A prediction system for dynamic optimisation-based execution time analysis

Hans-Gerhard Groß
Fraunhofer Institute
Experimental Software Engineering
Sauerwiesen 6, D-67661 Kaiserslautern, Germany.
E-Mail: grossh@iese.fhg.de

In: *First International Workshop on Software Engineering using Metaheuristic Innovative Algorithms (SEMINAL)*, ICSE 2001, Toronto, May 14, 2001.

Abstract

Evolutionary testing is an optimisation-based test-case generation technique. It can be applied to timing analysis of real-time systems where it is used to uncover temporal errors. Testability is the ability of the test technique to uncover faults. Evolutionary testability is the ability of an evolutionary algorithm to successfully generate test-cases with the goal to uncover faults, in this instance violations of the timing specification. Some properties of real-time programs were found to greatly inhibit evolutionary testability. These are small path domains, high-data dependence, large input vectors, and nesting. This paper defines source code measures which aim to express the effects of these properties on evolutionary testing. The measurement and prediction system developed from the experiments is able to forecast evolutionary testability with almost 90 % accuracy. The prediction system will be used to assess whether the application of evolutionary testing to a real-time system will be sufficient for successful dynamic timing analysis, or whether additional testing strategies are needed.

Introduction

The knowledge of the best- and worst-case task execution times (B/WCET) of a real-time system is essential for schedulability analysis. This analysis ensures that the system does not produce output too early or too late. The generation of results outside the predetermined time bounds is considered a temporal error. The traditional technique for schedulability analysis is static timing analysis. This initially determines the required time for executing a program's basic blocks based on the source code and then combines these into a cumulative total. The outcome represents the execution time for the entire task [1]. There are several criteria that software must satisfy before static timing analysis may be applied, for example the maximum number of loop iterations must be known, recursion and function variables must be absent, and possible sequences of program actions must be known (e.g. path information) [1, 2, 3]. Static timing analysis uses no information about input parameter values, since a program's source code does not provide these. It requires extensive human support and it is crucial that any provided information is correct and complete. All these requirements make static B/WCET analysis difficult, expensive and unreliable [4].

Evolutionary testing (ET) is based upon the application of an evolutionary algorithm (EA). It implements an optimisation technique that may be applied to analyse the timing behaviour of real-time systems. It can be used to verify whether the timing behaviour of a module complies with its specification. In this instance, timing analysis may be regarded as testing and the test target is to find temporal errors in the software. This is equivalent to finding the longest or shortest possible execution times for schedulability analysis, or to violating the timing specification of the task for verifying the correctness of timing [5].

Earlier experimental work was aimed at assessing the success rate of evolutionary testing to find the best- and worst-case execution times for a number of examples [6]. This identified difficulties of the technique on some test programs. Source code analysis of these examples indicated several attributes which inhibit the generation of test-cases according to the test objective under investigation.

This work introduces source-code measures with the intention to map program attributes which inhibit evolutionary testability (e.g. small path domains, high-data dependence, large input vectors, nesting) into figures and establish a prediction system for evolutionary testability. This measurement and prediction system can be applied to analyse real-

time software in order to verify whether evolutionary testing is likely to produce high-quality outcome in respect to the timing of a task.

ET: optimisation-based timing analysis

The dynamic best- and worst-case execution time analysis of real-time systems can be entirely regarded as a search or optimisation problem. The cost function for this optimisation is the execution time of the system or module under test [5], and the parameters to be optimised are the input situations with which the system is confronted during operation. The input parameters cause the system to exhibit different timing behaviour.

