IEE Workshop on applicable modelling, verification and analysis techniques. London, 11. January 1999.

Evolutionary Algorithms for the verification of execution time bounds for Real-Time Software

Hans-Gerhard Groß
Bryan Jones
David Eyres

School of Computing, University of Glamorgan, Pontypridd CF37 1DL, Wales, UK.

Abstract

Real-Time Systems must produce results according to a predefined time schedule. Therefore, their operation speed is critical. Timing Analysis is crucial for the verification of a program's timing behaviour and an important part of design, testing and assessment. It measures how well a system matches its specifications. Static Timing Analysis concentrates on the evaluation of a program's internal structure to predict execution time bounds. But it does not consider a program's input parameters which are mainly responsible for the dynamic behaviour. Dynamic Timing Analysis can produce the most accurate assessment of run-time behaviour through the analysis of the interactions of a module's input parameters, but it is considered to be impractical because of the possible combinatorial explosion of the parameter combination space.

However, powerful searching strategies such as Evolutionary Algorithms now make Dynamic Timing Analysis of systems feasible. They only need to search through a fraction of the input parameters' total combination space and can be used to replace or supplement static techniques. Experiments with the new methodology have already shown significant improvement in the prediction of timing constraints, although the technique is merely in an experimental state. It now requires a fast, accurate and reliable environment for the evaluation of its usability. This is particularly important for large systems where it would be most useful.

The aim of this paper is to create a basis for further investigations into the value of evolutionary techniques for system testing. It briefly introduces Evolutionary Algorithms and describes the idea for their use for the prediction of timing constraints. It concentrates on the development of an operational timing testing environment and discusses possible limitations and further research.

Introduction

Real-Time Systems must produce a correct result according to a well defined time schedule. Determining accurate timing constraints for software is difficult. Static Timing Analysis investigates the internal structure of programs and attempts to draw conclusions about their dynamic behaviour [1, 2, 3, 8]. However, dynamic behaviour is mainly dependent on a program's input parameters and cannot be sufficiently defined by its structure.

Ideally, to determine a program's worst case execution time (WCET) it should be executed for each possible input parameter combination. This is usually impractical due to a possible combination explosion in the program's parameter space. 'Intelligent' searching methodologies such as Genetic Algorithms and Evolution Strategies now make the search in huge combination spaces feasible. These techniques can be applied to search for a program's input domain causing its longest execution time.

This work briefly introduces the use of Evolutionary Algorithms (EA) in execution time assessment. The main emphasis is on the design of an operational experimental environment which makes an evaluation of EA techniques feasible. The integration of a commercially available timing package into the EA process is outlined and additional advantages of the technique are demonstrated. The paper concludes with an outlook on further research activities.

The use of Evolutionary Algorithms for the testing and assessment of systems is referred to as *Evolutionary Testing (ET)* [6].

Evolutionary Algorithms for testing

Evolutionary Algorithms are powerful search or optimisation strategies, loosely based upon the evolution of natural systems through reproduction and selection. They operate on populations consisting of strings of binary values for Genetic Algorithms [4] or real numbers for Evolution Strategies [10]. These strings are called *chromosomes*. Each of the chromosomes corresponds to an individual which represents a possible solution to the search problem. EA 'recombine' pairs of these individuals in order to produce offspring. This is called *crossover*. The resulting new individuals are 'mutated' and from those and the parents a new generation is selected according to the fitness function. This function defines how well each individual represents a possible solution to the search problem. Consequently, an Evolutionary Algorithm performs in a cycle of four basic steps: Reproduction, Mutation, Evaluation and Selection.

For software testing, the chromosome can be defined as the test program's input parameters, each individual representing the input parameters for one single execution. These parameters are recombined, mutated evaluated and selected according to the previously described EA procedure. Here, the fitness function determines the individual's ability to produce long execution times for the program under test.

The best (most fit) individuals are those that produce long execution times and they are therefore selected for the next generation. This technique is applied many times, each time tending to lead to more of the population having higher fitness and thus representing better solutions.

The search for input parameters which produce the WCET on the test program only involves the evaluation of a fraction of all possible locations in the search space.

Defining the fitness function

The fitness function is a key feature in an Evolutionary Testing process. It measures the execution time of a single program run for the input parameters given by the Evolutionary Al-

gorithm. The fitness function should be deterministic, so that it returns exactly the same execution time for the same set of input parameters.

The system's real-time clock is inaccurate as it includes many operating system activities such as context-switching and paging which occur in a multi-programming system. Consequently, timing cannot be performed by the timer. This method results in different test program execution times for the same input domain depending on the system load.

The single most accurate methodology to provide deterministic timing is to count the actual machine cycles during the test process. Rational's *Visual Quantify* may perform this task. It is the only (commercially available) product for this and it uses a technique referred to as *Object Code Insertion* extending the executable program code by inserting hardware dependent counting instructions. This is called *instrumentation.*

Visual Quantify was designed as a system to identify software performance bottle-necks [11], and it is unfortunately not optimal for Evolutionary Testing. The emphasis of the product is on an easy-to-use user interface which is able to display very detailed information about a program's execution. However, ET is only interested in a single figure – the number of executed machine cycles. The massively repeated use by Evolutionary Algorithms rather requires a sophisticated application interface (API).

Figure 1 depicts the integration of a fitness function based on the *Visual Quantify* package into an EA. Here, the EA fitness function writes the input parameters into a communication interface (file, pipe, socket). Then it executes the instrumented test program. This must be done for each individual separately. The test process reads the input parameters from the communication interface and proceeds with the execution of the actual test module. During execution the machine cycles are counted. The positions where to start and to stop the count can be controlled by API functions which may be placed into the test program's source code (see Figure 1).

