Simula Research Laboratory Technical Report (2012-12)

Combining Search-based and Adaptive Random Testing for Black-box System Testing of Real-time Embedded Systems

Muhammad Zohaib Iqbal[1, 2], Andrea Arcuri[1], Lionel Briand[1, 3]

[1] Certus Center for V & V, Simula Research Laboratory, P.O. Box 134, Lysaker, Norway
[2] Department of Informatics, University of Oslo, Norway
[3] SnT Center, University of Luxembourg, Luxembourg
{zohaib, arcuri }@simula.no, lionel.briand@uni.lu

Abstract. Effective system testing of real-time embedded systems (RTES) requires a fully automated approach. One such black-box system testing approach is to use environment models to automatically generate test cases and test oracles along with an environment simulator to enable early testing of RTES. In this paper, we propose a hybrid strategy, which combines (1+1) Evolutionary Algorithm (EA) and Adaptive Random Testing (ART), to improve the overall performance of system testing that is obtained when using each single strategy in isolation. An empirical study is carried out on a number of artificial problems and one industrial case study. The novel strategy shows significant overall improvement in terms of fault detection compared to individual performances of both (1+1) EA and ART.

1. Introduction

Real-time embedded systems (RTES) are widely used in critical domains where high system dependability is required. These systems typically work in environments comprising of large numbers of interacting components. The interactions with the environment are typically bounded by time constraints. Missing these time deadlines, or missing them too often for soft real-time systems, can lead to serious failures leading to threats to human life or the environment. There is usually a great number and variety of stimuli from the RTES environment with differing patterns of arrival times. Therefore, the number of possible test cases is usually very large if not infinite. Testing all possible sequences of stimuli is not feasible. Hence, systematic automated testing strategies that have high fault revealing power are essential for effective testing of industry scale RTES. The system testing of RTES requires interactions with the actual environment. Since, the cost of testing in actual environments tends to be high, environment simulators are typically used for this purpose.

In our earlier work, we proposed an automated system testing approach for RTES software based on environment models [1, 2]. The models are developed according to a specific strategy using the Unified Modeling Language (UML) [3], the Modeling and Analysis of Real-Time Embedded Systems (MARTE) profile [4] and our proposed profile [5]. These models of the environment were used to automatically generate an environment simulator [6], test cases, and obtain test oracle [1, 2].

In our context, a test case is a sequence of stimuli generated by the environment that is sent to the RTES. A test case can also include 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 affects the expected and actual behavior of the RTES. A test case can contain information regarding when and in which order to trigger such changes. So, at a higher level, a test case in our context can be considered as a setting specifying the occurrence of all these environment events in the simulator. Explicit “error” states in the models represent states of the environment that are only reached when RTES is faulty. Error states act as the oracle of the test cases, i.e., a test case is successful in triggering a fault in the RTES if any of these error states is reached during testing.

In previous work, we investigated several testing strategies to generate test cases. We used random testing (RT) [7] as baseline, and then considered two different approaches: Search-based Testing (SBT) [8] and Adaptive Random Testing (ART) [1]. For SBT, an *order function* was defined that utilizes the information in environment models to guide the search toward the error states. In contrast, with ART, test cases are rewarded based on their *diversity*. The results indicated that, apart from the failure rate of the system under test (SUT), the effectiveness of a testing algorithm also depends on the characteristics of the environment models. For problems where the environment model is easier to cover or where the failure rate of the RTES is high, even RT outperforms SBT. However, for more complex problems, SBT showed much better performance than RT. This raised the need for a strategy that combines the individual benefits of the two strategies and utilizes adaptive mechanisms based on the feedback from executed test cases.

In this paper, we extend our previous work by devising such a hybrid strategy that aims at combining the best search technique, i.e., (1+1) Evolutionary Algorithm (EA) in our experiments and ART (which is the algorithm that gave best results in our earlier experiments in [2]) in order to achieve better overall results in terms of fault detection. We defined two different strategies for combining these algorithms, but due to space constraints, in this paper, we only discuss the strategy that showed the best results. The hybrid strategy (HS) discussed here starts with running (1+1) EA and switches to ART when (1+1) EA stops yielding fitter test cases. The decision of when to switch (referred to as *configuration*) can have significant impact on the performance of the strategy and one main objective of this paper is to empirically investigate different configuration options. The other combination strategy started by running ART and later switched to (1+1) EA if consecutive test cases generated through ART showed better fitness compared to previously executed test cases. It did show improvements over the individual algorithms, but fared worse than HS.

We evaluate the fault detection effectiveness of HS by performing a series of experiments on 13 artificial problems and an industrial case study. The RTES of the artificial problems were based on the specifications of two industrial case studies. Their environment models were developed in a way to vary possible modeling characteristics so as to understand their effect on the performance of the test strategies. We could not have covered such variations in environment models with one or even a few industrial case studies, hence the motivation to develop artificial cases. The industrial case study used is of a marine seismic acquisition system, which was developed by a company leading in this industry sector. For all these cases, we compared the performance of HS (with best configuration) with that of ART, (1+1) EA, and RT. The results suggest that in terms of success rates (number of times an algorithm found a fault within a given test budget), for the problems where RT/ART showed better performance over (1+1) EA, HS results are similar to ART/RT and for the problems where (1+1) EA was better, HS results are similar to those of (1+1) EA, thus suggesting that HS combines the strength of both algorithms.

The rest of the paper is organized as follows. Section 2 discusses the related work, while Section 3 provides an introduction to the earlier proposed environment model-based system testing methodology that we improve in this paper. Section 4 describes the proposed hybrid strategy, whereas Section 5 reports on the empirical study carried out for evaluation purposes. Finally, Section 6 concludes the paper.

