Black-box System Testing of Real-Time Embedded Systems Using Random and Search-based Testing

Andrea Arcuri[1], Muhammad Zohaib Iqbal[1,2], and Lionel Briand[1,2]

[1] Simula Research Laboratory, P.O. Box 134, Lysaker, Norway.
[2] Department of Informatics, University of Oslo
{arcuri,zohaib,briand}@simula.no

Abstract. Testing real-time embedded systems (RTES) is in many ways challenging. Thousands of test cases can be potentially executed on an industrial RTES. Given the magnitude of testing at the system level, only a fully automated approach can really scale up to test industrial RTES. In this paper we take a black-box approach and model the RTES environment using the UML/-MARTE international standard. Our main motivation is to provide a more practical approach to the model-based testing of RTES by allowing system testers, who are often not familiar with the system design but know the application domain well-enough, to model the environment to enable test automation. Environment models can support the automation of three tasks: the code generation of an environment simulator, the selection of test cases, and the evaluation of their expected results (oracles). In this paper, we focus on the second task (test case selection) and investigate three test automation strategies using inputs from UML/MARTE environment models: Random Testing (baseline), Adaptive Random Testing, and Search-Based Testing (using Genetic Algorithms). Based on one industrial case study and three artificial systems, we show how, in general, no technique is better than the others. Which test selection technique to use is determined by the failure rate (testing stage) and the execution time of test cases. Finally, we propose a practical process to combine the use of all three test strategies.

Key words: Search based software engineering, branch distance, model based testing, environment, context, UML, MARTE, OCL.

1 Introduction

Real-time embedded systems (RTES) represent a major proportion of the software being developed [1]. The verification of their correctness is of paramount importance, particularly when these RTES are used for business or safety critical applications (e.g., controllers of nuclear reactors and flying systems). Testing RTES is particularly challenging since they operate in a physical environment composed of possibly large numbers of sensors and actuators. The interactions with the environment can be bound by time constraints. For example, if the RTES of a gate is informed by a sensor that a train is approaching, then the RTES should command the gate to close down before the train reaches the gate. Missing such time deadlines can have disastrous consequences in the environment in which the RTES works. In general, the timing of interactions with the real-world environment in which the RTES operates can have a significant effect on the resulting behavior of test cases.

In this paper our objective is to enable the black-box, automated testing of RTES based on environment models. More precisely, our goal is to make such environment modeling as easy as possible, and allow the testers to automate testing without any knowledge about the design of the RTES. This is typically a practical requirement for independent system test teams in industrial settings. In addition, the test must be automated in such a way to be adaptable and scalable to the specific complexity of a RTES and available testing resources. By adaptable, we mean that a test strategy should enable the test manager to adjust the amount of testing to available resources, while retaining as high a fault revealing power as possible.

The system testing of a RTES requires interactions with the actual environment or, when necessary and possible, a simulator. Unfortunately, testing the RTES in the real environment usually entails a very high cost and in some cases the consequences of failures would not be acceptable, for example when leading to serious equipment damage or safety concerns. In our context, a test case is a sequence of stimuli, generated by the environment or its simulator, that is sent to the RTES. If a user interacts with the RTES, then the user would be considered as part of the environment as well. There is usually a great number and variety of stimuli with differing patterns of arrival times. Therefore, the number of possible test cases is usually very large if not infinite. A test case can also contain changes of state in the environment that can affect the RTES behavior. For example, with a certain probability, some hardware components might break, and that has effect on the expected and actual behavior of the RTES. A test case can contain information regarding when and in which order to trigger such changes.

Testing all possible sequences of environment stimuli/state changes is not feasible. In practice, a single test case of an industrial RTES could last several seconds/minutes, executing thousands of lines of code, generating hundreds of threads/processes running concurrently, communicating through TCP sockets and/or OS signals, and accessing the file system for I/O operations. Hence, systematic testing strategies that have high fault revealing power must be devised.

The complexity of modern RTES makes the use of systematic testing techniques, whether based on the coverage of code or models, difficult to apply without generating far too many test cases. Alternatively, manually selecting and writing appropriate test cases based on human expertise for such complex systems would be far too challenging and time consuming. If any part of the specification of the RTES changes during its development, a very common occurrence in practice, then many test cases might become obsolete and their expected output would potentially need to be recalculated manually. The use of an automated oracle is hence another essential requirement when dealing with complex industrial RTES.

