Search-Based Test Case Generation for Cyber-Physical Systems

Aitor Arrieta*, Shuai Wang†, Urtzi Markiegi*, Goiuria Sagardui* and Leire Etxeberria*
* Mondragon Goi Eskola Politeknikoa, Spain
Email: {aarrieta,umarkiegi,gsagardui,letxeberria}@mondragon.edu
†Certus V&V Center, Simula Research Laboratory, Norway
Email: shuai@simula.no

***Abstract*—The test case generation of Cyber-Physical Systems (CPSs) face critical challenges that traditional methods such as Model-Based Testing cannot deal with. As a result, simulation-based testing is one of the most commonly used techniques for testing CPSs despite sometimes being computationally too expensive. This paper proposes a search-based approach which is implemented on top of Non-dominated Sorting Genetic Algorithm II (NSGA-II), the most commonly applied multi-objective search algorithm for cost-effectively generating executable test cases in order to test CPSs. With the aim of guiding the generation of the optimal set of so-called reactive test cases, the approach formally defines three cost-effectiveness measures: requirements coverage, test case similarity and test execution time. Furthermore, we design one crossover operator and three mutation operators (i.e., mutation at test suite level named *Mu_TS*, mutation at test case level named *Mu_TC* and mutation at both levels named *Mu_BO*) for test case generation. We evaluate our approach by comparing with Random Search (RS) using four case studies (one of them is an industrial system). Moreover, we evaluate the three mutation operators using the four case studies. The results of the experiment (with a rigorous statistical analysis) indicated that our approach in conjunction with the crossover operator operation and three mutation operators significantly outperformed RS. In general, *Mu_BO* achieved the best performance among the three mutation operators and managed to improve on average the test execution time by 14%, the requirements coverage by 34%, and the test similarity by 75% as compared with RS.**

I. Introduction

Nowadays, Cyber-Physical Systems (CPSs), which integrate digital cyber technologies (such as microprocessors or complex networks) with parallel physical processes [1], are attracting increasing attention from both academics and industry. Typically, microprocessors monitor and alter the physical processes by means of sensors and actuators (e.g., electrical engines). CPSs exist in our daily lives in several domains such as automotive, railway and aerospace [2]. Examples of CPSs include Unmanned Aerial Vehicles (UAVs), the adaptive cruise control of a car, elevators and pacemakers.

Moreover, CPSs have been cataloged as untestable systems and traditional testing techniques (e.g., model-based testing) are usually expensive, time consuming or infeasible to apply [3]. This is due to the fact that it is challenging for the traditional testing techniques to capture complex continuous dynamics and interactions between the system and its environment (e.g., people walking around when automatic braking systems are in use for automotive systems) [3][4]. As a result, simulation-based testing has been envisioned as an efficient means to test CPSs in a systematic and automated manner [3]. The use of simulation models permits several advantages such as (1) the execution of larger test cases, (2) selection of critical scenarios, (3) specification of test oracles for the automated validation of the system or (4) replication of safety-critical functions of a real system where it is expensive to execute test cases [3][5]. Therefore, testing CPSs face different particularities (e.g., concurrency or timing behavior) as compared to testing traditional software.

Aside from this, CPSs frequently interact with its environment, which make testing CPSs unpredictable [5]. To deal with this problem, different approaches have focused on testing CPSs using reactive test cases [2] since reactive test cases are able to observe different physical properties of the environment at runtime. Moreover, reactive test cases are composed with a set of signals that stimulate the inputs of CPSs under test and a set of properties (e.g., time, system outputs) are monitored to react on them and change the stimulation signal values [2]. Furthermore, the input space of a CPS is usually extremely large [4][6], with these inputs associated to requirements, which again bring great challenges for thoroughly testing a given CPS. In addition, diversification of the input space has helped the detection of faults in the past [7]. However, achieving higher requirements coverage usually requires more test cases for execution, which may become practically infeasible when a limited time budget for testing exists.

To address the above-mentioned challenges, we propose a search-based approach which considers three objectives with the aim to generate optimal reactive test cases to cost-effectively test CPSs. More specifically, our approach formally defines and formulates three cost-effectiveness measures, i.e., test execution time (cost), requirements coverage and test case similarity (effectiveness). Moreover, we design and implement one crossover operator and three mutation operators for generating optimal reactive test cases. The three mutation operators are designed from three perspectives, i.e., mutation at test case level named as *Mu_TC*, mutation at test suite level named as *Mu_TS* and mutation at both levels named as *Mu_BO*. Furthermore, our approach has been developed on the top of one of the most commonly applied multi-objective search algorithms named Non-dominated Sorting Genetic Algorithm

Explain CPSs and examples

the system only

interaction between system and its env.

why use reactive test case

Challenges

978-1-5090-4601-0/17/$31.00 ©2017 IEEE

II (NSGA-II) [8]. In addition, we provide tool support by developing a test case executor using our proposed approach for testing configurable CPSs. Notice that our tool support employs MATLAB/Simulink models, which is one of the most used simulation tools in the CPS context (e.g., [4][5][9]). Moreover, we empirically evaluate our approach together with tool support using one industrial case study and three open source case studies. More specifically, we first compared the approach with Random Search (RS), which is usually considered as a comparison baseline [10][9][2]. Secondly, we empirically evaluated the performance of the three proposed mutation operators (i.e., *Mu_TS*, *Mu_TC*, *Mu_BO*) using the four case studies. The results demonstrate that (1) our approach along with all three mutation operators achieved significantly better performance than RS in the four case studies; (2) the mutation operator *Mu_BO* achieved the best performance among the three designed mutation operators, especially for the complex CPSs while the mutation operator *Mu_TC* performed best in terms of the CPSs with less complexity; and (3) as compared with RS, our approach together with *Mu_BO* managed to reduce on average the test execution time by 14% and improve requirements coverage by 34% and test similarity by 75%.

