Automated System Testing of Real-Time Embedded Systems Based on Environment Models[1]

Muhammad Zohaib Iqbal, Andrea Arcuri, Lionel Briand

Certus Software V&V Center, Simula Research Laboratory, P.O. Box 134, Lysaker, Norway.
Email: {zohaib,arcuri,briand}@simula.no

Abstract

Testing real-time embedded systems (RTES) is in many ways challenging. Thousands of test cases may need to be 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 in an industrial context. In this paper we take a black-box approach and model the RTES environment using the UML/MARTE international modeling standards. Our main motivation is to provide a more practical approach to the model-based testing of RTES. To do so, we enable system testers, who are often not familiar with the system design but know the application domain well-enough, to model the environment, using well-supported modeling standards, to enable test automation. Environment models can support the automation of three tasks: the code generation of an environment simulator to enable testing on the development platform, 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 four test automation strategies using inputs from UML/MARTE environment models: Random Testing (baseline), Adaptive Random Testing, and Search-Based Testing (using Genetic Algorithms and (1+1) Evolutionary Algorithm). Results on artificial problems and one industrial case study show that Adaptive Random Testing performs best and, more importantly, that our automated testing framework is effective in finding faults.

Keyword: 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 [24]. 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 a large numbers of sensors and actuators. The interactions with the environment are usually bound by time constraints. For example, in a railway crossing, if the RTES of the gate controller 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 its resulting behavior.

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 test case and oracle generation without any knowledge about the internal design of the RTES. This is typically a practical requirement for independent system test teams in industrial settings. In addition, the testing 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

[1]This paper is an extension of a conference paper [14] published in the IFIP International Conference on Testing Software and Systems (ICTSS), 2010.

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. A simulator enables the execution of the RTES on the development platform, without requiring actual interactions with its environment. In our context, a test case is a sequence of stimuli, generated by the environment or its simulator, that are 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 this 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.

Testing all possible sequences of environment stimuli and state changes is not feasible. In practice, a single test case of an industrial RTES could last several seconds or even minutes, executing hundreds of thousands of lines of code, generating hundreds of threads and processes running concurrently, communicating through TCP sockets and operating system 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) [53] 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) [21] and Search-Based Testing (SBT) [41] (specifically, Genetic Algorithm and (1+1) Evolutionary Algorithm) to our specific problem and context. For our empirical evaluation, we use Random Testing (RT) [44] as a baseline of comparison. One main advantage of ART and SBT is that they 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. Furthermore, ART and SBT attempt, through different heuristics, to maximize the chances to trigger a failure within time constraints. 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 in industrial settings. 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, i.e., the RTES runs in a way that is identical to what would be observable in the actual environment.
- 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 automated 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. Because our focus in this paper is test automation, we only briefly discuss the use of

UML/MARTE for environment modeling of a RTES and automated generation of simulator code from these models (which we investigated in [34] and [33]). To date, no MBT automation results for ART and SBT on an industrial RTES, using international modeling standards, have ever been reported in the research literature. Since no freely available RTES was available, we also constructed four 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, the industrial case study of our industrial partner cannot be made public.

In our context, a test case is a setting of the environment simulator. It represents the possible values for various choices that can be taken during the simulation. These choices are for example indicating when hardware components should break, and what type of actions the users perform and when they occur. During simulation each of these choices can be taken several times. For example, consider a scenario where an environment component can loose its connection with the SUT during simulation. The test cases will decide when this scenario occurs (i.e., they will provide values for this choice). Assume that the value for this choice provided by a test case being executed is 2 seconds. This means that the component will loose the connection in 2 seconds after being connected. When a connection is lost, the component tries to reconnect to the SUT. After it reconnects, it can again loose a connection (based on the same choice). Again this value is decided by the test cases. Based on how many values a test case contains for this choice, the next occurrence will be decided. If the test case being executed only contained one value for this choice, then the value of 2 seconds will be used again. The decision of how many possible values for each choice should be there in test cases can have a significant impact on the fault detection effectiveness of testing strategies, as we will be investigating in this paper. We empirically evaluate the significance of using different number of these possible values and the representation of the test cases. We use two types of representation of test cases (ring and dynamic variable size) and evaluate their effect on the various testing strategies. In total, the empirical study on the four artificial problems took 653 days of CPU usage. Even by using a large cluster of computers, it took several days to carry it out. The experiments on the industrial RTES required a specialized machine, and could not be run on that cluster. It took more than 55 days to run those experiments. Although we only considered five RTES, of which only one was industrial, such empirical study required a very large amount of time and **??**sources. This is a general problem in system testing, where running test cases can be several orders more expensive than for example unit test cases.

As previously mentioned, because the approach described in this paper is fully automated but requires the development of environment models, one might question whether such an approach makes economical sense. Though modeling cost is far from negligible, the models are made only once and only need to be modified as needed. Every time there is a change in the RTES specification, this would typically result in a change in these models, but all the remaining tasks would be fully automated: the modification of the environment simulator, the choice of adequate test cases and the evaluation of their results. Our conjecture is that this one-time cost of creating the models is significantly less than manual test and oracle generation without models. Although controlled experiments in industrial contexts will be necessary to support this claim, note that an environment simulator is needed to support test execution regardless of whether test and oracle generation is automated or not.

The paper is organized as follows. Section 2 provides an overview of related work. How the environment of RTES is modeled and simulated is briefly 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 the automated testing of RTES. Most of these approaches are based on violating their timing constraints [22] or checking their conformance to a specification [36]. The specification is generally a formal model of the system and this model is then used to generate the test cases. Our approach is based on generating test cases based on environment models, rather than a model specifying the SUT. Still, in order to review the literature in RTES testing, we discuss some of the important works based on system modeling.

A number of approaches use Timed Automata [6] or its extensions to model the system specifications.

Neilsen *et al.* [45] proposed an approach for conformance testing of realtime system based on timed automata specifications. They provide a coverage criterion that covers the time domain of the SUT specification. The approach is applied to a specification of an artificial problem for test case generation. An evaluation of test suite size and the time and space required for test case generation for the artificial problem are also provided. Another approach based on timed automata for conformance testing is discussed in [36]. The work presents coverage criteria for the input automaton to generate test cases. The approach is demonstrated to generate test cases for two artificial specifications, one of which is of a network protocol. Timed Input Output Automata (TIOA) is an extension of timed automata that has also been used for test case generation (e.g., [50] [26] [17]). Most of these works are only applied to small examples to show how the tests are generated.

Similar to TIOA, another extension of timed automata, UPPAAL timed automata have also been discussed in literature regarding their application to testing. Among them, the work in [32] and [38] discuss an approach for conformance testing of RTES based on their specifications and environment assumptions. They propose an approach for generating test cases on the fly during testing. The specifications are modeled using UPPAAL timed automata.

Other approaches for model based testing for real-time systems typically use finite state-machines, UML state-machines or their extensions for modeling the specifications. An approach that uses UML statechart for specifying the SUT and generate test cases based on it was proposed in [43]. The statechart is converted to UPPAAL timed automata. Test cases are generated using an existing model-checker for UPPAAL. Merayo *et al.* [42] extend Finite State Machines (FSM) for modeling action durations and then use these models for testing timeouts and action durations. The approach is applied to simple examples in order to demonstrate the working of the approach. The work in [58] presents an approach to derive test cases based on FSM and their composition containing time outs. The approach is applied on the specification of a simplified behavior of a real life problem to support test case generation.

Search heuristics have been widely applied to software testing (SBT). The goal is to obtain test data that satisfy specified testing goals, such as branch coverage. Surveys of such SBT approaches are provided in [41], [29] and [3]. These search algorithms require the definition of a *fitness function* to evaluate the fitness of the test data with respect to the goals of the search. Based on these fitness evaluations the search is guided to generate new and better test data. Most the works related to SBT in the literature are based on fitness functions that are defined on source code (e.g., for white-box testing at unit level). Only a few approaches apply SBT for state machines and RTES.

Regarding works using SBT techniques for testing different aspects of RTES, examples are regarding identifing deadline misses [28], robustness testing [2], and functional testing [40]. A number of approaches in testing RTES have also focused on finding the best and worst case execution time for RTES. The idea is to exercise the SUT at both these extremes to see whether the SUT violates temporal constraints. Search based algorithms (mostly Genetic Algorithms (GAs) as noted by [1]) have largely been used to obtain inputs that cause the system to run at both these extremes (e.g., [52] and [54]).

Among the approaches that apply search algorithms for testing state machines, the work in [23] discusses test generation for timed extended FSMs based on GAs. The timed EFSM specifies the SUT behavior and temporal constraints. The fitness function is calculated based on the complexity of temporal constraints on the transitions in EFSM. The aim is to generate transition paths that are feasible for the timed EFSM and satisfy certain temporal properties (e.g., for stress testing). The approach is applied on a timed EFSM of a communication protocol to evaluate the test case generation of GA compared to random search. Regarding Simulink/stateflow models, an approach based on an improvement of GA, the *Messy GA*, to achieve the coverage of all transitions is presented in [47] . The fitness function is defined based on the branch distance for the predicates in the guards of the transitions. The approach is applied on three artificial MATLAB benchmark models and is shown to be effective in test generation for transition coverage.

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 have limited knowledge about and access to design information.

Adaptive Random Testing (ART) has been proposed as an improvement to Random Testing (RT) [20]. The assumption behind the approach is that the failing test cases in the SUT are grouped in contiguous regions. In [20], three common failure region distributions for numerical applications are provided, two of which showing that failure regions are typically contiguous (from a geometrical point of view). The idea of the approach is to improve RT in a way that it increases the overall test case diversity. The approach has been suggested to be better than RT for numerical applications in [20] and [21]. A number of extensions to the basic algorithm have also been proposed, ranging from optimized distance calculations (e.g., [37] and [19]) to combining it with evolutionary algorithms [51]. Though a large number of works in the literature apply ART on numerical applications, it has also been applied to other testing domains, such as test case prioritization [35] and model-based test case selection [31]. In [10], an empirical study shows that ART is not effective in several contexts, specially where test case execution time is short or the oracle is not automated.

