Improving Evolutionary Real-Time Testing

Marouane Tlili
Daimler Chrysler AG,
Reasearch and Technology
Alt-Moabit 96a
D-10559 Berlin, Germany
marouane.tlili@daimler-chrysler.com

Stefan Wappler
Technical University of Berlin,
DaimlerChrysler Automotive IT Institute
Ernst-Reuter-Platz 7
D-10587 Berlin, Germany
stefan.wappler@tu-berlin.de

Harmen Sthamer
Daimler Chrysler AG,
Reasearch and Technology
Alt-Moabit 96a
D-10559 Berlin, Germany
harmen.sthamer@daimler-chrysler.com

ABSTRACT

Embedded systems are often used in a safety-critical context, e.g. in airborne or vehicle systems. Typically, timing constraints must be satisfied so that real-time embedded systems work properly and safely. Execution time testing involves finding the best and worst case execution times to determine if timing constraints are respected. Evolutionary real-time testing (ERTT) is used to dynamically search for the extreme execution times. It can be shown that ERTT outperforms the traditional methods based on static analysis. However, during the evolutionary search, some parts of the source code are never accessed. Moreover, it turns out that ERTT delivers different extreme execution times in a high number of generations for the same test object, the results are neither reliable nor efficient. We propose a new approach to ERTT which makes use of seeding the evolutionary algorithm with test data achieving a high structural coverage. Using such test data ensures a comprehensive exploration of the search space and leads to rise the confidence in the results. We present also another improvement method based on restricting the range of the input variables in the initial population in order to reduce the search space. Experiments with these approaches demonstrate an increase of reliability in terms of constant extreme execution times and a gain in efficiency in terms of number of generations needed.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*

General Terms

Verification

Keywords

Software tools, Software engineering

1. INTRODUCTION

Classical methods for determining extreme execution times are based on static analysis and manual estimations [8]. Static analysis methods are limited when the test object contains program control structures such as loops. In this case, loop bounds must be specified manually or can only be estimated. With manual estimations, the program code must be analyzed by hand and the possible execution paths which lead to extreme execution times must be figured out. This process is resource-consuming and error-prone. Furthermore, the shortest execution time can be too optimistic and the longest execution time too pessimistic due to manual estimations.

ERTT is an approach to testing the timing constraints of real-time systems. The test objective is transformed into an optimization problem. An evolutionary algorithm is used to try to solve this problem. The search space is formed by the input domain of the test object (parameter data and global variables). The objective function is based on the execution time needed to execute the test object with the respective input data. This time is either maximized for the search for the worst case execution time (WCET) or minimized for the search for the best case execution time (BCET).

Previous work [2] showed that ERTT outperformed manual testing and random testing of timing constraints. However, in experiments, it can be shown that the results of ERTT are not always reliable. Different test runs produced different extreme execution times. Moreover, the number of generations needed to find the extreme execution times is usually high, even for simple test objects. This means that this method is not efficient. Furthermore, not every part of the test object was executed during the tests. Incomplete coverage decreases the confidence in the test results because the execution of the uncovered parts might lead to improve the extreme execution times.

In order to increase the reliability and the confidence in the results achieved by ERTT, we propose two methods to find longer and shorter execution times in less number of generations and in a robust way. The first method is based on seeding the evolutionary algorithm with structural test data which help the evolutionary search to explore new regions in the search space. Previous works [11, 2] suggest seeding the initial population of the evolutionary algorithm with input

data which contain information from the tester or information about the pathology of the program. Our idea is to seed the initial population with test data that lead to a high structural coverage of the test object.
The second method is based on the idea that most of the input and global variables are not used in their full data type range. When restricting the range of these variables in the initial population, the optimization process is guided to regions in the search space where the range of the variables is actually used. In doing this, more code is executed and therefore the confidence in the results delivered by ERTT is reinforced. In our experiments, we compare these two approaches with the standard ERTT.
This paper is structured in sections as follows: section 2 describes the ERTT. Section 3 shows the limitations of this approach. The new designs of the standard ERTT used to improve its reliability are explained in section 4 and 5. Section 6 presents the experiments and their settings, this includes the testing tools, the environment for testing the real-time systems, the evolutionary algorithm settings, the test objects and the analysis of the results. The last section concludes the paper and gives insights on future work and new ideas.

