An Evaluation of Dynamic, Optimisation-based Worst-case Execution Time Analysis[1]

Hans-Gerhard Gross
Fraunhofer Institute Experimental Software Engineering
Sauerwiesen 6, 67661 Kaiserslautern, Germany
grossh@iese.fhg.de

Abstract

Evolutionary testing is a relatively new software testing technique based upon the application of evolutionary algorithms. It can be used to generate input situations for procedures automatically which satisfy a given test criterion. In this scope, testing is entirely regarded as a search or optimisation problem with the test criterion as cost function and the program input as parameters which must be optimised according to that cost function. For assessing the worst-case execution time of real-time software dynamically, evolutionary testing can be applied to optimise input parameter combinations with the aim to maximise the execution time of a program.

This paper shows the application of evolutionary testing to the automatic generation of maximal execution times for 15 example test programs. The outcome of this optimisation-based timing analysis is compared with random test case generation, and additionally, with the performance of an experienced human tester who attempted to generate test cases manually on the basis of the source codes which maximise execution times. Experiments show a high success rate of evolutionary testing for generating worst-case timing behaviour compared with the outcome of random testing. Random testing can only generate about 85 % of the execution times found by evolutionary testing. For only 4 out of the 15 test objects, the human tester was found to be more successful than the automatic test case generators. This can be explained by the pathology of these test objects.

1 Introduction

The knowledge of the worst-case execution time (WCET) is an essential requirement for the safe and correct operation of a real-time system. Assessing that all deadlines during system execution will be met is a difficult task in the development of such a system. 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. Software testing is a widely used and accepted technique for verification and validation of software products. It is used to generate modes of operation which show that the software is conforming to its specification, design and implementation requirements, and to support the confidence in its safe and correct operation. Evolutionary testing is a new optimisation-based automatic test case generation technique (O'Sullivan et al., 1998). It is based upon the application of genetic algorithms (GA) to generate test cases which satisfy a given test criterion. In the case of dynamic WCET analysis it can

[1]Proceedings of the International Conference on Information Technology: Prospects and Challenges in the 21st Century, Kathmandu, Nepal, May 23-26, 2003.

be used to generate test cases which maximise the execution time of the system under consideration (Wegener et al., 1997).

Evolutionary WCET analysis can be applied to replace or supplement static analysis techniques (Nilsen and Rygg, 1995; O'Sullivan et al., 1998) which are traditionally used for this task, but suffer from various deficiencies. For example, the software must comply with several criteria to permit the application of static analysis (Puschner and Koza, 1989), or path information and code annotations must be introduced into the technique (Puschner and Vrchoticky, 1997). This demands extensive user interaction and makes static analysis difficult, expensive and error-prone (Puschner and Nossal, 1998).

The aim of this paper is to assess and verify the performance of evolutionary testing to find or generate worst-case execution times for a number of test programs. Since the true worst-case times are not known for the example programs, evolutionary WCET analysis is compared with random testing which is the single most important validation technique for this purpose in industry (Storey, 1996), and with the performance of a human tester who knows the used test modules well and tried to find the longest execution through manual analysis and execution of selected input samples.

The following section (section 2) introduces the terminology of evolutionary testing for WCET analysis. Section 3 describes the set up of the experiment including the used methodology, the test programs, and it displays the results of WCET analysis on the test objects. The paper concludes with a discussion of the findings in section 4.

2 Evolutionary testing

Dynamic worst-case execution time analysis in real-time application development 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 (Wegener et al., 1997), and the parameters to be optimised are the input situations with which the system is confronted during operation. For example random testing represents an optimisation-based technique, although a very simple one. Evolutionary testing (ET) (O'Sullivan et al., 1998) implements a much more sophisticated optimisation technique than random testing. It is based on the application of genetic algorithms to the problem under consideration.

A genetic algorithm performs on a population of binary strings, so-called chromosomes. These represent possible solutions to the considered optimisation problem. Some of these chromosomes are selected and then recombined to form new strings. These are mutated and the resulting strings form a new generation of prospective 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 pseudocode, 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; for example, if a predetermined number of generations is reached, or if a predetermined number of generations does no improve the cost function. GA generate new solutions based on information of successful 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 (Goldberg, 1989; Holland, 1975).