Among the approaches using environment models (or a similar notion) for testing, an approach to functional testing for embedded systems is presented in [40]. In this approach the tester needs to provide annotated FSMs specifying the search space for test stimuli and a fitness function to assess the outputs of the test cases. The fitness function defines the various possible failure conditions during the test execution. Fitness of the results of each test run is obtained based on this function. The fitness value is then used to generate new test stimuli that are likely to take the SUT closer to violating the failure conditions. The approach is applied to a MATLAB implementation of an industrial case study in the avionics domain. The search space for input signals, provided as FSMs can be considered similar to our concept of environment models. Though, to be more usable in industrial contexts, our approach requires the use of standard modeling notations with which the software engineers are familiar with. Our approach is different from their work as it also simulates the behavior of environment components specified using the state machines. Apart from simulating the normal environment behavior, this also allows us to simulate unfavorable environment situations (like hardware breakdown of environment component). Moreover, the oracle in our case is provided as part of the environment models. The fitness function we use is not only based on the boolean constraints representing failure scenarios, but it also considers the notion of time events, which have a significant impact in testing RTES. The approach presented in [40] is focused on testing of embedded systems and does not consider the temporal constraints on the environment or the SUT.

Peleska *et al.* [48] present a benchmark model for testing RTES in the automotive domain. The proposed testing methodology uses information from environment models and system models to obtain test cases. Our approach is different because we consider the SUT as a black-box, which is a common approach for system level testing where the test engineers are not familiar with the SUT internal design. To the best of our knowledge, the only related work about black-box testing based on environment specifications for RTES is reported in [16]. It uses attributed event grammars to specify the hazardous situations in the environment and aims at traversing the environment model to obtain test scenarios. The test scenarios are selected at random. The models are only used to generate test drivers and not to simulate the environment. The technique was not applied to any real-world case study, but rather to a simplified abstraction of an actual system. The specifications are in textual format and written as a grammar. Writing the specification of the environment for various scenarios will result in a large set of specifications. Since, the environment models are prone to changes (as we discuss later, mostly because of change in specification of environment components or when doing configuration testing) these specifications can become cumbersome to write and express (e.g., lack of commercial tools, lack of proper modularization of specifications). Such specifications might not scale to environments of real industrial systems. Though their testing strategy is based on environment models, that work is significantly different from our approach in several ways.

Our methodology is based on international modeling standards: Unified Modeling Language (UML), UML profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE), and Object Constraint Language (OCL), which are widely used and typically familiar to many software engineers. This selection is intentional in order to make our approach applicable in industrial settings. The idea is to use notations for modeling the environment with which the software engineers are typically familiar. Our environment models contain information regarding non-determinism and error states of the environment (due to incorrect behavior of the SUT) that is used to guide test case generation and obtain automated oracles. We also apply different search-based testing strategies and compare their results. Last but not least, as opposed to published case studies (e.g., [16, 57]), we assess our test strategies on the *actual production code* of an industrial RTES.

3 Environment Modeling & Simulation

For RTES system testing, software engineers familiar with the application domain would typically be responsible for developing the environment models. Therefore, the modeling language should be familiar to them and preferably 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 material. The Unified Modeling Language (UML) and its extensions are therefore a natural choice to consider in our context [46].

Several modeling and simulation languages are available and can be used for modeling and simulating the RTES environment (e.g., DEVS [56]). But, in our context, using these simulation languages raises a number of issues, including the fact that software engineers in the development teams are usually not familiar with the notations and concepts of such languages. Moreover, writing integration code between these languages and the source code of the SUT is a significant challenge. For example, they run on a virtual machine and writing a communication layer to communicate with the SUT would likely be a challenge in most cases. Even if such communication layer exists, because the SUT would necessary run outside the virtual machine of the simulator, then most of the benefits of these simulation languages would be lost, as for example such a layer will effect the precise simulation of time. Another issue is that these simulations languages are developed for a different purpose than environment model-based black-box system testing of RTES. A number of features are required to be modeled for this type of testing, which cannot be modeled with these simulation languages. For example, the models need to provide information about the oracles (to verify the test cases pass or fail). Moreover, during simulations, appropriate values corresponding to the non-deterministic choices in the environment models are also required. In this type of testing, these values are provided by the test cases.

Higher level programming languages (such as Java or C) can also be used as simulation languages. There are a number of issues with using these languages for simulators as we encountered in the software development processes for such simulators of our industrial partners. A 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 develop such simulators at a low-level of abstraction while simultaneously focusing on the details of the environment, complex programming constructs (such as, multiple threads, timers), and the handling of test configurations. Modeling at a level of abstraction higher than programming languages will, for example, reduce the unnecessary complications added by managing threads for timers or concurrency. Another problem is that during the development cycle of RTES, the environment simulators often need to be modified, due to the changes in the specifications of the environment components or for the testing of various environment configurations. Modifying a simulator will be more complex if the language used to build it is at a lower level of abstraction.

In general, for the type of system testing we do, a communication layer is needed to make the simulated environment communicate with the actual RTES (e.g., to receive stimuli and to send responses). Depending on how RTES is actually communicating with the environment components, different approaches are possible to implement such a layer, such as using TCP connections or operating system signals. A communication layer should replace the actual low-level drivers (if any) that the RTES uses to handle the hardware components directly connected to it.

Another important issue for simulation approaches is the handling of time. Typically, the aim of these approaches is to simulate and analyze the behavior of a system or environment before it is actually available. For the type of testing we do, the aim is to simulate the environment in diverse situations, without involving the actual environment components. As mentioned earlier, these components can be anything worth simulating in the environment of the SUT, ranging from hardware components such as sensors to the people in the environment. The SUT in our approach is always the actual executable production code, which is seen as a black-box. We do not control the internal behavior of the SUT and how it handles time. SUT clock is typically simulated in simulation approaches, such as SystemC[2]. In our case, such simulation is not required. In fact, the SUT interacts with the simulated environment as it would interact with the actual environment. Therefore, we do not simulate the clocks of either the environment or the SUT. The notion of time in the environment is that of the physical time. To achieve this, we used the definition of time provided by Java time semantics which are based on the CPU clock. Use of a CPU clock can have a significant issue of jitter that might be introduced because

[2]Webpage: http://www.systemc.org, last accessed 07/11/2011

of computational overhead on the processor (e.g., other operating system processes that are running). 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 on regular virtual machines and operating systems. For such cases, we recommend to use real-time Java virtual machines (e.g., Sun Java Real-Time System[3]) running over real-time operating systems (e.g., SUSE Linux Enterprise Real Time Extension[4]) which provide a nanosecond level time precision. Our generated simulators are compatible to run with the real-time Java virtual machines.However, if the time constraints are in the order of seconds (as it happened for the case studies used in this paper and in many other RTES), then use of the CPU clock is not a serious problem.

In our work, we have used UML/MARTE as a simulation language. We have already proposed a modeling methodology for modeling the environment [34]. The methodology introduces a UML profile for environment modeling and requires the engineers to develop a domain model and various behavioral models for environment components. The domain model represents the overall structure of the RTES environment, focusing on the various environment components, their relationships, properties, and constraints. The domain model is developed using UML class diagrams annotated with the above-mentioned environment modeling profile.

For each environment component that has a significant behavior for the SUT, a behavioral model is developed as a UML state machine. Apart from the nominal behavior of these environment components and their interaction with the SUT, the models may also contain information regarding possible incorrect or even hazardous states in the environment components. An example of the former is when items are not reaching their designated destinations in a sorting machine system. An example of the latter is when a train is reaching a gate while it is open for a gate control system. Such states, which can be caused by a faulty implementation of the SUT, are marked by the stereotype «Error» in the state machines. The behavioral models can also contain failure states that represent possible failures in the environment components (marked as «Failure» states). The SUT is expected to behave in an appropriate way (even if in a degraded mode) when environment components reach failure states (e.g., if some sensors break, the entire system should not go down).

Consider the example shown in Figure 1 which contains a simplified domain model and a state machine modeled using our environment modeling methodology. The only environment component is a Sensor, which sends some data to the SUT after a time delay. The state machine contains two failure states, *SoftwareFailure* and *HardwareFailure*. In both of these states the Sensor does not send the required data to the SUT. The Sensor can recover from a *SoftwareFailure* if it receives a *reboot* signal from the SUT. In such a case, the Sensor will transition back to the *Working* state. The reboot signal should never be received by the environment component while it is in the *Working* state. If this happens it would imply that the SUT is faulty. This is modeled with the transition from *Working* state to *Environment Illegal* state. The latter is an example of error state of the environment and is labelled with the «Error» stereotype.

Our methodology also allows the modeling of non-deterministic behavior of environment components using specialized notations provided by the profile. This is important to model, since an RTES environment can have a number of events whose occurrence cannot be determined, due for example to different arrival patterns of events or varying durations for completing a task. For verification purpose, our methodology allows four different types of non-determinisms to be modeled in the environment models. For all these cases, the models only provide a range of the possible values for the non-deterministice choices and the actual values are assigned based on these ranges during simulation. The first two types of non-determinism are defined based on the properties of components in the domain model and are modeled by using the stereotype «NonDeterministic». For numerical variables, upper and lower bounds for the possible values can be provided for these properties in the models. For other types of variables such as strings, or for not contiguous numerical ranges, the domains of the variables can be defined with OCL constraints.

For the first type of non-determinism, the values are assigned only once during the simulation when the instance of the component is created for the first time. This type is used for those attributes of the environment components whose values are left for the test cases to decide, but the initialization is only required once during the lifetime of the instance, for example types of items (glass bottles, cans, pet bottles) to be inserted in a recycling machine. In the example shown in Figure 1, the attribute *data* shows this type of non-determinism. The attribute is stereotyped with «NonDeterministic» with the stereotype attribute *scope* equal to *class* and stereotype attributes *upperBound* and *lowerBound* specifying the range of possible values for this attribute.