2. EVOLUTIONARY REAL-TIME TESTING

Time testing aims at finding input data for a given test object which result in extreme execution times. For safety-critical systems such as the airbag system of a modern vehicle, the timing constraints play an essential role. It is not acceptable that the system reacts after a specified time limit. During time testing, it must be ensured that the maximum and minimum time limits are satisfied by the function under test.
The traditional test methods lack the property of automating test case generation. For this reason, evolutionary testing (ET) is used. It can be applied to testing non-functional properties, like safety testing, robustness testing and temporal behavior testing. ET can also be employed for the automation of existing test methods like functional testing, structural testing or mutation testing [3].
With ERTT, the extreme execution times are searched for using an evolutionary algorithm. This input domain comprises the parameter data and the global variables which are referenced by the test object. Each individual represents a particular value set for the input space of the test object. For the fitness evaluation, the individuals are converted into test data with which the test object is executed. The execution time is measured and returned to the evolutionary algorithm as an objective value.
The evolutionary algorithm uses these values to select promising individuals for recombination and mutation in order to create better ones. Typically, the initial population of the evolutionary algorithm is generated by random. The succeeding iterations of the algorithm try to improve this initial data. The search usually is terminated if no improvement of the extreme execution time can be achieved over a number of predefined generations or if a configured number of generations has been reached (see section 6.3 for details).

3. LACK OF RELIABILITY

Although ERTT is more successful in finding the extreme execution times than the traditional methods based on static analysis, this method presents limitations. In order to practically demonstrate them, experiments on the standard ERTT were conducted (see section 6.3.1) on different test objects (see table 1). The following observations have been made:

- In order to have an idea on what are the parts of the program covered during the ERTT, the structural branch coverage of the individuals generated during the search for the WCET has been measured. The average branch coverage achieved is 75%. This shows that the results given by ERTT might not lead to the WCET. Indeed, the execution of the corresponding branches of the code which were not visited might help to produce a longer execution time. This observation shows that the confidence in ERTT can be further increased if a larger part of the source code is investigated during the evolutionary search.
- The delivered execution times are not always the same for the same test object. This is due to the different conditions in the code of the test object which must be satisfied in order to reach an extreme execution time. For instance, if the true branch of a complex condition leads to an execution path which needs a large amount of time during the execution, ERTT is successful only when this complex condition is satisfied. However, the objective function which is solely based on the execution time does not direct the evolutionary search to satisfy this condition. Consequently, the extreme execution time can only be found if the condition is satisfied by random.
  In figure 1, there is a common situation where the evolutionary search is not able to access some control statements. This happens because the search space is too large and there is no strategy favoring the execution of a certain branch in the control-flow graph.

```
void fun1(int a, int b){
  if(a == 0 && b == 0){
    do_something; //cost = 5
  }
  else{
    do_something; //cost = 1
  }
}
```

Figure 1: The probability to execute the if statement is very low since the search space is too large.

- The number of generations needed for the evolutionary algorithm is high even for simple test objects. Moreover, this number is not always the same in all the test runs. This means that the ERTT suffers from a lack of efficiency.

In order to find better extreme execution times in a constant number of generations, the evolutionary search has to access new regions in the search space which were not executed by the standard ERTT. We propose two modified designs of the standard ERTT. Both designs come from the idea that if the individuals in the initial population can access new regions in the search space, then this will help to find better values for the longest and the shortest execution times in the next generations. The first design is based on seeding the initial population of the evolutionary algorithm with test data which might cover regions in the code which were not accessible by the standard ERTT. The second new design does not use seeding but consists in restricting the range of the input and global variables in situations like the one described in figure 1. If the range of the integer input variables is restricted to the values 0 and 1, then the probability of executing the if statement is much bigger than without restricting the range of the variables.

4. EVOLUTIONARY REAL-TIME TESTING USING SEEDING

The idea behind this work is to provide the evolutionary algorithm with a high quality initial population to evolve from and therefore orienting the evolutionary search to new regions in the search space which might cover new parts in the source code of the test object. The first strategy is to seed the initial population of the evolutionary algorithm. As the seeding should be efficient, the new evolutionary algorithm settings should be different from the ERTT with standard settings. For this standard configuration, the initial population is made of randomly generated individuals in the search space.