The rest of the paper is structured as follows: The approach is evaluated in Section VI. Our work is positioned with other studies in Section II. Section III gives an overview of the background related to CPS testing and reactive test cases, which are the type of test cases we use for testing CPSs. Section IV explains our multi-objective approach for generating test cases for the context of CPSs. The developed tool and its integration with our other approaches is explained in Section V. Finally, conclusion and future work are outlined in Section VII.

II. Related Work

Search-based algorithms have recently been studied to generate test cases in the CPS context. Matinnejad et al. proposed a search approach that obtains objective functions by simulating the system in each iteration to test continuous controllers [6]. Ben Abdessalem et al. combined multi-objective optimization algorithms with neural networks to test automatic driving assistant systems [9]. Matinnejad et al. proposed a search algorithm to produce small set of test cases to test Simulink models [4]. Vos et al. proposed an evolutionary testing framework to test automotive systems, where objective functions were obtained using Software-in-the-Loop and Handware-in-the-Loop simulations [11]. All of them tried to find worst case scenarios by somehow simulating the system in each iteration to calculate the objective functions and guide the search. The main drawback of employing simulation models for finding worst case scenarios is that it might be extremely time consuming to generate test cases (e.g., in [9] a test generation budget of 120 minutes was used). Our approach does not use simulation to guide the search towards optimal solutions. Instead of trying to find worst case scenarios, we focus on system testing by trying to produce cost-effective reactive test cases taking into account requirements coverage, test execution time and test similarity.

Some works focused on the generation of reactive test cases. Zander captured the reactivity of the system with test oracles and later employed a model-based approach to generate one test case to test one requirement [12]. Mjeda captured the reactivity of the system with Simulink models and later used them to generate reactive test cases for safety-critical systems [13]. Lehmann proposed a tool named Time Partition Testing for the elaboration of model-based reactive test cases and it automatically generated executable test cases for different simulation tools (e.g., Simulink) [14]. These works focused on testing requirements, but they did not take other properties into account, such as the test execution time or test similarity. In addition, in all of them the process of generating test cases is semi-automatic as they all need to specify some reactivity behaviors.

In our previous works we focused on test system generation [5], test selection [15] and test prioritization [2], with a strong focus on configurable CPSs. Test system generation focuses on the automatic generation of the test infrastructure (e.g., test oracles, test controllers, etc.) to test configurable CPSs using simulation. Test selection focused on the efficient selection of test cases for the different levels where the CPSs are tested. The last step was to prioritize these test cases taking into account historical data (e.g., fault detection capability). However, all these approaches obtained test cases as input and none of them considered automatic test generation, which is considered in this paper.

III. Background

A. Cyber-Physical Systems Testing

As developing a CPS prototype is often costly, the use of simulation-based testing for CPSs is increasing. Simulation is one of the most frequently used techniques for testing system models in domains where software interacts with physical processes such as CPSs [4]. CPS models are heterogeneous due to encompassing software, networks and parallel physical processes. These models therefore provide an accurate representation of the real world and continuous dynamics [16][4]. Different simulation tools have been employed to test CPSs using simulation including MATLAB/Simulink [4][17][5], and a combination of System C and Open Dynamics Engine [18]. Moreover, as CPSs are involved in several engineering domains, it is very common to use different simulation environments integrated by a co-simulation engine. Although this integrated CPS model presents an important advance in terms of engineer flexibility, it increases the test and simulation time due to requiring the transfer of data between tools. In addition, CPSs simulation often involves the use of complex mathematical models to represent the continuous dynamics of the physical layer. Consequently, computer resources are allocated by the solvers used by the simulation tools.

In addition, CPSs are often tested at different levels such as Model-in-the-loop (MiL), Software-in-the-loop (SiL), Processor-in-the-loop (PiL) and Hardware-in-the-Loop (HiL)

3 measures
limitation of prev. works
author's prev. works
=> only talk about test selection and test prioritization
to simulate sw interact phsy. => most freq. use simulation-based testing
diverse
CPSs model is deiverse
4 levels to test CPSs
Traditional method
Drawback
Diff.

[19][15]. *MiL* aims to test the software model to obtain reference values [19]. This software model is then replaced by executable code to test the CPS at the *SiL* level, permitting software testing with fixed-point arithmetic logic [19]. This code is later compiled and embedded in the target processor to perform a *PiL* simulation, where the embedded system communicates with the physical layer simulated on the computer. The *PiL* simulation aims to test the compiled code [19]. Finally, the embedded software is integrated with the real-time infrastructure (e.g., operating systems, drivers, etc.) and the physical layer is embedded into real-time platform to perform a test at the *HiL* level [19]. At this stage, the objective is to test the system's real-time performance (e.g., tasks deadlines, use of memories, etc.) performing a real-time simulation.