[3]Webpage: http://www.oracle.com/technetwork/java/javase/tech/index-jsp-139921.html, last accessed 07/11/2011
[4]Webpage: http://www.suse.com/products/realtime/, last accessed 07/11/2011

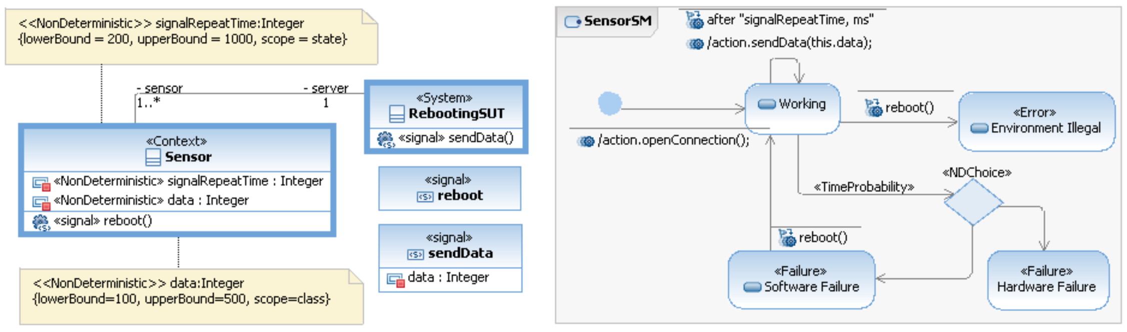
<<NonDeterministic>> signalRepeatTime:Integer
{lowerBound = 200, upperBound = 1000, scope = state}
- sensor
1..*
- server
1
«System»
RebootingSUT
«signal» sendData()
«Context»
Sensor
«NonDeterministic» signalRepeatTime : Integer
«NonDeterministic» data : Integer
«signal» reboot()
«signal»
reboot
«signal»
sendData
data : Integer
<<NonDeterministic>> data:Integer
{lowerBound=100, upperBound=500, scope=class}
SensorSM
after "signalRepeatTime, ms"
/action.sendData(this.data);
Working
reboot()
«Error»
Environment Illegal
/action.openConnection();
«NDChoice»
«TimeProbability»
reboot()
«Failure»
Software Failure
«Failure»
Hardware Failure

Figure 1: Environment models of a Sensor component in one of the artificial problems.

The entire range of values (from 100 to 500) are acceptable for the SUT. The value of the *data* variable is initialized when an instance of Sensor component is created during simulation.

For the second type of non-determinism, modeling of the property is similar to the first type with the only difference that the *scope* attribute of the stereotype «NonDeterministic» is set to *state*. This type of non-determinism is used mostly to model non-deterministic time transitions in the state machines of environment components. Such transitions can for example be used to model the non-deterministic arrival time of events. In this case, the values for the properties are assigned during simulation on entering the states which have outgoing transitions using the stereotyped attribute. The attribute *signalRepeatTime* in the environment model of Figure 1 is an example of this type of non-determinism. The attribute can have a value between 200 and 1000 as specified by the attributes *lowerBound* and *upperBound* of the stereotype «NonDeterministic». The attribute is used in the self transition of the *Working* state in the expression of the time event *after "signalRepeatTime, ms"*. This means that the value of this time event can vary between 200 milliseconds and 1 second. The actual value is provided by the test cases, which in this case will represent the actual value of this timeout.

The other two types of non-determinism can be modeled only in the state machines. One is used to model the non-deterministic time of event occurrences for events that can happen at any time during simulation, for example the breakdown of a sensor. This is done by labeling the transitions with the stereotype «TimeProbability». The other type is to model non-deterministic choices for selecting one of multiple possible valid transitions at a given time. This is modeled by assigning a stereotype «NDChoice» to a choice node. In the environment models shown in Figure 1, the transition between *Working* and the failure states is stereotyped as «TimeProbability» which is leading to a choice node stereotyped as «NDChoice». Using the «TimeProbability» on a transition specifies that this transition can be taken at any time during simulation and its exact occurrence is decided by the test cases during simulation. The choice node with «NDChoice» refers to the fact that the decision of which type of failure to select is also decided based on the information from the test cases, in this case there are two possible choices of failures.

Java is used as action language to specify various activities and effects and OCL is used to specify various constraints and guards in the models. For simple actions, the code can be written within the models. For complex action code and code related to communication with the SUT, external action code files are written in Java.

These models are then automatically transformed into environment simulators in Java code using model to text transformations. The transformations follow specific rules that we discussed in detail in [33]. During simulation a number of instances for each environment component can be created, which interact with each other and the SUT. The generated simulators are linked with the test framework that provides that appropriate values for each simulation execution. For all our case studies, the generated simulators communicated with the SUT using TCP sockets. As mentioned earlier, the code for this communication layer is written manually as an external action code file. We make use of the adapter pattern to provide a standard interface between the simulator and communication layer. This allows the simulator to be independent of the language in which the SUT is written as long as an adapter for communication is provided.

4 Automated Testing

In this section, we discuss the automated testing methodology based on the environment models. As mentioned earlier, automated testing is the main focus of this paper. Following, we first discuss the representation of a test case in our context and the possible issues associated with this representation (Section 4.1) and later we discuss the various testing strategies that we devised for automated system testing of RTES based on environment models (Section 4.2).

4.1 Test Case Representation

In our context, a test case execution is akin to executing the environment simulator. The domain model represents various components in the RTES environment. As mentioned earlier, there can be multiple instances for each of these environment components during simulation. For example, in a gate controller RTES, we can have an environment component representing trains in general. And then, during simulation, we can simulate multiple trains where each simulated train will be represented by an independent running instance of the train environment component. Similarly for the example shown in Figure 1, we can have multiple instances of the component *Sensor* during simulation.

The domain model is used to identify how many instances can run in parallel for each environment component. We refer to the configuration of instances created for simulation as an *environment configuration*. Based on the domain model, there are usually many possible environment configurations, but in this paper we focus only on one fixed configuration during testing (i.e., a fixed number of sensors or trains). Handling different types of configurations in an automatic way is an important problem that is the subject of extensive research [27]. As discussed in Section 3, in the behavioral models of the environment (i.e., state machines) there can be non-deterministic parts. For example, recall that timeout transitions are used to model the arrival of events in the environment and those are non-deterministic events. These transitions could be triggered between a minimum and a maximum time value but the exact value cannot be determined. This is a typical situation when real-world components are modeled, for which there is often natural variance in time-related properties. For example, the time a user takes to insert the required amount after selecting a desired item in a vending machine is non-deterministic. This case is also visible in the state machine shown in Figure 1. The self transition from *Working* state can have a variance between 200 milliseconds and 1 second. This transition is modeling the fact that a sensor is sending data to the SUT after every time interval which can vary within the mentioned range. Another example is when we model the failure scenarios, as for example the breakdown of sensors and actuators. As already discussed, this is modeled by applying the stereotype «TimeProbability» on the transition from *Working* towards *SoftwareFailure* and *HardwareFailure* in the state machine shown in Figure 1.

In our context, input data of a test case are the choices of the actual values to use for all these non-deterministic events. For example, this means that the decision on when a sensor should experience a software failure is made based on the values provided by a test case. In our example, it will require values for two non-deterministic choices to reach the *Software Failure* state from *Working* , i.e., a value for «TimeProbability» and a value for «NDChoice». We refer to these values for non-deterministic choices as a *simulation configuration*. A complete test case in our methodology is a combination of an environment configuration and a simulation configuration. As mentioned earlier, for this paper we are considering a fixed environment configuration and therefore, the rest of the discussion in the paper will be about generating simulation configurations.

In our modeling methodology, we can have non-determinism both in the domain model and in the behavioral models. For the former, the value during simulation is only assigned once for every instance at the time of instance creation (as for example the attribute *data* of Sensor). The value remains the same for the lifetime of the instance. The number of instances is known at the time of simulation so the number of non-deterministic values to be included in a test case corresponding to the domain model is known before simulation.

The case for behavior models is different. We can have non-deterministic choices for various modeling elements in the state machines. They can be in triggers (e.g., time event), on transitions (modeled with «TimeProbability»), and on choice nodes (modeled with «NDChoice»). A transition might be taken several times, and this number might be unknown before executing a test case (e.g., if the transition is in a cyclic path). Therefore, for each instance of the environment component, for each non-deterministic choice in the behavior models, we allocate in the test case a test data vector of possible values (values in different vectors can be of different types). A test case can be seen as a matrix, with a test data vector row for each non-deterministic

$$\begin{pmatrix} 103 & 463 & 311 \\ 400 & 220 & 560 \\ 3300 & 271 & 2153 \\ 0 & 1 & 0 \end{pmatrix}$$

Figure 2: A matrix of $l = 3$ showing non-deterministic choices for the Sensor component

choice of length l, which needs to be set. Each time a non-deterministic choice needs to be made, a new value from the corresponding vector is selected.

Figure 2 shows a matrix with $l = 3$, corresponding to non-deterministic choices of the example shown in Figure 1. Each row of the matrix represents the vector row of possible values for each non-deterministic choice. The first row shows the candidate values for the *data* attribute. The second row shows possible values for the attribute *signalRepeatTime* between the range [200,1000] as specified by the constraint in the environment model. The third row represents the values for the «TimeProbability» transition leading to failures. The fourth row represents the possible choice values for «NDChoice». Since the choice node has two outgoing transitions, the values are either 0 or 1.

Note that since we do know the number of values used from the test data vector before test execution, it is possible that during test execution all values in the data vector are not used. We still refer to the matrix containing these values as a test case. Similarly, it is also possible that during the execution of a test case, more than l values are needed from the same vector. We need a strategy regarding how to handle these cases. In this paper, we propose and evaluate two different approaches: vectors are either used as *rings* or extended *dynamically* during test case execution.

In a ring representation, the values from a vector are taken in order, and after l requests for values, it starts again from the beginning of the vector. In Figure 1, the time transition *after "signalRepeatTime, ms"* has a non-deterministic choice in [200,1000], i.e., signalRepeatTime $\in$ [200,1000]. Given for example $l = 2$, we would have a data vector containing for example $\{400,220\}$. The first time the transition $Working \rightarrow Working$ is taken, the value 400 is used for the non-deterministic choice. The second time, the value 220 is used. The third time, the value 400 is used again, and so on.

