Empirical Investigation of Search Algorithms for Environment Model-Based Testing of Real-Time Embedded Software

Muhammad Zohaib Iqbal
Certus Software V & V Center,
Simula Research Laboratory and
University of Oslo, Norway
zohaib@simula.no

Andrea Arcuri
Certus Software V & V Center,
Simula Research Laboratory, Norway
arcuri@simula.no

Lionel Briand
SnT Center, University of Luxembourg,
Luxembourg and Certus Software V & V
Center,
Simula Research Laboratory, Norway
lionel.briand@uni.lu

ABSTRACT

System testing of real-time embedded systems (RTES) is a challenging task and only a fully automated testing approach can scale up to the testing requirements of industrial RTES. One such approach, which offers the advantage for testing teams to be black-box, is to use environment models to automatically generate test cases and oracles and an environment simulator to enable earlier and more practical testing. In this paper, we propose novel heuristics for search-based, RTES system testing which are based on these environment models. We evaluate the fault detection effectiveness of two search-based algorithms, i.e., Genetic Algorithms and (1+1) Evolutionary Algorithm, when using these novel heuristics and their combinations. Preliminary experiments on 13 carefully selected, non-trivial artificial problems, show that, under certain conditions, these novel heuristics are effective at bringing the environment into a state exhibiting a system fault. The heuristic combination that showed the best overall performance on the artificial problems was applied on an industrial case study where it showed consistent results.

Categories and Subject Descriptors

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

Keywords

Automated model-based testing, real-time embedded systems, search-based software engineering, UML, branch distance.

1. INTRODUCTION

Real-time embedded systems (RTES) are part of a vast majority of computing devices available today. They are widely used in critical domains where high system dependability is required. These systems typically work in environments comprising of large numbers of interacting components. The interactions with the environment can be bound by time constraints. For example, if a gate controller RTES on a railroad intersection is informed by a sensor that a train is approaching, then the RTES should command the gate to close before the train reaches it. Missing such time deadlines, or missing them too often for soft real-time systems, can lead to serious failures leading to threats to human life or the environment. There is usually a great number and variety of stimuli from the RTES environment with differing patterns of arrival times. Therefore, the number of possible test cases is usually very large if not infinite. Testing all possible sequences of stimuli is not feasible. Hence, systematic automated testing strategies that have high fault revealing power are essential for effective testing of industry scale RTES. The system testing of a RTES requires interactions with the actual environment. Since, the cost of testing in actual environments tends to be high, environment simulators are typically used for this purpose.

In our earlier work, we proposed an automated system testing approach for RTES software based on environment models [1, 2]. The models are developed according to a specific strategy using the Unified Modeling Language (UML) [3], the Modeling and Analysis of Real-Time Embedded Systems (MARTE) profile [4] and our proposed profile for environment modeling [5]. These models of the environment were used to generate an environment simulator [6], test cases, and obtain test oracle [1, 2]. We applied various testing strategies to generate test cases, including search-based strategies, which turned out not to work very well as even Random Testing (RT) fared better.

In our context, a test case is a sequence of stimuli generated by the environment that is sent to the RTES. If a user interacts with the RTES, then she would be considered part of the environment as well. A test case can also include changes of state in the environment that can affect the RTES behavior. For example, with a certain probability, some hardware components might break, and that affects the expected and actual behavior of the RTES. A test case can contain information regarding when and in which order to trigger such changes. So, at a higher level, a test case in our context can be considered as a setting specifying the occurrence of all these environment events in the simulator. Explicit “error” states in the models represent states that should never be reached if the RTES is correct. If any of these error states is reached, then it implies a faulty RTES. Error states act as the oracle of the test cases, i.e., a test case is successful in triggering a fault in the RTES if an error state of the environment is reached during testing.

In this paper, we further extend the fitness function proposed in [1] to improve the disappointing results we had obtained with search-based testing. For this purpose, we present four new heuristics that are aimed to exploit potentially useful characteristics of the environment models. We evaluate the fault detection effectiveness of the new heuristics and their combinations by first performing a series of experiments on 13 artificial RTES that we developed based on the specifications of two industrial case studies. For all heuristics, we used two search algorithms: Genetic Algorithms (GA) and (1+1) Evolutionary Algorithms (EA). We also ran RT on the same problems as a comparison baseline. We then ran the heuristic combination that on average showed best results for the artificial problems on an industrial case study of a marine seismic acquisition system, which was developed by a company leading in this industry sector. We only ran the best combination because executing test cases on the industrial case study is very time consuming and we could not, for technical reasons, run it on a cluster. We compared the fault detection effectiveness of RT and this heuristic combination when used with GA and (1+1)EA on the industrial case study.

The rest of the paper is organized as follows: Section 3 discusses related work. Section 4 provides an introduction to the earlier proposed environment modeling methodology and testing approach. Section 5 discusses the new search heuristics, whereas Section 6 discusses the empirical study carried out to evaluate the new search heuristics. Finally, Section 7 concludes the paper.

2. BACKGROUND

Several software engineering problems can be reformulated as a search problem, such as test data generation [7]. An exhaustive evaluation of the entire search space (i.e., the domain of all possible combinations of problem variables) is usually not feasible. There is a need for techniques that are able to produce "good'' solutions in reasonable time by evaluating only a tiny fraction of the search space. Search algorithms can be used to address this type of problem. Several successful results by using search algorithms are reported in the literature for many types of software engineering problems [8].

To use a search algorithm, typically a fitness function needs to be defined. The fitness function should be able to evaluate the quality of a candidate solution (i.e., an element in the search space). The fitness function is problem dependent, and proper care needs to be taken for developing adequate fitness functions. The fitness function will be used to guide the search algorithms toward fitter solutions. Eventually, given enough time, a search algorithm will find a satisfactory solution.

There are several types of search algorithms. Genetic Algorithms (GAs) are the most well-known [8], and they are inspired by the Darwinian evolution theory. A population of individuals (i.e., candidate solutions) is evolved through a series of generations, where reproducing individuals evolve through crossover and mutation operators. (1+1) Evolutionary Algorithm (EA) is simpler than GAs, in which only a single individual is evolved with mutation. To verify that search algorithms are actually necessary because they address a difficult problem, it is a common practice to use Random Search (or Random Testing (RT) for testing problems) as a comparison baseline [8].

To cope with several problems related to combining together different heuristics/objectives with different priorities, we rather use an *order* function *h.* An order function takes two solutions as parameters and returns whether the first is better, equivalent, or worse than the second solution (e.g., by returning 1, 0, and -1 respectively). In a search algorithm, an order function *h* can always replace a fitness function *f* as long as the raw fitness values are not used besides comparing solutions' fitness. For example, *h* can be used in a GA using tournament or rank selection, but not for fitness proportional selection. For more details, examples and discussions regarding order functions for search algorithms in software testing can be found in [9].

3. RELATED WORK

Depending on the goals, testing of RTES can be performed at different levels: model-in-the-loop, hardware-in-the-loop, processor-in-the-loop, and software-in-the-loop [10]. Our approach can be considered as software-in-the-loop testing, in which the embedded software is tested on the development platform (e.g., Linux, Windows, or Solaris-based machines) with a simulated environment. The only variation is that, rather than simulating the hardware platform, we use an adapter for the hardware platform that forwards the signals from the system under test (SUT) to the simulated environment. This helps focus on testing the embedded software. This approach is especially helpful when the software is to be deployed on multiple hardware platforms or the hardware platform it is deployed on is stable (such as the case with our industry partners, working in the area of marine seismic acquisition and automated bottle recycling machines).

A large body of research has been carried out for RTES testing. Due to space constraints, here we only discuss a brief overview of the literature. Most of these approaches are based on testing the violation of timing constraints [11] or checking their conformance to a specification [12]. The specification is generally a formal model of the system and this model is then used to generate the test cases. As specification of the system, a number of approaches use Timed Automata or one of its extensions (e.g., [13], [14]). For the same purpose, UML statechart [15], Extended Finite State Machines [16] and Attributed Event Grammar [17] have also been used. There are also several works using search-based testing techniques for testing different aspects of RTES, as for example identify deadline misses [18] and testing functional properties [19]. Most of the work on search-based software testing has been focused on unit testing [20], and not system level testing as we do in this paper.