B. Reactive Test Cases

What is test reactivity

Test reactivity is known as the capacity of the test system to react on the outputs of the System Under Test (SUT), verdicts data, or internal signals of the test oracle [20]. Accordingly, reactive test cases are a set of stimulation signals that excite a system and observe some predefined properties (e.g., SUT outputs, time, etc.) to react on them and change the stimulation signals to other values [2]. These systems are typically employed to test embedded systems [21] or CPSs [2] of different domains such as in the automotive industry [12]. Reactive test cases are usually employed to test functional requirements at system level [12].

What is reactive test cases

As an example, consider three reactive test cases depicted in Figure 1, which aim at testing the cruise control system of a car. With this objective, the test cases can turn on the engine of the car, set the speed of the car (v), or push the brake pedal to reduce the speed of a car should the driver wish. In the illustrated example, the first reactive test case (i.e., *TC1*) contains three states.[1] The first state turns the engine on and sets the car to a speed of 0 km/h. Once the system achieves this state, the transition triggers the[2] second state of the reactive test case, which aims at setting the speed of the car to 100 km/h. When this state is achieved with certain stability (i.e., acceleration (a) $<$ 0.25 m/s2), the third and[3] last state is triggered, which sets the car at a speed of 180 km/h. Once the car achieves this speed the test case is finished. Unlike *TC1*, reactive test cases 2 and 3 (i.e., *TC2* and *TC3*) have 2 states. In *TC2*, the first state turns the engine on and sets the speed of the car to 150 km/h, whereas the second state sets the speed of the car into 0 km/h while the brake pedal is released. On the other hand, in *TC3*, the first state turns the engine on and sets the speed of the car to 180 km/h, while the second state keeps the speed of the car at 180 km/h but the brake pedal is pushed. When considering these three test cases, it can be easily deduced that *TC1* tests acceleration functionalities, *TC2* tests deceleration functionalities and *TC3* tests functionalities related to the braking system.

green: properties

Reactive test cases can also serve as test oracles. These test cases observe the output values of CPSs and they changed their state when the system achieves the set value. For instance, in the illustrated example in Figure 1, consider that the first state

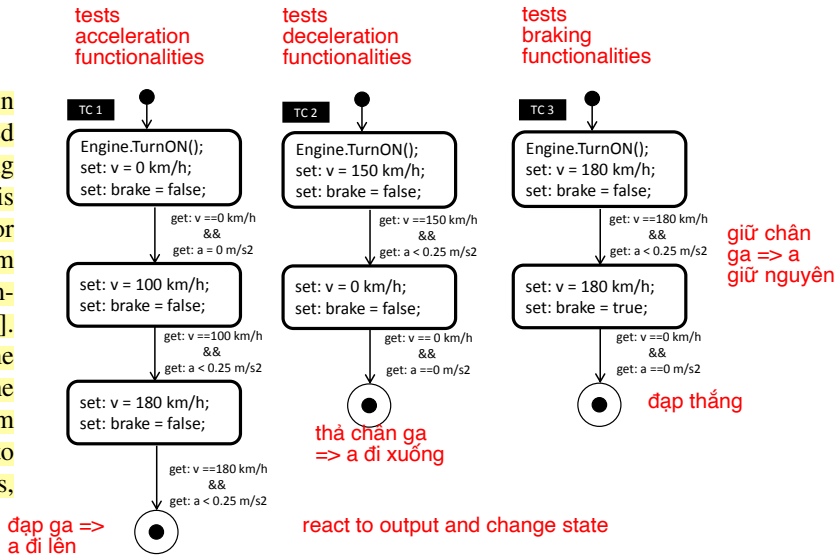

Fig. 1: Example of three reactive test cases for the cruise control system testing of a vehicle

of *TC2* is triggered, but for an unknown system fault, the car does not achieve the set speed. By means of a timeout mechanism, it can be assumed that the system does not achieve that state and that a fault has been detected. In addition, in the CPS context, the physical world is not predictable, which means the CPSs do not operate in a controlled environment [22]. CPSs have to be robust to withstand unexpected environmental conditions [22], and a key characteristic is to have adaptive capabilities to dynamically reconfigure themselves [23]. Thus, reactive test cases play an important role when testing CPSs as they support this unpredictability by observing the system behavior. Why???

IV. Search-Based Test Case Generation Approach

In this section, we present our search-based multi-objective approach for generating optimal reactive test cases cost-effectively. Section A formally defines three cost-effectiveness measures (i.e., objective functions), followed by solution representation (Section B). Sections C and D present the designed crossover and mutation operators respectively.

A. Cost-Effectiveness Measures

As aforementioned, we mathematically define three cost-effectiveness measures to guide search towards generating optimal reactive test cases, i.e., requirements coverage and test case similarity (effectiveness) and test execution time (cost). We present each measure in detail below.

