1$)	DPC(P$_2$)
N_{D_1}	1.5	0
N_{D_2}	1.5	0
N_{D_3}	1.5	0
N_{D_4}	1.5	5.5
ADPC	1.5	1.375

Table 8: The validation of ADPC shows that it is no measure for evolutionary testability. It is apparent that ADPC(P_1) $\geq$ ADPC(P_2) but $P_1 <^{\tau} P_2$.

number of input references in decisions can be designed which leads to higher filtering and to far worse evolutionary testability, as a test object with a higher number of input references in decisions but lower filtering. Here, it is not the number of input references which is decisive for evolutionary testability but the sizes of the domains which are generated by the decisions, but this is measured by DPC. The expression in equation 15 is therefore not valid, in contrast to the discussion in section 3.6. The same applies for the expression in equation 16 (ADIR).

Equation 17 is valid for the measure **DN** as illustrated in table 9 and displayed in figure 7. However, it must be noted that the reaction of ET on the measure DN is closely coupled with the measure DPC as indicated in table 9. Each additional nesting level typically increases the filtering effect of the program. Although, this is usually not caused through the actual nesting but through the decreasing number of values for which each additional decision switches to either of the two branches. DN has no effect if decisions do not depend upon input references as this is the case for the first two entries for DN in table 9. The results (100 % Target for WCET for DN=1 and DN=3, table 9) are caused through two static loops. These do not decrease evolutionary testability. It could never be observed that the outcome of ET was better for an increase in DN, so that DN is always a monotonic mapping according to equation 17. The measure **ADN** cannot be used as an individual indicator for evolutionary testability for the same reason as was put forward for the measure ADPC. Although, ADN complies to equation 18 for the example displayed

DN	ADN	*target %* WCET	BCET	Average B/WCET	DPC	DIR
1	1.0	100	100	100	0.0	0
3	1.5	100	100	100	0.0	0
6	2.0	93.3	100	96.7	0.0	1
10	2.5	76.3	100	88.2	0.3	2
15	3.0	64.1	100	82.1	0.9	3
21	3.5	54.8	100	77.4	1.8	4
28	4.0	45.9	100	73	3.0	5

Table 9: Illustration of the effect of the measure DN on the performance of ET (*target* %). This is in combination with the measure DPC for increasing decision nesting of the test program. Additional measurable program properties are unchanged (IS=3.0, ADIR=1). The corresponding graph is displayed in figure 7.

	DN(P_1)	DN(P_2)
N_{D_1}	1	1
N_{D_2}	2	2
N_{D_3}	3	3
N_{D_4}	4	1
N_{D_5}		2
N_{D_6}		3
N_{D_7}		1
N_{D_8}		2
N_{D_9}		3
ADN	2.5	2

Table 10: The validation of ADN shows that it is not fit for indicating evolutionary testability. A scenario may be constructed easily with ADN(P_1) $\geq$ ADN(P_2) but $P_1 <^{\tau} P_2$.

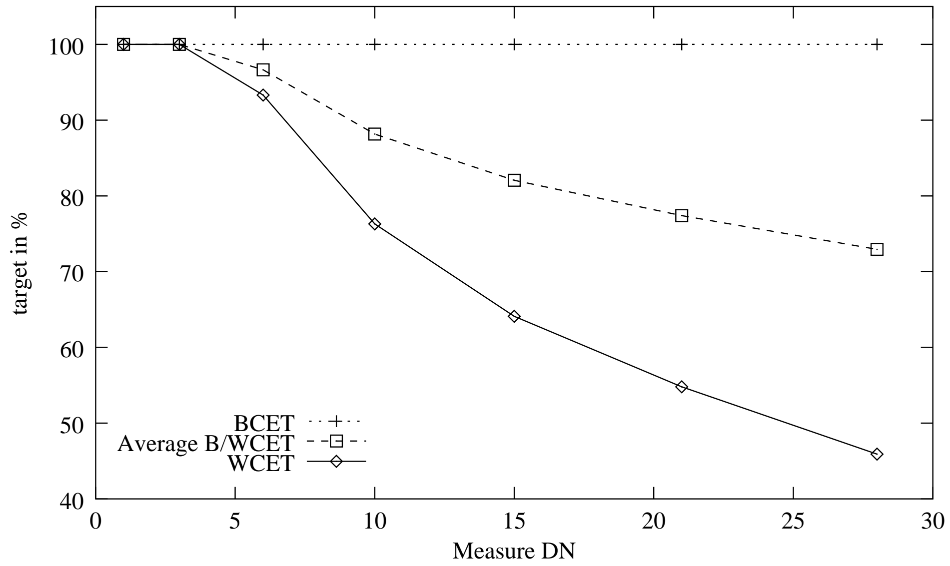

Figure 7: Graph of the measure DN mapped against ET performance (*target* %) according to table 9.

in table 9, a scenario can be easily constructed where $\text{ADN}(P_1) \geq \text{ADN}(P_2)$, but this does not lead to $P_1 \gtrsim^\tau P_2$ (see table 10). ADN is merely an extension for the measure DN which indicates the distribution of the decisions – according to the nesting level – over the whole test object.

The measures are based upon the ordinal scale, so that the class of all admissible transformations is the set of all monotonic mappings [7].

4.2 Axioms which are specific for the scale type

Since an ordinal scale is assumed for the introduced measures, the following axiom must be valid for the empirical relational system $((\mathbf{P}, \gtrsim^\tau), (\mathbb{R}, \geq), \mu)$ and for $P_1, P_2, P_3 \in \mathbf{P}$, $\mu : \mathbf{P} \rightarrow \mathbb{R}$:

Axiom 7 *Symmetry (completeness), reflexivity and transitivity of μ.*

The so-called 'weak order' of μ can be formalised by the following equations as adapted from Zuse [58, 59]:

$$\text{Symmetry} \qquad P_1 \gtrsim^\tau P_2 \quad \text{or} \quad P_2 \gtrsim^\tau P_1 \tag{19}$$

$$\text{Reflexivity} \qquad P_1 \approx^{\tau} P_1 \tag{20}$$

$$\text{Transitivity} \quad P_1 \gtrsim^{\tau} P_2 \quad \text{and} \quad P_2 \gtrsim^{\tau} P_3 \Longrightarrow P_1 \gtrsim^{\tau} P_3 \tag{21}$$

The mechanisms of evolutionary testing are entirely based upon chance. Repeated execution of one test object almost never produces the same outcome. The technique lacks accurate reproducibility, although most results tend to be within a distinct small range. In terms of the empirical relational system, introduced in the preceding paragraphs, the equality relation is always to be seen as an approximation, therefore the use of $P_1 \approx^{\tau} P_1$ instead of $P_1 =^{\tau} P_1$. It is very important to note that single test processes may certainly violate all the requirements which were previously introduced. However, the testing process does not exhibit this behaviour for the average case. For this work, the average of at least ten repetitions of tests on one single test object have been performed in order to reason about the behaviour of evolutionary testing. Equations 19 to 21 are valid as long as trends are considered and a threshold is allowed to compensate for small differences in outcome. It is interesting that the threshold tends to increase as complexity increases (decrease in evolutionary testability).

4.3 Axioms which are specific to the model

Axioms which are specific to the model of the measure can be defined as follows:

Axiom 8 *Two objects which implement the same function can be assigned to different equivalence classes.*

This states, that even if two programs perform the same task, their implementation can be entirely different [54] and therefore, evolutionary testing may perform differently on the two objects. This is trivially true for the introduced measures but it shows an interesting implication. It emphasises the desirable requirement of software measures that they can be used as a basis to reveal possible deficiencies in the design [59] and to improve the implementation and assess its improvement. This is discussed in chapter 5.

Axiom 9 *Increasing the input combination space must increase the measure IS.*

This is trivially valid, since IS= $\log(\text{InputSize}_{Bytes} + 1)$.

Axiom 10 *Reducing the probability of one of two branches being taken at a decision must increase the value for the measure DPC.*

This is valid according to how DPC is calculated from the probability in equation 9. The main problem here is the determination of these probabilities from the source code of the program.