2. Related Work

Depending on the goals, testing of RTES can be performed at different levels: model-in-the-loop, hardware-in-the-loop, processor-in-the-loop, and software-in-the-loop [9]. Our approach falls in the software-in-the-loop testing category, in which the embedded software is tested on the development platform with a simulated environment. The only variation is that, rather than simulating the hardware platform, we use an adapter for the hardware platform that forwards the signals from the SUT to the simulated environment. This approach is especially helpful when the software is to be deployed on multiple hardware platforms or the target hardware platform is stable.

There are only a few works in literature that discuss RTES testing based on environment models rather than system models. Auguston *et al.* [10] discusses the modeling of environment behaviors for testing of RTES using an event grammar. The behavioral models contain details about the interactions with the SUT and possible hazardous situations in the environment. Heisel *et al.* [11] propose the use of a requirement model and an environment model along with the model of the SUT for testing. Adjir *et al.* [12] discuss a technique for testing RTES based on the system model and assumptions in the environment using Labeled Prioritized Timed Petri Nets. Larsen *et al.* [13] propose an approach for online RTES testing based on time automata to model the SUT and environmental constraints. Iqbal *et al.* [5] propose an environment modeling methodology based on UML and MARTE for black-box system testing. Fault detection effectiveness of testing strategies based on these models was evaluated and reported in [8], including RT/ART [1], GA, and (1+1) EA. The results indicate that SBT show significantly better performance over RT for a number of cases and significantly worse performance than RT for a number of other cases.

There has been some work to combine SBT with RT. Andrews *et al.* propose the use of GA to tune parameters for random unit testing [14]. An evolutionary ART algorithm that uses the ART distance function as a fitness function for GA is proposed in [15]. In [16], the authors propose a search-based ART algorithm by using a variant of ART distance function as the fitness function for Hill Climbing to optimize the results of ART when the input domains are more than two dimensional.

The work presented here improves the work on environment model-based testing presented in [8] by combining the strengths of both ART and (1+1) Evolutionary Algorithm. Approaches discussed in the literature for combining ART/RT with SBT are restricted to improving ART or tuning RT by using search techniques. In contrast, here we want to use (1+1) EA to generate test cases that exploit the characteristics of environment models as well as benefit from the test diversity generated by ART, thus combining the two approaches.

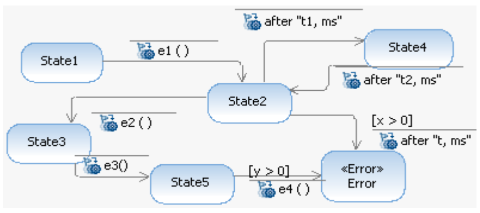

Fig. 1. A dummy state machine to explain search heuristics

3. Environment Model-based Testing

In this section, we discuss in more details the various components of our environment model-based testing approach.

3.1. Environment Modeling & Simulation

For RTES system testing, as we observed with our industry partner, software engineers familiar with the application domain would typically be responsible for developing the environment models. Therefore, we selected UML and its extensions as the environment modeling language, which is a standard modeling language that is widely taught and accepted by software engineers, in addition to be widely supported in terms of tools and training material. These are important considerations for successful industry adoption.

The environment models consist of a domain model and several behavioral models. The domain model, represented as a class diagram, captures the structural details of the RTES environment, such as the environment components, their relationships, and their characteristics. The behavior of the environment components is captured by state machines. These models are developed, based on our earlier proposed methodology by using UML, MARTE, and our proposed profile for environment modeling [5]. These models not only include the nominal functional behavior of the environment components (e.g., booting of a component) but also include their robustness (failure) behavior (e.g., break down of a sensor). The latter are modeled as "failure" states in the environment models. The behavioral models also capture what we call "error" states. These are the states of the environment that should never be reached if the SUT is implemented correctly (e.g., no incorrect or untimely message from the SUT to the environment components). Therefore, error states act as oracles for the test cases.

An important feature of these environment models is that they capture the non-determinism in the environment, which is a common characteristic for most RTES environments. Non-determinism may include, for example, different occurrence rates and patterns of signals, failures of components, or user commands. Each environment component can have a number of non-deterministic choices whose exact values are selected at the time of testing. Java is used as an action language and OCL (Object Constraint Language) is used to specify constraints and guards.

Using model to text transformations, the environment models are automatically transformed into environment simulators implemented in Java. The transformations follow specific rules that we discussed in detail in [6]. During simulation a number of instances can be created for each environment component, which can interact with each others and the SUT (for example multiple instances of a sensor component). The generated simulators communicate with the SUT through a communication layer (e.g., TCP layer), which is written by software engineers. They are also linked with the test framework that provides the appropriate values for each simulation execution. The choice of Java as target language is based on actual requirements of our industrial partner, where the RTES under study only involves soft real-time constraints.

3.2. Testing RTES based on Environment Models

In our context, a test case execution is akin to executing the environment simulator. During the simulation, values are required for the non-deterministic choices in the environment models. A test case, in our context, can be seen as a test data matrix, where each row provides a series of values for a non-deterministic choice of the environment component (the number of rows is equal to the number of non-deterministic choices). Each time a non-deterministic choice needs to be made, a value from the corresponding matrix row is selected.

During the simulation, a query for a non-deterministic choice can be made several times and the number of queries cannot be determined before simulation. To resolve this problem, each matrix row (a data vector) is represented as a variable size vector so that whenever the end of a vector is reached, its size is increased at run time and new values are added. In our earlier work [2], we evaluated the effect of the representations and starting lengths of the test data vectors on the fault detection effectiveness and showed that such a variable size vector is a suitable solution to this problem. In [1], we applied various testing strategies to generate test cases from the environment models, including ART, RT, and Genetic Algorithms (GA).