1) Requirements coverage: Reactive test cases are typically employed for functional testing at system level, meaning that functional requirements must be taken into account when generating test cases. For this reason, we defined the Requirements Coverage (*RC*) as the first effectiveness measure, which considers the number of requirements covered by a specific test suite with respect to the total number of requirements that a CPS has. The *RC* is calculated following the Equation 1, where $|FR \lessdot sk|$ refers to the number of functional requirements covered by a solution while $|FR|$ denotes the total number of

- generate TC take into account functional requirements of system
- # of requirement cover (1 TC) / total # requirement

$x \lessdot y$ means that x is covered by y.

functional requirements that the CPS has. Notice that *RC* is developed for each system with a script function by linking each requirement with a specific property of state or sequence of states. For instance, if the acceleration functionality of the cruise control was tested, a state would set the car to 0 km/h whereas the proceeding state would set the speed to more than 150 km/h.

$$RC_{sk} = \frac{|FR \lessdot sk|}{|FR|} \tag{1}$$

Similarity in different TCs => reduce => tăng diverse => higher chance to detect faults

2) Test Case Similarity: Existing literature has shown that a set of diverse test cases has a higher chance to detect faults [7]. For this reason, we defined test case similarity (*TCS*) as the second effectiveness measure based on the Hamming Distance (*HD*), which is a widely used similarity measure [24]. More specifically, *TCS* measures the distance between two test cases and returns a value within the range [0,1]. A *TCS* value with 0 denotes that both test cases are identical, whereas when the *TCS* returns a value of 1, both of the test cases are considered as totally different. In addition to that, it can be followed in the context of reactive test cases that a reactive test case with more states is more likely to find errors than a test case with less states. With this concern in mind, we also take into account the number of states present when calculating *TCS*.

In RTC - more states => higher find faults

The *TCS* of two test cases (i.e., TC_a and TC_b) is calculated as expressed in Equation 2, where $|S_a|$ and $|S_b|$ are the number of states of each of the test cases and $|ss|$ is the number of stimuli signals for the CPS. $ss_{j_{tc_{ai}}}$ is the j-th signal's value for the a test case's i-th state and $ss_{j_{tc_{bi}}}$ is the j-th signal's value for the b test case's i-th state. In addition, max_{ss_j} is the maximum value and min_{ss_j} is the minimum value that the j-th signal can have. We consider that these maximum and minimum values will always be different, and thus, the interval between max_{ss_j} and min_{ss_j} will not be 0. Finally, *MaxStates* refers to the maximum number of states a test case can have, which is predefined by the test engineer.

$$TCS(TC_a, TC_b) = \frac{\sum_{i=1}^{min(|S_a|,|S_b|)} \sum_{j=1}^{|ss|} \frac{abs(ss_{j_{tc_{ai}}} - ss_{j_{tc_{bi}}})}{abs(max_{ss_j} - min_{ss_j})} / |ss|}{MaxStates} \tag{2}$$

Consider as an example the *TCS* between the *TC1* and *TC2* from the sample Figure 1. Let us assume that the system has two stimuli signals (i.e., v and brake), the maximum speed that the system can be set to is 180 km/h, and the maximum number of states is 3. Thus, the *TCS* between *TC1* and *TC2* would be calculated as follows:

$TCS(TC1, TC2) = ((abs(0-150)/180 + abs(0-0)/1)/2 + (abs(100-0)/180 + abs(0-0)/1)/2)/3 = 0.2315$

Furthermore, given a solution sk with *NTC* number of test cases, the average similarity function can be measured by calculating the average *TCS* values for each test case pair. This is obtained following Equation 3, where *NTC* is the number of test cases in sk, i is the i-th test case, j is the j-th test case and $NTC \times (NTC+1)/2$ is the total number of test case combinations in sk.

1 solution có 1 đống TCs => avg từng pair => avg similarity

$$Sim_{sk} = \frac{\sum_{i=1}^{NTC} \sum_{j=i+1}^{NTC} TCS(TC_i, TC_j)}{NTC \times (NTC+1)/2} \tag{3}$$

3) Test Execution Time: The time for executing a test suite is essential in the CPS testing context [2][15]. This is, to a large extent, caused by the high computational resources that simulation solvers consume to compute complex mathematical models related to the physical layer. Thus, we defined Test Execution Time (*TET*) as a cost measure for the generation of test cases for the CPS context. Given a solution sk of *NTC* test cases, each test case i has NS_{tc_i} number of states, the *TET* of sk is calculated as proposed in Equation 4.

$$TET_{sk} = \sum_{i=1}^{NTC} \sum_{j=2}^{NS_{tc_i}} time(S_j, S_{j-1}) \tag{4}$$

Notice that the function time, which sets the time required by the system to change from one state to another, is system specific. This means that before launching the test generation approach, the test engineer must specify how the function time is computed. For instance, in the case of the cruise control example, where the test cases are depicted in Figure 1, the time function follows Equation 5. ΔV is the difference between both state speeds and a_c is the acceleration coefficient, which changes if the car is accelerating, decelerating or braking.

$$time(s1, s2) = a_c \times \Delta V. \tag{5}$$

B. Solution Representation and Selection Operator

1) Solution Representations: A solution in our context is a test suite (TS) composed by at least one Test Case (TC), i.e., $TS = \{TC1, TC2, ..., TC_N\}$, where N is the total number of TCs in TS. Accordingly, a TC in our context is a reactive test case. Each of these test cases are composed by a set of states (S), (i.e., $TC_i = \{S_1, S_2, ..., S_{N_{tc_i}}\}$, where N_{tc_i} is the number of states that the i-th TC is composed of). Each state S has a predefined set of stimuli signals (ss) that must be connected to the simulation model of the CPS. These stimuli signals are based on the simulation model, and can be of different types (i.e., Boolean, integer or double), and each of them has a maximum and a minimum value. Typically, CPSs test suites are composed of around 100 reactive test cases [2], although this number can optimally be reduced to around 20 after selecting them with a search-based approach [15].