Axiom 11 *Adding another decision node to the test object must increase the measure DN. Adding sequential code to the test object must not increase the measure DN.*

This is true, although the effect of adding another decision node might not be observed empirically as discussed under axioms 5 and 6. This is only apparent if a decision node with input references ($N_{D,IR}$) is added. Adding sequential code certainly has no effect on the measure DN through the way it is retrieved from the decision nodes.

4.4 Prediction models for evolutionary testability

The validation of the introduced measures has identified DN, DPC and IS as valid measures according to measurement theory. All additional introduced measures were found as not being proper measures in the sense of this theory. The fact that a measure is valid does not imply its usefulness for prediction. If a measure is not valid, it does not necessarily mean that it is not indicating a useful or interesting property of a program. In this instance, the development and validation of measures is strongly related to finding a suitable prediction system which can anticipate the likely success of evolutionary testing on a new test object. Therefore, it is important to clarify the difference between measures and prediction systems:

- A measurement system is used to assess an existing entity by numerically characterizing one or more of its attributes [7]. This has been demonstrated in the preceeding sections.

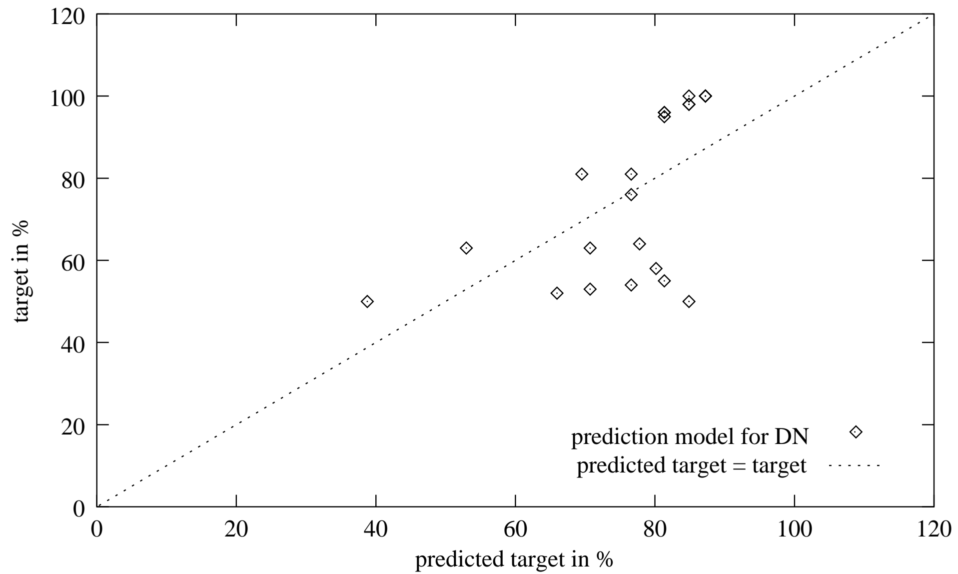

Figure 8: The prediction of evolutionary testability through the measure DN, predicted performance (predicted target in %) against the actual performance (target in %) for all test objects , $b_0 = 88.4288$, $b_1 = -1.18214$ (DN), prediction model $f(x) = b_0 + b_1 * x$, correlation coefficient $r = 0.582472$.

- A prediction system is used to anticipate some attribute of a future entity, involving an underlying mathematical model [7]. This is briefly discussed in the following paragraphs.

As seen previously, no measure has been introduced which is suitable to predict evolutionary testability individually. For example, the prediction for the measure DN is illustrated in figure 8. The underlying prediction model has been retrieved through linear regression.

The number of predicate nodes N_P (table 2) as measure seems to be even more suitable as the measure DN for the prediction of evolutionary testability according to the data for the test objects under consideration. The correlation coefficient r is for this case even higher ($r = 0.625125$, figure 9) than for the measure DN ($r = 0.582472$, figure 8), although, the number of predicates (N_P) is much easier to retrieve than the sum of the decision nesting levels (DN) of a program. The prediction for N_P is illustrated in figure 9. This suggests that the simpler measure N_P is performing

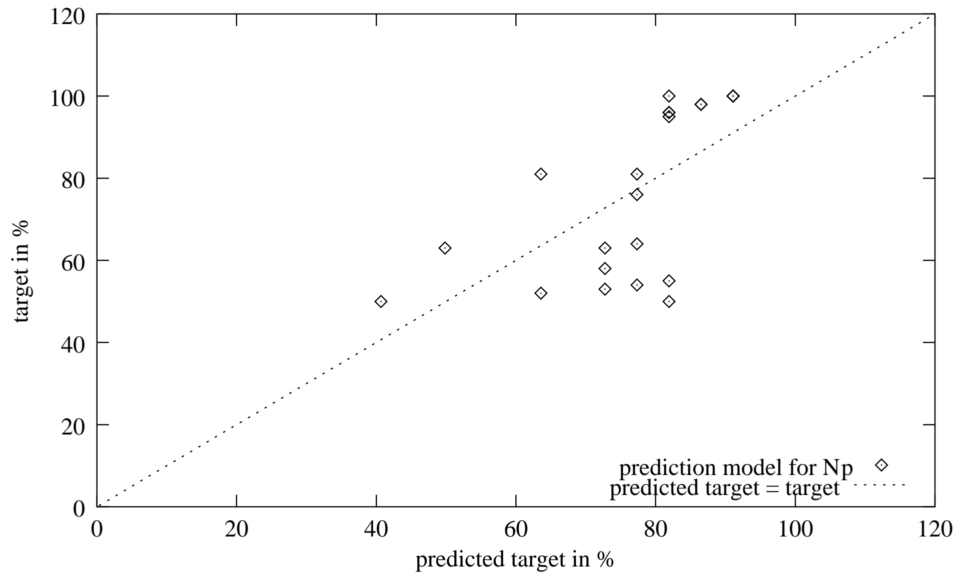

Figure 9: The prediction of evolutionary testability through the measure N_p, predicted performance (predicted target in %) against the actual performance (target in %), $b_0 = 95.6623$, $b_1 = -4.5823$, for the model $f(x) = b_0 + b_1 * x$, correlation coefficient $r = 0.625125$.

better than the more complex measure DN. However, the prediction becomes much more accurate if the individual measures from table 5 on page 48 are combined to form a prediction system. This is demonstrated in the following.

The results in table 5 suggest that the values for the individual measures tend to increase with a decrease in evolutionary testability. This is only a very general observation. Multiple regression can determine whether a combination of these indicators can predict evolutionary testability of a program. In order to keep the proposed model as simple as possible, rank correlation may identify interdependence between the measures. Table 11 shows the correlations for the measures taken from table 5. This identifies the measures DN/ADN ($r = 0.934$) and the measures DPC/ADPC ($r = 0.904$) as being highly correlated (table 11). The measures ADN and ADPC are therefore removed from the analysis. Multiple regression on the remaining attributes yields the prediction model in table 12. The mapping of the predicted and the actual performance of ET is displayed in figure 10 on page 67.

Corr	DN	ADN	DIR	ADIR	DPC	ADPC
DN	1.000					
ADN	0.934	1.000				
DIR	0.819	0.705	1.000			
ADIR	0.150	0.243	0.231	1.000		
DPC	0.417	0.359	0.670	-0.171	1.000	
ADPC	0.206	0.179	0.451	-0.071	0.904	1.000

Table 11: Rank correlation for the individual measures; data taken from table 5.

regression coefficients, b_i	measure, x_i	mathematical model
$b_0 = 99.084435$		$f(x_1, x_2, x_3, x_4) = b_0$
$b_1 = 0.278420$	DN	$+b_1 * x_1$
$b_2 = -5.522765$	DIR	$+b_2 * x_2$
$b_3 = -2.156085$	ADIR	$+b_3 * x_3$
$b_4 = -2.786464$	DPC	$+b_4 * x_4$
correlation coefficient $r = 0.826880$		

Table 12: Multiple regression for the measures DN, DIR, ADIR and DPC. The graph of the prediction is displayed in figure 10.