There are also a few works discussing RTES testing based on environment models rather than system models. Auguston *et al.* [17] discusses the development of environment behavioral models using an event grammar for testing of RTES. The behavioral models contain details about the interactions with the SUT and possible hazardous situations in the environment. Heisel *et al.* [21] propose the use of a requirement model and an environment model along with the model of the SUT for testing. Adjir *et al.* [22] discuss a technique for testing RTES based on the system model and assumptions in the environment using Labeled Prioritized Timed Petri Nets. Larsen *et al.* [23] propose an approach for online RTES testing based on time automata to model the SUT and environmental constraints. Peleska *et al.* [24] present a benchmark model for testing RTES in the automotive domain. Their testing methodology uses information from environment models and system models to obtain test cases.

The work presented here is significantly different from most the above approaches as we adopt, for practical reasons, a black-box approach to system testing that relies exclusively on modeling the RTES environment rather than its internal design properties. This is of practical importance as independent system test teams usually do not have easy access to precise design information. Most existing works do not focus on system testing, hence their emphasis is on modeling the RTES internal behavior and structure. Another difference of practical importance, though this is not in the focus of this paper, is that we use UML and its standard extensions for modeling the environment [5].

4. ENVIRONMENT MODELING AND ENVIRONMENT MODEL-BASED TESTING

This section introduces our previous work on which we build in this paper.

4.1 Environment Modeling & Simulation

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

The environment models consist of a domain model and several behavioral models. The domain model captures the structural details of the RTES environment, such as the environment components, their relationships, and their characteristics. The behavior of the environment components is captured by state machines. These models are developed, based on our earlier proposed methodology by using UML, MARTE, and our proposed profile for environment modeling [5]. To minimize modeling effort, the methodology aims at capturing only the details in the environment which are visible and relevant to the SUT. This not only includes the nominal functional behavior of the environment components (e.g., booting of a component) but also includes their robustness (failure) behavior (e.g., break down of a sensor). The latter are modeled as *failure* states in the environment models. The environment behavioral models also capture what we call *error* states. These are the states of the environment that should never be reached if the SUT is implemented correctly (e.g., no incorrect or untimely message from the SUT to the environment component). Error states act as oracles for the test cases. For example, consider a system under test that controls a physical gate on a railroad intersection. The gate should always be down whenever a train is reaching the intersection and should be raised in other situations. The various trains approaching the intersection and the gate will together compose the environment of the SUT. The domain model will comprise of a train component, a gate component, and the SUT. A state machine each for the train and gate components will specify their behavior. A possible failure state can for example be when the physical gate is stuck in a position (in which case the trains should be stopped before reaching the intersection) and a possible error state can be the situation when a train arrives at the gate while it is still open.

An important feature of these environment models is that they capture the non-determinism of the environment, which is a common characteristic for RTES environments (for example, the time it takes to change a gate position can have a variation of few seconds). Non-determinism may include, for example, different occurrence rates and patterns of signals, failures of components, or user commands. The environment modeling profile provides special constructs to model non-deterministic behavior of the environment. Each environment component can have a number of non-deterministic choices whose exact values are selected at the time of testing. Java is used as an action language and OCL (Object Constraint Language) is used to specify constraints and guards. 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). Such a communication layer is written by the software engineer separately from the models. This allows for the simulators and models to be independent of the language in which SUT is written.

The environment 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 [6]. During simulation a number of instances for each environment component (modeled in the domain model) can be created, which interact with each other and the SUT (for example multiple instances of a train component). The generated simulators are linked with the test framework that provides the appropriate values for each simulation execution. For all our case studies, the generated simulators communicated with the SUT using TCP sockets. The choice of Java and TCP is based on actual requirements of one of our industrial partners, where the RTES under study involves soft real-time constraints. Sensors and actuators are geographically distributed on large distances and communicate through a complex network, where delays of a few milliseconds are acceptable.

4.2 Environment Model-Based Testing

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, during a simulation there can be multiple instances for each of the environment components and multiple components run in parallel to form the RTES environment (for example, the train component for the gate controller RTES represent trains in general). During simulation, we can then simulate multiple trains where each simulated train will be represented by an independent running instance of the train environment component. During the simulation, values are required for the non-deterministic choices in the environment models. A test case in our context provides information for both the number of instances for each component (which we refer to as the environment configuration) and the values for various non-deterministic choices (referred to as the simulation configuration). For the scope of this paper, we only consider one fixed environment configuration; therefore in the rest of the paper, a test case is alternatively used for referring to a simulation configuration.

A test case can be seen as a test data matrix, where each row provides a series of values for a non-deterministic choice of the environment component (the number of rows is equal to the number of non-deterministic choices). Each time a non-deterministic choice needs to be made, a value from the corresponding matrix row is selected. During simulation, a query for a non-deterministic choice can be made several times, and so the actual number of queries cannot be determined before simulation. To resolve this problem, each matrix row (a data vector) can be represented in two possible forms: a fixed length ring or a variable length vector. On one hand, in the fixed-length ring vector, the vector is considered as a ring and upon reaching the end/tail of the vector. Then, the values are again selected from the start/head of the vector. On the other hand, in the variable size vector, whenever the end of a vector is reached, its size is increased at run time and new values are added. In our earlier work [2], we evaluated the effect of the representations and starting lengths of the test data vectors on the fault detection effectiveness.

In [1], we also applied various testing strategies to generate test cases from the environment models. For search-based testing, we developed a new fitness function *f* that can be seen as an extension of the fitness function developed for model-based testing based on system specifications [25]. The original fitness function uses the so-called “approach level” and normalized “branch distance” to evaluate the fitness of a test case, as further described below. For environment model-based testing, we introduced the novel concept of normalized “time distance”.

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

The approach level can be helpful to reward test case executions that get closer to an error state, but it does not provide any gradient (guidance) to solve the possible guards on the state transitions. The branch distance (B) is used to heuristically score the evaluation of the guards (if any) on the outgoing transitions from the closest executed state. In [26] we have defined a specific branch distance function for OCL expressions that is reused here for calculating the branch distance. In the dummy state machine in Figure 1, we need to solve the guard *"y > 0"* so that whenever *e4* is triggered, then the simulation can transition to the *Error* state. Note that branch distance is less important than approach level, since it is required only when the transition towards an error state is guarded and the approach level cannot be reduced any further. Therefore, we normalize the branch distance in the range of 0 to 1 [9].

The third important part of the fitness function is the time distance (T), which comes into play when there are timeout transitions in the environment models. For example, in Figure 1, the transition from *State2* to *Error* state is a timeout transition. If a transition should be taken after *z* time units, but it is not, we calculate the maximum consecutive time *c* the component stayed in the source state of this transition (e.g., *State2* in the dummy example). To guide the search, we use the following heuristic: $T = z - c$, where $c \leq z$. Again, the importance of time distance is less than that of approach level, therefore it is normalized in the range 0 to 1. The fitness function *f* using these three heuristics for a test data matrix *m* is defined as:

$$f(m)= min_e\,((A_e(m) + nor(T_e(m)) + nor(B_e(m))) \qquad (1)$$

where for an error state *e*, A_e represents the approach level, T_e represents the time distance, and B_e represents the branch distance. *nor()* is the normalizing function. For guarded time transitions, B_e was only calculated after the corresponding time event was triggered. Since, there can be multiple error states in the environment models, the function *f(m)* only takes the minimum value over all error states (represented by min_e in (1)).