In our work, the selected seeding strategy is to seed the initial population with input data achieved a high structural code coverage. As described in section 3, some parts of the source code in the test objects are never covered, the initial population is seeded with test data which might access these uncovered regions. In order to make sure that the seeded test data really cover these new regions, a maximum structural code coverage is performed prior to seeding. This ensures that the seeded individuals could execute all the reachable code.

Different coverage criteria exist for structural testing, for instance path coverage. A 100% path coverage is achieved when every possible path in the program is executed. Unfortunately, path coverage is too resource-consuming because of the significant computational time needed to test the infeasible paths. Moreover, the efficient automation of such a method is difficult to achieve [5].

Branch coverage is another structural coverage criterion, it consists in ensuring that each branch is traversed at least once in the control-flow graph. Every statement and every condition in the branches are executed. The automation of such coverage method is relatively easy to do [1]. Branch coverage testing was used in this work to generate the test data for seeding, the testing tool which was used to do such testing is the ETS-tool [10].

The branch coverage generates test cases. A test case contains values for the input and global variables of the function under test. These values are seeded in the initial population of all the test objects. If the branch coverage generates a smaller number of test cases than the number of individual contained in the initial population, the remaining individuals are generated at random as it is the case for the standard ERTT. The opposite situation (number of test cases generated bigger than the number of individuals) has never occurred in our experiments and was therefore not handled.

5. EVOLUTIONARY REAL-TIME TESTING USING RANGE RESTRICTION

The test objects used in the experiments present in their sources situations where the input and global variables often don't use the full range of their data type. Programmers do not take into account this important detail when writing a software. This results in a waste of computational efforts during the optimization process and it might lead to prevent the search from hitting the optimum since no search direction is preferred and the fitness function is only based on the execution times. The example given in figure 1 shows that if the range of the input variables is restricted to 0 and 1, then the probability of executing the if statement is higher than the case without range restriction because the search space is smaller with the range restriction technique. The test object N3 (see table 1) has 50 input and global variables of different data types, the range restriction technique was applied to 23 of them. The original search space bit size is 93 bits and the reduced search space bit size is 86 bits. The ratio of the second number over the first one gives a very small number which means that this method permits to save up to 99% of the original search space.

The range restriction was manually applied in our work for the test objects whenever it was possible. This was done in a very careful way. It was only performed when it was clear that a variable uses only a part of its data type. The experiences with ERTT were done with the testing tool TESSY (see next section). TESSY allows to restrict the range of the input and global variables with specifying new bounds. During the evolutionary search, the values of the variables can only be between these bound.

6. EXPERIMENTS

6.1 Test Environment

6.1.1 *Testing Tool TESSY*

TESSY [12] is a software testing tool commercially available through Hitex. Its main characteristic is to provide a support for different test activities for C-based programs. These activities include test execution, test monitoring, test evaluation as well as a systematic design of test cases, particularly for the functional test. The most important strength of TESSY is that it provides support for the whole testing life cycle. TESSY is a tool that can be used without a profound knowledge of the C-programming language and that permits the separation between the test quality process and the software development process. TESSY facilitates the combination of black-box and white-box tests. For black-box testing, test cases are determined using the classification-tree editor CTE, a graphical editor for the descriptive and systematic design of black-box test cases using the classification-tree method [9].

In our work, TESSY was used to automate the temporal test execution of the functions under test. The automation of ERTT is first done by generating the test driver and establishing the communication between the testing host and the target host. Working in an embedded environment is different from working in a native program development environment; the target host and the testing host system are not on the same hardware as it is the case for desktop applications. In order to run an application in this specific environment, a simulator or a target hardware is required. Two hardware/software configurations have been used in this project. This choice was made in order to test different software functions working on different platforms and to make the experimental test environment be as close as possible to the industrial one. The following configurations have been used to simulate the real-time systems:

- The debugger-simulation environment fast-view66-win from Hitex. It is a windows-based high level language debugger for the C166-ST10 microcontroller families. The C compiler used is the Tasking Compiler 16 bits. It is a special embedded compiler used for a big range of microcontrollers and microprocessors.
- A 32 bit microcontroller MPC555, a Motorola product, it contains a floating point unit designed to accelerate the advanced algorithms operations necessary to support complex applications. It is commonly used in the automotive applications such as engine and transmission control as well as robotics and avionics control. The compiler used with this microcontroller is the DIAB complier from the company Wind River, it creates executable files which will be downloaded on the MPC555 target board.