The other option is to use a variable size vector that can change dynamically. In such case, the values are taken in order and, after l request for values, the size of the vector is increased, where the new positions are filled at random. For example, consider Figure 1 if $l = 2$, and the data vector containing for example $\{400,220\}$. When the transition from $Working \rightarrow Working$ is taken for the third time, the length of the vector l is changed to 3 at runtime, such that we may obtain a new data vector $\{400,220,560\}$ and the value 560 is used.

The choice of l is arbitrary, but it has significant consequences. For example, in a ring representation, a small number l of possible values could make it impossible to represent sequences of event patterns that lead to failures in the RTES (e.g., if $l = 1$, each time we need a non-deterministic value in a loop, we will always take exactly the same value). On the other hand, as we will see in the next sections, 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. This can happen because, during execution, only few values are used if l is too long. Therefore, all these unused values just end up in unnecessary "noise" that the testing techniques have to deal with (recall that, before execution, we cannot know how many values are going to be used by the simulator).

Note that the two discussed representations have another limitation. In our case studies, the l 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 dependent on the dynamic execution of the test cases. For example, if a variable is constrained within minimum and maximum bounds, then they 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, an issue to be addressed by future research. Notice that, this could be partially accomplished by using the variable size representation discussed above with $l = 0$. Each time a new value is needed, then the vector is increased by only one (with a new random value from the appropriate domain). However, in this context a test case would

```
1: $Z$ ={}
2: add a random test case to $Z$ and execute it
3: while environment error state not reached $OR$ maximum test case executions are not completed
4:     sample $W$ random test cases
5:     for each $w \in W$
6:         $w.minD = min(dis(w,z \in Z))$
7:     execute and add $w$ with maximum $minD$ to $Z$
```

Figure 3: Pseudo-code of the basic algorithm of ART.

contain no information (i.e., being $l = 0$, all the test cases will be simply "empty"), and the only feasible testing strategy left would hence be random testing (because all test cases are empty with no information, we cannot exploit the previously executed test cases to smartly choose new ones to run and evaluate).

4.2 Testing Strategies

As described in the previous section, a test case can be seen as a matrix of test data vectors. Given n environment component instances with m non-deterministic choices in each of their state machines (for simplicity, because in general instances of different machines will have a different number of non-deterministic choices), then the matrix would have $n \times m$ rows, each one of length l. Elements in the rows of this matrix can be of different types, but their domain of valid values should be known before test case execution. Given $D(i)$ the domain of the ith variable type in the ith row, we obtain that the number of possible valid test cases is $\prod_i |D(i)|^l$, which is typically an extremely large number. An exhaustive execution of all possible test cases is therefore 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 (i.e., an error state in the environment has been reached). 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 the matrix, 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 [25, 15, 11].

Another technique that we investigate is Adaptive Random Testing (ART) [21], 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. We are aware of no previous application of ART to test RTES. In this paper we use the basic ART algorithm described in [21]. The pseudo-code of the algorithm used is shown in Figure 3.

Because in our case studies all the variables in the matrix are numerical, to calculate the distance between two test case data matrices M_1 and M_2 we use the following function: $dis(M_1,M_2) = \sum_r \sum_c abs(M_1[r,c] - M_2[r,c])/|D(r)|$. 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 in the triggering time of events. When a dynamic size representation is used (rather than a ring one), matrices can have rows with different lengths l_1 and l_2. In this case, in each row we only compare pairs of values up to $min(l_1,l_2)$, i.e., $dis(M_1,M_2) = \sum_r \sum_{c=1}^{min(l_{1r},l_{2r})} abs(M_1[r,c] - M_2[r,c])/|D(r)|$, where l_{1r} and l_{2r} are lengths of the rows of the two matrices. Notice that other types of distance functions could be used, but a complete analysis of their effects is not in the scope of this paper.

Since the distance calculation between new test cases is calculated before their execution, it can only exploit static test case information. However, when we calculate the distance of a new sampled test case t_a from the *already executed* test cases t_e, we can exploit the execution information of these test cases to remove "noise" in the distance calculations introduced by unused values. For example, in a dynamic variable size representation, we know how many values were actually used during the test case execution for t_e, regardless of whether they were less or more than the original l. So instead of using $min(l_{t_a r},l_{t_e r})$, we can replace $l_{t_e r}$ with the actual number of times values from that row r were used. For example, consider the models shown in Figure 1 and

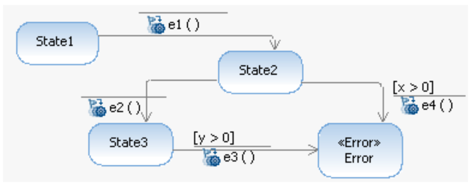

Figure 4: A dummy state machine to explain branch distance

the corresponding matrix shown in Figure 2. If only two values ($\{400,220\}$) for the attribute *signalRepeatTime* from the given vector $\{400,220,560\}$ are used, then we will utilize the value $min(3,2)$ in the distance formula above, rather than $min(3,3)$.

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) [29], and also the use of (1+1) Evolutionary Algorithm (EA). (1+1) EA is a simplified version of GA in which a single individual is evolved overtime with mutation rather than using an initial population as is the case with GA.

To use search algorithms to tackle a specific problem, a fitness function needs to be defined that is 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, as determined by an analysis of the environment model.

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 [41] and MBT [39]. 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 M is executed, then $f(M) = 0$. 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 calculates how close are the current values of the test case to satisfy a possible guard on the transition leading to the error state.

We will explain this by using a dummy example of a UML state machine shown in Figure 4. If the desired state is *Error* and currently the simulation is in *State2*, then the approach level is 1. By calculating the approach level (A) for the states that were reached, we can obtain the state that is closest to the desired state (i.e., it has the minimum approach level). In our example, the closest state based on the approach level is *State2*. Now, the goal is to transition in the direction of the desired state in order to reduce the approach level to 0. This goal is achieved with the help of branch distance. The branch distance (B) is used to heuristically score the evaluation of the guard (represented as an OCL constraint) on the outgoing transitions from the executed state that is closest to the error state. The closes executed state is the one having the minimum approach level to the error state.

In [4] and [5], we have defined a specific branch distance function for OCL expressions that is reused here in this paper. An event corresponding to a transition can occur several times but the transition is only triggered when the guard is true. The branch distance is calculated every time the guard is evaluated to capture how close the values used are from solving the guard. In the example of Figure 4, we need to solve guard $x > 0$ so that whenever $e4()$ is triggered we can reach the error state. Note that the branch distance is less important than the approach level. This is because the search always favors those test data vectors that reduce the approach level (i.e., they take the simulation closer to the Error state). The branch distance only comes into play when there is a guard on the transition, which cannot be set to true with the current setting of the simulation. 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 [8]. Note that, in the case of MBT, it is not always possible to calculate the exact branch distance when the related transition has never been triggered. In these cases, we assign to the branch distance B its highest possible value (i.e., the highest value a variable of double type can have). This is done to

```
1:  Generate random initial population $G$ of test cases and execute all $T_c \in G$
2:  while environment error state not reached $OR$ maximum fitness evaluations are not completed
3:      choose $P_1$, $P_2$ parents from $G$ with rank selection
4:      generate $O_1$, $O_2$ offsprings by crossover of $P_1$, $P_2$
5:      mutate $O_1$, $O_2$
6:      execute offsprings $O_1$, $O_2$
7:      select $P_b$ from $P_1$, $P_2$ that has best fitness
8:      select $O_b$ from $O_1$, $O_2$ that has best fitness
9:      if $fitness(O_b) \leq fitness(P_b)$
10:         then $P_1 = O_1$, $P_2 = O_2$
```

Figure 5: Pseudo-code of the employed steady state GA.

favor the test data vectors that have the same approach level, but were able to trigger the related transition with the guard to obtain an exact branch distance.

The extension of the fitness function proposed in this paper exploits the time properties of the RTES. Some of the transitions are triggered when a time-threshold is reached. For example, an error state could be reached if a sensor or 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 getting the execution closer to violating those time constraints. If a transition should be taken after z time units but is not, then we calculate the maximum consecutive time t the state machine stayed in the source state of the transition (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 M is defined as:

$$f(M) = min_E(A_E(M) + nor(T_E(M)) + nor(B_E(M)))$$

where A_E represents the approach level to error state E, T_E represent the time distance to E, and B_E represents the branch distance to state E. Since there are typically multiple error states in the models, the function $f(M)$ only considers the minimum value over all the error states. 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 the simulator code is generated from the environment models.

Once the fitness function is defined, we can use it to guide the search algorithms to select test cases. The first search algorithm we consider in this paper is a steady state GA [41]. GAs rely on four basic features: population, selection, crossover and mutation. More than one solution is considered at the same time (population). At each generation (i.e., at each step of the algorithm), some good solutions in the current population are selected to generate offspring using the crossover operator. This operator combines parts of the chromosomes (i.e., the solution representation) of the offspring with a certain probability; otherwise it just produces copies of the parents. These new offspring solutions are included in the population of the next generation. The mutation operator is applied to make small changes in the chromosomes and better explore the solution space. To avoid the possible loss of good solutions, some of the best solutions can be copied directly to the new generation without any modification. Another option is to use an approach in which only the offspring that are not worse than their parents are added to the next generations. Fitter individuals should have more chances to survive and reproduce. This is represented by the selection mechanism, for which there are several variants. Eventually, after a number of generations, an individual that solves the addressed problem may be obtained, e.g., a test case reaching an error state. The pseudo-code of the employed GA is shown in Figure 5.