Here, the proposed model of the combined measures is a much more accurate predictor for evolutionary testability as any of the single measures. This is represented through the much higher correlation coefficient $r = 0.826880$. The inclusion of the previously omitted indicators ADN and ADPC produces an even higher outcome. Here, the correlation coefficient is $r = 0.897923$. The two measures seem to have an impact on evolutionary testability. The results of this regression and the model are displayed in table 13. The resulting graph of the prediction for the test objects under consideration is displayed in figure 11 on page 68.

The produced results support the assumption that evolutionary testability may be predicted from the measures introduced in this thesis. The prediction accuracy of almost 90 % for the last prediction system is promising. Since some aspects

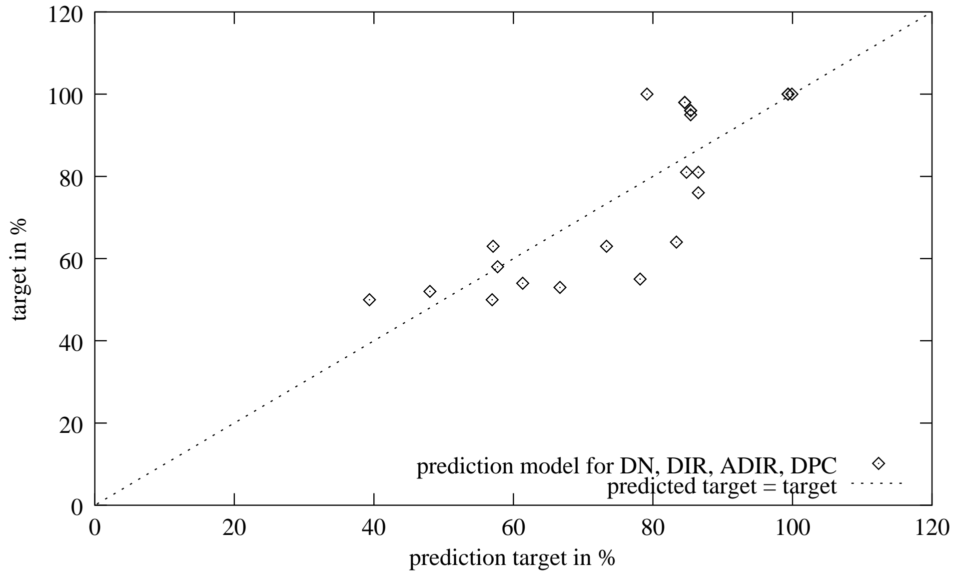

Figure 10: The prediction of evolutionary testability through the combination of the measures DN, DIR, ADIR and DPC, predicted performance (predicted target in %) against the actual performance (target in %).

regression coefficients, b_i	measure, x_i	mathematical model
$b_0 = 119.742449$		$f(x_1, x_2, x_3, x_4, x_5, x_6) = b_0$
$b_1 = 2.06803$	DN	$+b_1 * x_1$
$b_2 = -21.92733$	ADN	$+b_2 * x_2$
$b_3 = -7.411842$	DIR	$+b_3 * x_3$
$b_4 = 6.915588$	ADIR	$+b_4 * x_4$
$b_5 = -2.913803$	DPC	$+b_5 * x_5$
$b_6 = -0.888723$	ADPC	$+b_6 * x_6$
correlation coefficient $r = 0.897923$		

Table 13: Multiple regression for the measures DN, ADN, DIR, ADIR, DPC and ADPC. The graph of the prediction is displayed in figure 11.

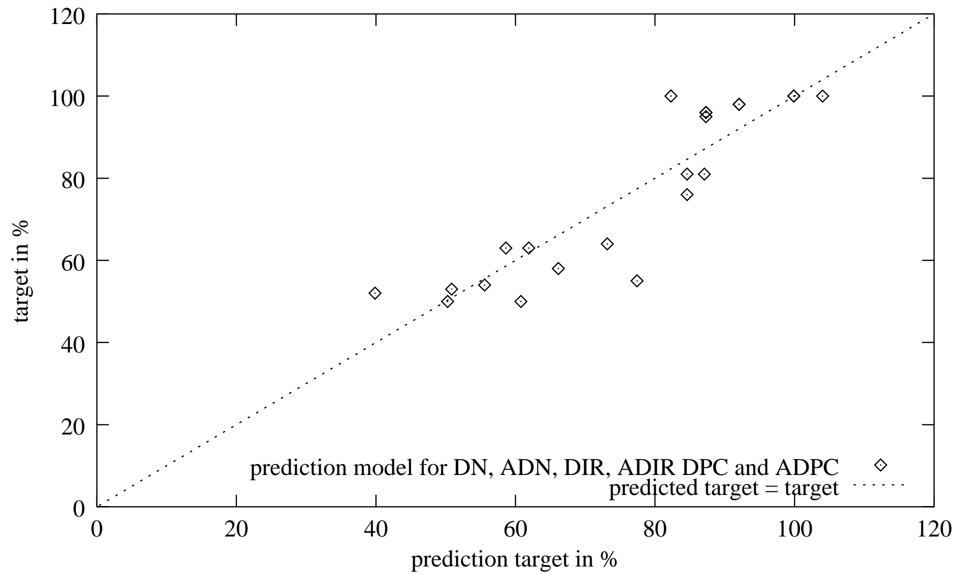

Figure 11: The prediction of evolutionary testability through the combination of the measures DN, ADN, DIR, ADIR, DPC and ADPC predicted performance (predicted target in %) against the actual performance (target in %).

of evolutionary testability have not yet been considered, this must be seen as a preliminary outcome which is still to change. The results from this brief statistical analysis must not be regarded as a proof of the hypothesis of the thesis. The number of considered data samples is too low and the test objects are not representative for the purpose of this analysis. It must rather be seen as an indicator for the overall validity of this work.

4.5 Composition operations for the valid measures

The definition of composition operators for the introduced measures logically belongs to the definition of the measures (chapter 3). Though, it makes no sense to reason about composition operators as long as the measures have not been validated and found applicable to the purpose of measurement. Composition operators are only introduced for measures which have been found valid according to the previously introduced criteria.

The introduced measures have been based on single modules. As seen from the

discussion in section 3.6, a measure is only useful if it can be extended to the system level. Therefore, composition operations are required which calculate the complexity of several integrated modules as one single unit. This is important for evolutionary testing as it can be easily applied to the system level, where it is most useful.

The input space is determined at the top-level of the integration, so that composition is not needed. **IS** is simply calculated from the top-level module.

The measure **DPC** for a system is the sum of the DPC values of all occurrences of each individual module of which the system is consisting (each procedure call is counted). The composition operation $\circ(P_1, P_2) : \mathbf{P} \times \mathbf{P} \to \mathbf{P}$ for DPC for one module P_1 calling a second module P_2 can be defined as

$$\mathrm{DPC}(P_1 \circ P_2) = \mathrm{DPC}(P_1) + \mathrm{DPC}(P_2) \tag{22}$$

and the measure ADPC can be defined as

$$\mathrm{ADPC}(P_1 \circ P_2) = \begin{cases} \frac{\mathrm{DPC}(P_1 \circ P_2)}{\mathrm{D}_{IR}(P_1) + \mathrm{D}_{IR}(P_2)} & \text{if } \mathrm{D}(P_1) + \mathrm{D}(P_2) > 0 \\ 0 & \text{if } \mathrm{D}(P_1) + \mathrm{D}(P_2) = 0 \end{cases} \tag{23}$$

Defining a composition operation $\circ(P_1, P_2) : \mathbf{P} \times \mathbf{P} \to \mathbf{P}$ for the measure **DN** is more complicated. If a program P_2 is inserted into a program P_1 by a procedure call then the overall value $\mathrm{DN}(P_1 \circ P_2)$ depends upon the nesting level L on which P_2 is inserted into P_1. The total measure $\mathrm{DN}(P_1 \circ P_2)$ can be formally defined as

$$\mathrm{DN}(P_1 \circ P_2) = \begin{cases} \mathrm{DN}(P_1) - \mathrm{L} + (\mathrm{ADN}(P_2) + \mathrm{L}) * \mathrm{N}_P(P_2) & \text{if } \mathrm{ADN}(P_2) > 0 \\ \mathrm{DN}(P_1) & \text{if } \mathrm{ADN}(P_2) = 0 \end{cases} \tag{24}$$

$\mathrm{N}_P(P_2)$ is the number of decision nodes in P_2. The composition for the measure **ADN** can be calculated as

$$\mathrm{ADN}(P_1 \circ P_2) = \begin{cases} \frac{\mathrm{DN}(P_1 \circ P_2)}{\mathrm{N}_D(P_1) + \mathrm{N}_D(P_2)} & \text{if } \mathrm{N}_D(P_1) + \mathrm{N}_D(P_2) > 0 \\ 0 & \text{if } \mathrm{N}_D(P_1) + \mathrm{N}_D(P_2) = 0 \end{cases} \tag{25}$$

Many authors of software measures consider the 'wholeness' of a measure as an important property. Wholeness states that the measure of two composed objects should be more than that of the sum of the individual objects [54, 59]. It is arguable whether this principle is useful for evolutionary testability as it is based upon 'cognitive complexity'. Experiments which use sub-systems as test objects must be performed in order to assess this.