In this paper we present a Model-Based Testing (MBT) [2] methodology to carry out system testing of RTES in a fully automated, adaptable, and scalable way. We tailor the principles of Adaptive Random Testing (ART) [3] and Search-Based Testing (SBT) [4] to our specific problem and context. For our empirical evaluation, we use Random Testing (RT) [5] as baseline. One main advantage of ART and SBT is that it can be tailored to whatever time and resources are available for testing: when resources are expended and time is up, we can simply stop their application without any side effect. A coverage-based strategy could not be, for example, interrupted at any time. Furthermore, ART and SBT attempt, through different heuristics, to maximize the chances to trigger a failure within time constraints. We will also see how their combined use can be helpful to gain the most out of testing resources in practice. The RTES under test (SUT) is treated as a black box: no internal detail or model of its behavior is required, as per our objectives. The first step is to model the environment using the UML standard and its MARTE profile, the latter being necessary to capture real-time properties. The use of international standards rather than academic notations is dictated by the fact that our solutions are meant to be applied by our industry partners. Environment models support test automation in three different ways:

- The environment models describe some of the structural and behavioral properties of the environment. Given an appropriate level of detail, they enable the automatic generation of an environment simulator to satisfy the specific needs of software testing.
- The models can be used to generate automated oracles. These could for example be invariants and error states that should never be reached by the environment during the execution of a test case (e.g., an open gate while a train is passing). In general, error states can model unsafe, undesirable, or illegal states in the environment. We used error states as oracles in our case studies.
- Test cases can be automatically selected based on the models, using various heuristics to maximize chances of fault detection. In our case studies we use ART and SBT.

In this paper we focus on the third item above and assess RT, ART, and SBT on the production code of a real industrial RTES. Due to space constraints, and because our focus in this paper is test automation, we do not explain in detail how to use UML/-MARTE to model the environment of a RTES and how simulator code can be automatically generated (which we investigated in [6]). To the best of our knowledge, no MBT automation results for ART and SBT on an actual RTES have ever been reported in the research literature. Since no freely available RTES was available, we also constructed three different artificial RTES in order to extend our investigation and better understand the influence of various factors on test cost-effectiveness such as the failure detection rate. The use of publicly available artificial RTES will also facilitate future empirical comparisons with our work since, due to confidentiality constraints, our industrial case study cannot be made public.

The paper is organized as follows. Section 2 provides an overview of related work. How the context is modeled and simulated is shortly discussed in Section 3. Section 4 describes the different strategies we used to generate test cases. Their empirical validation is described in Section 5 and threats to validity are discussed in Section 6. Finally, Section 7 concludes the paper.

2 Related Work

A large body of literature has been dedicated to test RTES. For reason of space, here we can only give a very brief and incomplete overview.

Most of the approaches to test RTES are based on violating their timing constraints [7] or checking their conformance to a specification [8]. The specification is generally a formal model of the system and this model is then used to generate test cases. A number of approaches have been proposed over the years to address the above problem. The most widely discussed approaches model the system using Timed Automata [9]. A number of Timed Automata extensions, such as Timed I/O Automata [10], have also been used for conformance testing. For the same purpose, UML statechart [11], Extended Finite State Machines [12] and Attributed Event Grammar [13] have also been used.

There are several works using SBT techniques for testing different aspects of RTES [14], as for example identify deadline misses [15] and testing functional properties [16].

The work presented here is significantly different from most the above approaches as we adopt, for practical reasons presented above, a black-box approach to system testing that relies on modeling the RTES environment rather than its internal design properties. As noted above, this is of practical importance as independent system test teams usually do not have easy access to precise design information. Most existing work does not focus on system testing, hence their emphasis on modeling the RTES internal behavior and structure. Another difference of practical importance, though this is not detailed in this paper, is that we use UML and its standard extensions for modeling the environment. Last but not least, as opposed to published case studies (e.g., [13, 12]), we assess our test strategies on the actual production code of an industrial RTES.

3 Environment Modeling & Simulation

For RTES system testing, software engineers would typically be responsible for developing the environment models. Therefore, the modeling language should be familiar to them and therefore based on software engineering standards. In other words, it is important to use a modeling language for environment modeling that is widely accepted and used by software engineers. Furthermore, standard modeling languages are widely supported in terms of tools and training. The Unified Modeling Language (UML) and its extensions are therefore a natural choice to consider in our context.

Several modeling and simulation languages are available and can be used for modeling and simulating the context (e.g., DEVS [17]). But in our case using these simulation languages raises a number of issues, including the fact that software engineers in the development team are usually not familiar with the notations and concepts of such languages.

Higher level programming languages (such as Java or C) can also be used as simulation languages. The major problem with the use of such languages is the low level of abstraction at which they “model” the environment. The software engineers will have to deal with all the programming language constructs (such as threads) while at the same time trying to focus on the details of the environment itself.

RTES testing through an environment simulator faces the question of how time is handled. Indeed, many properties of the RTES depend on whether some time constraints are fulfilled or not. Ideally, we would like to be able to simulate the passing of time in a deterministic way, but it is not always possible for large and complex RTES.

The opposite approach to time simulation would be to run the RTES with its simulated environment using the real clock of the CPU used to run the empirical analysis. On one hand, it has the benefit that we do not have any particular constraint on the type of RTES that can be analyzed. On the other hand, it adds noise and variance in the scheduled time events. If time constraints of the RTES are very tight (e.g., in the order of few milliseconds), then this approach is not a viable option.

In our work, we have used UML/MARTE as a simulation language. Models are developed in UML as classes and their state-machines. These models are then transformed into Java using model to text transformations. The activities and actions are written in Java and are converted into Java method calls. This was appropriate for the RTES considered in this paper. For other types of RTES, different programming languages could be necessary. Notice that our methodology is general. We chose Java only for practical reasons. In particular, in our empirical analyses we did not face the problem of the garbage collector interfering with time properties. The garbage collector was never called during the execution of a test case.