The results when using this fitness function, as reported in [1], were disappointing. The branch distance was calculated for the guards only after an event was triggered and this worked fine for signal events. But for time events, this meant that to get the branch distance, we first needed to trigger the time event. For this we focused first on reducing the time distance and then calculated the branch distance. It turned out that this assumption of favoring reduction of time distance whenever there is a time transition was naive. In some situations where the time transition had a guard, a test case with less time distance but with a greater branch distance was considered to be better than a test case with greater time distance but lower branch distance. However, there is no purpose in reducing the time distance (i.e., the error state will not be reached) if at the end the transition is not fired because the guard is false.

5. IMPROVED FITNESS FUNCTION

In this section, we present novel improvements in the fitness function *f* for environment model-based testing of RTES. As mentioned earlier, for problems related to combining various heuristics/objectives with different priorities, we can replace the use of a fitness function *f* with an order function *h*. For two test data matrices m_1 and m_2, the function *h* will return 1, 0, or -1 if m_1 is better, equal, or worse than m_2, respectively.

Following, based on *f(m)* we define a basic order function *h* for two test data matrices (m_1, m_2) that will be reused for definition of order functions for the three new heuristics: Time In Risky State (TIR), Risky State Count (RSC), and Coverage (COV).

$$h(m_1,m_2)=\begin{cases} \textbf{1} & \textbf{if } A_{min}(m_1)<A_{min}(m_2) \textbf{ or } (A_{min}(m_1)=A_{min}(m_2) \text{ and } B_{min}(m_1) < B_{min}(m_2)) \textbf{ or } (A_{min}(m_1)=A_{min}(m2) \text{ and } B_{min}(m_1)= B_{min}(m_2) \text{ and } T_{min}(m_1) < T_{min}(m_2)) \\ \textbf{0} & \textbf{if } A_{min}(m_1)=A_{min}(m_2) \textbf{ and } B_{min}(m_1)= B_{min}(m_2) \textbf{ and } T_{min}(m_1)=T_{min}(m_2)) \\ \textbf{-1} & \textbf{otherwise} \end{cases} \qquad (2)$$

where for a set of error states *es*, $A_{min}(m)$ is defined as the minimum approach level for the matrix *m* over *es*, $B_{min}(m)$ as the minimum branch distance for *m* over *es*, and $T_{min}(m)$ as minimum time distance for *m* over *es*. A_{min} takes precedence on B_{min} and $T_{min,}$ and B_{min} takes precedence on T_{min}. This is simply reflecting the relative importance of these three heuristics.

5.1 Improved Time Distance (ITD)

We improved the way the basic time distance was calculated in the earlier fitness function. The motivation behind the improved time distance is that to avoid fitness plateaus, a test case with a lower branch distance for a time transition should be preferred over the one having greater branch distance, irrespective of the time distance. This is due to the fact that during environment simulation, changing the values of a test case often has a direct impact on the time distance and it should therefore be easier to reduce it than the branch distance. For example in Figure 1, the time transition *after "t, s"* is guarded by [x > 0]. A test case with a positive value greater than 0 for *x* will be considered better than a test case with a negative or 0 value for *x*, irrespective of the value of *t*. The value of *t* is considered only after the branch distance of the guard equals 0. For this, we introduced the concept of a look-ahead branch distance (LB) for time transitions, which represents the branch distance of OCL guard on a time transition when it is not fired (i.e., the timeout did not occur). Because OCL evaluations are free from side-effects [26], this does not lead to any particular problem. The order function for two test data matrices m_1 and m_2 using this heuristic is:

$$h(m_1,m_2)=\begin{cases} \textbf{1} & \textbf{if } A_{min}(m_1)<A_{min}(m_2) \textbf{ or } (A_{min}(m_1)=A_{min}(m_2) \text{ and } B_{min}(m_1) < B_{min}(m_2)) \textbf{ or } (A_{min}(m_1)=A_{min}(m2) \text{ and } B_{min}(m_1)= B_{min}(m_2) \text{ and } ITD_{min}(m_1,m_2) = 1) \\ \textbf{0} & \textbf{if } A_{min}(m_1)=A_{min}(m_2) \textbf{ and } B_{min}(m_1)= B_{min}(m_2) \textbf{ and } ITD_{min}(m_1, m_2)=0) \\ \textbf{-1} & \textbf{otherwise} \end{cases} \qquad (3)$$

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

where for the set of error states *es* and a given error state $e \in es$, $A_{min}(m)$ represents the minimum approach level for matrix *m* over *es*, $B_{min}(m)$ is the minimum branch distance for *m* over *es*, $LB_e(m)$ represents the look-ahead branch distance for matrix *m* for the error state *e*, and $T_e(m)$ represents the time distance for *m* over *e*.

5.2 Time in Risky State (TIR)

A "risky state" is defined as a state adjacent to the error state (i.e., approach level = 1). For the order function, when two test cases have the same A_{min}, B_{min}, and T_{min}, then a test case that spends more time in risky states should have higher fitness. The motivation behind this heuristic is that, the more time spent in a risky state, the higher the chances of events happening in the environment or SUT leading to the error state (e.g., receive a signal from the SUT). For example, for the state machine shown in Figure 1, this heuristic will favor the test cases that spend more time in the risky states *State2* or *State5*. For instance in *State2*, it is possible to increase the value of *t1* in the time event *after "t1, ms"*, which will increase the time spent in this state. The overall order function based on *h* defined in (2), is given as:

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

where $TIR_{sum}(m)$ is the sum of time spent in risky states for all error states and the test data matrix *m*.

5.3 Risky State Count (RSC)

This heuristic is also based on utilizing the concept of risky states: When two test cases have the same A_{min}, B_{min}, and T_{min}, then a test case that enters a risky state more often should be preferred over a test case that does so less often. For example, for the state machine shown in Figure 1, this heuristic will assign higher fitness to the test cases that make the component enter *State2* more often, i.e., transitions to *State4* and come back. This would for instance result in minimizing the values of *t1* and *t2* for the timeout transitions *after "t1,s"* and *after "t2,s"* to increase the risky state count. Note that the heuristic will only be useful for the cases that allow a loop back to a risky state. The overall order function based on the basic order function *h* defined in (2) is:

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

where $RSC_{sum}(m)$ is total count of transitions made to all risky states for the test data matrix *m*.

5.4 Increase in Coverage (COV)

This heuristic is based on the concept of coverage of environment models. This heuristic, when two test cases have the same A_{min}, B_{min}, and T_{min}, calculates the environment coverage and assign higher fitness to the test cases that cover more environment states. The idea behind this heuristic is to increase the coverage of the environment models when the approach level, branch distance and time distance can no longer be improved. The assumption is that having higher environment coverage will result in more diversity in the test cases, which might lead to situations that help reach the error state. For example in Figure 1, this heuristic will favor a test case that visited *State4* over a test case that did not. The idea is to

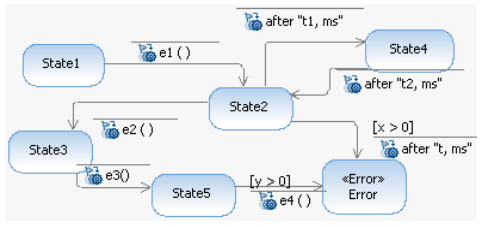

Figure 1. A dummy state machine to explain search heuristics

explore more states and transitions in the environment models. The overall order function for *COV* based on *h* (2) is:

$$h'(m_1,m_2)=\begin{cases} \mathbf{-1} & \textbf{otherwise} \\ \boldsymbol{h(m_1, m_2)} & \textbf{if } \mathrm{h}(m_1, m_2) \mathrel{!}= 0 \\ \mathbf{1} & \textbf{if } h(m_1, m_2) = 0 \textbf{ and } \mathrm{COV_{min}}(m_1) > \mathrm{COV_{min}}(m_2) \\ \mathbf{0} & \textbf{if } h(m_1, m_2) = 0 \textbf{ and } \mathrm{COV_{min}}(m_1) = \mathrm{COV_{min}}(m_2) \end{cases}$$

where $COV_{sum}(m)$ is the total coverage for all error states.