GAs have many parameters that need to be set. We employ rank selection with bias 1.5 to choose the parents. The population size is set to 10. A single point crossover is employed with probability $P_{xover} = 0.75$. This operator chooses a random row inside the data matrices of the parents. The rows after the selected splitting point are swapped between the two parents. For example, consider two matrices, $\left(\begin{smallmatrix} 10 & 46 \\ 40 & 22 \end{smallmatrix}\right)$ and $\left(\begin{smallmatrix} 20 & 31 \\ 33 & 52 \end{smallmatrix}\right)$. If the splitting point is 1, then the new offsprings after the crossover will be $\left(\begin{smallmatrix} 10 & 46 \\ 33 & 52 \end{smallmatrix}\right)$ and $\left(\begin{smallmatrix} 20 & 31 \\ 40 & 22 \end{smallmatrix}\right)$. Each of the $k = n \times m \times l$ elements in a data matrix is mutated with probability $1/k$. A mutation consists of replacing a value with another one at random within the range specified in the environment model for this non-deterministic choice. Notice that, as for in ART, information on how many values in each row were actually used is employed

```
1: Generate a random test case $T_c$ and execute it
2: while environment error state not reached $OR$ maximum fitness evaluations are not completed
3:     mutate $T_c$ to get $T_m$
4:     execute $T_m$
5:     if $fitness(T_m) \leq fitness(T_c)$
6:         then $T_c = T_m$
```

Figure 6: Pseudo-code of the employed (1+1) Evolutionary Algorithm.

in the calculation of k (instead of using the pre-defined l for each row). It can be argued that all the unused values should be removed from a data matrix after its test case execution. However, in evolutionary computation this may not be a good idea, as redundant genetic material in the genotype could be still useful throughout the evolving generations even if they do not immediately manifest themselves in the phenotype (for more details on this topic, please see [49]). In other words, even if some values are not used during test case execution, they might be still useful for generating new test cases in future generations. At any rate, whether this is actually the case for the testing of RTES is a matter of future investigations.

The optimal configuration of search algorithms is in general problem dependent [55]. Due to the large computational cost of running our empirical analysis, we have not tuned the GA. We simply use reasonable parameter values recommended in the GA literature, which seem to work reasonably well, though their *tuning* could significantly improve performance [13].

Another search algorithm we study in this paper is (1+1)EA. At a high level, (1+1)EA can be described as single individual GA, where only mutation is applied. Pseudo-code of (1+1)EA that we used is presented in Figure 6. (1+1)EA has been chosen for two reasons: first, due to the tight number of available fitness evaluations in system testing (since each test execution is expensive in terms of time, e.g., in our industrial case study it took 60 seconds for each test run) a single individual search algorithm could be preferred, as it puts more emphasis on the exploitation rather than the exploration of the search landscape; second, previous work in software testing show that (1+1)EA can be more effective than GAs in some situations (e.g., [9], [4], [30]).

As we discussed earlier in the paper, we run the actual production code of the RTES on a target hardware, where time constraints are based on the actual passing of time. Therefore, if other processes suddenly pop-up consuming most of the CPU, this may affect execution times and lead to test case execution failures (i.e., an error state is reached in the environment model) even if the SUT does not contain any fault. Furthermore, test cases could fail due to the operating system not properly deallocating resources (e.g., TCP sockets) when thousands of test cases are run in series. Synchronization faults in the SUT, that occur with very low probability, may also lead to problems as re-running corresponding failing test case generated by our framework for debugging purposes would be nearly useless. To cope with all these problems, we apply the following practical strategy. When the testing framework finds a test case that fails, it does not output that test case immediately to the user. The framework waits for five minutes, and then re-run the same test case again. If it is still failing, then the framework waits another five minutes, and checks if it is still failing. A test case is outputted to the user (and so the search ends) if and only if a test case fails for three times in a row. Otherwise, the search resumes. In a real industrial context in which the search can run for many hours/days, a 10 minute wait is much more preferable than prematurely stopping the search and providing the user with useless test results.

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 [5]. The analyzed system is a very large and complex seismic acquisition system that interacts with several sensors and 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. The SUT consists of two processes running in parallel, requiring a high performance, dedicated machine to run. Information of the environment models of this RTES is provided

[5]Notice that, since the publication of [14], we have made many modifications, improvements and fixes on our testing framework and models. Therefore, we re-ran the original experiments in [14], and the data presented in this paper are hence not exactly the same.

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	16	37	2	5	9
S2	5	16	1	1	1
S3	2	3	0	1	0

Table 2: Properties of the four 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	284	1871	1	8	7	10	20
AP2	471	975	1	3	5	2	4
AP3	401	2396	2	9	13	5	18
AP4	516	5545	4	23	38	13	33

in Table 1. Notice that for this case study there are several environment components, and for each of them there can be one or more instances running in parallel at the same time.

For each test case, $seven$ instances of environment components run in parallel, and each of them can start several threads. The total number of non-deterministic choices (NDCs) is $5 \times 9 + 1 = 46$. The UML/MARTE environment models were developed in IBM Rational Software Architect according to our methodology (Section 3). 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 will be made 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 four artificial RTES, called AP1, AP2, AP3 and AP4. 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 [57]. The last one is based on another industrial system produced by a different industrial partner. We use the same environment, but we had to recreate a dummy/simplified SUT because, due to technical reasons related to the communication layer and undocumented legacy code, at the time of writing of this paper it was not possible to use the actual production code of that industrial system.

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 at most two error states. We introduced by hand a single non-trivial fault in each of these RTES. We could have rather seeded those faults in a systematic way, for example by using a mutation testing tool [7]. We did not follow such procedure because the SUTs are highly multi-threaded and use a high number of network features (e.g., opening and reading/writing from TCP sockets), which could be a problem for 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 on RTES and by analyzing the bug repository of the industrial case study used in this paper.

5.2 Experiments

In this paper, we want to answer the following three research questions:

RQ1: What is the effect of test case representation on fault detection effectiveness of the testing strategies?

RQ2: Which testing strategy is best in terms of failure detection amongst RT, ART, GA, and (1+1)EA?

RQ3: Is environment model-based system testing an effective approach in detecting faults for industrial RTES?

To answer these research questions, we have carried out two different sets of experiments. One for the artificial problems, and one for the industrial RTES.

In the first set of the experiments, we ran RT, ART, GA and (1+1)EA on each of the four artificial problems. For each search algorithm, we consider both a ring and a dynamic test case representation, both with three different values for the starting length: $l \in \{3,5,10\}$. In total, we had $4 \times 4 \times 2 \times 3 = 96$ experiment configurations (4 artificial problems, 4 algorithms, 2 test case representations with each having 3 different lengths).

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 [9] (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 times. To increase the realism of the artificial problems, we chose to run them for 10 seconds.

Because the execution of a single test case takes 10 seconds, we stop each algorithm after 1000 sampled test cases or as soon as one of the error states is reached. Therefore, a single experiment can take up to more than two hours and half. In our context, the test case execution time is fixed, and it does not depend on the used execution platform and using faster hardware would not change the experiments' execution time. 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 cluster of computers used in the experiments were appropriate.

For each of the 96 experiment configurations, 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 $96 \times 1000 \times 100 = 9{,}600{,}000$. In our experiments, we had a total of 5,643,315 test cases (recall, we stop the search as soon as a test case fails). Running all these test cases would take more than 653 days on a single computer and, to alleviate this problem, we used a cluster to run these experiments.

The second set of experiments has been carried out on an industrial RTES. We run each test case for 60 seconds, where 1000 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.

We once again evaluated the use of RT, ART, GA and (1+1)EA to find failures, but this time in the industrial RTES. We could not run this empirical analysis on a cluster due to several technical reasons. We had to use a single dedicated computer with high performance, and it took more than 55 days to run these experiments. For this reason, we only evaluated a dynamic test case representation with length $l = 5$ (which showed the best results for artificial problems), and we only had 39 repeated runs per experiment (instead of 100). For this industrial RTES we did not need to seed any fault since the goal was to find new, previously uncaught critical faults.

To compare search algorithms (e.g., is ART better than RT?) in a sound statistical way, we followed the guidelines described in [12]. Differences in success rates are validated with a Fisher Exact test, where the effect size is quantified with odds ratio ψ with a 0.5 correction. When comparing two algorithms (say $\mathcal{A}$ and $\mathcal{B}$), the odds ratio measures the ratio of the odds of success rate of $\mathcal{A}$ to the odds of success rate of $\mathcal{B}$. This is calculated as $\psi = \frac{a+p}{m+p-a} / \frac{b+p}{n+p-b}$, where a is the number of times $\mathcal{A}$ was successful, b is the number of times $\mathcal{B}$ was

Table 3: Rank comparisons of six configurations for RT on all the four artificial problems.

Artifact			Ring			Dynamic
	3	5	10	3	5	10
AP1	3.5	3.5	3.5	3.5	3.5	3.5
AP2	3.5	3.5	3.5	3.5	3.5	3.5
AP3	6	3.5	1	3.5	3.5	3.5
AP4	3.5	3.5	3.5	1	6	3.5
Average	4.125	3.5	2.875	2.875	4.125	3.5

Table 4: Rank comparisons of six configurations for ART on all the four artificial problems.

Artifact			Ring			Dynamic
	3	5	10	3	5	10
AP1	3.5	3.5	6	3.5	3.5	1
AP2	3.5	3.5	3.5	3.5	3.5	3.5
AP3	5	6	4	2.5	1	2.5
AP4	4	5.5	5.5	2	2	2
Average	4	4.625	4.75	2.875	2.5	2.25

successful, n is the total number of runs for $\mathcal{A}$, m is the total number of runs for $\mathcal{B}$, and p is the correction applied (which in our case is 0.5). If $\psi = 1$ then the two algorithms have no difference in terms of odds ratio. If $\psi > 1$ then algorithm $\mathcal{A}$ has higher chances of being successful than $\mathcal{B}$. When there are no differences in success rates of the algorithms, we look at how long each algorithm takes in finding a fault (i.e., number of sampled test cases). Differences are validated with a Mann-Whitney U-test, where the effect size is quantified with the Vargha-Delaney $\hat{A}_{12}$ statistics. When comparing two algorithms $\mathcal{A}$ and $\mathcal{B}$, $\hat{A}_{12}$ statistics measures the probability that algorithm $\mathcal{A}$ will require less test cases to be sampled for finding a fault than that required by algorithm $\mathcal{B}$. If the two algorithms are equivalent, then $\hat{A}_{12} = 0.5$. All statistical tests are done at $\alpha = 0.05$ significance level (we do not report all p-values as it would take a lot of space, and not particularly useful for the specific analyses in this paper).