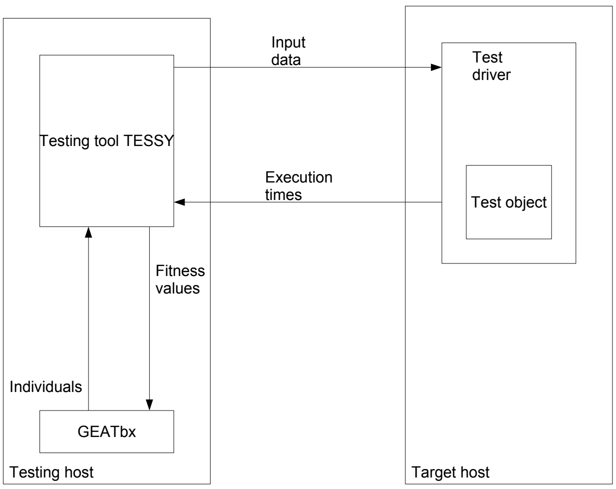


Figure 2: Host-Target configuration; communication between the testing tool TESSY and the test driver.

Figure 2 shows the communication with the target host during the optimization process. The evolutionary algorithm is implemented in an application called the peanuts server. The algorithms implementation is based on the Genetic and Evolutionary Algorithm Toolbox for Matlab (GEATbx [6]). The different phases of the evolutionary algorithm are executed on the testing host. When the evolutionary algorithm generates the relevant test data whose fitness values (execution times) have to be computed, TESSY sends these test data to the target board or the simulator. The longest or the shortest execution times are measured on the target host and sent back to the testing host computer in order to perform the next steps (selection, recombination, mutation and reinsertion) of the evolutionary algorithm.
For the fast-view66/tasking compiler configuration, the host and the target are on the same computer, whereas the microcontroller MPC555 is on an evaluation board (but installed on the host computer as a peripheral). The host computer used has a Pentium 4 2,66GHz processor, 760MB of RAM and runs Microsoft Windows XP Professional.

6.1.2 Testing Tool ETS-tool

The testing tool used to generate the seeded test data is the evolutionary testing system (ETS) tool. Its function is to generate test data based on coverage criteria such as statement, branch and condition coverage. It uses an evolutionary algorithm to perform the structural coverage. In prior works [10], it has been successfully applied to generate test data achieving a high structural coverage and it has proven to outperform the structural random testing.

6.2 Test Objects

Table 1 contains software metrics describing the complexity of the test objects. The *CYC* is the cyclomatic complexity. It corresponds to the number of decisions plus one, a high value corresponds to a complex control flow. The maximum nesting level is the measure *NL*. *KC* is the *Knot Count* which counts the number of break, continue, goto and return statements. The number of logical operators is described by the metric *Myer's Interval (MI)*. The *ELOC* is the number of executable lines of code.
These test objects were chosen because they present different structural properties. Some of them have a high cyclomatic complexity value such as BK-4, A-3 and MUZ-11. These examples present also high values for *NL* and *KC*. In fact, most of the functions are relatively complex since they have a cyclomatic complexity higher than 10. A recommended maximum value of the cyclomatic complexity is 10 [4].

Name	CYC	NL	KC	MI	ELOC
N1	42	4	0	25	156
N2	7	2	0	4	27
N3	51	4	19	18	232
N4	71	7	32	101	293
N5	23	2	0	21	90
N6	20	2	4	4	179
N7	49	6	25	21	266
N8	23	4	16	21	110
N9	44	17	6	2	157
N10	29	8	0	8	86
N11	50	4	0	22	320
N12	10	2	0	10	35

Table 1: Software structural measurements on the tested functions. The test objects are ordered by cyclomatic complexity value. These test objects are used in the automotive industry.