The fitness is written to a data file for the Evolutionary Algorithm (`QuantifySaveData()` in Figure 1) and the test process exits. The fitness function in the EA process reads the result file and extracts the number of machine cycles [1].

This procedure must be performed for each individual in the EA population.

Positive side-effects

The large numbern of generations and executions of test cases by the EA strategy provides some additional benefits. During the test phase, thousands of test cases are produced and their influence on the timing of the program is assessed. These cases may be stored and used by a static analysis technique for *symbolic execution.* This tool can then visualise the visited branches in the program's source code. In this way, the tester can determine if the program has been thoroughly tested and decide if further evaluation is required.

Introducing the use of an *alarm clock* and a *signal handler* in the testing process can determine if the test program came to an end (halting problem). Here, the time for the EA procedure to wait for the end of the *Visual Quantify* process can be set and the EA process woke up after the waiting time has expired. The calling process can then resume operation and save the input parameters which have caused the problem for later analysis. The alarm should be set to a sufficient value for the testing environment.

[1] The text and Figure 1 refer to the Windows NT version of *Visual Quantify.* The performance of the Unix version is better as a whole population may be evaluated within a single *Visual Quantify* process.

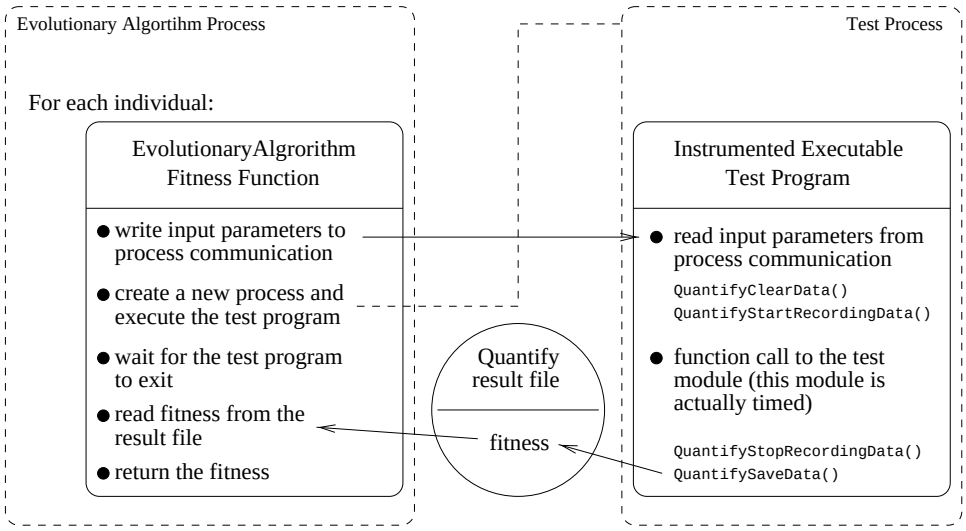

Figure 1: EA fitness function with *Visual Quantify*

A similar technique can be used for the discovery of run time errors. The calling process is informed by the operating system (through the `wait()` system call) how the called process actually finished. Again, the input data causing the process to crash can be saved for later analysis.

Conclusion and further research

Evolutionary Testing is a new technique for the assessment of WCET behavior. As it is a typical search strategy, it can never guarantee to find the actual maximum execution time of a program. This is a very important statement for the application of Evolutionary Testing which should always be kept in mind. However, experiments with the new technique have already confirmed the high value for Real-Time System testing [5, 6, 7].

In combination with Static Analysis it can be applied as guide for software testers/designers to analyse how good their estimation of a program's execution time bound is. Here, the technique is able to yield high-quality results and shows a potential to detect errors in Static Analysis [9].

For safety critical systems it certainly always depends on the implementation and the decision of the design/testing team to apply such a methodology. For many test cases the technique is indeed able to find the input domain causing the WCET and here it often can expose Static Analysis as being unsufficient [6, 7].

At the current stage, a final evaluation of all aspects of Evolutionary Testing is not yet possible. It must still be considered experimental.

The research in this field now concentrates mainly on the development of more sophisticated Evolutionary Algorithms. One of the key issues here is to establish a technique which is able to explore and analyse 'complicated' test programs. Other investigations are concerned with the decision when to stop the search process [7]. These approaches are focussed on the "EA side" of the problem.

A different approach is to analyse the software under test. Here, a main issue is the question of which attributes of a test program make it difficult for the EA to find the WCET,

or what makes a test program 'complicated' for Evolutionary Testing. Current research by the authors tries to establish a metric which is able to indicate if an evolutionary approach is likely to be successful on a given test program. The idea here is to define guidelines for the actual design/development of Real-Time Software which make the application of the EA methodology for testing more likely to succeed.

One of the main aspects here is the question of how well the EA adapts to increased system complexity. The outcome of this research could finally lead to the application of Evolutionary Testing to increasingly complex systems.

References

[1] Chapman R. Burns A. Wellings A. Integrated program proof and worst-case timing analysis of spark ada. In *Proceedings of the ACM Workshop on Language, Compiler and Tool Support for Real-Time Systems*, pages K1–K11, June 1994.

[2] Chapman R. Burns A. Wellings A. Combining static worst-case timing analysis and program proof. *Real-Time Systems*, 11:145–171, 1996.

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

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

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

[6] Müller F. Wegener J. 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.

[7] O'Sullivan M. Vösser S. Wegener J. 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.

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

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

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

[11] Rational Software. *Quantify User's Guide - Version 3.1.* Rational Software, 1997.