Given a test data matrix, a test case can be run for any arbitrary length of time (e.g., 10 seconds, one hour). The choice of the duration has high impact on the testing performance. Is it better to have many quick simulations, or fewer longer ones? This is conceptually similar to the choice of test length in test data generation of object-oriented software. In this paper, we choose a fixed duration based on the properties the models (e.g., if there are time transitions that take 10 seconds, then we should have test cases running for at least 10 seconds, otherwise those transitions will never be taken).

To calculate the distance between two test data matrices m_1 and m_2 for ART we use the function $dis(m_1, m_2) = \sum_r \sum_c abs(m_1[r,c] - m_2[r,c]) / |D(r)|$, where r and c represent the rows and columns of the matrices. In other words, 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 for the values of non-deterministic choices. The results of the first experiments we conducted showed that RT/ART perform better than SBT [1].

For search-based testing, rather than using a fitness function, we use an *order* function. An order function is used to determine whether one solution is better than another, without having the problem of defining a precise numerical score (this is often difficult when several objectives need to be combined and tight budget constraints do not allow a full multi-objective approach). The new order function *h* can be seen as an extension of the fitness function developed for model-based testing based on system specifications [17]. The original fitness function uses the so-called "approach level" and normalized "branch distance" to evaluate the fitness of a test case. For environment model-based testing, we introduced the concept of "time distance" with a look-ahead branch distance and the concept of "time in risky states" [8].

In our context, the goal is to minimize the order function *h*, which heuristically evaluates how far a test case is from reaching an error state. If a test case with test data *m* is executed and an error state of the environment model is reached, then $h(m) = 0$. The approach level (*A*) refers to the minimum number of transitions in the state machine that are required to reach the error state from the closest executed state. Fig. 1 shows a dummy example state machine to elaborate the concept. The state named *Error* is the error state. Events *e1, e2,* and *e3* are signal events, whereas events *after "t, s"*, *after "t1, ms"*, and *after "t2, ms"* are time events with *t*, *t1*, and *t2* as the time values and *ms* and *s* as time units referring to milliseconds and seconds. Events *e3* and *after "t, s"* are guarded by constraints using OCL. If the desired state is Error and the closest executed state was State5, then the approach level is 1.

The approach level is helpful to reward test case executions that get closer to an error state, but it does not provide any gradient (guidance) to solve the possible guards on the state transitions. The branch distance (*B*) is used to heuristically score the evaluation of the guards on the outgoing transitions from the closest executed state. In previous work [18], we have defined a specific branch distance function for OCL expressions that is reused here for calculating the branch distance. In the dummy state machine in Fig. 1 we need to solve the guard *"y > 0"* so that whenever *e4* is triggered, then the simulation can transition to *Error*. Note that branch distance is less important than approach level, since it is required only when the transition towards an error state is guarded and the approach level cannot be reduced any further.

The third important part of the order function is the time distance (*T*), which comes into play when there are timeout transitions in the environment models. For example, in Fig. 1, the transition from *State2* to *Error* is a timeout transition. If a transition should be taken after *z* time units, but it is not, we calculate the maximum consecutive time *c* the component stayed in the source state of this transition (e.g., *State2* in the dummy example). To guide the search, we use the following heuristic: $T = z - c$, where $c \leq z$. For transitions other than time transitions, we initially decided to calculate branch distance after an event is triggered. As investigated in our earlier work [8], this is not suitable for time transitions and therefore the concept of a look-ahead branch distance *(LB)* was introduced. *LB* represents the branch distance of OCL guard on a time transition when it is not fired (i.e., the timeout did not occur). Because OCL evaluations are free from side-effects [18], this approach is feasible in our context.

The fourth important part of the order function is "time in risky states" *(TIR)*. *TIR* favors the test cases that spent more time in the state adjacent to the error state (i.e., the *risky* state). The motivation behind this heuristic is that, the more time spent in a risky state, the higher the chances of events happening in the environment or SUT that lead to the error state (e.g., receive a signal from the SUT). For example, for the state machine shown in Fig. 1, this heuristic will favor the test cases that spend more time in the risky states *State2* or *State5*. For instance in *State2,* it is possible to increase the value of *t1* in the time event *after "t1, ms"*, which will increase the time spent in this state. *TIR* is less important than the other three heuristics and is only used when the other heuristics fail to guide the search. The order function *h* using the four previously described heuristics, given two test data matrices m_1 and m_2 as input, is defined as:

$$\boldsymbol{h(m_1,m_2)} = \begin{cases} \boldsymbol{v(m_1, m_2)} & \textbf{if } v(m_1, m_2) \mathrel{!=} 0 \\ \mathbf{1} & \textbf{if } v(m_1, m_2) = 0 \textbf{ and } \mathrm{TIR}_{\mathrm{sum}}(m_1) > \mathrm{TIR}_{\mathrm{sum}}(m_2) \\ \mathbf{0} & \textbf{if } v(m_1, m_2) = 0 \textbf{ and } \mathrm{TIR}_{\mathrm{sum}}(m_1) = \mathrm{TIR}_{\mathrm{sum}}(m_2) \\ \mathbf{-1} & \textbf{otherwise} \end{cases} \quad (1)$$

$$v(m_1,m_2)= \begin{cases} \mathbf{1} & \textbf{if } A_{\min}(m_1)<A_{\min}(m_2) \textbf{ or } (A_{\min}(m_1)=A_{\min}(m_2) \text{ and } B_{\min}(m_1) < B_{\min}(m_2)) \\ & \textbf{or } (A_{\min}(m_1)=A_{\min}(m2) \text{ and } B_{\min}(m_1)= B_{\min}(m_2) \text{ and } ITD_{\min}(m_1,m_2) = 1) \\ \mathbf{0} & \textbf{if } A_{\min}(m_1)=A_{\min}(m_2) \textbf{ and } B_{\min}(m_1)= B_{\min}(m_2) \textbf{ and } ITD_{\min}(m_1, m_2)=0) \\ \mathbf{-1} & \textbf{otherwise} \end{cases}$$