6.3 Experiment Configurations

Each test object has been tested 10 times for the search for the WCET and 10 times for the search for the BCET.

For all the experiments, the same evolutionary algorithm settings were applied. This was done in order to be able to compare the outcome of the experiments. These settings for testing the temporal behavior use the so-called *Extended Evolutionary Algorithms* [7]. The idea is to use different subpopulations. Each subpopulation follows its own search strategy and competes with the other subpopulations. Six subpopulations have been used, each of them contain 40 individuals, which makes 240 individuals in total. The input variables are in integer format, the selection method is stochastic universal sampling, and the selection generation gap is set to 0.9, which means that a new generation is composed of 10% of individuals from the former generation (parents) and 90% from newly created individuals. Recombination is done with the help of discrete recombination. Mutation has also been applied. Each subpopulation is assigned a value for the mutation range, the value used are 0.01, 0.05, 0.001, 0.005, 0.0001 and 0.0005. This ensures that subpopulations are affected differently by mutation. Having different values is important for a local and global search strategy. Since there are parallel subpopulations, competition and migration are applied. Competition takes place every 10 generations, 10% of unsuccessful individuals in the subpopulations are being transferred to successful ones. When the size of a subpopulation reaches the number of 10, no further transfer is done. Migration is used to make sure that information is exchanged between the isolated subpopulations, which happens every 13 generations, and the best individuals (10%) migrate from every subpopulation.
The objective function is the measurement of the execution times of the test data. The termination criterion is the maximum number of generations decided by the tester. This was done by experimenting after how many generations the optimization's result was not improved (plus 30 to 50% of this value). Such a decision was made with the help of a visualization tool contained in TESSY. The time measure is the number of CPU clock ticks that every individual needed.

6.3.1 *Experiment 1 - Standard ERTT*

In this experiment, the tester provides no indications to optimization process. The initial population is created randomly according to an internal procedure implemented in the GEATbx matlab toolbox. The input variables use the full range of their data type, which means that during the optimization, the entire search space might be investigated.

6.3.2 *Experiment 2 - ERTT with seeding*

In this configuration, test data which achieved a high structural branch coverage are seeded as described in section 4. Table 2 shows the branch coverage achieved using the ETS-tool. The achieved branch coverage is maximum, the seeded test data cover all the reachable code in the test objects.

6.3.3 *Experiment 3 - ERTT with range restriction*

The range restriction technique was applied to all test objects as it is described in the section 5. For some test objects, many variables were not using the full range of their data type. Whereas for other functions, the range of only a few variables was subjected to range restriction.

Name	Branch Coverage
N1	93,42
N2	100
N3	96,7
N4	94,89
N5	97,44
N6	100
N7	100
N8	100
N9	93,02
N10	100
N11	94,79
N12	100

Table 2: Structural coverage of the test objects in %. For some of the test objects, 100% coverage was not possible because the functions contain unreachable statements in their source code.

6.4 Experimental Results

During the experiments, the longest and shortest execution times, the standard deviation of the values found in the different test runs and the number of generations needed to reach such values were measured.

6.4.1 *Longest Execution Time*

Figure 3 shows the longest execution times for the test objects listed in the table 1. For almost all test objects, the configuration using seeding and the configuration using range restriction outperform the default configuration of ERTT. The improvement ranges from 0% to 41%. Figure 4 shows that the number of generations needed to find the longest execution time has been shortened in practically all the examples in RR-ERTT and S-ERTT. Concerning the