5 Conclusions and further research

The motivation for this work was initiated because it is difficult to assess the outcome of evolutionary testing as a dynamic timing analysis technique. Experiments showed that for some test objects, evolutionary testing can be applied successfully to generate test-cases which comply to the test criterion. However, ET was found to perform poorly on more 'complicated' test objects. The analysis of these programs revealed a strong relationship between the 'complexity' of the test objects and the quality of the outcome produced by evolutionary testing. The aim of this work was:

- The confirmation of the hypothetical relation between complexity and evolutionary testability.

  Due to the difficulty of describing evolutionary testability as an overall aspect of complexity, an overall relation between evolutionary testability and program complexity as formulated in this thesis, could not be established. Since the model of evolutionary testability complexity is still incomplete, it is difficult to confirm this. As far as trends are considered, this assumption seems to be correct. This was illustrated through the statistical analysis. It confirmed the assumption that the introduced measures might indeed be able to assess and predict evolutionary testability of programs. An important aspect for future work is to investigate the interactions of the individual measures and the overall impact on the performance of ET much more thoroughly. It could be shown that each individual aspect of complexity can indicate an impact on the performance of ET.

  An interesting outcome of this work is that, for the test programs under investigation, the determination of either the BCET or the WCET, but not both, is more difficult for ET. This needs further verification.

- The identification of program properties which inhibit evolutionary testability.

  Program properties which make an assessment of evolutionary testing difficult were identified. These are small input domains, high parameter and algorith-

mic interdependence, the size of the input vector and the nesting of a program. These properties were found to inhibit the success of evolutionary testing and to be the core factors responsible for its poor performance.

- The derivation and validation of measures to capture these attributes.

  A combined measure to capture all aspects of complexity was found to be insufficient and inexpressive. This shifted the research effort towards investigating aspects of evolutionary testability individually. For some of these, new measures were derived and introduced.

  The average of the two test objectives WCET and BCET has been used to argue about evolutionary testability. This seems a valid approach, since it cannot be decided a priori whether a branch is contributing to the longest or the shortest execution path. In order to determine this, both branches at a decision must be executed. The measures can therefore be used to indicate the difficulty of finding the WCET and the BCET.

  The individual measures are believed to provide reasonable prediction of ET performance for those aspects of complexity for which they are intended. No suitable measure could be identified and derived for 'algorithmic interdependence or interaction', and for the effects of input data types. This can be seen as major deficiency of this work and must be considered the most important area of future research.

The validation of the individual measures identified three core metrics which can act as indicators for evolutionary testability as discussed in section 3.7. These are IS ('input size'), DPC ('decision probability complexity') and DN ('decision nesting'). Although, decision nesting was found to be coupled with the measure DPC. It could be shown that they are based upon the ordinal scale and that they comply with the introduced requirements of measurement theory, their scale and their underlying models. The additional measures DIR, ADIR, ADPC and ADN could not be shown to be individual indicators for evolutionary testability, though ADPC and ADN are

useful for providing additional information about the complexity of the test program and DIR is used to determine DPC. The derivation of a prediction system from the introduced individual measures illustrates the usefulness of all these indicators for predicting evolutionary testability. The high prediction accuracy of nearly 90 % for the proposed model is quite promising. Though the validity of this outcome must not be overrated, since only 22 data samples were used.

Automation. Measures should be easy to retrieve, ideally by an automatic process [59]. The introduced measures have been developed with automation in mind. The measure IS is very easy to determine automatically, as its calculation is trivial (equation 11). The measure DN is based upon the source code of a program. It can be easily produced by a scanner which is typically contained in compiler packages or it can be retrieved from an existing software-engineering package which generates control flowgraphs. Producing the measure DPC is more difficult. It is based upon decisions with input references (D_{IR}) and these must be determined by an automatic process. Program slicing can be used to implement this. The probability of simple predicates which are consisting of single relations can be calculated, although arbitrarily complex predicates must be executed (tested) to determine the probability. This can also be implemented by program slicing. It is not a simple solution, but it is feasible in general. Future work should concentrate on the application of slicing techniques for this particular case.

The problem of determining 'decision probabilities' (DPC) from the source code can be circumvented by retrieving this from the design phase. The values or ranges of values for which an action is performed in a function, must have been determined prior to coding. Hence, it must be available during the design phase of the module. Here, it is much easier to retrieve than from the source code. The same applies for the input ranges from which the size and data type of the input vector is specified. Moving measurement into earlier software development phases is most desirable and should be encouraged [59]. If evolutionary testability can be determined within

the design phase, then the applicability of ET can be predicted much earlier. The measure DN cannot be retrieved from earlier phases of the development process as nesting is dependent upon the implementation of the design. Research into the early application of testability measures is still needed and worthwhile.

The derivation of specific design principles is an important property of software measures. Many researchers have emphasised this (compare [59]). Ideally, a software measure should be able to indicate and assess possible changes of the design. As discussed in section 3.1, evolutionary testability of a test object can be improved through specific design principles. Low coupling and high cohesion [57] have been identified as being advantageous for evolutionary testing. Low nesting [25] is another principle which has a positive effect on evolutionary testing. These are only examples of how the design of a program can be changed in order to achieve improved evolutionary testability. Overall, every effort which attempts to decrease the values of the core measures (IS, DN and DPC) identified in this thesis, makes the outcome of ET more accurate and predictable. Subsequent work should concentrate on the derivation of design principles which are able to increase evolutionary testability.

Consistency of outcome. There are two alternatives for improving the outcome of evolutionary testing:

- improving the optimisation technique, or
- improving the software which is to be tested.

This work belongs into the scope of the second alternative. It concentrates on the software-engineering aspects of evolutionary testing.

There is possible potential for improving evolutionary algorithms in order to make them more readily applicable to software testing. This may include the development of new genetic operators which are optimised for this purpose, the definition

of new fitness functions, or alternative representations for the input parameters. Some effort has already been committed to the development of alternative genetic operators [28]; for example their optimisation for specific classes of test programs. In this case, the work of this thesis could possibly be extended to classify programs according to the genetic operators which are most likely to grant maximum success.

The results in this thesis only apply in combination with the used evolutionary algorithm. A change in the genetic operators may cause a change in the outcome of evolutionary testing. In this case, this work will have to be repeated for any change in the evolutionary technique in order to re-adjust the measures. This could be greatly simplified through the development of a library of typical modules on which each specific algorithm can be assessed.

Improving the process of evolutionary testing. The role of the fitness function has been neglected in this thesis. The use of a test object's execution time as fitness, or the coverage of its source code annotations as a simplification for this particular instance, seemed a reasonable approach. For this purpose, code annotations were found advantageous, as they are easy to implement and easy to control. The longest or shortest execution paths occur, where the search technique encounters the code annotations. They can be placed at arbitrary locations in the program. In this way, it is straightforward to conduct experiments according to many different aspects. The longest or shortest paths are simply 'defined' during the experiment. Through the representation of execution time with source code annotations, the whole problem of timing analysis is seen as a problem of structural coverage. As a simplification, timing analysis can be seen as structural testing (path testing), plus finding the execution times along the tested paths. Therefore, this work is also valid for the application of evolutionary techniques to structural testing.