4 Automated Testing

4.1 Test Case Representation

In our context, a test case execution is akin to executing the environment simulator. Each state machine represents a component of the environment. There can be more instances of a state machine with different settings to represent different sensors/actuators of the same type. For example, in a gate controller RTES, we can have a state machine representing the trains. For each simulated train we will have an independent running instance of that state machine. The domain model is used to identify how many instances can or should run in parallel for each state machine. Based on the domain model, there could be different possible configurations of the environment, but in this paper we focus only on one fixed configuration.

In the behavioral models of the environment (i.e., the state machines) there can be non-deterministic parts. For example, a timeout transition could be triggered within a minimum and a maximum time value but the exact value cannot be determined. This is very typical when real-world components are modeled, in which for example there is always a natural variance when time-related properties are represented. Another example is when we assign probabilities p in the models to represent failure scenarios, as for example the breakdown of sensors/actuators. In our context, input data of a test case are the choice of the actual values to use in these non-deterministic events.

In our modeling methodology, we have non-deterministic choices only in the transitions between states. They can be in the trigger, the guard and the action of the transition. A transition might be taken several times, and this number might be unknown before executing the test case. Therefore, for each instance of the environment state machines, for each non-deterministic choice, we allocate in the test case a vector of possible values. The length of this vector is l. Each time such non-deterministic choice needs to be made, a value from the corresponding vector is selected. Because the vector has finite length l, it is used as a ring: The values are taken in order, and after l request for values, it starts again from the beginning of the vector. Figure 1 shows an example.

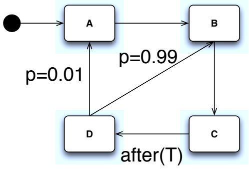

Fig. 1. Example of a reduced UML/MARTE state machine.

Let the transition $C \rightarrow D$ have a non-deterministic choice in $[0,1]$, for example the timeout $T \in [0,1]$. Given for example $l = 2$, we would have a data vector containing for example $\{0.4,0.32\}$. The first time the transition $C \rightarrow D$ is taken, the value 0.4 is used fort the non-deterministic choice. The second time, the value 0.32 is used. The third time, the value 0.4 is used again, and so on.

Given n state machine instances, and m non-deterministic choices in each of them (for simplicity, because in general instances of different machines will have a different number of non-deterministic choices), we would have that each test case contains $L = n*m*l$ values, which can be represented as a vector. The choice of l is arbitrary but has significant consequences. On one hand, a small number of possible values could make it impossible to represent sequences of event patterns that lead to failures in the RTES. On the other hand, a high number of possible values will lead to long vectors and might harm the effectiveness of test selections techniques such as ART and SBT (discussed in more details in the next sections).

In our case studies, the values to include in the test case data are chosen before the execution of the test cases. This means that the domain of these values should be static and not depending on the dynamic execution of the test cases. For example, if a variable is constrained within a minimum and maximum limit, then these boundaries should be known before test execution. This is the case for the industrial RTES analyzed in this paper and for other RTES we have worked with. When this is not the case, we would need to enable the choice of non-deterministic options at runtime.

4.2 Testing Strategies

As described in the previous section, a test case can be seen as a vector V. Elements in this vector can be of different types, but their domain of valid values should be known. Given $D(i)$ the domain of the ith variable in V, we obtain that the number of possible valid test cases is $\prod |D(i)|$, which is an extremely large number. An exhaustive execution of all possible test cases is infeasible.

In this paper we consider the testing problem of sampling test cases to detect failures of the RTES with automated oracles derived from the environment models. For all test strategies, the oracle checks whether a transition to an error state specified in the model occurs during test execution. We choose and execute test cases one at a time. We stop sampling test cases as soon as a failure has been found. A test strategy that requires the sampling of fewer test cases to detect failures should obviously be preferred.

The simplest, automated technique to choose test cases is Random Testing (RT). For each variable in V, we simply sample a value from its domain with uniform probability. Although RT can be considered to be a naive technique, it has been shown to be effective in many testing situations [18, 19].

Another technique that we investigate is Adaptive RT (ART) [3], which has been proposed as an extension of RT. The underlying idea of ART is that diversity among test cases should be rewarded, because failing test cases tend to be clustered in contiguous regions of the input domain. ART can be automated if one can define a meaningful similarity function for test cases. To the best of our knowledge, we are aware of no previous application of ART to test RTES. In this paper we use the basic ART algorithm described in [3].

Because in our case studies all the variables in V are numerical, for the distance between two test case data vectors $V1$ and $V2$ we use the following $dis(V1{,}V2) = \sum abs(V1[i] - V2[i])/|D(i)|$. We sum the absolute difference of each variable weighted by the cardinality of the domain of that variable. Often, these variables represent the time in timeout transitions. Therefore, ART rewards diversity in the triggering time of events.