5.3 Discussion

Regarding the performance of the testing strategies, Tables 3 to 6 report on the analysis of each technique in isolation on each of the four artificial problems. Regarding RQ1, we evaluate the effects of test case representation (ring and dynamic) and the lengths l of the vectors on the algorithm performance. For each problem, there are $2 \times 3 = 6$ configurations of the algorithm to compare. Those tables provide a relative ranking, based on the statistical difference of the compared configurations. Configurations which are statistically equivalent (i.e., high p-values) are expected to show the same or 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. Then, based on the final scores, each configuration is assigned ranks ranging from 1 (best configuration) to 6 (worst configuration). In case of ties, ranks are averaged.

For RT, in Table 3 we can see no difference among the six configurations for AP1 and AP2, whereas on AP3 a ring representation is better when $l = 10$, while on AP4 it is best when a dynamic representation with $l = 3$ is used. So there is no clear trend in the data. On the other hand, for ART, in Table 4 there is a clear preference for a dynamic representation, with length either 5 or 10. The trend is the same for both (1+1)EA and GA, in Table 5 and Table 6 with the difference that for (1+1)EA the length does not seem to matter. We can hence answer **RQ1**: overall, a dynamic representation seems better than a ring one, in particular when the lengths are rather long (i.e., $l = 5$ or $l = 10$).

Table 7 shows full details of the probability distributions of the performance of each algorithm (but only

Table 5: Rank comparisons of six configurations for (1+1)EA on all the four artificial problems.

Artifact			Ring		Dynamic	
	3	5	10	3	5	10
AP1	3.5	3.5	3.5	3.5	3.5	3.5
AP2	3.5	3.5	3.5	3.5	3.5	3.5
AP3	4	5	6	2	2	2
AP4	4	5	6	2	2	2
Average	3.75	4.25	4.75	2.75	2.75	2.75

Table 6: Rank comparisons of six configurations for GA on all the four artificial problems.

Artifact			Ring		Dynamic	
	3	5	10	3	5	10
AP1	3	3	5.5	5.5	3	1
AP2	3.5	3.5	3.5	3.5	3.5	3.5
AP3	4.5	1.5	6	3	4.5	1.5
AP4	4	5.5	5.5	2.5	2.5	1
Average	3.75	3.375	5.125	3.625	3.375	1.75

for the best configuration on average out of the six configurations) on each of the four artificial problems. If we look at the success rates, it is clearly visible that the first two problems are quite difficult to solve, whereas the remaining two are much easier (in the sense that they can be solved nearly each time when a budget of 1000 fitness function evaluations is employed). Recall that we manually seeded faults with the aim of making them challenging to find. At any rate, even if two cases ended up being simple, they can still be used to compare the techniques since we can study how long they take to trigger the first failure. For example, the difference in mean values between RT and ART is nearly 30 test cases. Such a difference entails a 5 minute difference, where RT is 50% slower than ART. In some contexts, this can be considered of practical value when testing is done in parallel with coding and debugging.

In Table 8 we compare each algorithm on each artificial problem, and we show the resulting ranking. To provide more details about these comparisons, in Tables 9 to 12 we show the effect sizes of each of these comparisons. The results are rather surprising. ART appears to be the best algorithm, while the search techniques are even worse than RT. This can also be seen from the success rates of the best algorithms shown in Table 7. Notice that the involved dynamics of these algorithms are very complex, and so we can only provide *plausible conjectures* to explain such behaviors.

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, whose trigger and guard usually depend on the entire state of the environment and SUT. The fitness function we adapted from white-box SBT does not seem appropriate for this kind of system level testing based on environment models. As mentioned earlier, we have already enhanced the fitness function defined for structural testing in the literature ([41] and MBT [39]), but the results suggest that there is still a need to design more sophisticated fitness function (e.g., by exploiting further information/properties of the environment models). Overall, results suggest, however, that the distance function used of ART for the experiments is suitable.

Another complementary explanation is that the search budget was relatively "small", i.e., only 1000 fitness evaluations. This choice was due to the high cost of running each single test case (i.e., 10 seconds). Evolutionary algorithms, such as GA, tend to explore different areas of the search landscape, with the aim of avoiding local optima. This is an extremely useful property in many search problems, but it reduces the convergence speed of these algorithms. For small search budgets there would be no need to escape from local optima, because anyway there would be no time to effectively explore other areas of the search landscape that might have better local optima.

Table 7: Full details of each algorithm (in its best configuration of test case representation and length) when applied on each of the four artificial problems. The success rate indicates the proportion of times out of 100 repeated experiments a fault was found. For the successful runs, we report the distribution of how many test cases were sampled and evaluated before finding the fault.

Artifact	Algorithm	Success Rate	Average	Std. Deviation	Median	Skewness	Kurtosis
AP1	RT	0.06	561.33	337.72	658.50	-0.39	1.42
	ART	0.19	418.47	263.83	346	0.56	2.62
	(1+1)EA	0.00	-	-	-	-	-
	SSGA	0.044	426.75	309.33	403.00	0.18	1.47
AP2	RT	0.00	-	-	-	-	-
	ART	0.020	280.00	103.24	280.00	0.00	1.00
	(1+1)EA	0.00	-	-	-	-	-
	SSGA	0.022	863.50	61.52	863.50	0.00	1.00
AP3	RT	1.00	25.70	24.11	16	1.41	4.35
	ART	1.00	21.85	18.96	16	1.70	7.04
	(1+1)EA	0.99	192.51	199.59	129	1.74	6.37
	SSGA	0.81	139.52	210.25	27	1.84	5.70
AP4	RT	1.00	97.69	107.06	67	3.47	22.19
	ART	1.00	69.58	61.70	52	1.26	4.15
	(1+1)EA	0.94	308.18	261.02	225	0.89	2.86
	SSGA	0.88	302.94	277.30	205	0.94	2.72

Table 8: Rank comparisons of each of the four testing techniques on all the four artificial problems.

Artifact	RT	ART	(1+1)EA	SSGA
AP1	2.5	1	4	2.5
AP2	2.5	2.5	2.5	2.5
AP3	1.5	1.5	3	4
AP4	1.5	1.5	3.5	3.5
Average	2	1.625	3.25	3.125

Though we can explain why GA does not work well on these artificial problems, why does it behave even worse than RT? The reason is exactly the same for which ART is better than RT: the lack of diversity of the test cases. In our case, there was little gradient in the fitness function, hence all the sampled test cases had similar fitness value (i.e., the fitness landscape would have a large plateau). So any new sampled test case was accepted and added to the next generation in GA. The crossover operator did not produce any new value in the data matrices, it simply swapped the values between two parent test cases. The mutation operator resulted in only small changes to a data matrix, because on average only one variable was mutated. During the search, the offspring had genetic material (i.e., the data matrices) that was similar to the one of the parents. Therefore, the diversity of test cases during a GA and (1+1)EA evolution was much lower than the one of RT. If the hypothesis of contiguous regions of faulty test cases is true for a given RTES and when there is no or little gradient in the fitness function, we would a-priori expect this following relationship regarding the performance of testing strategies: $SBT \leq RT \leq ART$. This hypothesis seems to be confirmed by our experiments.

The results of the experiments on the industrial problem are summarized in Table 13, whereas the effect sizes of the comparisons among the four testing techniques are reported in Table 14. In the industrial case study, we can see exactly the same trend as in the artificial problems, i.e., ART is best, while GA and (1+1)EA are even worse than RT. However, there is not enough evidence to claim that ART is indeed better than RT (although this is suggested by the effect sizes) as the results are not statistically significant. In particular, the Mann-Whitney U-test resulted in a 0.09 p-value. In other words, there is a 9% probability of wrongly claiming

Table 9: Effect sizes of the comparisons of each testing technique on AP1. Statistically significant effect sizes at level $\alpha = 0.05$ are reported in bold.

Algorithm	RT	ART	(1+1)EA	SSGA
RT	-	$\boldsymbol{\psi = 0.28}$, $\hat{A}_{12} = 0.62$	$\boldsymbol{\psi = 13.83}$, $\hat{A}_{12} = NA$	$\psi = 1.32$, $\hat{A}_{12} = 0.62$
ART	$\boldsymbol{\psi = 3.52}$, $\hat{A}_{12} = 0.38$	-	$\boldsymbol{\psi = 48.69}$, $\hat{A}_{12} = NA$	$\boldsymbol{\psi = 4.66}$, $\hat{A}_{12} = 0.47$
(1+1)EA	$\boldsymbol{\psi = 0.072}$, $\hat{A}_{12} = NA$	$\boldsymbol{\psi = 0.021}$, $\hat{A}_{12} = NA$	-	$\boldsymbol{\psi = 0.096}$, $\hat{A}_{12} = NA$
SSGA	$\psi = 0.76$, $\hat{A}_{12} = 0.38$	$\boldsymbol{\psi = 0.21}$, $\hat{A}_{12} = 0.53$	$\boldsymbol{\psi = 10.46}$, $\hat{A}_{12} = NA$	-

Table 10: Effect sizes of the comparisons of each testing technique on AP2. Statistically significant effect sizes at level $\alpha = 0.05$ are reported in bold.

Algorithm	RT	ART	(1+1)EA	SSGA
RT	-	$\psi = 0.20$, $\hat{A}_{12} = NA$	$\psi = 1.00$, $\hat{A}_{12} = NA$	$\psi = 0.18$, $\hat{A}_{12} = NA$
ART	$\psi = 5.10$, $\hat{A}_{12} = NA$	-	$\psi = 4.64$, $\hat{A}_{12} = NA$	$\psi = 0.91$, $\hat{A}_{12} = 0.00$
(1+1)EA	$\psi = 1.00$, $\hat{A}_{12} = NA$	$\psi = 0.22$, $\hat{A}_{12} = NA$	-	$\psi = 0.20$, $\hat{A}_{12} = NA$
SSGA	$\psi = 5.62$, $\hat{A}_{12} = NA$	$\psi = 1.10$, $\hat{A}_{12} = 1.00$	$\psi = 5.11$, $\hat{A}_{12} = NA$	-

Table 11: Effect sizes of the comparisons of each testing technique on AP3. Statistically significant effect sizes at level $\alpha = 0.05$ are reported in bold.