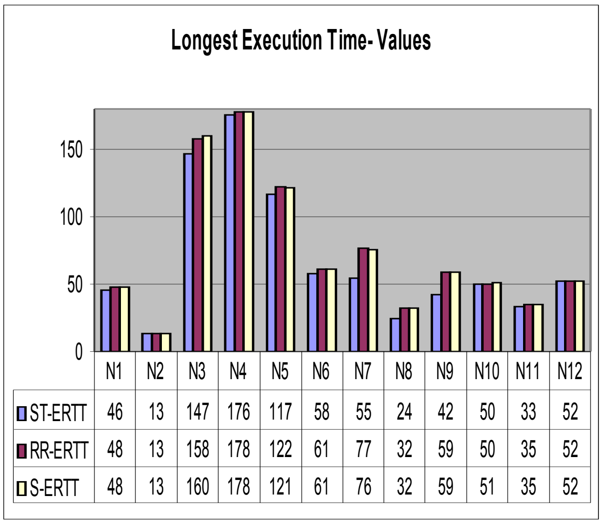
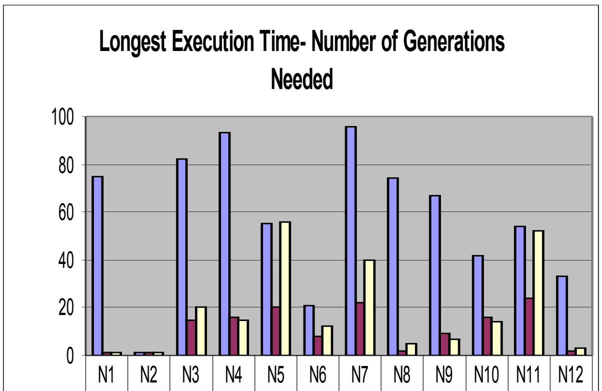


Figure 3: Longest execution time needed for all the test objects, ST-ERTT corresponds to the standard ERTT configuration, RR-ERTT corresponds to ERTT with range restriction and S-ERTT corresponds to the ERTT with seeding. For almost all the test objects, a longer execution time was found in RR-ERTT and S-ERTT.

standard deviation of the values found in the 10 test runs for the different test objects, the values found are bigger than 0 in ST-ERTT, whereas in RR-ERTT and S-ERTT, very small values are found (very close to 0). The extreme values found are more reliable in the improvement methods since the standard deviation is very low and they are more efficient since the number of generations needed has been decreased.
The results also show that in RR-ERTT and S-ERTT longer execution times were found for most of the test objects except for N2 and N12. For these two test objects, in all three experiments, the same execution times were found. One explanation can be that these two test objects have a relatively simple structure (see table 1). However, there is a difference in the results of the two methods, a shorter number of generations was needed for the test object N12 in RR-ERTT and S-ERTT, this confirms the fact that the improvement methods help to find more efficient results even if the same execution times were found for all methods.
From the experiments, longer execution times were found in S-ERTT. However, one can not be sure in advance if the longest execution time found is actually the WCET. Indeed, the longest execution time for N5 and N7 in RR-ERTT gives a bigger value than in S-ERTT. This shows that although the individuals forming the initial population in S-ERTT achieved a high structural coverage, they might not always help the evolutionary search to explore all parts of the code as these individuals might be discarded if they are not fit in terms of their objective value (the execution time).

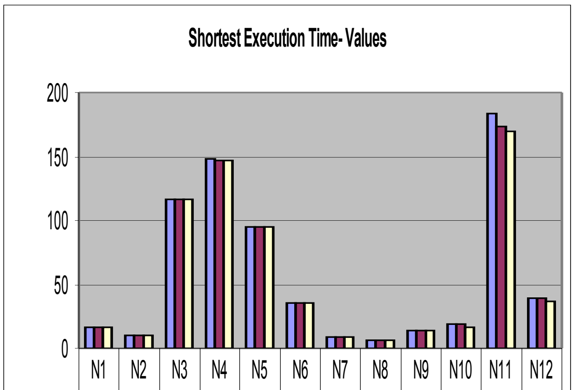


	N1	N2	N3	N4	N5	N6	N7	N8	N9	N10	N11	N12
ST-ERTT	75	1	82	93	55	21	96	74	67	42	54	33
RR-ERTT	1	1	15	16	20	8	22	2	9	16	24	2
S-ERTT	1	1	20	15	56	12	40	5	7	14	52	3

Figure 4: Number of generations needed to find the longest execution time. For almost all the test objects, less number of generations were needed in RR-ERTT and S-ERTT.