$$ITD_e(m_1,m_2) = \begin{cases} \mathbf{1} & \textbf{if } LB_e(m_1) < LB_e(m_2) \textbf{ or } (LB_e(m_1) = LB_e(m_2) \text{ and } T_e(m_1) < T_e(m_2)) \\ \mathbf{0} & \textbf{if } (LB_e(m_1) = LB_e(m_2) \text{ and } T_e(m_1) = T_e(m_2)) \\ \mathbf{-1} & \textbf{otherwise} \end{cases}$$

where for an error state e, $A_{min}(m)$ represents the minimum approach level over all error states, $B_{min}(m)$ represents the minimum branch distance, T_e represents the time distance, LB_e is the look-ahead branch distance for an error state e, and $TIR_{sum}(m)$ is the sum of time spent in risky states for all error states for test data matrix m.

The results, based on our extensive experiments evaluating various heuristics [8], suggested that (1+1) EA with the order function in (1) gave best results in cases where the approach to a risky state was non-trivial (i.e., simulation cannot reach a risky state in <5 random test cases). But in cases where the approach was easy, RT outperformed evolutionary algorithms.

```
Algorithm   HybridStrategy(mx, n, w)
Input       mx: number of maximum fitness evaluations
            n: number of consecutive test cases with no improved fitness
            w: number of random test-cases to generate for comparison in ART
Declare     Y: set of executed test cases = {}, W: set of randomly generated test cases = {}
            ev: number of fitness evaluations performed = 0
            z: number of consecutive test cases with no improved fitness found so far = 0
            $T_c$: a random test case, $T_m$: mutated test case, $T_w$: a test case from W, $T_e$ test case
            from W selected according to ART criteria
            $D_w$: minimum distance of test case $T_w$ with all the test cases in Y
            d: stores the maximum value of $D_w$ obtained over W
1.  begin
2.    Generate a random test case $T_c$
3.    Execute $T_c$ and evaluate whether environment error state is reached
4.    Add $T_c$ to Y
5.    while environment error state not reached OR  ev <= mx OR z <= n
6.        Mutate $T_c$ to get $T_m$
7.        Execute $T_m$ and evaluate whether environment error state is reached
8.        Add $T_m$ to Y
9.        Increment ev
10.       if fitness($T_m$) >= fitness($T_c$)
11.         then $T_c = T_m$ , $z = 1$
12.       else
13.         Increment z
14.   while environment error state not reached OR ev <= mx
15.       Sample w random test cases and add them to W
16.       $d = 0$
17.       for each $T_w \in W$
18.         Calculate $D_w$
19.         if $D_w > d$
20.            then $d = D_w$, $T_e = T_w$
21.       Execute $T_e$ and evaluate whether environment error state is reached
22.       Add $T_e$ to Y
23.       Increment ev
24. end
```

Fig. 2 Pseudo code of the proposed hybrid strategy (HS)

4. Hybrid Strategy by Combining Adaptive Random and Search-based Testing

In this section we present our proposed hybrid strategy (HS) that combines (1+1) EA and ART to improve the overall fault detection effectiveness of our system testing approach. As discussed earlier (Section 3), previous studies showed that, in some cases, RT/ART could perform better than SBT. The difference between their performances was mostly significant and at times even extreme. In [2] and [8], we identified two possible reasons for this behavior. First of all, for the problems with high failure rates, randomized algorithms were found to be much better than SBT [2]. For high failure rates, there is no need for search, as solutions are anyway found quickly. Crossover produces similar genes, while mutation only performs small modifications. This can have a negative effect as, given just few fitness evaluations, only similar solutions are evaluated (in contrast to RT/ART). Secondly, the performance of the algorithms also depended on the properties of environment models, and in particular how easy is it to traverse the models in order to reach the error states. In other words, by combining ART and (1+1) EA, we hope to achieve is a consistently good result regardless of the properties of the SUT or its environment.

In the environment models, there are transitions on paths leading to error states that depend only on the behavior of the SUT (i.e., they can only be triggered when the SUT behaves in a certain way). Transition from a risky state to an error state is one such example as it is only triggered when the SUT behaves in an erroneous way. Another example can be when a guard on a transition depends on a specific response from the SUT. To execute this behavior of SUT, the overall environment (combination of environment components) needs to behave in a particular way. This particular behavior of the environment that is required to trigger SUT behavior cannot be determined before simulation, since for practical reasons discussed earlier the design of the SUT is not visible. Hence, the information of what should be executed in the environment to trigger this behavior is not available in the environment models. The fitness function for SBT (which exploits the environment models to guide the search towards error states) in this case does not give enough gradient to generate fitter test cases (i.e., a search plateau). In these cases maximizing the diversity of the environment behavior (e.g., by using entirely different values for the test data matrix, irrespective of their effect on the fitness) appears to be a better option, thus favoring RT/ART. This can explain the scenarios where RT/ART show better results than (1+1) EA.

On the other hand, if in the environment models, there are transitions on the path to error states which are triggered by specific behaviors of the environment (e.g., a transition triggered as a result of a specific non-deterministic event in the environment, such as a failure of an environment component) or time transitions, then fitness function for SBT is specifically designed to deal with these cases and are more suitable for such cases than RT. For example, in the fitness function, the time distance heuristic is defined specifically for time transitions and favors test cases that are closer to executing the transitions (i.e., with a value of c closer to z, see Section 3.2). OCL constraints in guards that are independent of SUT behavior but dependent on the state of environment components (e.g., a constraint requiring a sensor to be broken), can be solved by directly changing the values of these components' attributes. For such constraints, our previous results showed that SBT are an order of magnitude better than RT [18].