Each individual step of an evolutionary algorithm is explained in the following:

- The **initialisation** of the chromosomes determines random locations in the search volume for the population to start the optimisation process (Goldberg, 1989). This corresponds to a single execution of a random test.

- **Selection** determines the individuals to be recombined. *Tournament selection* determines the 'fittest' individual of a number of randomly selected individuals (tournament) from the population. The *tournament size* deter-

mines how 'elitist' the selection operation performs (Miller and Goldberg, 1996).

- **Recombination** exchanges on average $L/2$ binary items between two selected individuals. L is the number of binary locations on the chromosome. This applies to *uniform crossover* as recombination operator with crossover probability $p_c = 0.5$. Two individuals can be selected and cloned. Each binary location in the chromosomes of the two clones is exchanged according to this probability (Spears and De Jong, 1991). The outcome of this process are two new individuals.

- **Mutation** introduces a small change to each new individual. This is applied with a very low mutation probability, for example $p_m = 0.001$. Each binary location in the chromosome is flipped according to this probability (Goldberg, 1989).

- The **fitness** of an individual is determined by the cost function. In this instance it measures the execution time of the test object for a particular input situation. Program input and chromosome are exactly the same. The EA attempts to maximise the execution time. Section 3 shows a code excerpt of the used 'fitness function'.

3 Experiments with evolutionary testing

The EU funded Experimental Software Engineering Network (ESERNET) provides guidelines on how to perform software engineering experiments. The final report template (Esernet-Consortium, 2002) identifies a number of criteria that should be outlined for an experiment. The work described in this paper is only very roughly abiding by these principles, that is, for instance, the context of the study, its description, and main results as well as their interpretation. The context of this study is a university environment. Its description is laid out in the following paragraphs including a brief characterization of the used test objects in Table 1. The outcomes of the study are summarized in Table 2, and the evaluation concludes this section.

The used test objects are 15 modules taken from different applications. Their names, functions and input sizes are displayed in table 1. The input size of a module is important for evolutionary testing, since it determines the size of the search volume.

All experiments were performed on a PIII-500 under the *Linux* operating system with the *RTLinux Version Two* hard real-time extension. The real-time extension is available at http://www.rtlinux.org, under a public license. This system is used because it provides hard real-time scheduling, a high-resolution timer with microsecond resolution, and additionally, it is freely available. Timing is reasonably accurate under this system. Repeated timing of the same test case almost always leads to the same execution time in microseconds ($+/-2\mu$s). The following C source code segment was used as test harness (cost function). The routine can communicate with a test case generator via *fifo message queues*. It executes the actual test object with input provided by a test case generator, and measures its execution time. The real-time commands used in the source code example are described in the *RTLinux* manual.

```
pthread_t thread;
/* ---------- TEST HARNESS */
void * test_harness (void *arg) {
  long diff, start, stop;
  struct sched_param p;
  p . sched_priority = 1;
  pthread_setschedparam
    (pthread_self(), SCHED_FIFO, &p);
  while (1) {
    pthread_wait_np ();
    rtf_get(1, (char*)
        &input_msg, INPUT_SIZE);
    /* get start time from
       the high-resolution timer */
    start = (int) gethrtime();
    test_object (input_msg,sz);
    /* get stop time from the
       high-resolution timer */
    stop = (int) gethrtime();
    diff = stop - start;
    diff = diff / 1000;
    rtf_put(2, (char*)
        &diff, sizeof(diff));
  }
  return 0;
}

/* ---------- Handler for fifo access */
int my_handler (unsigned int fifo) {
  /* found something in the fifo,
     so wake up thread */
  pthread_wakeup_np (thread);
  return 0;
}
```

Two automatic test case generators *EvolTest* and

Module	Description	Input (bytes)
delevat	Elevate the degree of a Bezier line interpolation.	28
dzz1	Contour plotting package, noise filter, part 1	1080
dzz2	Contour plotting package, noise filter, part 2	1080
dzz3	Contour plotting package, noise filter, part 3	1080
dzch	Contour plotting package, noise filter.	1080
bsa	Bubble sort.	1024
bsb	Different bubble sort.	1024
ins	Insertion sort.	1024
ses	Selection sort.	1024
diff	Robot vision system, difference of two frames.	1024
sobel	Robot vision system, edge detector.	1024
min	Robot vision system, filter.	1024
median	Robot vision system, noise filter.	1024
gstretch	Robot vision system, grey scale filter.	1024
train	Train control system module.	656

Table 1: Description of the test objects including input size.