ET implements such an optimisation technique. It is based on evolutionary algorithms. An evolutionary algorithm (EA) performs on a population of strings, so-called chromosomes. These represent possible solutions to the optimisation problem under investigation. Some of these chromosomes are selected and then recombined to form new strings. These new strings are mutated and the resulting strings form a new generation of perspective solutions. The cost function determines their fitness, and this represents their ability to successfully solve the problem. More successful individuals have a higher chance of being selected and recombined to create 'offspring' for subsequent generations. The operation of this process can be represented by the following pseudo-code, with P, P1, P2, P3 as 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
```

This process is continued over many generations until the stopping criterion is satisfied (e.g. predetermined number of generations). EA generate new solutions based on information of successful existing solutions, so that the population consists of fitter individuals after many generations. These individuals are able to yield better results and represent better solutions [7, 8].

Software measures for ET

Small or single-value path domains and parameter dependent loops are caused by decision statements which execute one branch with low probability. This corresponds to covering a branch for very few or even only one single value of the input vector. It restricts the ability of an EA for large search spaces as it is unlikely to generate the required values to cover the low-probability branch purely by chance. Loops whose number of iterations depend upon input variables are equivalent to such decisions since the EA must generate input to perform the lowest or highest number of loop iterations. This property can be measured through *decision probability complexity* (DPC)

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

The measure is based upon the probability of a predicate to become *true* (p_{true}) or *false* (p_{false}). D_{IR} are all decisions with references to input. The reciprocal ensures that high complexity ($\equiv$ low probability) is represented by a high value for the measure. The expression +log(0.5) sets DPC = 0 if $p_{true} = p_{false} = 0.5$, so that the measure is zero if the domains of the decision are balanced. The sum of all decision complexities makes no statement about the distribution of individual decision complexities over the program. This can be indicated through ADPC (*average decision probability complexity*)

$$\mathrm{ADPC} = \begin{cases} \frac{\mathrm{DPC}}{\mathrm{D}_{IR}} & \text{if D} > 0 \\ 0 & \text{if D} = 0 \end{cases} \quad (2)$$

High data dependence or large input-vectors may be caused through input references in decisions. These decisions require some of the input values to be in a specific relation, for example a specified pattern, in order to lead the program flow into a distinct branch. High interdependence may also be caused through calculations on input variables. The values of the variables determine the time it takes to

perform a calculation, although this depends upon the system architecture. The total number of *input references in decisions* (DIR) can be calculated for a module

$$\mathrm{DIR} = \sum_{i=1}^{\mathrm{N}_{P,IR}} \mathrm{IR}(i) \tag{3}$$

$\mathrm{N}_{P,IR}$ is the total number of decision nodes with input references (typically all decisions except static loops). $\mathrm{IR}(i)$ is the number of input references in each predicate of a decision i. DIR does not indicate whether a program consists of many conditions with few input references – in this case data dependence is not as critical for ET – or whether it consists of only a few decisions which contain many input references. This is captured by the measure *average decision input references* (ADIR) which indicates the distribution of parameter references over the predicate nodes.

$$\mathrm{ADIR} = \begin{cases} \frac{\mathrm{DIR}}{\mathrm{N}_{P,IR}} & \text{if } \mathrm{N}_{P,IR} > 0 \\ 0 & \text{if } \mathrm{N}_{P,IR} = 0 \end{cases} \tag{4}$$

Nesting and sequencing is the combination of all previous items. It can be measured by the *decision nesting* (DN) of a program.

$$\mathrm{DN} = \sum_{i=1}^{\mathrm{N}_P} \mathrm{L}(i) \tag{5}$$

N_P is the number of predicate nodes in branch- or loop-conditional expressions in the program's flowgraph. This captures the nesting and sequencing of the decision-nodes. It concentrates on a program's predicates since once the program flow enters a branch behind a decision all statements on that branch will be executed and their timing determined. 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. The measure *average decision nesting* (ADN) indicates this difference.

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

Experiments