As it is, the fitness function is only capable of producing execution times for sections of the program which can be easily reached during testing. With this

in mind, it becomes obvious that for test objects with extremely low 'decision probabilities', the assumed global search is in fact a local search within those areas of the search volume which are represented by the initial population. In order to produce execution times for all paths in the program, key combinations must be found which allow access to branches behind such decisions. For many applied input combinations no change in execution time is observed, until a key combination is produced which causes a change in the expression of the decision, and makes a new path available to be inspected. This results in a change of the produced execution time and generates feedback for the evolutionary search. In terms of the search volume, this situation can be thought of as 'a pinnacle sticking out of a flat plateau' at a decision. It makes the deficiency of a fitness function apparent which is entirely based upon execution time. Around the decision, the search of an EA is like a random search, as no feedback is generated by the fitness. The real power of evolutionary algorithms is not effectively exploited for these cases. An ideal fitness function should not only pursue the test target (longest or shortest execution time) but also guide the search into areas which are insufficiently sampled. This may be achieved through a hybrid fitness function which takes this into account and effectively 'attracts' individuals into areas where there is a possible optimal solution. A fitness function which aims to find data domain boundaries in a program is introduced by Sthamer [46]. This could possibly be used to extend the fitness function used for this work. Hybrid fitness functions are definitely going to be an important issue for future work in the area of evolutionary testing and their development is likely to have a major impact on the performance of ET. At this point in time it is not clear how to implement such a hybrid fitness function.

Cognitive complexity. For this work, complexity was defined as 'difficulty for the search technique to generate satisfactory test cases'. This was regarded in contrast to 'cognitive complexity'. Considering the aspects of complexity which were

put forward for evolutionary testability, it becomes apparent that complexity as it is 'seen' by an evolutionary algorithm is not much different from the way humans may experience this. This was most obvious for programs which were difficult to inspect and analyse in order to determine their extreme timing paths. In general, ET performed poorly on these test objects as well. Especially programs which exhibit extreme algorithmic complexity, were difficult to analyse manually and difficult to test by ET. The same applies for deep nesting, many input interactions, small domains, and large and complex input vectors which are particularly difficult to handle by humans.

After all, it may turn out that complexity as defined for a search technique is also valid for human cognition. An attempt to investigate and verify this might reveal very interesting results.

References

[1] R. Bache and M. Müllerburg. Measures of testability as a basis for quality assurance. *Software Engineering Journal*, pages 86–92, March 1990.

[2] V.R. Basili and H.D. Rombach. The tame project: towards improvement-oriented software environments. *IEEE Transactions on Software Engineering*, 14(6):758–773, 1988.

[3] B. Beizer. *Software Testing Techniques.* Van Nostrand Reinhold, New York, 1990.

[4] L.A. Belady and C.J. Evangelisti. A graph representation of structured programs. *IBM Systems*, 19(4):542–553, 1980.

[5] L.A. Belady and C.J. Evangelisti. System partitioning and its measure. *Systems and Software 2*, pages 23–29, 1981.

[6] K.A. De Jong and W.M. Spears. An analysis of the interacting roles of population size and crossover in genetic algorithms. In *International Conference on Parallel Problem Solving from Nature*, Dortmund, Germany, October 1990.

[7] N.E. Fenton and S.L. Pfleeger. *Software Metrics - A Rigorous & Practical Approach.* Thomson Computer Press, London, 1996.

[8] D.E. Goldberg. *Genetic algorithms in search, optimization and machine learning.* Addison-Wesely, Reading, MA, 1989.

[9] H.G. Groß, B.F. Jones, and D.E. Eyres. Evolutionary algorithms for the verification of execution time bounds for real-time software. In *IEE Workshop on applicable modelling, verification and analysis techniques*, London, 11.01.1999.

[10] H.G. Groß, B.F. Jones, and D.E. Eyres. Predicting the effectiveness of evolutionary testing for the measurement of extreme execution times. In *ICSE 2000,*

1st International Workshop on Automated Program Analysis, Testing and Verification, Limerick, Ireland, June 2000.

[11] H.G. Groß, B.F. Jones, and D.E. Eyres. A structural performance measure of evolutionary testing applied to worst-case timing of real-time systems. *IEE Proceedings Software*, 2000. To appear.

[12] M.H. Halstead. *Elements of Software Science*. Elsevier, New York, 1977.

[13] M. Harman and K.B. Gallagher. Program slicing. *Information and Software Technology*, 40:577–581, 1998.

[14] M.G. Harmon, T.P. Baker, and D.B. Whalley. A retargetable technique for predicting execution time of code segments. *Real-Time Systems*, Sep 1994.

[15] W. Harrison and K. Magel. A complexity measure based on nesting level. *Sigplan Notices*, 16(3):63–74, 1981.

[16] J. Holland. *Adaption in natural and artificial systems*. MIT Press, Cambridge, MA, 1975.

[17] S.T. Holmes. *Heuristic Generation of Software Test-Data*. PhD thesis, School of Computing, University of Glamorgan, Pontypridd, Wales, UK, 1996.

[18] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):35–46, 1990.

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

[20] B.F. Jones, H. Sthamer, X. Yang, and D.E. Eyres. The automatic generation of software test data sets using adaptive search techniques. In *3rd International Conference on Software Quality Management*, Seville, Spain, 1995.

[21] B.F. Jones, H.H. Sthamer, and D.E. Eyres. Automatic structural testing using genetic algorithms. *Software Engineering Journal*, pages 299–306, September 1996.

[22] J. Koza. *Genetic Programming: on the programming of computers by means of natural selection.* MIT Press, Cambridge, Mass., 1992.

[23] K. Kuhlmann. Complexity, coupling and binding in object-orientated software development. In *5. GI Workshop Software Metrics, Technische Universität Berlin*, Berlin, 1995.

[24] T. McCabe. A complexity measure. *IEEE Transactions of Software Engineering*, SE-1(3):313–327, 1976.

[25] S. McConnell. *Code Complete.* Microsoft Press, Redmond, Washington, 1993.

[26] A.C. Melton, D.A. Gustafson, J.M. Bieman, and A.L. Baker. A mathematical perspective for software measures research. *Software Engineering Journal*, pages 246–254, September 1990.

[27] B.L. Miller and D.E. Goldberg. Genetic algorithms, tournament selection and the effects of noise. *Complex Systems*, 9, 1996.

[28] F. Müller and J. Wegener. A comparison of static analysis and evolutionary testing for the verification of timing constraints. In *Fourth IEEE Real-Time Technology and Applications Symposium*, Denver, Colorado, June 1998.

[29] G.L. Myers. An extension of the cyclomatic measure of program complexity. *Sigplan Notices*, 12(10):61–64, 1997.

[30] K.D. Nilsen and B. Rygg. Worst-case execution time analysis on modern processors. *ACM SIGPLAN Notices*, 30(11):20–30, November 1995.

[31] M. O'Sullivan, S. Vössner, and J. Wegener. Testing temporal correctness of real-time systems – a new approach using genetic algorithms and cluster analysis.

In *6th International European Conference on Software Testing, Analysis and Review*, München, Nov/Dec 1998.

[32] C.Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, 5:31–62, 1993.

[33] R.S. Pressman. *Software Engineering*. McGraw-Hill, New York, fourth edition, 1997.

[34] P. Puschner. A tool for high-level language analysis of worst-case execution times. In *10th Euromicro Workshop on Real-Time Systems*, Berlin, 1998.

[35] P. Puschner. Worst-case execution-time analysis at low cost. *Control Engineering Practice*, 6:129–135, 1998.

[36] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *Real-Time Systems*, 1:159–176, 1989.

[37] P. Puschner and R. Nossal. Testing the results of static worst-case execution-time analysis. In *19th IEEE Real-Time Systems Symposium (RTSS98)*, Madrid, Dec 1998.

[38] P. Puschner and A. Schedl. Computing maximum task execution times – a graph based approach. *Real-Time Systems*, 13:67–91, 1997.