5.5 Combination of heuristics

Apart from the individual heuristics, we also investigate their combinations. In total, for the latter three heuristics (*TIR, RSC*, and *COV*) there are eight possible combinations. They can be combined with the basic order function *h* and an order function containing the improved time distance ITD instead of T in *h*, which results in a total of 16 possible combinations

$$h'(m_1,m_2)\begin{cases} \boldsymbol{h(m_1, m_2)} & \textbf{if } h(m_1, m_2) \mathrel{!}= 0 \\ \mathbf{1} & \textbf{if } h(m_1, m_2) = 0 \textbf{ and } \mathrm{comb}\,(m_1) > \mathrm{comb}(m_2) \\ \mathbf{0} & \textbf{if } h(m_1, m_2) = 0 \textbf{ and } \mathrm{comb}\,(m_1) = \mathrm{comb}(m_2) \end{cases}$$

where *comb(m)* is a given combination of the heuristics.

When combining these heuristics, we follow the Pareto dominance principle - a key concept for multi-objective optimization in evolutionary algorithms [27]. In our context this means that, given a combination of heuristics, a test data matrix m_1 will dominate another matrix m_2, if it is better than m_2 for at least one heuristic and is not worse than m_2 in any of the other heuristics. The reasons for using a Pareto dominance is that, in contrast to approach level and branch distance, we do not know which is the most important heuristic among the three that were proposed: this is a research question that we address in this paper.

Table 1. Summary of environment models*

Problem	GtP	ToP	LtR	GET	TtE	Approach
AP1	Yes	Yes	No	Yes	Yes	Non-trivial
AP2	Yes	Yes	No	Yes	Yes	Non-trivial
AP3	No	No	Yes	No	No	Trivial
AP4	No	Yes	No	No	Yes	Non-trivial
AP5	No	Yes	No	No	Yes	Non-trivial
AP6	No	Yes	No	No	Yes	Non-trivial
AP7	Yes	Yes	Yes	Yes	Yes	Non-trivial
AP8	Yes	Yes	Yes	Yes	Yes	Non-trivial
AP9	Yes	Yes	Yes	Yes	Yes	Non-trivial
AP10	Yes	Yes	No	Yes	Yes	Trivial
AP11	Yes	Yes	No	Yes	Yes	Trivial
AP12	Yes	Yes	No	Yes	Yes	Trivial
AP13	Yes	Yes	Yes	Yes	Yes	Trivial
IC	Yes	Yes	Yes	Yes	Yes	Trivial

* GtP = Guard on Path, ToP = Time Transition on Path, LtR = Loop to risky state, GET = Guard on error transition, TtE = Time transition to error state, Approach = Approach to risky state

Table 2. Success rates of various heuristic for GA & EA

Problem	Basic		ITD		TIR		RSC		COV	
	GA	EA	GA	EA	GA	EA	GA	EA	GA	EA
AP1	0.3	0	0.05	0.15	0.9	1	0.2	0.05	0.4	0
AP2	0.65	0.3	0.5	0.5	1	1	0.65	0.55	0.6	0.25
AP3	0.75	0.65	0.8	0.45	0.6	0.4	0.9	1	0.45	0.45
AP4	0.4	1	0.5	0.9	0.5	0.9	0.45	0.9	0.5	1
AP5	0.9	1	0.95	1	1	1	0.95	1	0.95	0.95
AP6	0	0.55	0.05	0.6	0.05	0.95	0.05	0.5	0.05	0.6
AP7	0.65	0.5	0.85	0.9	0.65	0.75	0.4	0.35	0.5	0.15
AP8	1	0.9	1	0.9	1	1	0.95	0.9	0.95	0.3
AP9	0.15	0.1	0.15	0.55	0	0.3	0	0.05	0	0.05
AP10	0.55	0.75	0.75	0.8	0.6	0.7	0.65	0.45	0.65	0.45
AP11	0.25	0.25	0.3	0.1	0.25	0.05	0.25	0	0.15	0.1
AP12	0.95	1	1	1	0.85	0.9	0.95	0.85	1	0.75
AP13	1	0.9	1	0.85	1	0.9	1	0.95	0.85	0.15
Average	0.58	0.61	0.61	0.67	0.65	0.76	0.57	0.58	0.54	0.4

6. EMPIRICAL STUDY

The objective of this empirical study is to evaluate the effectiveness, in terms of fault detection, of the proposed heuristics and their combinations. We selected two search algorithms for this empirical study: Genetic Algorithms (GA) and (1+1)Evolutionary Algorithm (EA). Though (1+1) EA is simpler than GA, it has shown better results in our previous testing works (e.g., [26]). We use the convention *Algorithm-Heuristic* to denote an algorithm using a heuristic or its combination. For example, to denote that GA is used with the basic fitness function defined in (1), we use the terms GA-Basic.

6.1 Case Study

For the sake of experimenting with diverse environment models and RTES, we developed 13 different artificial RTES that were inspired by two industrial cases we have been involved with [2] and one case study discussed in the literature [16]. Since, there are no benchmark RTES available to researchers, we specifically designed these artificial problems to conduct our experiments (called AP1 – AP13). The goal while developing the models of these RTES was to vary various characteristics of the environment models (e.g., guarded time transitions, loops) in order to evaluate the impact of these characteristics on the test heuristics. Nine of these artificial problems were inspired by a marine seismic acquisition system developed by one of our industrial partners. These problems covered various subsets of the environment of the industrial RTES. Three of the 13 problems were inspired by the behavior of another industrial RTES (part of an automated recycling machine) developed by another industrial partner. The thirteenth artificial problem was inspired by the train control gate system described in [16].

The industrial case study we also report on is a very large and complex seismic acquisition system that interacts with several sensors and actuators. The timing deadlines on the environment are in the order of hundreds of milliseconds. The company that provided the system is a market leader in its field. For confidentiality reasons we cannot provide full details of the system. The SUT consists of two processes running in parallel, requiring a high performance, dedicated machine to run. A summary of the characteristics of the environment models for the SUT is provided in Table 1 (row with problem IC).

To facilitate the discussion of our results, a summary of relevant characteristics for the environment models of the RTES under study is provided in Table 1. The columns 'Guard to path' (GtP) and 'Time transition on path' (ToP) represent whether these features were present on a path to the error state. The column 'Loop to risky state' (LtR) reports whether there was a loop back to a risky state (i.e., an outgoing transition to a state and then returning back to the risky state). The columns 'Guard to error transition' (GET) and 'Time transition to error' (TtE) show whether these features were present on the transition from the risky state to the error state. The column 'Approach 'shows if the approach to the risky state (i.e., obtaining a test case in which the closest executed state is the risky state) is trivial or not. It is considered to be trivial if a risky state is reached on average by the first ten randomly executed test cases.

These RTES are written in Java to facilitate their use on different machines and operating systems. The communication between the RTES and their environments is carried out through TCP. All these RTES are multithreaded. Each of the artificial problems had one error state in their environment models and non-trivial faults were introduced by hand in each of them. We could have rather seeded those faults in a systematic way, as for example by using a mutation testing tool [28]. 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 with the industrial RTES and from the feedback of our industry partners. For the industrial case study, we did not seed any fault and the goal was to find the real fault that we initially uncovered in [1].

6.2 Experiments

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

RQ1: What is the effect on fault detection of new order functions having each one of the proposed heuristics: Improved Time Distance (ITD), Time In Risky State (TIR), Risky State Count (RSC), and Coverage (COV) compared to the previously defined basic fitness function for GA and (1+1) EA? **RQ2:** Which combinations of the proposed heuristics are best in terms of fault detection? **RQ3:** Between the two search-based algorithms, GA and (1+1) EA, which one works better in terms of fault detection with the new heuristics? **RQ4:** How do the search-based algorithms compare to random testing (RT)? **RQ5:** How does the best combination of the proposed heuristics compare to RT, GA-Basic, and (1+1) EA-Basic on the industrial case study?