Figure 2 depicts an instance for representing a particular solution, which denotes a test suite with *N* reactive test cases. As shown in Figure 2, each test case has a set of states and each state includes three stimuli signals, i.e., (1) *Eng* shows the current status of the engine (*on* or *off*), which can be represented as a Boolean input; (2) *V* refers to the set speed, which can be represented as an integer input; and (3) Brake means the brake pedal state, which can be represented as a Boolean input. In the example, the first test case (i.e., *TC1*) is composed by three states, while the third test case (i.e., *TC3*)

- Test suite (TS) composed by n-th TC
- TC composed by n-th States-S
- State composed by n-th Signals-ss ~ properties: ech prop. := int or bool + min/max values

by five. Notice that each of the states is composed by the three stimuli signals (Eng, V and Brake), which are directly connected to the inputs of the system with their specific values.

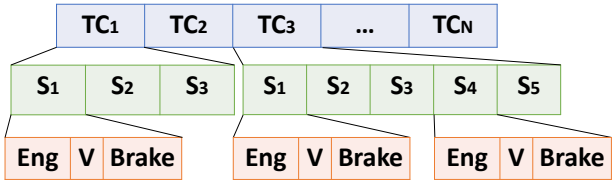

Fig. 2: Representation of a Test Suite for N test cases with 3 stimuli signals.

- To do so, first two solutions are selected randomly from the population. Then, if the solutions are from different front, the one with lower rank is chosen otherwise the one with higher CD would be chosen

2) Selection Operator: We chose the widely used binary tournament selection operator to evaluate the quality of solutions with the aim of including the best ones into the offspring solutions which are based on the defined fitness functions [8][25].

C. Crossover Operator

2 different solutions = 2 different TS

The crossover operator we implemented exchanges the test cases between two different solutions. Given two test suites, a randomly generated crossover point is selected, in the range [1,N], where N is the number of test cases that the smallest test suite has. When the crossover point is selected, two children are generated combining test cases between both test suites. We used single point crossover but the algorithm can also be configured to use multi-point crossover.

Consider as an example two parent test suites as shown in Figure 3. One of them has 6 test cases, whereas the other has 10. Thus, the crossover function will randomly select a crossover point in the range [1,6]. In the illustrated example, the crossover point is the number 5. The child 1 maintains the first 5 test cases of parent 1, while it inherits the sixth to tenth test cases of parent 2. On the other hand, child 2 maintains the first 5 test cases of parent 2, while it inherits the sixth test case of parent 1.

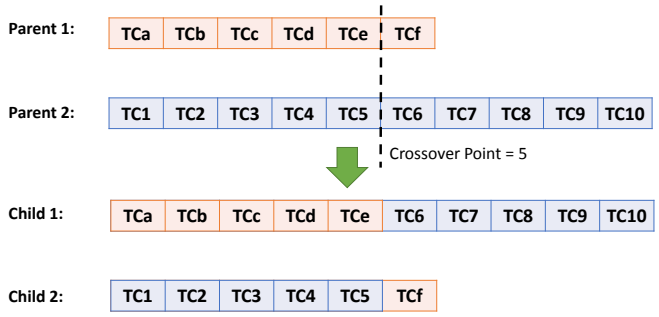

Fig. 3: Crossover operator example for two test suites of 6 and 10 test cases, having the crossover point at the fifth place

D. Mutation Operators

We design our mutation operators from two levels. The first level is the test suite level, whereas the second one is the test case level. The mutation operator at the test suite level randomly mutates test cases by adding or removing them from the test suite. The mutation operator at the test case level mutates the states by modifying them and stimuli signals of the test case. In addition, we define the third mutation operator that combines the above-mentioned two mutation operators. We present each mutation operator in detail below.

1) Mutation operator at test suite level (named Mu_TS): For the *Mu_TS* mutation operator, two sub-mutation operators have been developed. When the mutation operator *Mu_TS* is selected, one of these sub-mutation operators is randomly chosen by the algorithm. On the one hand, the first sub-mutation operator consists of the addition of a new test case into the test suite. When a new test case is added, this operator randomly decides the number of states that this test case must have. When the number of states is decided, it randomly selects the values of their stimuli signals based on the type and the maximum and minimum values they can be set to. On the other hand, the second sub-mutation operator consists of the removal of a test case from the test suite. It randomly selects the test case to be removed from the test suite and a child test suite is generated without the selected test case. Figure 4 illustrates both sub-mutation operators for *Mu_TS*. In the first one, a new test case is selected, whereas in the second one, the fourth test case is removed.