HS combines ART, which showed best results in our initial experiments [2], with our proposed SBT strategy that showed best performance [8], i.e., (1+1) EA with improved time distance and the "time in risky state" heuristic (ITD-TIR). The strategy is designed to combine the strengths of both (1+1) EA and ART. This strategy starts by applying (1+1) EA. If (1+1) EA does not find fitter test cases after running *n* number of test cases, the testing algorithm is switched to ART. All the test cases that were executed so far are now used for distance calculations in ART. Fig. 2 shows the pseudo-code for HS. The idea behind switching from (1+1) EA to ART is that there is not enough time for a random walk to get out of a fitness plateau. And so, in this scenario, applying ART can yield better results. Running system test cases is very time consuming, so only few fitness evaluations are feasible within reasonable time (e.g., 1000 test cases can already take several hours). Therefore, in case of fitness plateau, it is reasonable to switch strategy, and rather reward diversity instead of the fitness value. Though the choice of *n* is arbitrary it can have significant consequences on the performance of this strategy. A too small value of *n* will result in an early switch to ART. If the given problem matches the case where (1+1) EA performs better, then the performance of HS will be affected. Similarly, if *n* is too large then the remaining testing budget might not be sufficient for ART to perform well.

5. Empirical Study

The objective of this empirical study is to evaluate the fault detection effectiveness of the proposed hybrid strategy.

5.1. Case Study

To enable experimentation with diverse environment models and RTES, we developed 13 different artificial RTES that were inspired by two industrial cases we have been involved with [2] and one case study discussed in the literature [19]. Since, there are no benchmark RTES available to researchers, we specifically designed these artificial problems to conduct our experiments (called P1 – P13). The goal while developing the models of these RTES was to vary various characteristics of the environment models (e.g., guarded time transitions, loops) that were expected to have an impact on the test heuristics. Nine of these artificial problems were inspired by a marine seismic acquisition system developed by one of our industrial partners. They covered various subsets of the environment of that RTES. Three problems were inspired by the behavior of another industrial RTES (automated recycling machine) developed by another industrial partner. The thirteenth artificial problem was inspired by the train control gate system described in [19].

These RTES are multithreaded, written in Java and they communicate with their environments through TCP. Each of the artificial problems had one error state in their environment models and non-trivial faults were introduced by hand in each of them. We could have rather seeded the faults in a systematic way, as for example by using a mutation testing tool [20] but opted for a different procedure since the SUTs are highly multi-threaded and use a high number of network features (e.g., opening and reading/writing from TCP sockets), features that are not handled by current mutation testing tools. Furthermore, our testing is taking place at the system level, and though small modifications made by a mutation testing tool might be representative of faults at the unit level, it is unlikely to be the case at the system level for RTES. On the other hand, the faults that we manually seeded came from our experience with the industrial RTES and from the feedback of our industry partners.

The industrial case study we also report on (called IC) is a very large and complex seismic acquisition system (mentioned above) that interacts with many sensors and actuators. The timing deadlines on the environment are in the order of hundreds of milliseconds. The company that provided the system is a market leader in its field. For confidentiality reasons we cannot provide full details of the system. The SUT consists of two processes running in parallel, requiring a high performance, dedicated machine to run. For the industrial case study, we did not seed any fault and the goal was to find the real fault that we uncovered earlier [1].

5.2. Experiment

In this paper, we want to answer the following research questions:
RQ1. Which configuration is best in terms of fault detection for the proposed hybrid strategy (HS)?
RQ2. How the fault detection of the best HS configuration compares with the performance of ART, (1+1) EA, and RT for (a) the artificial problems (P1-13) and (b) the industrial case study (IC)?

To answer these research questions, we have conducted two distinct sets of experiments, one for the artificial problems (to answer RQ1 and RQ2a) and one for the industrial RTES (to answer RQ2b). For test case representation in these experiments we used a dynamic representation with a length equal to 10 for the test cases (which correspond to each row of the test data matrix m). In our earlier experiments this setting showed the best results [2]. For (1+1) EA we calculated the mutation rate as 1/*k*, where *k* is the number of total elements in a test data matrix. This strategy is widely used for SBT and was initially suggested in [21]. We used the fitness function that performed best in our previous experiments [8], as discussed in Section 4: Improved Time Distance with Time in Risky State (ITD-TIR).

Table 1. Success Rates (SR) for 12 configurations of HS on the 13 problems

Configurations → Problems ↓	**10**	**20**	**50**	**60**	**70**	**80**	**90**	**100**	**200**	**300**	**400**	**500**
P1	0.5	0.75	0.95	1	1	1	1	1	1	1	1	1
P2	0.85	0.95	1	1	1	1	1	1	0.9	1	1	1
P3	1	1	1	1	1	1	1	1	0.9	0.8	0.6	0.5
P4	0.05	0.2	0.8	0.85	0.7	0.75	0.9	0.9	1	1	0.9	1
P5	0.85	1	1	1	1	1	1	1	1	1	1	1
P6	0	0.15	0.45	0.4	0.45	0.5	0.45	0.6	0.7	0.7	0.5	0.6
P7	0.3	0.4	0.8	0.8	0.85	0.95	0.8	0.8	0.8	0.8	0.8	1
P8	1	1	1	1	1	1	1	1	1	1	0.95	1
P9	0.05	0.05	0.45	0.55	0.55	0.35	0.6	0.4	0.8	0.45	0.5	0.55
P10	1	1	1	1	0.95	0.85	1	0.95	0.65	0.55	0.4	0.45
P11	1	1	1	0.95	0.95	0.9	1	0.9	0.65	0.05	0.1	0.4
P12	1	1	1	1	0.95	1	1	1	0.9	0.9	0.75	0.65
P13	1	1	1	1	1	1	1	1	0.9	0.7	0.95	0.85
Average SR	0.66	0.73	0.88	0.89	0.88	0.87	0.9	0.89	0.86	0.77	0.73	0.77
Average Rank	6.38	6.73	5.19	5.77	5.23	6.31	6.50	6.19	6.73	8.46	7.73	6.69

To answer RQ1, we used 12 different values for the number of test cases which fitness should be considered before switching from (1+1) EA to ART: $n \in \{10, 20, 50, 60, 70, 80, 90, 100, 200, 300, 400, 500\}$. We ran these 12 configurations on each of the 13 artificial problems. To answer RQ2a, we selected the configuration of HS that gave the best result in terms of fault detection for the 13 artificial problems. We compared this configuration with the results of (1+1) EA, ART, and RT on these problems. RT was used as a comparison baseline.