In this paper we also investigate the use of search algorithms to tackle the testing of RTES. In particular we consider the use of Genetic Algorithms (GAs), which are the most used search algorithms in the literature on search-based testing (SBT) [14]. To use search algorithms to tackle a specific problem, a fitness function needs to be defined tailored to solve that problem. Search algorithms exploit the fitness function to guide the search toward promising areas of the search space. The fitness function is used to heuristically evaluate how "good" a test case is. In our case, the fitness function is used to estimate how close a test case is from triggering a failure in the RTES, that is when at least one component of the environment enters an error state. This is once again determined by analyzing the environment models.

To tackle the testing problem described in this paper, we developed a novel fitness function f that can be seen as an extension of the fitness functions that are commonly used for structural testing [4] and MBT [20]. In our case, the goal is to minimize the fitness function f. If at least one error state is reached when a test case with test data V is executed, then $f(V) = 0$. For each error state E in each state machine instance we employ the so called approach level A and branch distance B. The approach level calculates the minimum number of transitions in the state machine to reach an error state from the closest executed state. The branch distance is used to heuristically score the evaluation of the Object Constraint Language (OCL) constraints in the closest executed state from which the approach level is calculated. The branch distance is used to guide the search to find test data that satisfy those OCL constraints. A transition could be triggered several times but never executed because the guard fails. For the branch distance, we calculate it every time but then we only consider the minimum value it obtains. Because the branch distance is less important than the approach level, it is normalized in the range $[0{,}1]$. We use the following normalizing function $nor(x) = x/(x+1)$, which has been shown to be better than other normalizing functions used in the literature [21]. Notice that, in the case of MBT, it is not always possible to calculate the branch distance when the related transition has never been triggered. In these cases, we assign to the branch distance B its highest possible value.

The extension of the fitness function we make in this paper exploits the time properties of the RTES. Some of the transitions are triggered when a time-threshold is violated. For example, an error state could be reached if a sensor/actuator does not receive a message from RTES within a time limit. If such transitions exist on the path toward the execution of the error states, then we need a way to reward test data that get the execution closer to violate those time constraints. If a transition is taken after a threshold z, then we calculate the maximum consecutive time t the state machine stays in the state from which that transition can be triggered (this would be the same state from which the approach level is calculated from). Then, to guide the search we can use the following heuristic $T = z - t$, where $t \leq z$.

Finally, the fitness function f for a test data vector V is defined as:

$$f(V) = min_E(A_E(V) + nor(T_E(V)) + nor(B_E(V))).$$

Notice that, to collect information such as the approach level, the source code of the simulator needs to be instrumented. This is automatically done when this code is generated from the environment models.

Once the fitness function is defined, we can use it to guide the GA to select test cases. But GAs have many parameters that need to be set. In this paper we use a Steady State GA [4]. We employ rank selection with bias 1.5 to choose the parents. A single point crossover is employed with probability $P_{xover} = 0.75$. This operator chooses a random point inside the data vectors V of the parents sx and sy. The elements in the data vector after that splitting point are swapped between the two parent solutions. Each of the L elements in a data vector is mutated with probability $1/L$. A mutation consists of replacing a value with another one at random from the same domain. The population size is chosen to be 10. The optimal configuration of search algorithms is in general problem dependent [22]. Due to the large computational cost of running our empirical analysis, we have not tuned the GA. We simply use reasonable parameter values given in the literature of GAs.

5 Empirical Study

5.1 Case Study

To validate the novel approach presented in this paper, we have applied it to test an industrial RTES. The analyzed system is a very large and complex controller that interacts with several sensors/actuators. The company that provided the system is a market leader in its field. For confidentiality reasons we cannot provide full details of the system. Information of the environment models of this RTES is provided in Table 1. Notice that for this case study there are several state machines, and for each of them there can be one or more instances running in parallel at the same time. For each test case, 23 instances of state machines run in parallel, each of them can start several threads. The total number of non-deterministic choices (NDCs) is 82. The UML/MARTE context models

Table 1. Summary of the state machines of the environment of the industrial RTES. NDC stands for "Non-Deterministic Choice".

State Machine	States	Transitions	Error States	Instances	NDCs for Instance
S1	19	29	1	10	6
S2	4	7	0	11	2
S3	3	8	1	1	0
S4	5	5	0	1	0

were developed in IBM Rational Software Architect. Constraints, such as guards, were expressed in OCL.

To facilitate future comparisons with the techniques described in this paper, it would be necessary to also employ a set of benchmark systems that are freely available to researchers. Unfortunately, we have not found any RTES satisfying this criterion. Therefore, in addition to our industrial case study, we have designed three artificial RTES, called AP1, AP2 and AP3. Two of them are inspired by the industrial RTES used in this paper, whereas the third is inspired by the control gate system described in [12]. The RTES are written in Java to facilitate their use on different machines and operating systems. For the same reason, the communications between the RTES and their environments are carried out through TCP. The use of TCP was also essential to simplify the connection of the RTES with its environment. For example, if the simulator of the environment is generated from the models using a different target language (e.g., C/C++), then it will not be too difficult to connect to the artificial RTES written in Java. These RTES are all multithreaded. Table 2 summarizes the properties of these artificial RTES. In each of them, there is only one error state. We introduced by hand a single non-trivial fault in each of these RTES.