	EvolTest μs			RandTest μs			manual test μs
Module	min	avg	max	min	avg	max	max
delevat	3	5	**7**	3	3	4	3
dzz1	12	12	12	12	12	12	**18**
dzz2	25	27	**29**	21	21	21	24
dzz3	34	35	**36**	29	32	34	33
dzch	51	53	**54**	46	48	50	51
bsa	755	778	**806**	611	615	623	618
bsb	1262	1276	**1287**	1091	1096	1104	1079
ins	239	244	247	161	164	167	**271**
ses	296	305	**317**	291	293	296	295
diff	12	18	**21**	12	13	17	14
sobel	109	111	**112**	99	101	104	101
min	379	385	**388**	353	354	355	354
median	386	388	390	376	378	382	**408**
gstretch	81	82	**88**	80	81	85	85
train	4	6	8	2	4	6	**13**

Table 2: Experimental outcome. Found worst-case execution time (WCET) in microseconds for evolutionary testing (EvolTest), random testing (RandTest) and manual testing. Displayed are minimal found WCET, maximal found WCET and average over 10 repetitive trials (for the automatic techniques).

RandTest were used for assessing the timing behaviour of the test objects. *EvolTest* is a simple evolutionary testing toolbox which was developed by the author (any other GA-toolbox that is available through the Web will be sufficient for the purpose). The following pseudo-code represents the used algorithm:

```
begin GA
  create population of size
      parents + children;
  foreach (parent) do begin
    initialise individual's
       chromosome randomly;
    start = read timer;
    call test_object with chromosome;
    stop = read timer;
    fitness of the individual
        = stop - start;
  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
        start = read timer;
        call test_object with chromosome;
        stop = read timer;
        fitness of the individual
            = stop - start;
      end-for
    end-foreach
    sort population according to fitness;
    determine stopping_criterion;
  end-while
end GA
```

The following genetic operators were used for the GA:

- Binary string as chromosome (Goldberg, 1989). The chromosome is an exact mapping of the memory for each parameter set.
- The population size equals 40 individuals. The algorithm always keeps the best 40 chromosomes.
- The number of generations is set to 1,000. 40,000 testcases are generated in total.
- Tournament selection, tournament size equals 4 individuals (Miller and Goldberg, 1996).
- Discrete recombination, uniform crossover, $p_c = 0.5$ (Spears and De Jong, 1991).
- Low constant mutation rate, $p_m = 0.001$ (Goldberg, 1989).
- Rank based fitness (Whitley, 1989).
- Random initialisation of the chromosomes (Goldberg, 1989).

RandTest is a very simple uniform generator for random test cases. The number of randomly generated test cases was limited to 40,000 in order to allow a fair comparison between the two automatic test case generators. This is simply the number of individuals in the used GA (40) multiplied by the number of generations (1,000).

The number of tests generated by the human tester was in the order of magnitude of ten to twenty. The test generation was solely based on the experience of the tester with respect to the source codes. Overall, it was an extremely fuzzy trial and error strategy that aimed at finding longer execution times than that of random or evoltuionary testing. Manual test case generation was perceived as being extremely tedious.

The automatic testing typically finished within minutes after launch, for the used hardware. The 40,000 random tests usually took only about 10% of the time which was needed to run the 1,000 generations on the genetic algorithm. This is due to the much longer and more complex internal algorithm which is applied by the evolutionary technique with recombination and evaluation compared with the very straight-forward random generation of test data. The manual tests took several hours or even days depending on the test object under consideration. Table 2 displays the obtained measurements (timing in microseconds) for each testing strategy. For the two automatic techniques *EvolTest* and *RandTest*, the average over ten repetitive trials is displayed as well as the minimal and maximal worst-case execution times which were found during these experiments (table 2). For the manual tests, only the absolute maximal found timing is displayed.

The results clearly illustrate that evolutionary testing always outperforms random testing. For all cases, evolutionary testing is at least as good in finding long execution times as the random technique.

This seems most obvious, since a GA is inherently founded upon random techniques. Though, a GA is much more than a random methodology, through the mechanism with which new 'good locations' in the search volume are generated by 'remembering' and recombining existing 'good locations'. On average, random testing could only produce about 85-% of the maximum execution times found by evolutionary testing. This is the case for *avg* and *max* in table 2.