For the artificial problems, the execution time of each test case was fixed to 10 seconds and we stopped each algorithm after 1000 sampled test cases or as soon as we reached any of the error states. The choice of running each test case for 10 seconds was based on the properties of the RTES and the environment models. The objective was to allow enough time for the test cases to reach an error state. We ran each of the strategies 20 times on each artificial problem with different random seeds. The total number of sampled test cases was 1,561,390, which required around 180 days of CPU resources. Therefore, we performed these experiments on a cluster of computers.

To answer RQ2b, we carried out experiments on the described seismic acquisition system. We run each test case for 60 seconds, where 1000 test case executions (fitness evaluations) can take more than 16 hours. This choice has been made based on the properties of the RTES and discussions with the actual testers. Due to the large amount of resources required, we only ran the configuration that on average gave best results for the artificial problems (i.e., $n=50$) and compared its fault detection rate with that of (1+1) EA, ART, and RT. We carried out 39 runs for each of these four test strategies. The total number of sampled test cases was 55,283, which required over 55 days of computation on a single, high-performance, dedicated machine.

To analyze the results, we used the guidelines described in [22] which recommends a number of statistical procedures to assess randomized algorithms. First we calculated the success rates of each algorithm: the number of times it was successful in reaching the error state out of the total number of runs. These success rates are then compared using the Fisher Exact test, quantifying the effect size using an odds ratio (ψ) with a 0.5 correction. When the differences between the success rates of two algorithms were not significant, we then looked at the average number of test cases that each of the algorithms executed to reach the error state. We used the Mann-Whitney U-test and quantified the effect size with the Vargha-Delaney A_{12} statistics. The significance level for these statistical tests was set to 0.05.

5.3. Results & Discussion

Table 1 provides the success rates (in terms of fault detection) for various HS configurations. The last row of the table shows the average ranking of each configuration based on the statistical differences among them. Configurations that are statistically equivalent (i.e., p-values above 0.05) are assigned a similar ranking. This is done by assigning scores based on pairwise comparisons of configurations. Whenever a configuration is better than the other and the difference is statistically significant, its score is increased (for details, see [22]). Then, based on the final scores, each configuration is assigned ranks ranging from 1 (best configuration) to 12 (worst configuration). In case of ties, ranks are averaged. As the success rates and average rankings indicate, using a very low (< 50) or very high value (>=200) of n results in a degraded performance for HS. With a low value of n, HS makes the switch from (1+1) EA to ART too early, which does not give sufficient time for (1+1) EA to converge and hence running HS becomes similar to only running ART. In cases where ART performs well, such configurations of HS also perform well (see Table 2 for the performance of ART on artificial problems). For instance, for $n = 10$, the average success rate is 66% and average ranking is 6.38. Similarly, when HS switches too late, it does not give enough time to ART (given the upper bound of 1000 iterations) and hence running HS is similar to running (1+1) EA in such cases. These configurations perform well in cases where (1+1) EA performs well (Table 2) and poor otherwise. The best results are provided for values between *50* and *100* and the differences in results in this range are not significant. Though the results are not fully consistent across all problems, configuration $n = 50$ has the best average rank across all problems and is always very close to the maximum success rates. We can hence answer RQ1 by stating that, overall, $n=50$ shows the best results for HS and therefore this configuration can be used when applying HS on new problems.

For RQ2a we compared the best HS configuration ($n = 50$) with RT, ART, and (1+1)EA. Table 2 shows the corresponding success rates of these algorithms and Table 3 shows a comparison of HS with the other three algorithms based on statistical tests. The statistics for the situations where HS is significantly better are bold-faced and are italicized where it is significantly worse. Table cells with a '-' denote no significant differences. P-values obtained as a result of Fisher Exact test on the success rates are denoted as p and odds ratio as ψ. In cases where there is no statistical difference in success rates, the number of iterations is considered and the p-values of the Mann-Whitney U-test are denoted as *it-p* and corresponding effect sizes by A_{12}.

When compared to (1+1) EA, HS showed better fault detection performance in four of the artificial problems (P3, P10 – P12) and had similar results otherwise. These are the problems where (1+1) EA, when ran in isolation, showed poor results when compared to RT and ART (as visible from Table 2). For example in the case of P11, (1+1) EA was not able to find the a in any of the runs. On the other hand it is 100% for HS, RT, and ART, which means that these strategies were able to find a fault in every run. Hence, HS shows significant improvement over (1+1) EA.