To answer the research questions RQ1 – RQ4, we carried out a series of experiments on the above-mentioned thirteen artificial problems with (1+1) EA, GA, and RT. The latter was selected as a comparison baseline as it is the simplest solution to implement. For GA, we employ rank selection with bias 1.5 to choose the parents, the initial population size is 10 and a single point crossover is used with probability $P_{xover} = 0.75$. Different settings of these parameters could lead to different performance, but we selected reasonable parameter values following recommendations in the GA literature [29]. For test case representation we used a dynamic representation with a length equal to 10 for the test cases (which correspond to each row of the test data matrix m). In our earlier experiments this setting showed the best results [2].

Table 3. Results of ITD compared with basic fitness function

Problem	GA-ITD vs. GA	EA-ITD vs. EA
AP7	-	**p=0.0138, ψ =7.4**
AP9	-	**p=0.0057, ψ =8.96**

Table 4. Results of TIR compared with basic fitness function

Problem	GA-TIR vs. GA	EA-TIR vs. EA
AP1	**p= 0.00024, ψ = 16.51**	**p=1.45e-11, ψ =1681**
AP2	**p= 0.00832, ψ = 22.78**	**p=3.34e-06, ψ = 91.46**
AP4	-	**it-p = 0.00167, A_{12} = 0.8**
AP6	-	**p=0.00836, ψ = 10.74**
AP7	*it-p= 0.03125, A_{12} = 0.24*	-
AP13	-	**it-p= 0.02677, A_{12} = 0.7**

Table 5. Results of RSC compared with basic fitness function

Problem	GA-RSC vs. GA	EA-RSC vs. EA
AP3	-	**p=0.00831, ψ =22.78**
AP11	-	*p=0.047, , ψ =22.78*
AP13	**it-p= 0.0073, A_{12} = 0.74**	-

Table 6. Results of COV compared with Basic fitness function

Problem	GA-COV vs. GA	EA-COV vs. EA
AP7	-	*p = 0.0407, ψ = 5*
AP8	-	*p = 0.0002, ψ =16.5*
AP12	-	*p = 0.0471, ψ =14.5*
AP13	-	*p = 3.36e-06, ψ =37*

For the experiments, we ran RT, GA, (1+1) EA on each of the thirteen problems. We have three order functions for the individual heuristics and can combine them in 12 different ways (as described in Section 5.5). We ran these fifteen combinations with both the basic order function and an order function using ITD. We compared these order functions with the basic fitness function (defined in (1)) and the order function for ITD (defined in (3)). In total we therefore executed 2 * (8 * 2) *13 + 13 = 429 experiment configurations (two search algorithms, 16 order functions, 13 artificial problems, on which RT is also run). The execution time of each test case was fixed to 10 seconds and we stopped each algorithm after 1000 sampled test cases or as soon as we reached any of the error states. The choice of running each test case for 10 seconds was based on the properties of the RTES and the environment models. The objective was to allow enough time for the test cases to reach an error state. For each of these 429 experiment configurations, we ran each algorithm 20 times with different random seeds. The total number of sampled test cases was 7,676,635, which required around 888 days of CPU resources. Therefore, we performed the experiments on a cluster of computers.

To answer the research question RQ5, we carried out experiments on the industrial case study. We run each test case for 60 seconds, where 1000 test case executions (fitness evaluations) can take more than 16 hours. This choice has been made based on the properties of the RTES and discussions with the actual testers. Due to the large amount of resources required, we only ran the combination of heuristics that on average gave best results for the thirteen artificial problems. We compared its fault detection rate with that of GA-Basic, (1+1) EA-Basic, and RT. We carried out 20 runs for each of these four experiment configurations. The total number of sampled test cases was 39,948, which required over 27 days of computation on a single, high-performance, dedicated machine.

To analyze the results, we used the guidelines described in [30] which recommends a number of statistical procedures to assess randomized test strategies. First we calculated the success rates of each algorithm: the number of times it was successful in reaching the error state out of the total number of runs. These success rates are then compared using the Fisher Exact test, quantifying the effect size using an odds ratio (ψ) with a 0.5 correction (p-values of this test are denoted as *p* in the tables showing the results). When the differences between the success rates of two algorithms were not significant, we then looked at the average number of test cases that each of the algorithms executed to reach the error state. We used the Mann-Whitney U-test and quantified the effect size with the Vargha-Delaney A_{12} statistics (p-values of this test are denoted as *it-p* in the tables showing the results). The significance level for these statistical tests was set to 0.05. In all the tables showing the odds ratio and A_{12} statistics when comparing two algorithms, say *q* and *r*), a bold-faced font shows that *q* is significantly better than *r* and an italicized font shows that *q* shows significantly worse performance than *r*. Table cells with a '-' denote no significant results for the comparison.

6.3 Results and Discussion

We decompose RQ1 into four sub questions (RQ1a - RQ1d), one for each heuristic. Table 2 shows the success rates for the 13 artificial problems and the four heuristics with GA and (1+1) EA.

Results when applying ITD (RQ1a) to the artificial problems with GA and EA are shown in Table 3 and are compared with results obtained when using the basic fitness function. The table shows the p-values and odds ratio when success rates were significantly different and otherwise, the p-value and the A_{12} statistics on the difference in the number of test case executions to reach the error state. Using ITD with (1+1) EA yields significantly better results for two of the artificial problems. In other cases the performance of the algorithm with this order function was the same as that for the basic algorithm. ITD relies on information regarding guarded time transitions in the models. Among the thirteen artificial problems, AP3 – AP6 did not have any guard or time transition leading to the error state. Even in these cases, ITD shows similar performance to basic fitness with no significant drawbacks. To answer RQ1a, using the fitness function with ITD can bring improvements in fault detection effectiveness for (1+1) EA and has no significant difference when used with GA.

Turning now to Table 4, when TIR was used with GA (RQ1b), it gave significantly better results in two of the artificial problems and was worse in one problem (AP7). For other artificial problems, the results of the two algorithms were comparable. When TIR was used with (1+1) EA, it gave significantly better results for five of the 13 artificial problems. In other cases there were no significant differences. To answer RQ1b, TIR performs better or similar to the basic fitness for all but one of the artificial problems, whereas the performance of TIR with EA is better or equal to the (1+1) EA-Basic in all the cases. Hence the use of TIR in the order function seems to be an effective option in most cases.

Table 5 addresses RQ1c and evaluates the RSC heuristic. When RSC was used with GA, it gave significantly better results in one of the artificial problems (AP13) and showed no significant difference for the other artificial problems. When RSC was used with (1+1) EA, it gave significantly better results for one artificial problem (AP3), worse results for another one (AP11), and no statistical differences otherwise. RSC depends on the presence of a loop back to a risky state. According to the information in Table 1, AP3, AP7, AP8, AP9 and AP13 had a loop back to the risky state. Hence, we can answer RQ1c by stating that for all the problems that have a loop to risky states, an order function using the RSC heuristic performs significantly better or similar to the basic fitness function. But for the problems without such a loop, it can negatively affect performance. Table 6 addresses RQ1d and evaluates the Coverage (COV) heuristic. When COV was used with GA, there were no statistical differences between the results. When it was used with (1+1) EA, it gave significantly worse results for four of the artificial problems and yielded no significant differences in other cases. To answer RQ1d, using the order function with coverage only can result in significant deterioration in the performance of (1+1) EA.

Next, we answer RQ2, for which we evaluate the various combinations of the four proposed heuristics. As discussed we had a total of 16 possible order functions for each search algorithm. Table 7 provides the relative ranking based on the statistical difference of the compared configurations. Configurations which are statistically equivalent (i.e., p-values above 0.05) are expected to show a similar ranking. This is done by assigning scores based on pairwise comparisons of configurations. Whenever a configuration is better than the other and the difference is statistically significant, its score is increased. Then, based on the final scores, each configuration is assigned ranks ranging from 1 (best configuration) to 32 (worst configuration). In case of ties, ranks are averaged. The configurations in the table are sorted by their average ranking (last column) in an ascending order.