[39] P. Puschner and A. Vrchoticky. Problems in static worst-case execution time analysis. In *ITG/GI-Fachtagung Messung, Modellierung und Bewertung von Rechen- und Kommunikationssystemen, Kurzbeiträge und Toolbeschreibungen*, pages 18–25, Freiberg, Germany, Sep 1997.

[40] N. Ramsey and M.F. Fernandez. The new-jersey machine code toolkit. In *Usenix Technical Conference*, New Orleans, LA, 1995.

[41] N.F. Schneidewind. Methodology for validating software metrics. *IEEE Transactions on Software Engineering*, 18(5):410–421, May 1992.

[42] H.P. Schwefel and R. Männer. *Parallel problem solving from nature.* Springer, Berlin, 1990.

[43] M. Shepperd and D. Ince. *Derivation and Validation of Software Metrics.* Clarendon, Oxford, 1993.

[44] I. Somerville. *Software Engineering.* Addison-Wesley, Harlow, fifth edition, 1995.

[45] W.M. Spears and K.A. De Jong. On the virtues of parameterized uniform crossover. In *Proceedings of the fourth International Conference on Genetic Algorithms*, 1991.

[46] H.H. Sthamer. *The Automatic Generation of Software Test Data.* PhD thesis, Department of Electronics and Information Technology, University of Glamorgan, Pontypridd, Wales, UK, 1995.

[47] N. Storey. *Safety-critical computer systems.* Addison-Wesley, Harlow, England, 1996.

[48] N. Tracey, J. Clark, and K. Mander. The way forward in unifying dynamic test case generation: the optimisation-based approach. In *Proceedings of the IFIP'98 International Workshop on Dependable Computing and its Applications*, South Afrika, Jan 1998.

[49] P.J.M. von Laarhoven and E.H.L. Aarts. *Simulated Annealing: Theory and Applications.* Mathematics and its Applications. Kluwer, Dordrecht, 1987.

[50] A.E.L. Watkins. A tool for the automatic generation of test data using genetic algorithms. In *Proceedings of Software Quality Conference*, Dundee, Scotland, 1995.

[51] J. Wegener and M. Grochtmann. Verifying timing constraints of real-time systems by means of evolutionary testing. *Real-Time Systems*, 3(15):275–298, 1998.

[52] J. Wegener, H. Sthamer, B.F. Jones, and D. Eyres. Testing real-time systems using genetic algorithms. *Software Quality Journal*, 6(2):127–135, June 1997.

[53] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[54] E.J. Weyuker. Evaluating software complexity measures. *IEEE Transactions on Software Engineering*, 14(9):1357–1365, September 1988.

[55] D. Whitley. The genitor algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. In *International Conference on Genetic Algorithms*, pages 116–129, 1989.

[56] S. Xanthakis, C. Ellis, C. Skourlas, A Le Gall, and S. Katsikas. Application of genetic algorithms to software testing. In *5th International Conference on Software Engineering*, Toulouse, France, 1992.

[57] E. Yourdon and L.L. Constantine. *Structured Design.* Prentince Hall, Englewood Cliffs, NJ, 1979.

[58] H. Zuse. *Software Complexity.* Walter de Gruyter, New York, 1991.

[59] H. Zuse. *A Framework of Software Measurement.* Walter de Gruyter, New York, 1998.

Appendix: Source codes of the test programs

Module	Source code on page
delevat	85
bhorner	85
bcastel	85
bs1	87
bs2	87
is	88
polex	89
polex1	90
dzz	92
dzz1	93
dzz2	93
dzz3	94
exp	95
diff	97
sobel	97
min	98
median	99
gstretch	99
train	no source code
train1	no source code
dummy1/2	103

delevat. Input size is 28 bytes, *coeff*[6] and *cx*. *Degree* equals 6 for this instance.

```
float delevat (short degree,
               float* coeff,
               float* cx     )
{
  int i;
  int degree1 = degree + 1;
  cx[0] = coeff[0];
  for (i=1; i<=degree; i++)
    cx[i] = i + coeff[i-1] + (degree1 - i) * coeff[i];
  cx[degree1] = coeff[degree]
}
```

bhorner. Input size is 28 bytes, *coeff*[6] and *t*. *Degree* equals 6 for this instance.

```
float bhorner (short degree,
               float* coeff,
               float* t      )
{
  int n_choose_i = 1;
  float fact     = 1.0;
  float t1       = 1.0 - t;
  float aux      = coeff[0] * t1;
  for (int i=1; i<degree; i++) {
    fact       = fact * t;
    n_choose_i = (int) n_choose_i * (degree-i+1) / i;
    aux        = (aux + fact * n_choose_i * coeff[i]) * t1;
  }
  aux = aux + fact*t*coeff[degree];
  return aux;
}
```

bcastel. Input size is 28 bytes, *coeff*[6] and *t*. *Degree* equals 6 for this instance.

```
float bcastel (short degree,
               float* coeff,
               float  t     )
{
  float t1 = 1.0 - t;
  float coeff_ret;
  float* coeff_tmp = new float [degree];
  memcpy ((char*) coeff_tmp, 0, degree*sizeof(float));
  for (int i=1; i<= degree; i++)
    for (int j=0; j<=degree - i; j++)
      coeff_tmp[j] = t1 * coeff_tmp[j] + t* coeff_tmp[j+1];
  coeff_ret = coeff[0];
  delete [] coeff_tmp;
  return coeff_ret;
}
```

bs1. Input is 256 4-byte integers (1024 bytes). The variable *sz* is set to be 256.

```
template <class T>
void bs1 (T* List, int sz) {
  T tmp;
  for (int i=sz; i>=0; i--) {    /* fixed coverage = 257     */
    for (int j=1; j<i; j++) {    /* fixed coverage = 32896   */
      if (List[j-1] > List[j]) { /* fixed coverage = 32640   */
        COVER (0,1);
        tmp = List[j-1];         /* variable cover = 0-32640 */
        List[j-1] = List[j];     /* variable cover = 0-32640 */
        List[j] = tmp;           /* variable cover = 0-32640 */
      }
    }
  }
}
```

bs2. Input is 256 4-byte integers (1024 bytes). The variable *size* is set to be 256.

```
template <class T>
void bs2 (T* List, int sz) {
  char sorted;
  T tmp;
  do {
    sorted = 1;                  /* variable cover = 1-256     */
    COVER(0,1);
    for (int i=0; i<sz-1; i++) { /* variable cover = 257-65536 */
      if (List[i] > List[i+1]) { /* variable cover = 0-65280   */
        COVER(1,1);
        sorted = 0;              /* variable cover = 0-65280   */
        tmp = List[i];           /* variable cover = 0-65280   */
        List[i] = List[i+1];     /* variable cover = 0-65280   */
        List[i+1] = tmp;         /* variable cover = 0-65280   */
      }
    }
```

```
  } while (!sorted);             /* variable cover = 1-256     */
}
```

is. Input size is 256 4-byte integers (1024 bytes). The variable *size* is set to be 256.

```
template <class T>
void is (T* List, int sz) {
  T tmp;
  int j;
  for (int i=1; i<sz; i++) {     /* fixed coverage = 256       */
    tmp = List[i];               /* fixed coverage = 255       */
    j = i;                       /* fixed coverage = 255       */
    while (j>0 &&                /* variable cover = 255-32895 */
      COVER(0,1);
      List[j-1]>tmp) {           /* variable cover = 0-32640   */
      List[j] = List[j-1];       /* variable cover = 0-32640   */
      j--;                       /* variable cover = 0-32640   */
    }
    List[j] = tmp;               /* fixed coverage = 255       */
  }
}
```