The outcome of the manual testing is quite diverse. Manual testing is used as comparison as previous work has identified specific pathological conditions of the programs which are detrimental for search techniques (Gross, 2000). For four out of the 15 test programs, the human tester was most successful. These are the test objects *dzz1, ins, median* and *train*. For these cases, ET could only achieve about 76 %, random testing only about 64 % of the performance of the human tester. An analysis of the source codes reveals common properties which are exhibited by these four test objects. They all have in common that the longest execution paths through these programs can be determined from the source code. For *dzz1, median* and *train*, the longest sections can only be reached trough distinct pattern in the input. As there are only very few combinations leading into the longest branches of these programs, input situations according to these branches are difficult to find by the automatic techniques. This causes the obvious shorter paths to be taken for most input situations, because the search process does not provide any information through the fitness function that would lead to the evaluation of these very few 'interesting' locations of the search space. Only an analysis of the source code can determine a suitable input situation which causes the program flow to follow the longest path, although this analysis becomes increasingly difficult, if not impossible, for large and complex test objects.

In terms of algorithmic properties, the sorting procedures exhibit their longest execution time if they are faced with reversely sorted lists as input (every number in the list must be swapped a maximum number of times). Although, this is not the case if real execution time is considered. Except for the module *ins*, the time for the swapping in this module is relatively short compared with the time to execute the remaining sections of the programs. For the sorting algorithms, it is not the swapping which causes most of the run-time but everything else apart from that; hence, the lower timing for a reversely sorted list (created by the human tester) on the sorting algorithms compared with the outcome of the automatic techniques for the sorting algorithm. The human tester could only find longer execution times by applying a reversely sorted list for the *ins* module because the way this sorting is implemented creates extremely small input value domains which are difficult to find through automatic testing.

4 Conclusions and further research

Subject of this paper is the application of evolutionary testing as a new technique for verifying the worst-case timing behaviour of real-time software. Its main focus is on the comparison of evolutionary testing with random test case generation. Random testing is the single most important validation technique for timing properties in the industry to date. The paper illustrates through experiments that evolutionary testing is much more powerful at finding long execution times than random testing, and this may serve as a strong advocator for a paradigm shift in industry towards using evolutionary testing as a replacement technique for random testing. The application of evolutionary testing is exactly the same as for random testing, so its adoption cost for software companies is minimal. Random testing can only produce about 85 % of the execution times found by evolutionary testing. Since computational power and hardware speed are growing, the approximately ten times slower processing of evolutionary testing compared to random testing is of minor importance, specially since the overall automatic testing process only takes a few minutes.

For some pathological programs, the human tester performs better in finding appropriate test cases which maximise execution time. This is caused through very few key combinations which must be found in order to direct the program flow into the longest branches. Previous work thoroughly discusses this (Gross, 2000). The ideal testing strategy is clearly a combination of evolutionary testing that is supported by human knowledge of the test object. In that respect the tester may augment the initial

population with information of a test object's pathology, and then run the evolutionary testing process with that information to support the search. This can be simply achieved through a seeding of test cases that are predetermined by the human tester, into the randomly generated initial population. The GA can then use this information to drive the search process towards areas of the search volume that are otherwise too difficult to reach because the fitness function does not provide information to generate such unlikely input combinations.

Since evolutionary testing is performing so well under the traditional procedural development paradigm, current work in this area now focuses on the application of evolutionary testing in object-oriented and component-based real-time applications.

Acknowledgements

This work has been partially supported through the European Union funded Experimental Software Engineering Network (ESERNET), http://www.esernet.org, and through the German Federal Department of Education and Research under the MDTS Project acronym http://www.fokus.fhg.de/mdts.

References

Esernet-Consortium. 2002. Esernet experiments template: Final report template guide. Technical report, Experimental Software Engineering Network, http://www.esernet.org.

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

Hans-Gerhard Gross. 2000. *Measuring Evolutionary Testability of Real-Time Software*. Ph.D. thesis, School of Computing, University of Glamorgan, Pontypridd, Wales, UK, July.

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

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

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

M. O'Sullivan, S. Vössner, and J. Wegener. 1998. 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.

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

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

P. Puschner and A. Vrchoticky. 1997. 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.

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

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

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

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