Table 2. Properties of the three artificial problems. LoC stands for "Lines of Code", whereas NDC stands for "Non-Deterministic Choice".

Artificial Problem	LoC of RTES	LoC of Environment	State Machines	States	Transitions	Instances	Total NDCs
AP1	227	259	1	5	7	10	20
AP2	409	271	1	5	7	2	4
AP3	337	318	2	9	13	5	18

5.2 Experiments

We have carried out two different sets of experiments. One for the artificial problems, and one for the industrial RTES. In all these experiments, the value l for the non-deterministic choices is set to $l = 3$. This means that the number of input variables in each test case is 60 for AP1, 12 for AP2, 54 for AP3 and finally 246 for the industrial RTES.

In the first step of the experiments, we ran RT, ART and GA on each of the three artificial problems. Because the execution of a single test case takes 10 seconds, we stop each algorithm after 1000 sampled test case or as soon as one of the error state is reached. Notice that the value 10 seconds is fixed, and it does not depend on the used execution platform. Using faster hardware would not change the amount of time required to run these experiments. The only requirement is that the hardware used for the experiments is fast enough to sustain the CPU load without introducing delays higher than a few milliseconds. Because in these simulations most of the time the CPU is in idle state, the computers used in the experiments were appropriate.

For each test strategy and each case study, we ran the algorithms 100 times with different random seeds. Because these algorithms are randomized, a large number of experiments is required to obtain statistically significant results. The total number of sampled test cases is hence at most $3 * 3 * 1000 * 100 = 900{,}000$, which can take up to 104 days on a single computer. To cope with this problem, we used a cluster to run these experiments.

Given an upper bound of 1000 test cases, it is not always the case that any of the test strategies is able to trigger a failure in the RTES. In Table 3 we report how many times each algorithm was able to do so out of the 100 experiments. Because the process of detecting failures in 100 experiments can be considered to be a binomial process with unknown probability [23], we use the Fisher Exact test to compare the success rate of RT with the ones of ART and GA. The significance level of the tests is set to 0.05. Results show that the only case in which there is no significant difference in the success rate is for problem AP2 when RT is compared to ART.

Table 3. Success rate (out of 100 runs) for the three artificial problems.

Algorithm	AP1	AP2	AP3
RT	6	35	49
ART	0	40	74
GA	90	21	31

Table 4. Number of sampled test cases to detect the first failure in the considered industrial RTES. “SD” stands for Standard Deviation.

Algorithm	Min	Median	Mean	Max	SD
RT	1	73.0	131.9	912	164.9
ART	1	75.5	104.6	525	99.7
GA	1	99.0	160.0	767	155.2

The second set of experiments has been carried out on an industrial RTES. In system testing of RTES, the simulation of the environment can in general be run for any arbitrary amount of time. But there should be enough time to render possible the execution of all the functionalities of the RTES. For example, in the RTES for a train/gate controller, we should run the simulation at least long enough to make it possible for a train to arrive and then leave the gate. Choosing for how long to run a simulation (i.e., a test case) is conceptually the same as the choice of test sequence length in unit testing [24] (i.e., many short test cases or only few ones that are long?). But in contrast to unit testing in which often the execution time of a test case is in the order of milliseconds, in the system testing of RTES we have to deal with much longer execution time. In this paper, we run each test case for 20 seconds. This choice has been made based on the properties of the RTES and discussions with its software testers.

We evaluated the use of RT, ART and GA to find failures in this RTES. We could not run this empirical analysis on a cluster due to technical reasons. We used a single dedicated computer, and it took nearly ten days to run these experiments. The failure rate of the SUT in these experiments was quite high, so we did not use any upper bound for the number of sampled test cases. The results of experiments are shown in Table 4.

To analyze the results in a sound manner we carried out a set of statistical tests on the data presented in Table 4. We used parametric *t*-tests to see whether there is any statistical difference between the mean values of sampled test cases among the three analyzed algorithms. The scientific or practical significance of these differences is evaluated using the Cohen D coefficient. We also carried out non-parametric Mann-Whitney U tests to see whether any of the results of these algorithms is stochastically greater than the others. The scientific significance of this test is measured with the Vargha-Delaney A statistic. For both *t*-tests and Mann-Whitney U tests the significant level is set to 0.05. For the Cohed D coefficient (value d), we classify the effect size as follows [25]: small for $abs(d) = 0.2$, medium for $abs(d) = 0.5$, and finally large for value $abs(d) = 0.8$. In the case of Vargha-Delaney A statistic (value a), we use the following classification [26]: small for $abs(a - 0.5) = .06$, medium for $abs(a - 0.5) = 0.14$ and large for $abs(a - 0.5) = 0.21$. Table 5 summarizes the results of these statistical tests.

Table 5. Results of the statistical tests for the data in Table 4.