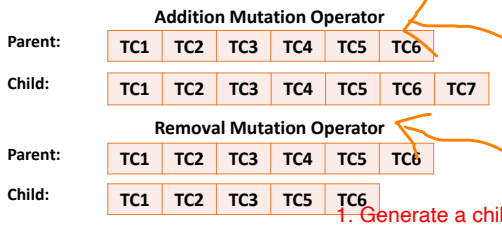

Fig. 4: Two sub-mutation operators at test suite level

1. Generate a child TS
2. Randomly add / remove TC from TS
3. In case of addition, randomly decide # of states and select value of `ss`

2) Mutation operator at test case level (named as Mu_TC): The mutation operator *Mu_TC* operates within the states that certain test cases have. At this level, four sub-mutation operators have been developed, consisting of state addition, state removal, exchange of states and change of variable. When the mutation operator *Mu_TC* is selected, one of these sub-mutation operators is randomly chosen.

1 For the state addition operator, a test case from the test suite is randomly chosen and a new state is added in a random position of that test case. Consider as an example Figure 5, where a new state is added to the second position of the test case.

2 For the state removal operator, a test case is randomly chosen from the test suite and one of its states is randomly removed. Consider as an example the test case depicted in Figure 6, where the second state is removed.

3 For the state exchange operator, a test case is randomly chosen from the test suite. Later, two states are randomly selected and their positions are exchanged. Consider as an example Figure 7, where the second state is exchanged with the third one to form a new child test case.

. **Four** sub-mutation operators: state addition / removal / exchange and variable changes

1. St. addition: pick TC randomly > add new state in random position
2. St. removal: pick TC randomly > remove a state in random position
3. St. exchange: pick two TCs randomly > exchange their position
4. Change of var. operator: pick TC randomly >

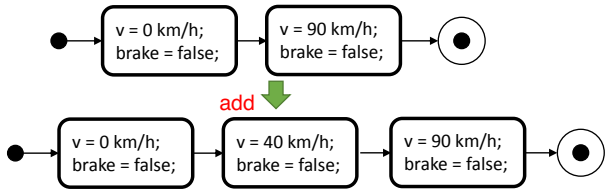

Fig. 5: Addition mutation operator for the test case level

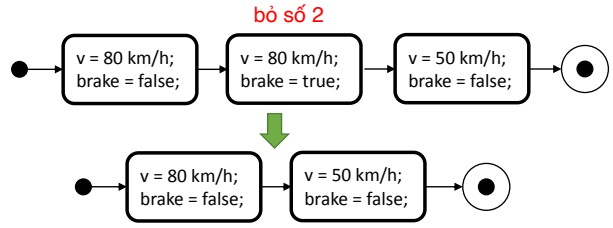

Fig. 6: Removal mutation operator for the test case level

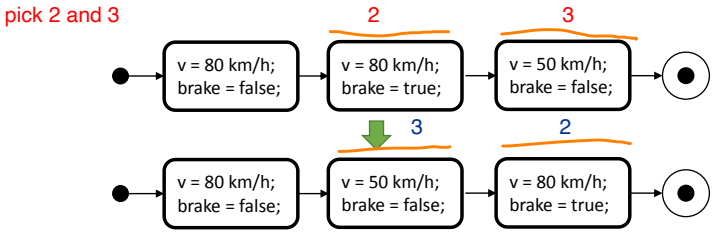

Fig. 7: Exchange mutation operator for the test case level

4 Lastly, the change of variable operator randomly chooses one of the stimuli signals of a test case and its value is changed according to its type and maximum as well as minimum values. In the example provided in Figure 8, the brake stimuli signal of the second state is changed.

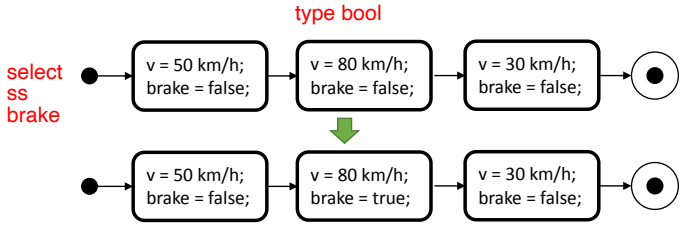

Fig. 8: Change of variable mutation operator for the test case level.

3) Mutation at test suite and test case level (named Mu_BO): *Mu_BO* combines both *Mu_TS* and *Mu_TC* explained in Sections IV-D1 and IV-D2. Every time *Mu_BO* is called, one mutation operator from *Mu_TS* and *Mu_TC* is randomly chosen followed by the same mutation process as explained above.

V. Tool Support

We implemented our approach for test case generation into an integrated tool framework for the efficient validation of

To manage variability, authors integrate their tools with feature models.

In the feature model, variability is modeled in "a test feature model"

configurable CPSs [5][2][15]. A particularity of configurable CPSs is that variability must be taken into account. To support variability we integrated our previous tools with a variability notation named feature models [5][15]. In these feature models, the variability that affected the test process was modeled in a "test feature model". The test feature model managed variability of stimuli signals (used in [5]) as well as requirements (used in [5][15]).

TS, TC, State, ss

The developed tool employs feature models to manage the solution representation. Specifically, in the test feature model the stimuli signals are modeled (as proposed in [5]). The tool we employ is FeatureIDE [26], a user-friendly, open source feature modeling tool. In addition, this tool permits embedding small notes inside the features. By using these notes we were able to embed the type of data (i.e., if they were boolean, integer or double), as well as the maximum and minimum values of each of the stimuli signals. Later, our test generation algorithm was able to parse the feature model, which was saved into an *.xml format, to obtain the representation of the solution. Specifically, it obtains the total amount of stimuli signals, the type of each of them and their maximum and minimum values.

managed 4 cái???