6.4.2 *Shortest Execution Time*

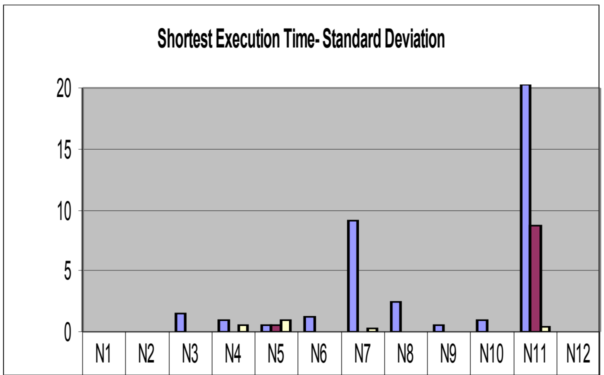
Figure 5 shows that in the three experiments, the same execution time values were found for all the test objects except for four of them. For these test objects (N4, N10, N11 and N12), shorter execution times were found in S-ERTT, this shows that the seeding method helps the evolutionary search to explore new regions in the search space, which caused to



	N1	N2	N3	N4	N5	N6	N7	N8	N9	N10	N11	N12
ST-ERTT	16	10	117	148	95	35	9	6	14	19	183	39
RR-ERTT	16	10	117	147	95	35	9	6	14	19	173	39
S-ERTT	16	10	117	147	95	35	9	6	14	17	169	37

Figure 5: Shortest execution time needed for all the test objects, a shorter execution time was found for 4 test objects out of 12 in S-ERTT.

find better optimum. The results found in RR-ERTT and S-ERTT are also more robust since the standard deviation is very close to 0 (see figure 6). The number of generations



	N1	N2	N3	N4	N5	N6	N7	N8	N9	N10	N11	N12
ST-ERTT	0	0	1.5	0.97	0.6	1.26	9.15	2.5	0.5	1	20.3	0
RR-ERTT	0	0	0	0	0.6	0	0	0	0	0	8.74	0
S-ERTT	0	0	0	0.5	0.9	0	0.3	0	0	0	0.4	0

Figure 6: Standard deviation of the shortest execution times found between the different test runs. In ST-ERTT, the standard deviation varies between 0 and 20. Whereas in S-ERTT and RR-ERTT, it is very close to 0.

(figure not shown in this paper) needed to search for the BCET can be higher in S-ERTT and RR-ERTT comparing to ST-ERTT. This happens when the evolutionary search needs more 'computational efforts' to find the shortest execution time. Figure 7 shows an example of a possible 'shortcut' situation for the test object N7. The values for (a,b,c) which lead to the shortest execution time are (1,0,0) because evaluating a, b and c takes longer than evaluating only a.

However, it was noticed that the seeded individuals in S-ERTT present values for (a,b,c) = (1,1,1). Such individuals were generated by the structural branch testing because in order to execute the branch containing the if statement, one of the variables must be evaluated to true, and if all of them are evaluated to true, the branch is also traversed, the structural testing generated (1,1,1) and the branch was executed. There was no need to generate the individual (1,0,0). During S-ERTT, the individuals with values for (a,b,c) which are not equal to (1,0,0) might be discarded since their execution time can be further reduced. This explains why the evolutionary search in S-ERTT might need a higher number of generations in order to get out of the local optima (1,1,1) and to find values for (a,b,c) = (1,0,0).

```
void fun5(char a, char b, char c){
  if(a || b || c){
   do_something; //cost = 1
  }
  else{
   do_something; //cost = 4
  }
}
```

Figure 7: Situation of a possible 'shortcut', if the input (a,b,c) = (1,0,0) is not found, the shortest execution time will not be reached.

7. CONCLUSIONS AND FUTURE WORK

The results of the conducted experiments confirmed the expectations. The improvement methods showed that they outperform the classical ERTT configuration. Longer and shorter execution times were found in more robust and efficient ways.

Concerning the search for the WCET, longer execution times were found for almost all the test objects when these test objects have complex structural properties. The number of generations needed to find the longest execution times was also noticeably reduced and the standard deviation of the values found between the different test runs was decreased. Although the results of the standard ERTT showed that the shortest execution time was found for 66% of the test objects. The seeding method and the range restriction technique helped to find shorter execution times for the rest of the functions. They also helped to reduce the standard deviation, thus giving reliable and efficient results

Nevertheless, seeding with structural data shows the case where the fitness function of the ERTT only computes the execution time and does not take into account the structural properties of the seeded individuals. This results in discarding some individuals that have interesting structural properties. A future work might include a new design of the fitness function which should account for the structural properties of the individuals. For example, the fitness function will not only consist of the execution time, but also of the ability of the seeded individuals to execute complex conditions or to access different nesting levels.