Comparison	*t*-tests p-value	Cohen D	U-test p-value	Vargha-Delaney A
RT vs ART	0.1588	0.2012	0.9708	0.5015
RT vs GA	0.2150	-0.1768	0.0334	0.4129
ART vs GA	0.0030	-0.4272	0.0193	0.4042

5.3 Discussion

In the results of the experiments on the artificial problems shown in Table 3, we can see that no testing technique generally dominates the others. GA is statistically better on the first problem, but it is the worst on the other two problems. Regarding RT and ART, they are equivalent on the second problem, but RT is best on the first, whereas ART is best on the third problem.

The results in Table 3 for GA can be precisely explained. Covering all the non-error states and transitions in the environment models of these problems is very easy, practically all test strategies achieve this. The only difficult part is the transition to the error state. For the first problem AP1, that transition is a time transition with no guard. After a time threshold, that transition is triggered. The novel fitness function proposed in this paper can take advantage of this information, rewarding test cases that get closer to violate that time constraint. In fact, for each test case we can automatically calculate the time that it spends in the state that could lead to the error state. This automated fitness function produces an easy fitness landscape that can be efficiently searched by GA. This explains the fact that GA gets to the error state 90% of the time, whereas RT reaches it only in 6% of the time. However, why do we obtain so much worse results in the other two problems AP2 and AP3? The reason is that the fitness function in these cases is practically a needle-in-the-haystack function. In the transition to the error state, there is a guard that is checking whether one Boolean variable is equal to true. The value of this variable depends on the interactions with the SUT, particularly whether a specific message has been received or not. This type of guard in search-based testing is a known, very difficult problem denoted as the flag problem [27]. In this case, the fitness function provides no gradient, and this makes the search difficult. Unfortunately, testability transformations [27] cannot be used in this case, because in our context the SUT is a black box. Even if we had access to the SUT, it would still be problematic, because we are aware of no work dealing with the flag problem for the system testing of concurrent programs. Though the above issue is a limitation, in practice, we can automatically determine before running GA whether it will work.

Though we can explain why GA does not work well on AP2 and AP3, why does it behave even worse than RT? The reason is exactly the same for which ART is better than RT: the diversity of the test cases. If there is no gradient in the fitness function, all the sampled test cases would have same fitness value (i.e., the fitness landscape would have a large plateau). So any new sampled test case would be accepted and added to the next generation in GA. The crossover operator does not produce any new value in the data vector V, it simply swaps values between two parent test cases. The mutator operator does only small changes to a data vector, because on average only one variable is mutated. During the search, the offspring have genetic material (i.e., the data vectors) that is similar to the one of the parents. Therefore, the diversity of test cases during GA evolution is much lower than the one of RT. If the hypothesis of contiguous regions of faulty test cases is true for a RTES, then, when there is no gradient in the fitness function, we would a-priori expect this following relationship regarding the performance of testing strategies: $GA \leq RT \leq ART$. For problems AP2 and AP3, this is verified in the results of Table 3.

In the experiments on the industrial RTES, we can see that GA is statistically worse than the other approaches, although the difference is only small/medium in size from a scientific point of view. The results on the industrial RTES shown in Table 4 are important to stress out that the choice of a testing strategy is also heavily dependent on when the SUT is tested. The version of the industrial RTES used in this paper was not a finished product. It was in an early phase of development. The types of failure scenarios introduced with our models were not something that was fully tested before. This explains the high failure rate shown in Table 4. Notice that the failure rate θ can be simply estimated from the mean value of RT, i.e. $\theta = 1/mean(RT)$. The reason is that RT follows a geometric distribution with parameter θ, therefore $mean(RT) = 1/\theta$. In our case, we have $\theta = 1/131.9 = 0.007$, which can be considered to be a high failure rate.

5.4 Practical Guidelines

For high failure rates, it makes sense to use a simple RT instead of more sophisticated techniques, since the expected number of sampled test cases would be low on average. In practice, we would expect high failure rates at the beginning of the testing phase. The failure rate would hence be expected to decrease throughout the development process as faults get fixed. Therefore, we would expect to get good results for RT at the beginning, but then more sophisticated techniques could be required at later stages.

Our results lead us to suggest the following heuristics to apply RT, ART, and SBT in practice: In the early stages of development and testing, when failure rates are still high, one should use RT as it will be very efficient and quick to detect the first failure, without requiring any overhead like ART or SBT. One exception to this rule is when the time of executing a test case is high (e.g., in the order of several seconds or minutes), where we then suggest to use ART as one should enforce test execution diversity to prevent the execution of too many test cases. Once the failure rate decreases due to the fixing of *easy-to-detect* faults, then use SBT, but only if a proper fitness function can be derived automatically from the models, that is a fitness function that is likely to provide effective guidance for the search of failing test cases. Otherwise, use RT. ART should not be used when the failure rate is low as the overhead of distance calculations would get too high, due to the large number of test cases executed.

Figure 2 summarizes the above heuristic in a decision tree and it shows when to apply each testing technique. We provide practical advice regarding when to switch from ART to RT below. But for the switch from RT to SBT, we need more empirical/theoretical analyses to provide practical guidelines.