Figure 9 depicts an overview of our tool framework where our test case generation approach presented in this paper has been integrated. Test feature models are in charge of managing variability of the system as well as the test items. The test case generator presented in this paper reads the test feature model and generates test cases to save them in the repository. When a specific configuration from the repository needs to be tested, a tool named ASTERYSCO generates the test system (including test cases, test oracles, etc.) to execute test cases in MATLAB/Simulink [5]. Before launching the test, test cases are selected [15] and prioritized [2]. When the test is finished, test results are stored in a historical test database to later perform test selection and prioritization in an efficient manner.

1

2

3

bên trong Test exec. box

4

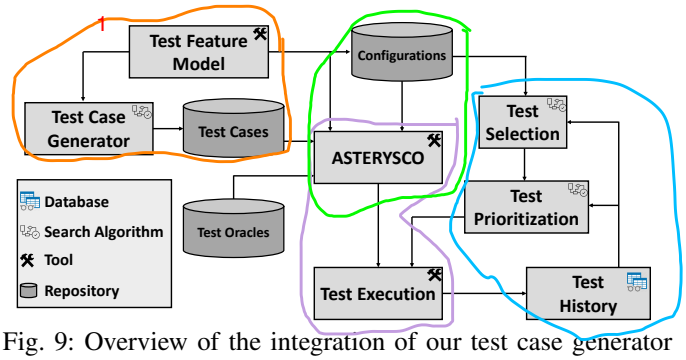

Fig. 9: Overview of the integration of our test case generator with the rest of the approach for testing configurable CPSs

VI. Empirical Evaluation

This section reports an empirical evaluation for our approach using four different case studies. We first present the research questions to be addressed in Section VI-A followed by explaining the experimental setup (Section VI-B). Section

VI-C details the experiment results and Section VI-D provides an overall discussion based on the obtained results. Last, section VI-E discusses the threats to validity.

A. Research Questions

To evaluate our approach, we aimed to answer two Research Questions (RQs), which are detailed as below. Random Search (*RS*) is used as a comparison baseline (as proposed in other similar studies [9]) to assess that the test case generation problem is not trivial (RQ1) [9][10][27]. In addition, we aim at evaluating three designed mutation operators (Section IV-D) and assessing which of them achieved the best performance for solving our problem (RQ2).

- **RQ1:** Is our approach cost-effective when compared to *RS* for solving the test case generation problem?
- **RQ2:** Which of the mutation operators developed fares best when solving test case generation problem?

B. Experimental Setup

1) Case Studies: Four case studies were employed in the proposed empirical evaluation. An open source system involving the automatic control of a DC engine with safety-critical functionalities was selected as a first case study from the web-page of the mathworks. Similar case studies have been previously used for evaluation in other test generation approaches (e.g., [6]). The second case study was the model of a configurable drone we adapted [17] and used in other evaluations (e.g., test case selection approach [15] and test system generation approach [5]). The third case study was the adaptation of an industrial system from Daimler AG concerning the Cruise Control of a car. Prior to our research, this case study was used to evaluate test case generation approaches [12] and test prioritization approaches [2]. The last case study consisted of a tank system we previously used in other evaluations [2][15].

Table I reports the key characteristics of the selected case studies. Specifically, the *Reqs* column specifies the number of requirements of the systems. The *Stimuli Signals* column is the number of stimuli signals of their Simulink models (B means the number Boolean signals and I means the number of integer signals). The last two columns are related to the Simulink models of the systems; the *Blocks* column is related to the number of blocks that the systems has, whereas the column *Depth* refers to the number of hierarchical levels of the system model.

TABLE I: Key characteristics of the case studies

Case Study	Reqs	Stimuli Signals	Blocks	Depth
DC Engine	25	4 (2-B, 2-I)	257	3
UAV	22	10 (6-B, 4-I)	843	4
Cruise Control	10	7(4-B, 3-I)	415	5
Tank	9	3 (3-I)	112	4

2) Algorithms Parameters Configurations: Notice that our approach is on top of NSGA-II, which is a well-known multi-objective algorithm that has been commonly applied for solving multi-objective test optimization problems (e.g., [9]).

We integrated our designed crossover operator and three mutation operators (i.e., *Mu_TS*, *Mu_TC* and *Mu_BO*) into NSGA-II (Sections IV-C and IV-D). As recommended by one of the most commonly applied multi-objective optimization Java framework jMetal [28], we set the crossover rate as 0.9, the population size was 100 and the number of fitness evaluations was 100,000. The mutation probability is 1/N, where N is (1) the number of test cases for the mutation operator *Mu_TS*; (2) the number of states for the first three sub-mutation operators of *Mu_TC* and the number of stimuli signals for the fourth sub-mutation operator of *Mu_TC*. In addition, we run each algorithm 100 times to account for random variations as recommended by Arcuri and Briand [10].

3) Evaluation Metrics: Based on the guide [29], we selected the Hypervolume (*HV*) quality indicator as the evaluation metric for the empirical evaluation. To be specific, *HV* measures the volume in the objective space covered by the produced solutions [28] with the range from 0 to 1 and a higher value of *HV* denotes a better performance of the algorithm. It is important to note that *HV* has been applied to assess many multi-objective test optimization approaches (e.g., test case generation approach [9]).