Overall, based on the average ranks for the 13 artificial problems, (1+1) EA with TIR proved to be the best algorithm for both Basic and ITD versions of the heuristic. Analyzing the results of Table 7 according to the characteristics of artificial problem, we can conclude that in general search-based algorithms perform significantly worse than RT for the artificial problems where the approach to risky states is trivial (see discussion for RQ4 and a plausible detailed explanation at the end of this section). If we exclude the results of such artificial problems (i.e., AP3, AP10 – AP13), then in all the other problems, (1+1) EA with ITD and TIR performed significantly better than other combinations. According to the ranks shown, the only exception seems to be AP8, but even in that case, though the number of test case executions is significantly less for other order functions, the success rate of (1+1) EA with both the order functions (Basic-TIR and ITD-TIR) was 100%. If we only consider GA, then the best two algorithms were GA-ITD-TIR and GA-ITD-TIR-RSC. The good overall performance of TIR is likely to be due to the fact that it focuses on making the environment spend more time in the risky states, thus increasing the occurrence of situations that lead to the error state. When we compared the performance of (1+1) EA-Basic-TIR with (1+1) EA-ITD-TIR, there were no significant differences in the results. But looking at the results in Table 7, where for various combinations used with (1+1) EA-ITD and (1+1) EA-Basic, the combinations used with (1+1) EA-ITD showed better or statistically equal results. This also further confirms the findings of RQ1a, which suggested to use (1+1) EA-ITD over (1+1) EA-Basic.

Regarding RQ3 (about the comparison of GA and (1+1) EA), based on Table 7, (1+1) EA seems overall to provide significantly better results with various combinations when compared to GA using the same combinations of heuristics. An exception to this is when EA is used with the coverage heuristic, in which case it performs significantly worse than GA. Even for the problems with non-trivial approach level, the performance of most of the heuristic combinations for EA is significantly better than their performance with GA. Hence, we can conclude that the fault detection effectiveness of (1+1) EA is higher than that of GA for the kind RTES system testing we focus on.

To answer RQ4 (comparison of RT with EA and GA), we compare RT with the heuristic combinations giving the best results for GA and EA. According to RQ3, for (1+1) EA, EA-ITD-TIR and EA-Basic-TIR were the best two combinations and for GA, GA-ITD-TIR and GA-ITD-TIR-RSC were the best two combinations.

Table 8 shows a comparison of RT with these four algorithms. The statistics for the situations where RT is significantly worse than these algorithms are bold faced and the situations where it is significantly better are italicized. It can be observed that for all the artificial problems that have a trivial approach level (Table 1 : AP3, AP10–AP13), RT performs significantly better than both search algorithms. But in other cases, where the approach level is hard, EA and GA perform significantly better. This is especially true for EA who performs better in all the other problems, except AP8. For AP8, over 90% of the heuristics combinations had a 100% success rate and the remaining had a success rate of over 85%. Therefore, AP8 can also be considered to be a simple problem. Hence, we can answer RQ4 by stating that for simple problems (i.e., where the average success rate of all the algorithms is high or the approach level is trivial) RT performs significantly better than both search-algorithms, but for more difficult problems (i.e., lower success rates or non-trivial approach level), search algorithms perform significantly better. The best technique (1+1) EA-ITD-TIR has an average success rate of 73% for the 13 problems with an average number of 222 test case executions to find a fault. If we only consider the problems where approach level was non-trivial (i.e., excluding AP3, AP10 – AP13), then the average success rate is 84%. The worst success rate is 35% (AP9), which suggests that with r runs of the technique, we would achieve a success rate of $1 - (1-0.35)^r$. For example with only five runs ($r = 5$), we would obtain a success rate above 99%.

RQ5 is about comparing the best combination of heuristics with GA-basic, (1+1) EA-Basic and RT on the industrial case study. According to RQ2, the combination showing on average the best results for artificial problems was (1+1) EA-ITD-TIR. Table 9 shows the comparative results of running (1+1) EA-ITD-TIR, (1+1) EA-Basic, GA-Basic, and RT on the industrial case study. Table 10 shows the details of the results of this experiment including the average success rate (SR) and the average number of test case executions to find a fault (ATE). We can see that (1+1) EA-ITD-TIR shows significantly better performance over both GA-Basic and (1+1) EA-Basic. When compared to RT, there is no significant statistical difference. The best combination has relatively lower success rate (0.85 compared to 1 for RT), but it finds the fault with a lower, average number of test case executions (169 compared to 295 for RT). The better performance of RT can be explained by the fact that in the industrial case study, the approach level to risky state was again trivial as shown in Table 1 (i.e., on average it could be reached in less than 10 random test cases).

Following, we provide a plausible explanation as to why RT shows better performance when the approach level to risky state is trivial. The transition from a risky state to the error state represents the erroneous behavior of the SUT and will only be triggered if the interaction of the SUT with the environment was at some point incorrect. Therefore, triggering this transition is dependent on the behavior of the SUT.

Table 7. Rank of each heuristic combination on 13 artificial problems (sorted by average rank)

Algorithm	**AP1**	**AP2**	**AP3**	**AP4**	**AP5**	**AP6**	**AP7**	**AP8**	**AP9**	**AP10**	**AP11**	**AP12**	**AP13**	**Avg.**
(1+1)EA-Basic-TIR	2.5	2.5	31	4	4	1	3	26.5	8.5	17.5	24.5	23	12.5	12.35
(1+1)EA-ITD-TIR	2.5	2.5	23.5	2	3	2.5	2	22	5.5	23.5	30	27	18.5	12.65
GA-ITD-TIR	7	5	25	24.5	21	26	7.5	1	25	10.5	3.5	12	2.5	13.12
(1+1)EA-ITD-TIR-RSC	2.5	2.5	26	1	1	9.5	7.5	19	3.5	22	30	31	20.5	13.54
GA-ITD-TIR-RSC	10	9	3	24.5	21	26	7.5	8	16	26	17.5	4	4	13.58
GA-ITD-RSC-COV	17	18.5	4	24.5	12.5	26	13	4	21	17.5	2	14.5	9	14.12
RT	14.5	9	1	33	33	26	32	8	31	1	1	1	2.5	14.85
(1+1)EA-ITD-TIR-RSC-COV	5	9	22	9	2	9.5	7.5	4	1.5	32	30	32.5	30	14.92
(1+1)EA-ITD-RSC-COV	28	29	21	14	14.5	9.5	4	20	3.5	5	8.5	18.5	22	15.19
GA-ITD-RSC	19.5	22	19.5	24.5	17	26	22.5	12	16	2	5	3	9	15.23
GA-ITD-TIR-COV	23	13.5	16.5	24.5	26	26	10	12	16	7.5	8.5	6	9	15.27
(1+1)EA-ITD	23	27	32	15.5	6	14.5	1	21	1.5	6	17.5	20.5	20.5	15.85
GA-Basic-TIR-RSC	11	9	8	24.5	21	26	17	8	25	20	17.5	10.5	12.5	16.15
(1+1)EA-ITD-RSC	19.5	18.5	12	12	6	14.5	11	24	5.5	20	24.5	24.5	18.5	16.19
GA-Basic-RSC-COV	26.5	24.5	7	24.5	28.5	17.5	22.5	2	16	14.5	3.5	14.5	9	16.19
(1+1)EA-Basic-TIR-RSC	2.5	2.5	29.5	9	9	5	20.5	24	10.5	23.5	30	29	16	16.23
GA-Basic-RSC	23	22	6	24.5	21	26	26	18	31	3.5	8.5	5	1	16.58
(1+1)EA-ITD-TIR-COV	7	13.5	23.5	9	10.5	5	5	24	10.5	33	30	22	29	17.08
GA-Basic	18	22	14.5	24.5	24	26	14.5	8	16	14.5	8.5	17	16	17.19
GA-ITD-COV	26.5	18.5	10	24.5	27	17.5	14.5	15.5	16	14.5	17.5	8	14	17.23
GA-Basic-TIR-COV	13	18.5	5	24.5	30	26	17	14	25	14.5	17.5	14.5	5	17.27
GA-Basic-TIR	7	9	16.5	24.5	31	26	28.5	17	31	10.5	8.5	8	9	17.42
GA-ITD	30	27	12	24.5	28.5	26	12	8	16	10.5	8.5	10.5	16	17.65
GA-ITD-TIR-RSC-COV	23	15.5	14.5	24.5	25	26	20.5	12	16	7.5	17.5	8	24	18.00
GA-Basic-TIR-RSC-COV	14.5	15.5	12	24.5	32	26	24.5	15.5	25	10.5	17.5	14.5	6	18.31
(1+1)EA-Basic-TIR-COV	12	9	9	4	17	5	28.5	29.5	25	26	24.5	27	31	19.04
GA-Basic-COV	16	24.5	28	24.5	21	26	27	4	31	3.5	17.5	2	23	19.08
(1+1)EA-Basic-RSC-COV	23	30	18	13	14.5	16	17	31.5	8.5	26	17.5	20.5	26	20.12
(1+1)EA-Basic-TIR-RSC-COV	9	9	19.5	6	12.5	2.5	30.5	31.5	31	31	30	32.5	28	21.00
(1+1)EA-Basic-RSC	30	27	2	9	10.5	9.5	30.5	26.5	25	29	30	24.5	25	21.42
(1+1)EA-Basic	32.5	31	27	15.5	8	9.5	24.5	29.5	16	30	12	18.5	27	21.62
(1+1)EA-ITD-COV	30	32.5	33	4	17	9.5	19	28	7	20	24.5	27	32.5	21.85
(1+1)EA-Basic-COV	32.5	32.5	29.5	9	6	13	33	33	25	28	17.5	30	32.5	24.73