When compared to RT, HS showed significantly better results in terms of success rates for six artificial problems (P1, P4, P5, P6, P7, and P9) and had similar results for all the other problems. Similarly with ART, in terms of success rates, HS showed better results for six artificial problems (P1, P2, P4, P6, P7, and P9) and had similar results for the rest. P1, P4, P6, P7, and P9 are the problems where ART and RT showed poor results when compared to (1+1) EA (Table 2). For example in the cases of P4, P6, and P9, the success rate of both RT and ART is 0, but that of (1+1) EA and HS is 1 and 0.8, respectively. Hence, in terms of success rates, HS shows significantly better results when compared to RT and ART. However, in terms of number of iterations required to detect the fault, HS is significantly worse than RT in four problems (P8, P10, P12, and P13) and significantly worse than ART in six problems (P3, P8, P10, P11, P12, and P13). But, for all these problems, the success rate of HS, RT, and ART is 1, which means that whenever these algorithms run they find the fault (within the budget of 1000 test cases). Therefore, we can answer RQ2a by stating that HS shows overall significantly better performance than ART, RT, and (1+1) EA in terms of fault detection, but was slower than RT/ART in finding faults for problems where these two algorithms perform better than (1+1) EA. But since the success rate of HS is 100%, and therefore the first run is expected to reach the error state, this difference in execution time has limited practical impact.

Table 2. Success Rates of HS (Best configuration), RT, ART, and (1+1) EA

	P1	**P2**	**P3**	**P4**	**P5**	**P6**	**P7**	**P8**	**P9**	**P10**	**P11**	**P12**	**P13**	**Avg.**	**IC**
HS	0.95	1	1	0.8	1	0.45	0.8	1	0.45	1	1	1	1	0.88	1
ART	0.4	0.75	1	0	0.95	0	0.15	1	0	1	1	1	1	0.63	1
EA	1	1	0.5	1	1	0.7	0.85	1	0.35	0.45	0	0.7	0.95	0.73	0.74
RT	0.45	1	1	0	0.65	0	0.2	1	0	1	1	1	1	0.64	0.97

Table 3. Comparison of best HS configuration with RT, ART, & (1+1)EA*

Problem	**HS vs. (1+1) EA**	**HS vs. RT**	**HS vs. ART**
P1	-	**p = 0.0012, ψ =15.74**	**p = 0.0004, ψ =19.12**
P2	-	**it-p = 0.0065, A_{12} = 0.25**	**p = 0.047, ψ =14.55**
P3	**p = 0.0004, ψ = 41.00**	-	*it-p = 0.013, A_{12} = 0.73*
P4	-	**p = 1.5e-07, ψ = 150.33**	**p = 1.5e-07, ψ = 150.33**
P5	-	**p = 0.0083, ψ = 22.78**	-
P6	-	**p = 0.0012, ψ = 33.87**	**p = 0.0012, ψ = 33.87**
P7	-	**p = 0.0004, ψ = 13.44**	**p = 8.7e-05, ψ = 18.33**
P8	-	*it-p = 0.009, A_{12} = 0.74*	*it-p = 0.0004, A_{12} = 0.825*
P9	-	**p = 0.0012, ψ = 33.87**	**p = 0.0012, ψ = 33.87**
P10	**p = 0.0001, ψ = 49.63**	*it-p = 0.0006, A_{12} = 0.81*	*it-p = 0.0002, A_{12} = 0.85*
P11	**p = 1.4e-11, ψ = 1681.00**	-	*it-p = 0.0032, A_{12} = 0.77*
P12	**p = 0.02, ψ = 18.38**	*it-p = 0.0016, A_{12} = 0.79*	*it-p = 0.0008, A_{12} = 0.81*
P13	-	*it-p = 0.0199, A_{12} = 0.71*	*it-p = 0.021, A_{12} = 0.71*
IC	**p = 0.0004, ψ = 28.83**	-	*it-p = 0.015, A_{12} = 0.66*

For RQ2b we compared the performance of the best configuration of HS (*n = 50*) with that of ART, RT, and (1+1) EA on the industrial case study. The last row of Table 3 shows a comparison of the results of the four strategies on this case study (IC) and the last column of Table 2 shows the corresponding success rates. The results are similar to that obtained for those artificial problems where RT and ART perform better than (1+1) EA. HS outperformed (1+1) EA. When compared with the results of ART and RT, there is no significant difference though (100% success rate). These results are consistent with RQ2a and we can therefore answer RQ2 by stating that, overall, HS shows significantly better results when compared to (1+1) EA, RT, and ART. However, as for RQ2a, for problems where ART performed much better than (1+1) EA, though the success rates of HS and ART are similar, ART find the faults faster than HS.

HS starts with (1+1) EA and switches only when *fifty* consecutive test cases do not show better fitness. Fitness evaluations make HS slower than ART/RT but its effectiveness considerably improves over ART/RT for the problems where they showed poor results. In the light of these results, we can conclude that when applying our testing approach, using HS seems to be the most practical choice as its performance, unlike that of (1+1) EA, ART, and RT, is not drastically affected by the properties of the SUT and its environment models. As a result, testers can apply this strategy in confidence, knowing it will perform well in most circumstances.

5.4. Threats to Validity

Although the artificial problems that we developed were based on industrial RTES and are not trivial to test (they are multithreaded and hundreds of lines long), these artificial problems may not be representative of complex RTES. To reduce this threat, we used artificial problems inspired by actual RTES and intentionally varied the properties of their environments in ways that could affect the testing strategies.

A typical problem when testing RTES is the accurate simulation of time. Our approach focuses on RTES with soft time deadlines in the order of hundreds of milliseconds with an acceptable jitter of a few milliseconds. Therefore, we used the CPU clock to represent time. This might be unreliable if time constraints in the RTES were very tight (e.g., nanoseconds) since they could be violated due to unpredictable changes of load balance in the CPU in the presence of unrelated process executions. To be on the safe side, to evaluate whether our results are reliable, we selected a set of experiments and ran them again with exactly the same random seeds. We obtained equivalent results with a small variance of a few milliseconds, which in our context did not affect the testing results.