Algorithm	RT	ART	(1+1)EA	SSGA
RT	-	$\psi = 1.00$, $\hat{A}_{12} = 0.52$	$\psi = 3.41$, $\boldsymbol{\hat{A}_{12} = 0.15}$	$\boldsymbol{\psi = 48.57}$, $\hat{A}_{12} = 0.46$
ART	$\psi = 1.00$, $\hat{A}_{12} = 0.48$	-	$\psi = 3.41$, $\boldsymbol{\hat{A}_{12} = 0.13}$	$\boldsymbol{\psi = 48.57}$, $\hat{A}_{12} = 0.44$
(1+1)EA	$\psi = 0.29$, $\boldsymbol{\hat{A}_{12} = 0.85}$	$\psi = 0.29$, $\boldsymbol{\hat{A}_{12} = 0.87}$	-	$\boldsymbol{\psi = 14.24}$, $\boldsymbol{\hat{A}_{12} = 0.65}$
SSGA	$\boldsymbol{\psi = 0.021}$, $\hat{A}_{12} = 0.54$	$\boldsymbol{\psi = 0.021}$, $\hat{A}_{12} = 0.56$	$\boldsymbol{\psi = 0.07}$, $\boldsymbol{\hat{A}_{12} = 0.35}$	-

Table 12: Effect sizes of the comparisons of each testing technique on AP4. Statistically significant effect sizes at level $\alpha = 0.05$ are reported in bold.

Algorithm	RT	ART	(1+1)EA	SSGA
RT	-	$\psi = 1.00$, $\hat{A}_{12} = 0.58$	$\boldsymbol{\psi = 14.89}$, $\boldsymbol{\hat{A}_{12} = 0.22}$	$\boldsymbol{\psi = 27.20}$, $\boldsymbol{\hat{A}_{12} = 0.26}$
ART	$\psi = 1.00$, $\hat{A}_{12} = 0.42$	-	$\boldsymbol{\psi = 14.89}$, $\boldsymbol{\hat{A}_{12} = 0.18}$	$\boldsymbol{\psi = 27.20}$, $\boldsymbol{\hat{A}_{12} = 0.21}$
(1+1)EA	$\boldsymbol{\psi = 0.067}$, $\boldsymbol{\hat{A}_{12} = 0.78}$	$\boldsymbol{\psi = 0.067}$, $\boldsymbol{\hat{A}_{12} = 0.82}$	-	$\psi = 1.83$, $\hat{A}_{12} = 0.52$
SSGA	$\boldsymbol{\psi = 0.037}$, $\boldsymbol{\hat{A}_{12} = 0.74}$	$\boldsymbol{\psi = 0.037}$, $\boldsymbol{\hat{A}_{12} = 0.79}$	$\psi = 0.55$, $\hat{A}_{12} = 0.48$	-

Table 13: Full details of each algorithm (with dynamic representation and $l = 5$) when applied on the industrial case study. The success rate indicates the proportion of times out of 39 repeated experiments a fault was found. For the successful runs, we report the distribution of how many test cases were sampled and evaluated before finding the fault.

Artifact	Algorithm	Success Rate	Average	Std. Deviation	Median	Skewness	Kurtosis
Industrial	RT	0.97	342.45	277.48	272.00	0.82	2.52
	ART	1.00	254.74	260.01	142	1.40	4.42
	(1+1)EA	0.51	499.35	298.87	598.00	-0.29	1.79
	SSGA	0.46	321.72	217.58	313.50	0.92	4.19

Table 14: Effect sizes of the comparisons of each testing technique on the industrial case study. Statistically significant effect sizes at level $\alpha = 0.05$ are reported in bold.

Algorithm	RT	ART	(1+1)EA	SSGA
RT	-	$\psi = 0.32$, $\hat{A}_{12} = 0.61$	$\boldsymbol{\psi} = \mathbf{24.41}$, $\hat{A}_{12} = 0.36$	$\boldsymbol{\psi} = \mathbf{29.83}$, $\hat{A}_{12} = 0.49$
ART	$\psi = 3.08$, $\hat{A}_{12} = 0.39$	-	$\boldsymbol{\psi} = \mathbf{75.15}$, $\mathbf{\hat{A}_{12} = 0.27}$	$\boldsymbol{\psi} = \mathbf{91.81}$, $\hat{A}_{12} = 0.38$
(1+1)EA	$\boldsymbol{\psi} = \mathbf{0.041}$, $\hat{A}_{12} = 0.64$	$\boldsymbol{\psi} = \mathbf{0.013}$, $\mathbf{\hat{A}_{12} = 0.73}$	-	$\psi = 1.22$, $\mathbf{\hat{A}_{12} = 0.69}$
SSGA	$\boldsymbol{\psi} = \mathbf{0.034}$, $\hat{A}_{12} = 0.51$	$\boldsymbol{\psi} = \mathbf{0.011}$, $\hat{A}_{12} = 0.62$	$\psi = 0.82$, $\mathbf{\hat{A}_{12} = 0.31}$	-

that ART is better than RT if that is not actually the case. However, recall that for the artificial problems we had 100 runs per experiment, whereas due to the high computational cost of running the experiments for the industrial case study, we could only have 39 runs (which took more than 55 days). Since we see the same performance trend in the four artificial problems, using more runs *might* reduce that p-value.

Accounting for both the experimental results of the four artificial problems and the industrial case study, we can answer **RQ2** by stating that ART is overall the algorithm that performs best, while GA and (1+1)EA were even worse than RT.

Regarding the performance of ART, it is important to note that such result currently only holds for a budget of up to 1000 test case executions. ART has an overhead of distance calculations that is quadratic in the number of test cases, and a memory requirement that is linear. This is one of the reasons why we criticized its use in unit testing [10]. On the other hand, for a budget of up to 1000 test cases where each of them is very time consuming (e.g., 10 and 60 second per test case), then such overheads are simply negligible. However, if a practitioner wants to have longer test executions (e.g., 10 or 100 thousands test cases), then the ART overhead might not be negligible any more and other ART variants would need to be employed, as for example the one described in [18]. Its application in environment model-based testing of RTES is hence an important approach to further investigate in future.

Our framework when applied to the industrial case study, managed to find two critical faults in the production code of the industrial case study. The faults were a result of the combined behavior of various component instances that the SUT was not expecting. They could only be detected by an approach that tests at the system level rather than unit level. The development of environment models is the major effort required for using our methodology. It does not require the software engineers to be familiar with the internal details of the SUT, but it is concerned with its behavior when interacting with its execution environment. This is a common situation in industry with separate, independent system test teams. The environment modeling methodology involves the explicit modeling of erroneous situations of the RTES environment and therefore the determination of possible paths in the environment that can lead to such situations. These can only be reached in the presence of an implementation fault in the SUT.

To answer **RQ3**, since the experiment that we conducted on an industrial RTES yielded a success rate of 100% (ART), we can state that our testing approach showed very promising results. For artificial problems AP3 and AP4, RT and ART both have a success rate of 100%. For AP1, ART has a success rate of 19%, which suggests that with r runs of ART, we would achieve a success rate of $1 - (1 - 0.19)^r$. For example, with

$r = 22$, we would obtain a success rate above 99%. For AP2, none of the approaches were very successful. This was probably because the fault seeded was too hard to detect given the testing budget (i.e., 1000 test case executions). To detect such faults, ART might not be sufficient and, as suggested earlier, we would need for example more sophisticated fitness functions. In future work, we will also need to consider the effect of the way environment models are developed on the effectiveness of the testing strategies.

6 Threats to Validity

In this paper we have carried out a set of experiments on an industrial RTES. Although that system is large and complex, and the experiment took 45 days, it is at this stage difficult to generalize beyond this specific system. The results presented in this paper are not necessarily applicable to all other types of RTES.

To enable future comparisons and complete our analysis, we have also carried out experiments on four artificial problems inspired by actual systems. Although they are not trivial (they are multithreaded and hundreds of lines long), these artificial problems are not necessarily representative of complex RTES.

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 as to obtain a precise and deterministic handling of clock time. We used the CPU clock instead, which may be unreliable if time constraints in the RTES are very tight (e.g., milliseconds) since they could be violated because of unpredictable changes of load balance in the CPU in the presence of unrelated process executions. However, the time constraints in this paper were in the order of seconds. To be on the safe side, 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 running it again with the same seed, we obtained exactly 43 test cases again. However, the experiments were not exactly the same. For example, for debugging purposes we used time stamps on log files. In these time stamps, though small variances of a few milliseconds were present, they did not have any effect on the testing results. Notice that our novel methodology can obviously be applied also when the time is simulated.

7 Conclusion

In this paper we proposed a black-box system testing methodology for real-time, embedded systems (RTES). It is automated and 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 real-time embedded systems (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. Though significant, the only incurred cost by human testers is the development of the environment models. This paper has focused on the testing heuristics and an empirical study to determine the conditions under which they are effective.

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, we complemented this industrial study with artificial case studies, that are made publicly available to foster future empirical analyses and comparisons. The results in both the artificial and industrial case studies were however consistent, and this further supports their validity and representativeness.

We experimented with different test case representations and 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) and (1+1) Evolutionary Algorithm (EA). All these techniques can be adjusted to project constraints as they can be run as long as time and access to CPU are available.

The experiments clearly identified which test case representation should be used to obtain better results and showed that ART was the best among the studied algorithms. Surprisingly, both GAs and (1+1)EA were even worse than RT. We provided detailed arguments to explain these results. Furthermore, to support the claims above, we followed a rigorous experimental method based on statistical analyses, following the guidelines in [12]. This empirical analysis is a significant contribution, in part because of its realism and size, taking more than 708 days of computation (though some, but not all, could be parallelized on a cluster of computers).

Regarding future work, a first step to carry out is an empirical cost-benefit analysis of the proposed approach. The cost of building and modifying the environment models needs to be compared with that of the manual changes to simulators and test suites. Intuitively, the latter should be much larger than the former, but it nevertheless should be investigated. Estimates of the cost of field failures need to be considered as well to obtain more reliable and complete comparisons of cost-effectiveness among test strategies. Another research direction is to analyze how to properly use the domain models for effective automated testing of different configurations of the RTES environment.