This work illustrates the empirical correlation between the introduced measures and evolutionary testability. Evolutionary testability is expressed as percent of the maximal or minimal achievable coverage of the worst- or best-case execution path. This coverage is not equivalent with real-time. It is only used as a representation of the maximal or minimal real execution time since the actual timing of the test objects is not known. The cost function can be implemented through code annotations which are inserted into the test objects' source codes along their shortest and longest algorithmic execution paths. The determination of the longest and shortest paths in a test program with this technique is entirely arbitrary. The experiments verify to which extend an evolutionary algorithm will generate input situations which cause the execution of a test object to cover these source code annotations. The maximal and minimal possible annotation coverage must have been determined for each test object prior to the experiments. The coverage criterion is defined as follows:

For worst-case execution time as test criterion 100 % coverage means that all code annotations which result in the longest timing path were executed during evolutionary testing. This includes multiple executions of the same code annotations in loops.

For best-case execution time as test criterion 100 % coverage means that all code annotations which result in the best-case timing path were executed (e.g. minimal number of loop iterations).

A simple genetic algorithm with the following attributes was used for optimisation: **Binary string** as chromosome [7]. The chromosome is an exact mapping of the memory for each input parameter set. **Population size** = 40. The algorithm always keeps the forty best solutions. The choice of the population size is a trade-off between sampling accuracy and performance limitations. This requires some experimentation. **Tournament selection** with tournament size = 4. The tournament size depends upon the population size. The chosen value implements selection which is more "elitist" than random selection. **Discrete recombination** with uniform crossover and crossover probability $\mathrm{p}_c = 0.5$. This has become a standard for genetic algorithms and it performs well regardless of the distribution of important values in the chromosome [9]. **Low constant mutation rate**, $\mathrm{p}_m = 0.001$. This value was successfully used for

many problems, e.g. Goldberg [7]. **Rank-based fitness** [10]. The chance of an individual to be selected for recombination depends on its rank in the population and not its objective fitness. The fitness merely determines its rank. **Random initialisation** of the chromosomes [7].

The test objects are modules taken from different applications. Table 1 displays name, functional description and input vector size for each used module. 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 determines the size of the search space. The input size of most modules is about one kilobyte. This large size compensates the relatively low algorithmic complexity of the used test programs. Large search spaces impose considerable difficulty on the optimisation process, particularly when low sampling accuracy of the search space is used.

Table 2 displays for each test object the number of nodes (N), number of predicate nodes (N_P), measure decision nesting (DN), the measure average decision nesting (ADN), the measure input references in decisions (DIR), the measure average input references in decisions (ADIR), the measure decision probability complexity (DPC) and the measure average decision probability complexity (ADPC). Additionally, the results of the evolutionary testing process are displayed: the values for the worst-case execution time (WCET), the values for the best-case execution time (BCET) and their arithmetic mean (avg). The average coverage of the B/WCET paths was used to develop the prediction system. The results for WCET and BCET are average values from at least ten repetitive trials. Each trial is equivalent to one evolutionary testing process.

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 underrepresented. For assessing the behaviour of evolutionary testing, a test object must have been fully analysed and its best- and worst-case paths determined. The worst- and best-case branches for most test programs 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 test objects, for example *median*, demand a detailed inspection and many manual executions with different test data to determine the shortest and longest paths. The true best- and worst-case paths for all test programs should be accurate, although, giving a guarantee for the more complex programs is impossible. The choice of test objects minimises this source of error. They are quite simple for this reason.

A prediction model for evolutionary testability

A measurement system is used to assess an existing entity by numerically characterizing one or more of its attributes [11]. Some candidate measures for evolutionary testability were introduced above. A detailed view on the measures can be found in [12]. A prediction system is used to anticipate some attribute of a future entity, involving an underlying mathematical model [11].