In the literature, it has been shown that ART can be twice as fast as RT [3]. Let us consider t_{tc} the execution time of a test case, t_{dis} the execution time of a distance calculation with d the total number of distances computed, θ the failure rate, $E[RT]$ and $E[ART]$ the expected number of test cases sampled by RT and ART. We know that $E[RT] = 1/\theta$ and that, under optimal conditions, $E[ART] = E[RT]/2$. We can develop a heuristic that is based on the following equation: $E[RT]{\cdot}t_{tc} = E[ART]{\cdot}t_{tc} + d{\cdot}t_{dis}$, which is a *loose approximation* to determine the failure rate θ^* above which ART is going to yield better results than RT. From that equation, it follows $\theta^* \approx \frac{t_{dis}}{4 \cdot t_{tc}}$. This optimal threshold for ART for the failure rate can be estimated before test execution. Finally, we can suggest to run ART for $1/2\theta^*$ iterations, but only as long as the number of sampled test cases is not high enough to make the decision to switch to SBT. The above recommendations are heuristics and will need to be evaluated and refined as we gather more empirical data.

6 Threats to Validity

Due to the complexity of the industrial RTES used in the empirical study of this paper, we could not run the RTES and its simulated environment in such a way to obtain a precise and deterministic handling of clock time. We used the CPU clock instead. This could be unreliable if time constraints in the RTES are very tight, as for example in the order of milliseconds, because these constraints could be violated due to unpredictable

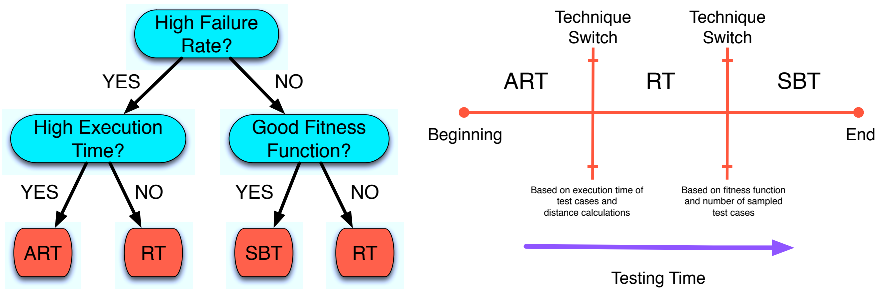

Fig. 2. Decision tree and and application timeline of the three analyzed testing strategies.

changes of load balance in the CPU because of unrelated processes. Although the time constraints in this paper were in the order of seconds, the problem could still remain. To evaluate whether our results are reliable, we hence selected a set of experiments, and we re-ran them again with exactly the same random seeds. We obtained equivalent results. For example, if RT for a particular seed obtained a failing test case after sampling 43 test cases, then, when we ran it again with the same seed, it was still requiring exactly 43 test cases. However, the experiments were not exactly the same. For example, for debugging purposes we used time stamps on log files. In these time stamps, small variances of a few milliseconds were present, but this did not have any effect on the testing results. Notice that our novel methodology can obviously be applied also when time clocks are simulated.

7 Conclusion

In this paper we proposed a black-box system testing methodology, based on environment modeling and various heuristics for test case generation. The focus on black-box testing is due to the fact that system test teams are often independent from the development team and do not have (easy) access to system design expertise. Our objective is to achieve full system test automation that scales up to large industrial RTES and can be easily adjusted to resource constraints. The environment models are used for code generation of the environment simulator, selecting test cases, and the generation of corresponding oracles. The only incurred cost by human testers is the development of the environment models. This paper, due to space constraints, has focused on the testing heuristics and an empirical study to determine the conditions under which they are effective, plus guidelines to combine them in practice.

In contrast to most of the work in the literature, the modeling and the experiments were carried out on an industrial RTES in order to achieve maximum realism in our results. However, in order to more precisely understand under which conditions each test heuristic is appropriate and how to combine them, we complemented this industrial study with artificial case studies, that will be made publicly available to foster future empirical analyses and comparisons.

We experimented with different testing heuristics, which have the common property to be easily adjustable to available time and resources: Random Testing (RT), Adaptive Random Testing (ART) and Search-Based Testing using Genetic Algorithms (GAs). All these techniques can be adjusted to project constraints as they can be run as long as time and access to CPU are available. Though RT was originally used as comparison baseline, it turned out to be the best alternative under certain conditions.

On the artificial problems, in one case GA is the best search algorithm, and the difference is very large. But on the other two cases, GA has the worst results, which are due to poor fitness functions. In one case RT and ART are equivalent, but in the other two, RT is better in one case and worse in the other.

However, on the industrial RTES, results are quite different from the artificial case studies: there is no statistical difference between RT and ART, whereas GA is slightly worse than the others (the effect size is between *small* and *medium*). After investigation, this was found to be due to the RTES high failure rate and a fitness function that offered little guidance to the search due to a Boolean guard condition. To support the claims above, we followed a rigorous experimental method based on five types of statistical analyses.