polex. Input size is 1080 bytes.

```
void polex (char stripe, short ypos, char direction, short* grid) {
  float a,b,c;
  float x1,x2,x3,y1,y2,y3;
  /* fixed coverage = 1 */
  if (direction == DOWN) {
    COVER(0,1); /* variable cover = 0-1 */
    y1 = *(grid+stripe*SCANLINES+ypos-7);
    x1 = (ypos-7)*6;
    y2 = *(grid+stripe*SCANLINES+ypos-4);
    x2 = (ypos-4)*6;
    y3 = *(grid+stripe*SCANLINES+ypos-1);
    x3 = (ypos-1)*6;
  }
  /* fixed coverage = 1 */
  if (direction == UP) {
    COVER(0,1); /* variable cover = 0-1 */
    y1 = *(grid+stripe*SCANLINES+ypos+7);
    x1 = (ypos+7)*6;
    y2 = *(grid+stripe*SCANLINES+ypos+4);
    x2 = (ypos+4)*6;
    y3 = *(grid+stripe*SCANLINES+ypos+1);
    x3 = (ypos+1)*6;
  }
  /* fixed coverage = 1 */
  a = - (x1 * (y2 - y3) - (x2 * (y1 - y3) + x3 * (y2 - y1))) /
      ((x1 * x2 - x1 * (x2 + x3) + x2 * x3) * (x2 - x3));
  b = - (a * (x1 * x1 - x2 * x2) + y2 - y1) / (x1 - x2);
  c = - (a * x1 * x1 + b * x1 - y1);
  *(grid+stripe*SCANLINES+ypos)
                      = (int) (a*ypos*ypos*36 + b*ypos*6 + c);
  /* fixed coverage = 1 */
  if ((*(grid+stripe*SCANLINES+ypos+direction) == 0)) {
```

```
    COVER(0,1) /* variable coverage = 0-1 */
    *(grid+stripe*SCANLINES+ypos+direction) =
       (int) (a * (ypos + direction) * (ypos + direction) + 36 +
    b * (ypos + direction) * 6 + c);
  }
}
```

polex1. Input size is 16 bytes.

```
void polex1 (
              /*IN*/ short diff7, /*IN*/ short pos7,
              /*IN*/ short diff4, /*IN*/ short pos4,
              /*IN OUT*/ short* diff1, /*IN*/ short pos1,
              /*IN OUT*/ short* diff0, /*IN*/ short pos0
            )
{
  float a,b,c;
  float x1,x2,x3,y1,y2,y3;
  /* fixed coverage = 1 */
  y1 = (float) diff7;
  x1 = (float) pos7 * 6;
  y2 = (float) diff4;
  x2 = (float) pos4 * 6;
  y3 = (float) (*diff1);
  x3 = (float) pos1 * 6;
  a = -( x1*(y2-y3) - ( x2*(y1-y3) + x3 * (y2-y1) ) );
  b = -( a*(x1*x1-x2*x2) + y2 - y1 ) / (x1 - x2);
  c = -( a*x1*x1 + b*x1 - y1 );
  *diff0 =  (short)(a*pos0*pos0*36 + b*pos0*6 + c);
  /fixed coverage = 1*/
  if (*diff1 == 0) {
    COVER(0,1) /* variable coverage = 0-1 */
    *diff1 = (short)(a*pos1*pos1*36 + b*pos1*6 + c);
  }
```

```
}
```

dzz. Input size is 1080 bytes.

```
int STRIPES = 20;
int SCANLINES = 27;
void dzz (short *grid) {
  int           stripe;
  int           ypos;
  int           a,b,c;
  /* fixed coverage = 21 */
  for(stripe=0; stripe<STRIPES; ++stripe) {
    /* fixed coverage = 520 */
    for(ypos=1; ypos<(SCANLINES-1); ++ypos) {
      /* fixed coverage = 500 */
      if((*(grid+stripe*SCANLINES+ypos) < 0) ||
         (*(grid+stripe*SCANLINES+ypos) > 510)) {
         COVER(0,1);
         /* variable cover = 0-500 */
         *(grid+stripe*SCANLINES+ypos)=0;
      }
      /* fixed coverage = 500 */
      a=*(grid+stripe*SCANLINES+ypos-1);
      b=*(grid+stripe*SCANLINES+ypos);
      c=*(grid+stripe*SCANLINES+ypos+1);
      /* fixed coverage = 500 */
      if( (abs(a-c) < 30)
          && (abs(b - (a+c)/2) > 20) ) {
        COVER(1,1);
        /* variable cover = 0-500 */
        *(grid+stripe*SCANLINES+ypos)=0;
      }
      /* fixed coverage = 500 */
      if( abs(a-b) > 40 ) {
        COVER(2,1);
        /* variable coverage = 0-500 */
```

```
        *(grid+stripe*SCANLINES+ypos-1) = 0;
        *(grid+stripe*SCANLINES+ypos)=0;
      }
    }
  }
}
```

dzz1. Input size is 1080 bytes.

```
void dzz1 (short* grid) {
  /* fixed coverage = 21 */
  for (int i=0; i<STRIPES; i++) {
    /* fixed coverage = 525 */
    for (int j=1; j<SCANLINES-1; j++) {
      /* fixed coverage = 500 */
      if ((*(grid+i*SCANLINES+j)) < 0  ||
          (*(grid+i*SCANLINES+j)) > 510  ) {
        COVER(0,1);
        /* variable coverage = 0-500 */
        *(grid+i*SCANLINES+j)=0;
      }
    }
  }
}
```

dzz2. Input size is 1080 bytes.

```
void dzz2 (short* grid) {
  /* fixed coverage = 21 */
  for (int i=0; i<STRIPES; i++) {
    /* fixed coverage = 520 */
    for (int j=1; j<SCANLINES-1; j++) {
      /* fixed coverage = 500 */
      short a = *(grid+i*SCANLINES+j-1);
      short b = *(grid+i*SCANLINES+j);
```

```
      short c = *(grid+i*SCANLINES+j+1);
      /* fixed coverage = 500 */
      if( (abs(a-c) < 30) &&
          (abs(b - (a+c)/2) > 20) ) {
        COVER(0,1);
        /* variable coverage = 0-500 */
        *(grid+i*SCANLINES+j)=0;
      }
    }
  }
}
```

dzz3. Input size is 1080 bytes.

```
void dzz3 (short* grid) {
  short a, b;
  /* fixed coverage = 21 */
  for (int stripe=0; stripe<STRIPES; stripe++) {
    /* fixed coverage = 520 */
    for (int ypos=1; ypos<SCANLINES-1; ypos++) {
      /* fixed coverage = 500 */
      a = *(grid+stripe*SCANLINES+ypos-1);
      b = *(grid+stripe*SCANLINES+ypos);
      /* fixed coverage = 500 */
      if (abs(a-b) > 40) {
        COVER(0,1);
        /* variable coverage = 0-500 */
        *(grid+stripe*SCANLINES+ypos-1) = 0;
        *(grid+stripe*SCANLINES+ypos)   = 0;
      }
    }
  }
}
```

exp. Input size is 1080 bytes.

```
inline int exp_pattern (short x, short x_1, short x_4, short x_7) {
  if ( x_1 > 0) {
    COVER(0,1);
    if (x_4 > 0) {
      COVER(1,1);
      if (x_7 > 0) {
        COVER(2,1)
        if (x == 0) {
          COVER(3,1);
          return 1;
        }
      }
    }
  }
  return 0;
}

void exp (short* grid) {
  int stripe, ypos;
  /* fixed coverage = 21 */
  for (stripe=0; stripe<STRIPES; stripe++) {
    /* fixed coverage = 420 */
    for (ypos=7; ypos<SCANLINES; ypos++) {
      /* fixed coverage = 400 */
      if (exp_pattern ( *(grid+stripe*SCANLINES+ypos),
                        *(grid+stripe*SCANLINES+ypos-1),
                        *(grid+stripe*SCANLINES+ypos-4),
                        *(grid+stripe*SCANLINES+ypos-7)  )
          )
      {
        /* EXTERNAL FUNCTION CALL */
        /* variable coverage = 0-10 */
```

```
        ypos=ypos+2;
      }
    }
  }
  /* fixed coverage = 21 */
  for (stripe=0; stripe<STRIPES; stripe++) {
    /* fixed coverage = 420 */
    for (ypos=SCANLINES-7; ypos>0; ypos--) {
      /* fixed coverage = 400 */
      if (expo_pattern ( *(grid+stripe*SCANLINES+ypos),
                         *(grid+stripe*SCANLINES+ypos+1),
                         *(grid+stripe*SCANLINES+ypos+4),
                         *(grid+stripe*SCANLINES+ypos+7)  )
          ) {
        /* EXTERNAL FUNCTION CALL */
        /* variable coverage = 0-10 */
        ypos=ypos-2;
      }
    }
  }
}
```