Furthermore, experiments shall be performed to know after how many generations and how often the seeding should be done in order to check if this method has an influence on the overall test performance. Given that some of the seeded individuals which have interesting structural properties are discarded during the search, it will be interesting to find out what might happen if these individuals are seeded again after a determined number of generations which will be investigated experimentally.

In our work, the branch coverage was used as a criterion to generate test data for seeding. This choice has a limitation. The branch coverage concerns only the execution of the conditions as true and false, but it does not handle the execution of all the possible values of the predicates forming the conditions. This has an impact on the execution time since it can differ depending on how many variables are evaluated in a condition. Other structural coverage criteria exist [4], among which a more suitable coverage criterion should be investigated. For instance, one could think of using the multi-condition coverage criterion, this ensures the generation of test cases which will cover all the operands of a given condition, thus helping to optimize the execution time. This idea is applicable only if the testing effort is not very high and if there exists support tools for the automation of this method.

8. ADDITIONAL AUTHORS

Joachim Wegener: joachim.wegener@daimler-chrysler.com.

9. REFERENCES

[1] I. Baxter. Branch Coverage Tools for Arbitrary Languages Made Easy. *International Software Quality Week , (QWE 2002)*, September 2002.

[2] H.-G. Gross. An Evaluation of Dynamic, Optimization-based Worst-Case Execution Time Analysis. *Proceedings of the International Conference on Information Technology: prospects and Challenges in the 21st Century May23-26*, 2003.

[3] C. Haubelt, S. Mostaghim, F. Slomka, J. Teich, and A. Tyagi. Hierarchical Synthesis of Embedded Systems Using Evolutionary Algorithms. In R. Drechsler and N. Drechsler, editors, *Evolutionary Algorithms for Embedded System Design*, Genetic Algorithms and Evolutionary Computation (GENA), pages 5–9, Boston, Dordrecht, London, 2003. Kluwer Academic Publishers.

[4] Y. Malaiya, N. Li, R. Karcich, and B. Skbbe. The Relationship Between Test Coverage and Reliability. In *Proc. 5th International Symposium on Software Reliability Engineering*, pages 69–80, 1994.

[5] N. Mansour and M. Salame. Data Generation for Path Testing. *Software Quality Journal*, 12(2):121–136, 2004.

[6] H. Pohlheim. Geatbx: Genetic and Evolutionary Algorithm Toolbox for Use With Matlab. http://www.geatbx.com/.

[7] H. Pohlheim. Competition and Cooperation in Extended Evolutionary Algorithms. In E. D. Goodman, editor, *2001 Genetic and Evolutionary Computation Conference Late Breaking Papers*, pages 331–338, San Francisco, California, USA, 9-11 July 2001.

[8] P. P. Puschner and R. Nossal. Testing The Results of Static Worst-Case Execution-Time Analysis. In *IEEE Real-Time Systems Symposium*, pages 134–143, 1998.

[9] H. Singh, M. Conrad, and S. Sadeghipour. Test Case Design Based on Z and the Classification-Tree Method. In *ICFEM '97: Proceedings of the 1st International Conference on Formal Engineering Methods*, page 81, Washington, DC, USA, 1997. IEEE Computer Society.

[10] J. Wegener, A. Baresel, and H. Sthamer. Evolutionary Test Environment for Automatic Structural Testing. *Information & Software Technology*, 43(14):841–854, 2001.

[11] J. Wegener and M. Grochtmann. Evolutionary Testing of Temporal Correctness. *Proceedings of the 2nd International Software Quality Week Europe,(QWE 1998)*, November 1998.

[12] J. Wegener and R. Pitschinetz. Tessy-Yet Another Computer-Aided Software Testing Tool? *Proceedings of the Second European International Conference on Software Testing, Analysis and Review EuroSTAR 94*, 1994.