Table 8. Comparison of RT with best combinations of GA and (1+1)EA on artificial problems*

Problem	**RT vs. GA1**	**RT vs. GA2**	**RT vs. EA1**	**RT vs. EA2**
AP1	**p=0.0012, ψ =15.74**	-	**p = 0.0001, ψ = 49.63**	**p = 0.0001, ψ = 49.63**
AP2	-	-	**it-p = 0.002, A_{12} = 0.2137**	**it-p = 0.0038, A_{12}= 0.2312**
AP3	*p=0.0004, ψ = 41*	*p=0.0202, ψ = 18.38*	*p = 4.5e-05, ψ = 60.29*	*p = 0.0004, ψ = 41.00*
AP4	**0.0202, ψ = 18.38**	**p = 0.0005, ψ = 41**	**p = 3.3e-09, ψ = 303.40**	**p = 1.5e-11, ψ = 1681.00**
AP5	-	-	**p = 0.0083, ψ = 22.78**	**p = 0.0083, ψ = 22.78**
AP6	-	-	**p = 3.0e-10, ψ = 533.00**	**p = 3.3e-06, ψ = 91.46**
AP7	**p = 1.7e-05, ψ = 27.13**	**p = 8.7e-05, ψ = 18.33**	**p = 0.0012, ψ = 10.33**	**p = 8.7e-05, ψ = 18.33**
AP8	-	-	*it-p = 0.0053, A12 = 0.759*	*it-p = 0.0425, A12 = 0.689*
AP9	-	-	**p = 0.0201, ψ = 18.38**	**p = 0.0083, ψ = 22.78**
AP10	*p = 0.0471, ψ = 14.55*	*p = 4.5e-05, ψ = 60.29*	*p = 0.0201, ψ = 18.38*	*p = 0.0001, ψ = 49.63*
AP11	*p = 1.3e-05, ψ = 73.80*	*p = 2.6e-08 , ψ = 205.00*	*p = 3.0e-10, ψ = 533.00*	*p = 1.4e-11, ψ = 1681.00*
AP12	-	-	*it-p = 0.0081, A_{12} = 0.7528*	*p = 0.0202, ψ = 18.38*
AP13	-	-	-	*it-p = 0.0114, A_{12} = 0.738*

* GA1 = GA-ITD-TIR, GA2 = GA-ITD-TIR-RSC, EA1 = EA-Basic-TIR, EA2 = EA-ITD-TIR

Table 9. Comparison of four algorithms on industrial case

Algorithm	**(1+1)EA-Basic**	**(1+1)EA-ITD-TIR**	**RT**	**GA-Basic**
(1+1)EA-Basic	×	ψ= 0.29 , *A_{12}= 0.82*	*ψ= 0.036 , A_{12}= 0.75*	ψ= 1.78 , *A_{12}= 0.82*
(1+1)EA-ITD-TIR	ψ= 3.40 , **A_{12}= 0.18**	×	ψ= 0.12 , A_{12}= 0.31	**ψ= 6.05** , A_{12}= 0.29
RT	**ψ= 27.88 , A_{12}= 0.25**	ψ= 8.20 , A_{12}= 0.69	×	**ψ= 49.63** , A_{12}= 0.47
GA-Basic	ψ= 0.56, **A_{12}= 0.18**	**ψ= 0.17**, A_{12}= 0.71	*ψ= 0.02* , A_{12}= 0.53	×

Table 10. Details of each algorithm on the industrial case*

Algorithm	**SR**	**ATE**	**StD**	**Med**	**Skewness**	**Kurtosis**
(1+1)EA-Basic	0.6	559	270.18	615.5	-0.8	3.03
(1+1)EA-ITD-TIR	0.85	169.24	221.81	89	1.93	5.4
RT	1	295.2	279.1	225	1.24	3.42
GA-Basic	0.45	273.22	186.97	246	0.18	1.88

* SR = Success rate, ATE = average test case executions for each run, StD = standard deviation, Med = Median

Once the environment reaches a risky state and is not able to proceed to the error state, a possible option is to try to maximize the diversity in the environment behavior (e.g., by using entirely different values for the test data matrix, irrespective of their effect on the fitness). Maximizing diversity could result in execution of a behavior of the environment that causes the SUT to interact in an erroneous way which will in turn result in the transition to the error state. When the approach to risky state is trivial then we can simply use RT (or a similar technique) to try to maximize diversity, instead of using a technique like (1+1) EA that generates similar individuals (which makes it hard for search algorithms to be successful in such cases). If this is not the case, then a likely reason for not reaching the risky state is a guard on the transition and/or a time transition. The heuristics for search-based algorithms that we discussed in this paper are specifically designed to deal with these cases and are more suitable for such cases than RT. Our previous results on solving constraints written in OCL, lead us to the conclusion that search-based algorithms are an order of magnitude better than randomized algorithms for this purpose [26]. Hence, if the guard on the transition can be solved by directly changing the values of attributes of the environment components or the transition is a time transition, then our best chance is to use the search algorithms (and more specifically in our context, (1+1) EA-ITD-TIR).

From a practical standpoint, a possible solution to deal with the above mentioned situations that arise due to the nature of environment models is to apply RT at the start of testing and evaluate whether risky states are easy to reach. If this is the case, and if the OCL guard on the transition does not provide gradient (i.e., the so called *flag* problem [31]), then RT is most likely to trigger the transition to the error states compared to search algorithms (because of the reasons discussed above). In case the approach is not trivial, then one should use (1+1) EA-ITD-TIR, which is the best combination to use in the cases when there are guards on time transitions located on the path to the error state and is at the same time no worse than its corresponding Basic version (i.e., (1+1) EA-Basic-TIR). One limitation to this can be situations in which the approach level is not trivial and at the same time the transition leading to the risky state is purely dependent on the behavior of the SUT (e.g., a guard that is set based on interactions with the SUT). This case will be similar to scenarios with a trivial approach to risky state in a way that the best chances of getting the SUT to behave in the required way are by invoking diverse environment behaviors. This, as we discussed earlier, is better done by RT than by the search algorithms with the proposed order functions. A possible solution to situations like these is to combine random testing with search-based algorithms and apply adaptive mechanisms based on the feedback from the executed test cases, which we will address in our future work.