6. Conclusion

In this paper, we proposed a hybrid strategy (HS) that combines (1+1) Evolutionary Algorithm (EA) and Adaptive Random Testing (ART) for black-box automated system testing of real-time embedded systems (RTES). The strategy was developed to combine the benefits of both algorithms, since their individual results varied greatly depending on the failure rate of the system under test and properties of its environment. The ultimate goal was to obtain a strategy with consistently good results. The proposed strategy starts with running (1+1) EA and switches to ART when the (1+1) EA search stops yielding fitter test cases. We empirically investigated when to switch to ART and identified an optimal setting for HS. Results indicate that switching too early or too late than the identified setting has a negative impact on the performance of the strategy. Based on the experiments, when using HS in practice, we propose switching to ART after (1+1) EA generates 50 consecutive test cases that do not improve fitness. We evaluated the proposed strategy and compared its performance with that of running (1+1) EA and ART individually. We also use random testing (RT) as a comparison baseline. The empirical evaluation uses an industrial case study and 13 artificial problems that were developed based on two industrial case studies belonging to different domains. The models of these artificial problems were developed in order to vary their characteristics, thus potentially affecting the performance of the evaluated testing strategies. Overall, the results indicate that HS shows significantly better performance in terms of fault detection (an overall 88% success rate for artificial problems and 100% for the industrial case study) than the other three algorithms (for artificial problems: ART: 63%, RT: 64%, and (1+1) EA: 74% and for the industrial case study: ART: 100%, RT, 97%, (1+1) EA: 74%). Unlike the other strategies, variations in environment properties do not have a drastic impact on the performance of HS and it is therefore the most practical approach, showing consistently good results for different problems.

References

1. Arcuri, A., Iqbal, M., Briand, L.: Black-Box System Testing of Real-Time Embedded Systems Using Random and Search-Based Testing. In: Petrenko, A., Simão, A., Maldonado, J. (eds.) Testing Software and Systems, vol. 6435, pp. 95-110. Springer Berlin (2010)

2. Iqbal, M.Z., Arcuri, A., Briand, L.: Automated System Testing of Real-Time Embedded Systems Based on Environment Models. Simula Research Laboratory, Technical Report (2011-19) (2011)
3. OMG: Unified Modeling Language Superstructure, Version 2.3, http://www.omg.org/spec/UML/2.3/. (2010)
4. OMG: Modeling and Analysis of Real-time and Embedded systems (MARTE), Version 1.0, http://www.omg.org/spec/MARTE/1.0/. (2009)
5. 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: Petriu, D., Rouquette, N., Haugen, Ø. (eds.) Model Driven Engineering Languages and Systems, vol. 6394, pp. 286-300. Springer Berlin / Heidelberg (2010)
6. Iqbal, M.Z., Arcuri, A., Briand, L.: Code Generation from UML/MARTE/OCL Environment Models to Support Automated System Testing of Real-Time Embedded Software. Simula Research Laboratory, Technical Report (2011-04) (2011)
7. Arcuri, A., Iqbal, M.Z., Briand, L.: Random Testing: Theoretical Results and Practical Implications. IEEE Transactions on Software Engineering (2011)
8. Iqbal, M.Z., Arcuri, A., Briand, L.: Empirical Investigation of Search Algorithms for Environment Model-Based Testing of Real-Time Embedded Software In: International Symposium on Software Testing and Analysis (ISSTA). ACM (2012)
9. Broekman, B.M., Notenboom, E.: Testing Embedded Software. Addison-Wesley.. (2003)
10. Auguston, M., B, M.J., Shing, M.: Environment behavior models for automation of testing and assessment of system safety. Information and Software Technology 48, 971-980 (2006)
11. Heisel, M., Hatebur, D., Santen, T., Seifert, D.: Testing Against Requirements Using UML Environment Models. In: Fachgruppentreffen Requirements Engineering und Test, Analyse & Verifikation, pp. 28-31. GI (2008)
12. Adjir, N., Saqui-Sannes, P., Rahmouni, K.M.: Testing Real-Time Systems Using TINA. Testing of Software and Communication Systems, vol. 5826. Lecture Notes in Computer Science, Springer Berlin / Heidelberg (2009)
13. Larsen, K.G., Mikucionis, M., Nielsen, B.: Online Testing of Real-time Systems Using Uppaal. Formal Approaches to Software Testing, vol. 3395. Lecture Notes in Computer Science, Springer Berlin / Heidelberg (2005)
14. Andrews, J.H., Menzies, T., Li, F.C.H.: Genetic algorithms for randomized unit testing. IEEE Transactions on Software Engineering 37, 80-94 (2011)
15. Tappenden, A.F., Miller, J.: A novel evolutionary approach for adaptive random testing. IEEE Transactions on Reliability 58, 619-633 (2009)
16. Schneckenburger, C., Schweiggert, F.: Investigating the dimensionality problem of Adaptive Random Testing incorporating a local search technique. In: International Conference on Software Testing Verification and Validation Workshop (ICSTW '08), pp. 241-250. (2008)
17. Lefticaru, R., Ipate, F.: Functional search-based testing from state machines. In: Proceedings of the International Conference on Software Testing, Verification, and Validation, pp. 525-528. IEEE Computer Society (2008)
18. Ali, S., Iqbal, M.Z., Arcuri, A., Briand, L.: A Search-based OCL Constraint Solver for Model-based Test Data Generation. International Conference on Quality Software, pp. 41-50. IEEE (2011)
19. Zheng, M., Alagar, V., Ormandjieva, O.: Automated generation of test suites from formal specifications of real-time reactive systems. The Journal of Systems & Software 81 (2008)
20. Andrews, J., Briand, L., Labiche, Y., Namin, A.: Using mutation analysis for assessing and comparing testing coverage criteria. IEEE Transactions on Software Engineering 32, (2006)
21. Mühlenbein, H.: How genetic algorithms really work: I. mutation and hillclimbing. Parallel problem solving from nature 2, 15-25 (1992)
22. Arcuri, A., Briand, L.: A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering. In: 33rd International Conference on Software Engineering (ICSE), pp. 1 - 10 (2011)