Each individual measure is not sufficient to indicate evolutionary testability accurately. For example, the correlation between evolutionary testability and the measure DN is only $r = 0.582472$ (determined through linear regression), or the correlation between the number of predicate nodes N_P and evolutionary testability is $r = 0.625125$.

A prediction becomes much more accurate if the individual measures from table 2 are combined to form a prediction system. The results in table 2 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. The results of a regression model which includes all individual measures is displayed in table 3. The resulting graph of the prediction for the test objects under consideration is displayed in figure 1. The correlation coefficient for this model is $r = 0.897923$.

Conclusion

The generated empirical data demonstrate that the introduced measures can predict evolutionary testability. The accuracy of almost 90 % for the introduced prediction model is promising.

Some aspects of evolutionary testability have not yet been considered. The generated results are preliminary and may still be improved upon if additional aspects are included. One important aspect is algorithmic complexity. It occurs when the state of the input is recursively altered according to its previous state. An example for this is a sorting algorithm where the execution of a branch in a loop depends not only upon the initial input values but additionally upon the entire state history of the input, in this case the list which is to be sorted. Such modules typically exhibit a low number of possible global maxima in the search space, although this depends upon the data type of the used variables. Future work will address ways to assess this kind of complexity.

The number of considered data samples is low for such an empirical investigation. Whether the used test objects are representative for all possible real-time components is questionable. A much broader study involving many more test objects will be performed to validate the evidence of this work.

This paper demonstrates that the prediction of evolutionary testability for real-time programs is feasible. The measurement and prediction system can be used to assess the expected evolutionary testability for components. The timing behaviour of components with high expected evolutionary testability can then be determined completely automatically with high confidence in the outcome. However, the tester must still provide additional information for timing analysis of modules with low expected evolutionary testability.

Measurement and prediction systems should always facilitate the improvement of the measured attributes. Further research will concentrate on the derivation of design guidelines which have the potential to improve the evolutionary testability of real-time components.

References

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

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

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

[4] 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.

[5] 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.

[6] 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*, 147(2), April 2000.

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

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

[9] 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.

[10] 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.

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

[12] H.-G. Groß. *Measuring evolutionary testability of real-time software*. PhD thesis, School of Computing, University of Glamorgan, Pontypridd, Wales, UK, July 2000.

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.

									coverage %		
Module	N	N_P	DN	ADN	DIR	ADIR	DPC	ADPC	WCET	BCET	avg
delevat	5	1	1	1	0	0	0	0	100	100	100
bhorner	6	1	1	1	0	0	0	0	100	100	100
polex1	13	1	1	1	1	1	4.5	4.5	100	100	100
bcastel	7	3	3	1	0	0	0	0	100	100	100
sobel	9	2	3	1.5	2	2	0	0	97	99	98
diff	6	2	3	1.5	2	2	0	0	96	100	98
bs2	10	3	6	2	2	2	0	0	96	95	96
is	9	3	6	2	2	2	0	0	96	95	96
bs1	7	3	6	2	2	2	0	0	95	95	95
dummy2	10	7	16	2.3	3	1	0	0	89	73	81
min	9	4	10	2.5	2	2	0	0	62	100	81
median	15	4	10	2.5	2	2	0	0	82	70	76
dzz1	6	4	9	2.3	2	1	1.8	1.8	100	28	64
dummy1	13	10	30	3	6	1	5.4	0.9	44	81	63
train1	16	5	15	3	4	1.3	1.8	0.5	25	100	63
gstretch	14	5	7	1.4	6	2	2.1	0.7	17	99	58
dzz3	5	3	6	2	2	2	2.6	2.6	100	9	55
dzz2	9	4	10	2.5	5	2.5	2.7	2.7	7	100	54
train	16	5	15	3	4	1.3	4.2	1.4	5	100	53
dzz	15	7	19	2.7	6	1.2	7.4	2.5	35	68	52
polex	17	3	3	1	3	1	8.7	2.9	0	100	50
exp	17	12	42	3.5	8	1	9	1.1	0	100	50

Table 2: Individual measures for evolutionary testability. The modules are ordered according to evolutionary testability (average achieved coverage 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 3: Multiple regression for the measures DN, ADN, DIR, ADIR, DPC and ADPC. The graph of the prediction is displayed in figure 1.

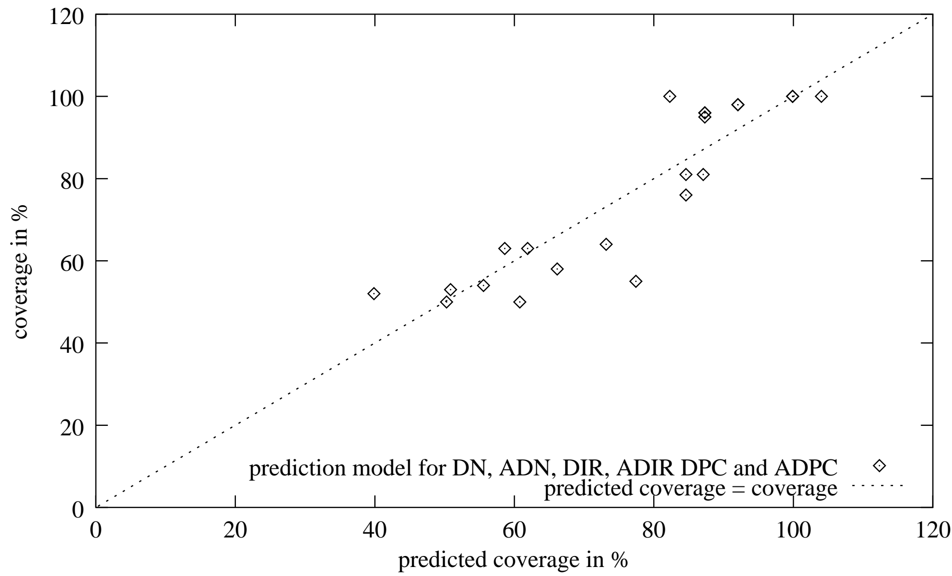

Figure 1: The prediction of evolutionary testability through the combination of the measures DN, ADN, DIR, ADIR, DPC and ADPC predicted performance (predicted coverage in %) against the actual performance (coverage in %). The correlation coefficient is $r = 0.897923$.