Based on our results, we have provided practical guidelines to apply the three testing techniques described in this paper, i.e. RT, ART, and GA. In fact, none of them dominates the others in all testing conditions and they must be, in practice, combined to achieve better results. However, more empirical and theoretical studies are needed to develop more precise, practical guidelines.

One current limitation of our testing approach is that the domains of valid values for the non-deterministic test inputs need to be static: they should be known before test case execution. Research will need to be carried out to design novel testing strategies for non-deterministic inputs that can only be determined at runtime.

Acknowledgements

The work described in this paper was supported by the Norwegian Research Council. This paper was produced as part of the ITEA-2 project called VERDE.

References

1. Douglass, B.P.: Real-time UML: developing efficient objects for embedded systems. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA (1997)
2. Utting, M., Legeard, B.: Practical model-based testing: a tools approach. Elsevier (2007)
3. Chen, T.Y., Kuoa, F., Merkela, R.G., Tseb, T.: Adaptive random testing: The art of test case diversity. Journal of Systems and Software (JSS) (2010) (in press).
4. McMinn, P.: Search-based software test data generation: A survey. Software Testing, Verification and Reliability **14**(2) (2004) 105–156
5. Myers, G.: The Art of Software Testing. Wiley, New York (1979)
6. Iqbal, M.Z., Arcuri, A., Briand, L.: Environment Modeling with UML/MARTE to Support Black-Box System Testing for Real-Time Embedded Systems: Methodology and Industrial Case Studies. In: ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS). (2010)

7. Clarke, D., Lee, I.: Testing real-time constraints in a process algebraic setting. In: IEEE International Conference on Software Engineering (ICSE). (1995) 51–60
8. Krichen, M., Tripakis, S.: Conformance testing for real-time systems. Formal Methods in System Design **34**(3) (2009) 238–304
9. Alur, R., Dill, D.L.: A Theory of Timed Automata. Theoretical Computer Science **126** (1994) 183–235
10. En-Nouaary, A.: A scalable method for testing real-time systems. Software Quality Journal **16**(1) (2008) 3–22
11. Miicke, T., Huhn, M.: Generation of optimized testsuites for UML statecharts with time. In: IFIP international conference on testing of communicating systems. (2004) 128–143
12. Zheng, M., Alagar, V., Ormandjieva, O.: Automated generation of test suites from formal specifications of real-time reactive systems. Journal of Systems and Software (JSS) **81**(2) (2008) 286–304
13. Auguston, M., Michael, J.B., Shing, M.T.: Environment behavior models for automation of testing and assessment of system safety. Information and Software Technology (IST) **48**(10) (2006) 971–980
14. Harman, M., Mansouri, S.A., Zhang, Y.: Search based software engineering: A comprehensive analysis and review of trends techniques and applications. Technical Report TR-09-03, King's College (2009)
15. Garousi, V., Briand, L.C., Labiche, Y.: Traffic-aware stress testing of distributed real-time systems based on uml models using genetic algorithms. Journal of Systems and Software (JSS) **81**(2) (2008) 161–185
16. Lindlar, F., Windisch, A., Wegener, J.: Integrating model-based testing with evolutionary functional testing. In: International Workshop on Search-Based Software Testing (SBST). (2010)
17. Zeigler, B.P., Praehofer, H., Kim, T.G.: Theory of modeling and simulation. Academic press New York, NY (2000)
18. Duran, J.W., Ntafos, S.C.: An evaluation of random testing. IEEE Transactions on Software Engineering (TSE) **10**(4) (1984) 438–444
19. Arcuri, A., Iqbal, M.Z., Briand, L.: Formal analysis of the effectiveness and predictability of random testing. In: ACM International Symposium on Software Testing and Analysis (ISSTA). (2010)
20. Lefticaru, R., Ipate, F.: Functional search-based testing from state machines. In: IEEE International Conference on Software Testing, Verification and Validation (ICST). (2010) 525–528
21. Arcuri, A.: It does matter how you normalise the branch distance in search based software testing. In: IEEE International Conference on Software Testing, Verification and Validation (ICST). (2010) 205–214
22. Wolpert, D.H., Macready, W.G.: No free lunch theorems for optimization. IEEE Transactions on Evolutionary Computation **1**(1) (1997) 67–82
23. Feller, W.: An Introduction to Probability Theory and Its Applications, Vol. 1. 3 edn. Wiley (1968)
24. Arcuri, A.: Longer is better: On the role of test sequence length in software testing. In: IEEE International Conference on Software Testing, Verification and Validation (ICST). (2010) 469–478
25. Cohen, J.: A power primer. Psychological bulletin **112**(1) (1992) 155–159
26. Vargha, A., Delaney, H.D.: A critique and improvement of the CL common language effect size statistics of McGraw and Wong. Journal of Educational and Behavioral Statistics **25**(2) (2000) 101–132
27. Harman, M., Hu, L., Hierons, R., Wegener, J., Sthamer, H., Baresel, A., Roper, M.: Testability transformation. IEEE Transactions on Software Engineering **30**(1) (2004) 3–16