diff. Input is two picture frames of 512 bytes length each (1024 bytes).

```
typedef unsigned char BYTE;
void diff (BYTE* image1, BYTE* image2, BYTE* imageOut) {
  int i;
  int delta;
  /* fixed coverage = 513 */
  for (i = 0; i < width * height; i++) {
    /* fixed coverage 512 */
    delta = image1[i] - image2[i];
    /* fixed coverage = 512 */
    if (delta < 0) {
      COVER(0,1);
      /* variable coverage = 0-512 */
      delta = - delta;
    }
    /* fixed coverage = 512 */
    imageOut[i] = (BYTE)delta;
  }
}
```

sobel. Input size is 1024 bytes plus 1 byte for *white.*

```
inline int edge_detected (BYTE grad, BYTE white) {
  if (grad > white)
    return 1;
  else
    return 0;
}

void sobel (BYTE white, BYTE *imageIn, BYTE *imageOut) {
  int i;
  int grad;
  int deltaX, deltaY;
  /* fixed coverage = 1 */
```

```
  memset(imageOut, 0, width);
  /* fixed coverage = 961 *
  for (i = width; i < (height-1)*width; i++) {
    /* fixed coverage 960 */
    deltaX = 2*imageIn[i+1]
              + imageIn[i-width+1]
              + imageIn[i+width+1]
              - 2*imageIn[i-1]
              - imageIn[i-width-1]
              - imageIn[i+width-1];
    deltaY = imageIn[i-width-1]
              + 2*imageIn[i-width]
              + imageIn[i-width+1]
              - imageIn[i+width-1]
              - 2*imageIn[i+width]
              - imageIn[i+width+1];
    grad = (abs(deltaX) + abs(deltaY)) / 3;
    /* fixed coverage = 960 */
    if (edge_detected(grad,white)) {
      COVER(0,1);
      /* variable coverage = 0-960 */
      grad = white;
    }
    /* fixed coverage = 960 */
    imageOut[i] = (BYTE)grad;
  }
  /* fixed coverage = 1 */
  memset(imageOut + i, 0, width);
}
```

min. Input is 1024 bytes plus 1 byte for *white*.

```
void min(BYTE white, BYTE *imageIn, BYTE *imageOut) {
  int i;
```

```
  int val;
  int x, y;
  /* fixed coverage */
  memset(imageOut, 0, width);
  /* fixed coverage = 961 */
  for ( i = width; i < (height-1)*width; i++) {
    /* fixed coverage = 960 */
    val = white;
    /* fixed coverage  = 3840 */
    for (y = -1; y <= 1; y++) {
      /* fixed coverage = 11520 */
      for (x = -1; x <= 1; x++) {
        /* fixed coverage 8640 */
        if (imageIn[i + x + y * width] < val) {
          COVER(0,1);
          /* variable coverage = 0-7740 */
          val = imageIn[i + x + y * width];
        }
      }
    }
    /* fixed coverage = 960 */
    imageOut[i] = (BYTE)val;
  }
  /* fixed coverage = 1 */
  memset(imageOut + i, 0, width);
}
```

median. Input is 1024 1-byte characters (1024 bytes).

```
template <class T>
void IPL_median(T * imageIn, T * imageOut)
{
  const int matrix_size = 9;
  T values[matrix_size];
```

```
int i;
/* fixed coverage = 1 */
memset(imageOut, 0, width);
/*
 * loop over the image, starting at the second row, second column
 */
/* fixed coverage 961 */
for (i = width; i < width * (height - 1); i++) {
  /* fixed coverage 960 */
  memcpy ((char*)&values,   &imageIn[i-width-1], 3 );
  memcpy ((char*)&values+3, &imageIn[i-1],       3 );
  memcpy ((char*)&values+6, &imageIn[i+width-1], 3 );
  /* ---------------------------------------------- */
  { /* insertion sort input is values[9] */
    int x, min;
    T tmp;
    /* fixed coverage = 9600 */
    for (x=0; x<9; x++) {
      /* fixed coverage = 8640 */
      tmp = values[x];
      min = x;
      /* variable coverage = 12139-39678 */
      while ( (min>0) &&
              (values[min-1] > tmp) ) {
        COVER(0,1);
        /* variable coverage = 3499-31038 */
        values[min] = values[min-1];
        min--;
      }
      /* fixed coverage = 8640 */
      values[min] = tmp;
    }
  } // end insertion sort
  /* ---------------------------------------------- */
```

```
    /* fixed coverage = 960 */
    imageOut[i] = values[4];
  }
  /* fixed coverage = 1 */
  memset(imageOut + i, 0, width);
}
```

gstretch. Input is 1024 1-byte characters (1024 bytes).

```
template <class T>
void gstretch(T* imageIn, T* imageOut) {
    int i;
    T maxVal;
    T minVal;
    T range;
    double factor;
    /* fixed coverage = 1 */
    maxVal = 0;
    minVal = 255;
    /* fixed coverage = 1025 */
    for (i = 0; i < width*height; i++) {
      /* fixed coverage = 1024 */
      if (imageIn[i] < minVal) {
        COVER(0,1);
        /* variable coverage = 0-1018 / 2 */
        minVal = imageIn[i];
      }
      /* fixed coverage = 1024 */
      if (imageIn[i] > maxVal) {
        COVER(0,1);
        /* variable coverage = 0-1018 / 2 */
        maxVal = imageIn[i];
      }
    }
```

```
    /* fixed coverage = 1 */
    range = maxVal - minVal; // maxVal und minVal sind INPUT
    if (range == 0) {
      COVER(1,1);
      /* variable coverage = 0-1 */
      range = 1;
    }
    /* fixed coverage = 1 */
    factor = (double)white / range;
    /* fixed coverage = 1025 */
    for (i = 0; i < width*height; i++)
        imageOut[i] = (BYTE)(factor * (imageIn[i] - minVal));
}
```

dummy1. Input size is 1080 bytes.

```
void dummy1 (short* grid) {
  int stripe;
  int ypos;
  /* fixed coverage = 21 */
  for (int stripe=0; stripe<STRIPES; stripe++) {
    /* fixed coverage = 525*/
    for (int ypos=1; ypos<SCANLINES-1; ypos++) {
      /* fixed coverage = 500 */
      if (*(grid+stripe*SCANLINES+ypos) < -2048 ||
          *(grid+stripe*SCANLINES+ypos) > 2048) {
        COVER(0,1);
        /* variable coverage = 0-500 */
        *(grid+stripe*SCANLINES+ypos)=0;
      }
    }
  }
  /* fixed coverage = 20 */
  for (stripe=1; stripe<STRIPES; stripe++) {
    /* fixed coverage = 400 */
    for (ypos=7; ypos<SCANLINES; ypos++) {
      /* fixed coverage = 380 */
      if (*(grid+stripe*SCANLINES+ypos-1) > 0) {
        COVER(1,1);
        if (*(grid+stripe*SCANLINES+ypos-4) > 0) {
          COVER(2,1);
          if (*(grid+stripe*SCANLINES+ypos-7) > 0) {
            COVER(3,1);
            if (*(grid+stripe*SCANLINES+ypos) == 0) {
              ypos += 2;
              COVER(4,1);
              /* variable coverage = 0-20 */
            }
```

```
          }
        }
      }
    }
  }
}
```

dummy2. INput size is 1080 bytes.

```
void dummy2 (short* grid) {
  int stripe;
  int ypos;
  /* fixed coverage = 21 */
  for (int stripe=0; stripe<STRIPES; stripe++) {
    /* fixed coverage = 520 */
    for (int ypos=1; ypos<SCANLINES-1; ypos++) {
      /* fixed coverage = 500 */
      if (*(grid+stripe*SCANLINES+ypos) < -28000) {
        COVER(0,1);
        /* variable coverage = 0-500 */
        *(grid+stripe*SCANLINES+ypos)=0;
      }
    }
  }
}
```

train* and *train1. Input size is 672 and 656 bytes. Unfortunately, there was no permission granted for publication.