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. We are thankful to Bjørn Nordmoen, Petter Eide, Tore Andre Nilsen, and Sofia Wegger, all from WesternGeco for their technical support.

References

[1] W. Afzal, R. Torkar, and R. Feldt. A systematic review of search-based testing for non-functional system properties. *Information and Software Technology (IST)*, 51(6):957–976, 2009.

[2] S. Ali, L. Briand, A. Arcuri, and S. Walawege. An Industrial Application of Robustness Testing using Aspect-Oriented Modeling, UML/MARTE, and Search Algorithms. In *ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS)*, 2011. to appear.

[3] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test case generation. *TSE*, 36:742–762, 2010.

[4] S. Ali, M. Iqbal, A. Arcuri, and L. Briand. A search-based ocl constraint solver for model-based test data generation. In *IEEE International Conference on Quality Software (QSIC)*, pages 41–50, 2011.

[5] S. Ali, M. Iqbal, A. Arcuri, and L. Briand. Solving OCL Constraints for Test Data Generation in Industrial Systems with Search Techniques. Technical Report Technical Report (2010-16), 2011.

[6] R. Alur and D. L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126:183–235, 1994.

[7] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering (TSE)*, 32(8):608–624, 2006.

[8] A. Arcuri. It really does matter how you normalize the branch distance in search-based software testing. *Software Testing, Verification and Reliability (STVR)*, 2011. http://dx.doi.org/10.1002/stvr.457.

[9] A. Arcuri. A theoretical and empirical analysis of the role of test sequence length in software testing for structural coverage. *IEEE Transactions on Software Engineering (TSE)*, 2011. http://doi.ieeecomputersociety.org/10.1109/TSE.2011.44.

[10] A. Arcuri and L. Briand. Adaptive random testing: An illusion of effectiveness? In *ACM International Symposium on Software Testing and Analysis (ISSTA)*, pages 265–275, 2011.

[11] A. Arcuri and L. Briand. Formal Analysis of the Probability of Interaction Fault Detection using Random Testing. *IEEE Transactions on Software Engineering (TSE)*, 2011.

[12] A. Arcuri and L. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *IEEE International Conference on Software Engineering (ICSE)*, pages 1–10, 2011.

[13] A. Arcuri and G. Fraser. On parameter tuning in search based software engineering. In *SSBSE*, pages 33–47, 2011.

[14] A. Arcuri, M. Z. Iqbal, and L. Briand. Black-box system testing of real-time embedded systems using random and search-based testing. In *IFIP International Conference on Testing Software and Systems (ICTSS)*, pages 95–110, 2010.

[15] A. Arcuri, M. Z. Iqbal, and L. Briand. Formal analysis of the effectiveness and predictability of random testing. In *ACM International Symposium on Software Testing and Analysis (ISSTA)*, pages 219–229, 2010.

[16] M. Auguston, J. B. Michael, and M. T. Shing. Environment behavior models for automation of testing and assessment of system safety. *Information and Software Technology (IST)*, 48(10):971–980, 2006.

[17] A. Bonifacio and A. Moura. A new method for testing timed systems. *STVR*, 2011.

[18] K. Chan, T. Chen, and D. Towey. Forgetting test cases. In *IEEE International Computer Software and Applications Conference (COMPSAC)*, pages 485–494, 2006.

[19] T. Chen, D. Huang, F. Kuo, R. Merkel, and J. Mayer. Enhanced lattice-based adaptive random testing. In *Proceedings of the ACM symposium on Applied Computing*, pages 422–429, 2009.

[20] T. Chen, H. Leung, and I. Mak. Adaptive random testing. In *Advances in Computer Science*, pages 320–329, 2004.

[21] T. Y. Chen, F. Kuoa, R. G. Merkela, and T. Tseb. Adaptive random testing: The art of test case diversity. *Journal of Systems and Software (JSS)*, 2010. (in press).

[22] D. Clarke and I. Lee. Testing real-time constraints in a process algebraic setting. In *IEEE International Conference on Software Engineering (ICSE)*, pages 51–60, 1995.

[23] K. Derderian, M. G. Merayo, R. M. Hierons, and M. Núñez. Aiding test case generation in temporally constrained state based systems using genetic algorithms. In *Proceedings of the 10th International Work-Conference on Artificial Neural Networks: Part I: Bio-Inspired Systems: Computational and Ambient Intelligence*, pages 327–334, 2009.

[24] B. P. Douglass. *Real-time UML: developing efficient objects for embedded systems*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1997.

[25] J. W. Duran and S. C. Ntafos. An evaluation of random testing. *IEEE Transactions on Software Engineering (TSE)*, 10(4):438–444, 1984.

[26] A. En-Nouaary. A scalable method for testing real-time systems. *Software Quality Journal*, 16(1):3–22, 2008.

[27] E. EngstrŽm and P. Runeson. Software product line testing - a systematic mapping study. *Information and Software Technology*, 53(1):2 – 13, 2011.

[28] V. Garousi, L. C. Briand, and Y. Labiche. Traffic-aware stress testing of distributed real-time systems based on uml models using genetic algorithms. *Journal of Systems and Software (JSS)*, 81(2):161–185, 2008.

[29] M. Harman, S. A. Mansouri, and Y. Zhang. Search based software engineering: A comprehensive analysis and review of trends techniques and applications. Technical Report TR-09-03, King's College, 2009.

[30] H. Hemmati, A. Arcuri, and L. Briand. Achieving Scalable Model-Based Testing Through Test Case Diversity. Technical Report Technical Report (2010-18), 2010.

[31] H. Hemmati, A. Arcuri, and L. Briand. Achieving Scalable Model-Based Testing Through Test Case Diversity. *TOSEM*, 2011.

[32] A. Hessel, K. G. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, and A. Skou. Testing real-time systems using uppaal. In *Formal methods and testing*, pages 77–117. Springer-Verlag, 2008.

[33] M. Iqbal, A. Arcuri, and L. Briand. Code Generation from UML/MARTE/OCL Environment Models to Support Automated System Testing of Real-Time Embedded Software. Technical Report Technical Report (2011-04), 2011.

[34] M. Z. Iqbal, A. Arcuri, and L. Briand. 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.

[35] B. Jiang, Z. Zhang, W. Chan, and T. Tse. Adaptive random test case prioritization. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 233–244, 2009.

[36] M. Krichen and S. Tripakis. Conformance testing for real-time systems. *Formal Methods in System Design*, 34(3):238–304, 2009.

[37] F. C. Kuo. An indepth study of mirror adaptive random testing. In *International Conference on Quality Software*, pages 51–58, 2009.

[38] K. G. Larsen, M. Mikucionis, and B. Nielsen. Online testing of real-time systems using uppaal. *Formal Approaches to Software Testing*, pages 79–94, 2005.

[39] R. Lefticaru and F. Ipate. Functional search-based testing from state machines. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 525–528, 2008.

[40] F. Lindlar, A. Windisch, and J. Wegener. Integrating model-based testing with evolutionary functional testing. In *International Workshop on Search-Based Software Testing (SBST)*, 2010.

[41] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, 2004.

[42] M. Merayo, M. Núñez, and I. Rodríguez. Extending EFSMs to Specify and Test Timed Systems with Action Durations and Time-Outs. *IEEE Transactions on Computers*, 57(6):835–844, 2008.

[43] T. Miicke and M. Huhn. Generation of optimized testsuites for UML statecharts with time. In *IFIP international conference on testing of communicating systems*, pages 128–143, 2004.

[44] G. Myers. *The Art of Software Testing*. Wiley, New York, 1979.

[45] B. Nielsen and A. Skou. Automated test generation from timed automata. *International Journal on Software Tools for Technology Transfer (STTT)*, 5(1):59–77, 2003.

[46] Object Management Group (OMG). *Unified Modeling Language Superstructure, Version 2.3*, 2010. ISBN 3-900051-07-0.

[47] J. Oh, M. Harman, and S. Yoo. Transition coverage testing for simulink/stateflow models using messy genetic algorithms. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, GECCO '11, pages 1851–1858, 2011.

[48] J. Peleska, F. Lapschies, E. Vorobev, H. Loeding, P. Smuda, H. Schmid, and C. Zahlten. A real-world benchmark model for testing concurrent real-time systems in the automotive domain. In *IFIP International Conference on Testing Software and Systems (ICTSS)*, 2011. to appear.

[49] M. Shackleton, R. Shipma, and M. Ebner. An investigation of redundant genotype-phenotype mappings and their role in evolutionary search. In *Evolutionary Computation, 2000. Proceedings of the 2000 Congress on*, pages 493 –500 vol.1, 2000.

[50] J. Springintveld, F. Vaandrager, and P. DÁrgenio. Testing timed automata. *Theoretical Computer Science*, 254(1-2):225–257, 2001.

[51] A. F. Tappenden and J. Miller. A novel evolutionary approach for adaptive random testing. *IEEE Transactions On Reliability*, 58(4):619–633, 2009.

[52] M. Tlili, S. Wappler, and H. Sthamer. Improving evolutionary real-time testing. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, GECCO '06, pages 1917–1924, 2006.

[53] M. Utting and B. Legeard. *Practical model-based testing: a tools approach*. Elsevier, 2007.

[54] J. Wegener and M. Grochtmann. Verifying timing constraints of real-time systems by means of evolutionary testing. *Real-Time Systems*, 15:275–298, 1998.

[55] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.

[56] B. P. Zeigler, H. Praehofer, and T. G. Kim. *Theory of modeling and simulation*. Academic press New York, NY, 2000.

[57] M. Zheng, V. Alagar, and O. Ormandjieva. Automated generation of test suites from formal specifications of real-time reactive systems. *Journal of Systems and Software (JSS)*, 81(2):286–304, 2008.

[58] M. Zhigulin, N. Yevtushenko, S. Maag, and A. Cavalli. Fsm-based test derivation strategies for systems with time-outs. In *IEEE International Conference on Quality Software (QSIC)*, pages 141–149, 2011.