In light of all the results and discussions, we can conclude that when applying our environment model-based testing approach in practice, one can achieve good results by combining RT and (1+1) EA-ITD-TIR. This can be done by running RT first and then, if no error state is reached within a short time, by running (1+1) EA-ITD-TIR for a few runs. Based on the results reported in this paper, this strategy would be expected to achieve a success rate close to 100%.

6.4 Threats to validity

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

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

7. CONCLUSION

In this paper, we proposed four new heuristics for search-based, black-box automated testing of Real-Time Embedded Systems (RTES) based on a model of their environment. The heuristics were developed to exploit various properties of these environment models in an attempt to reach environments states indicating a fault in the RTES (Error states). We provide an extensive empirical evaluation on an industrial case study and thirteen artificial RTES that we developed based on two industrial case studies belonging to different domains. The models of these artificial problems present varying properties that may affect the performance of these heuristics and are meant to help us understand the conditions under which they are beneficial. We evaluated the individual heuristics and their 16 combinations with two search algorithms, Genetic Algorithms (GA) and (1+1) Evolutionary Algorithm (EA). We also used Random Testing (RT) as a comparison baseline.

Results show that when reaching a state adjacent to the error state (risky state) is not trivial (i.e., reached by random test cases), RT is significantly worse than any of the proposed search algorithms. In this case, the best results are obtained when using (1) a heuristic favoring test cases maximizing the time spent in risky states and (2) (1+1) EA as a search algorithm, which showed to be overall superior to GA. However, the heuristic that favored higher coverage of states in the environment model (coverage) showed significantly poorer performance with (1+1) EA in four of the thirteen problems. Based on the results, we proposed a way to combine RT with (1+1) EA in order to achieve high fault detection rates in practice.

8. REFERENCES

[1] Arcuri, A., Iqbal, M., and Briand, L. 2010. Black-Box System Testing of Real-Time Embedded Systems Using Random and Search-Based Testing. in *Testing Software and Systems*. vol. 6435. A. Petrenko, *et al.*, Eds., ed: Springer Berlin / Heidelberg, pp. 95-110.

[2] Iqbal, M. Z., Arcuri, A., and Briand, L.2011. *Automated System Testing of Real-Time Embedded Systems Based on Environment Models*. Simula Research Laboratory, Technical Report (2011-19)

[3] Omg. 2010, Unified Modeling Language Superstructure, Version 2.3, http://www.omg.org/spec/UML/2.3/.

[4] Omg. 2009, Modeling and Analysis of Real-time and Embedded systems (MARTE), Version 1.0, http://www.omg.org/spec/MARTE/1.0/.

[5] Iqbal, M. Z., Arcuri, A., and Briand, L. 2010. Environment Modeling with UML/MARTE to Support Black-Box System Testing for Real-Time Embedded Systems: Methodology and Industrial Case Studies. in *Model Driven Engineering Languages and Systems*. vol. 6394. D. Petriu, *et al.*, Eds., ed: Springer Berlin / Heidelberg, pp. 286-300.

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

[7] Harman, M., Mansouri, S., and Zhang, Y.2009. *Search based software engineering: A comprehensive analysis and review of trends techniques and applications*. Department of Computer Science, King's College London, TR-09-03

[8] Ali, S., Briand, L. C., Hemmati, H., and Panesar-Walawege, R. K. 2009. A Systematic Review of the Application and Empirical Investigation of Search-Based Test Case Generation *IEEE Transactions on Software Engineering*. vol. 99.

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

[10] Broekman, B. M. and Notenboom, E. 2003. *Testing Embedded Software*: Addison-Wesley Co., Inc.

[11] Clarke, D. and Lee, I. 1995. Testing real-time constraints in a process algebraic setting. *in Proceedings of the 17th International Conference on Software Engineering*.

[12] Krichen, M. and Tripakis, S. 2009. Conformance testing for real-time systems *Formal Methods in System Design*. vol. 34. pp. 238-304.

[13] Nielsen, B. and Skou, A. 2003. Automated test generation from timed automata *International Journal on Software Tools for Technology Transfer*. vol. 5. pp. 59-77.

[14] Higashino, T., Nakata, A., Taniguchi, K., and Cavalli, A. 1999. Generating test cases for a timed I/O automaton model. *in Proceedings of the IFIP TC6 12th International Workshop on Testing Communicating Systems: Method and Applications*, pp. 197-214.

[15] Mucke, T. and Huhn, M. 2004. Generation of optimized testsuites for UML statecharts with time. *in Proceedings of16th IFIP international conference on Testing of communicating systems*, p. 128.

[16] Zheng, M., Alagar, V., and Ormandjieva, O. 2008. Automated generation of test suites from formal specifications of real-time reactive systems *The Journal of Systems & Software*. vol. 81. pp. 286-304.

[17] Auguston, M., B, M. J., and Shing, M. 2006. Environment behavior models for automation of testing and assessment of system safety *Information and Software Technology*. vol. 48. pp. 971-980.

[18] Briand, L., Labiche, Y., and Shousha, M. 2006. Using genetic algorithms for early schedulability analysis and stress testing in real-time systems *Genetic Programming and Evolvable Machines*. vol. 7. pp. 145-170.

[19] Lindlar, F., Windisch, A., and Wegener, J. 2010. Integrating Model-Based Testing with Evolutionary Functional Testing. *Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation Workshops*.

[20] Mcminn, P. 2004. Search based software test data generation: a survey *Software Testing, Verification and Reliability*. vol. 14. pp. 105-156.

[21] Heisel, M., Hatebur, D., Santen, T., and Seifert, D. 2008. Testing Against Requirements Using UML Environment Models. *in Fachgruppentreffen Requirements Engineering und Test, Analyse & Verifikation*, pp. 28-31.

[22] Adjir, N., Saqui-Sannes, P., and Rahmouni, K. M. 2009. Testing Real-Time Systems Using TINA. in *Testing of Software and Communication Systems*. vol. 5826, ed: Lecture Notes in Computer Science, Springer Berlin / Heidelberg.

[23] Larsen, K. G., Mikucionis, M., and Nielsen, B. 2005. Online Testing of Real-time Systems Using Uppaal. in *Formal Approaches to Software Testing*. vol. 3395, ed: Lecture Notes in Computer Science, Springer Berlin / Heidelberg.

[24] Peleska, J., Lapschies, F., Vorobev, E., Loeding, H., Smuda, P., Schmid, H., and C., Z. 2011. A real-world benchmark model for testing concurrent real-time systems in the automotive domain. *in Proceedings of IFIP International Conference on Testing Software and Systems (ICTSS)*, pp. 146-161.

[25] Lefticaru, R. and Ipate, F. 2008. Functional search-based testing from state machines. *in Proceedings of the International Conference on Software Testing, Verification, and Validation*, pp. 525-528.

[26] Ali, S., Iqbal, M. Z., Arcuri, A., and Briand, L. 2011. A Search-based OCL Constraint Solver for Model-based Test Data Generation. *11th International Conference on Quality Software*.

[27] Deb, K. 2001. *Multi-Objective Optimization Using Evolutionary Algorithms*: John Wiley and Sons.

[28] Andrews, J., Briand, L., Labiche, Y., and Namin, A. 2006. Using mutation analysis for assessing and comparing testing coverage criteria *IEEE Transactions on Software Engineering*. vol. 32. pp. 608-624.

[29] Arcuri, A. and Fraser, G. 2011. On Parameter Tuning in Search Based Software Engineering *in International Symposium on Search Based Software Engineering (SSBSE)*.

[30] Arcuri, A. and Briand, L. 2011. A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering. *in 33rd International Conference on Software Engineering (ICSE)*, pp. 1 - 10

[31] Harman, M., Hu, L., Hierons, R., Wegener, J., Sthamer, H., Baresel, A., and Roper, M. 2004. Testability transformation *IEEE Transactions on Software Engineering*. vol. 30.