4) Statistical Analysis: We first employed the Shapiro-Wilk test to test the normality of the obtained data samples. The results of this test showed that the obtained data samples were normally distributed and thus we employed the t-test (i.e., significance test) to determine the significance of the results produced by different algorithms, as recommended in [10]. As recommended by Arcuri and Briand [10], the significance level was set to 95%, whereby, there is a statistically significant difference between the results of two algorithms if the p-value is less than 0.05. To determine the difference existing between two test generation strategies (A strategy and B strategy) the Vargha and Delaney statistics was used to calculate $\hat{A}_{12}$ [10][30].

C. Results and Analysis

From Figure 10, we can observe that our approach along with the crossover operator and three mutation operators performed better than *RS* in terms of solving the test generation problem since all the values for median and mean produced by *RS* were lower than the values outputted by our approach. Such results were further corroborated with statistical analysis as shown in Table II, where results for the t-test and the $\hat{A}_{12}$ measures for the HV indicator are reported. As reported in Table II, our approach together with the mutation and crossover operators statistically outperformed *RS* for all the four cases (i.e., $\hat{A}_{12} > 0.5$ and p-value < 0.05). Moreover, in three out of four case studies, *RS* was the algorithm that required more time for the generation of the test cases.

Regarding the performance of the three mutation operators, *Mu_BO* showed the highest mean and median *HV* values in the UAV and cruise control case studies, whereas *Mu_TC* achieved the highest mean and median values for *HV* with respect to the DC engine and Tank case studies. In addition, the mutation operator *Mu_TS* always produced the worst average

UAV & CC are complex => HV low < 0.6 and 0.36
DC & Tank are easy => HV high > 0.65 and 0.6

RS fastest còn 3 kia RS chậm nhất

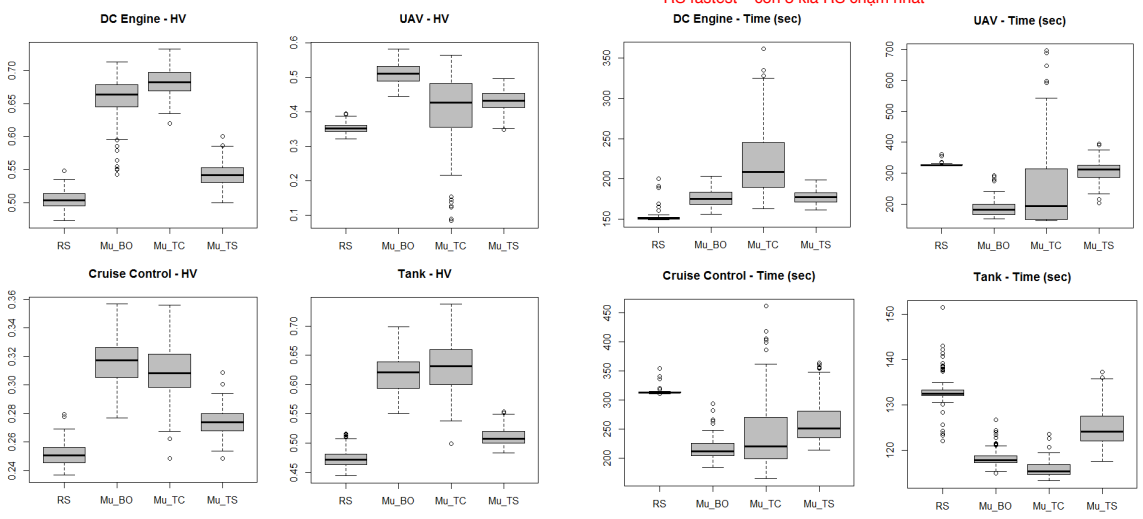

Fig. 10: Boxplots for the HV quality indicator and test generation time for the performed experiment runs

TABLE II: Results for the t-test for each study subject for the HV indicator. $\hat{A}_{12}<0.5$ means that the algorithm in the left performed better than the algorithm in the right. $\hat{A}_{12}>0.5$ means the opposite. The p-value shows the statistical significance between both algorithms (i.e., there is a statistical significance if p-value < 0.05) can reject null hypo.

RQ	PAIRS	DC Engine		UAV		Cruise Control		Tank	
		$\hat{A}_{12}$	p-value	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$	p-value
RQ1	**RS vs Mu_BO**	0.99	<0.00001	1	<0.00001	1	<0.00001	1	<0.00001
	RS vs Mu_TC	1	<0.00001	0.75	0.0014	0.99	<0.00001	0.99	<0.00001
	RS vs Mu_TS	0.95	<0.00001	0.98	<0.00001	0.95	<0.00001	0.93	<0.00001
RQ2	**Mu_BO vs Mu_TC**	0.72	<0.00001	0.15	<0.00001	0.40	0.01094	0.59	0.01718
	Mu_BO vs Mu_TS	0.01	<0.00001	0.03	<0.00001	0.01	<0.00001	0	<0.00001
	Mu_TC vs Mu_TS	0	<0.00001	0.52	0.0073	0.06	<0.00001	0.01	<0.00001

and median *HV* values for all the case studies except for the UAV, where the mutation operator *Mu_TC* showed lower *HV* mean and median values.

When taking the statistical significance into account (Table II), the mutation operator *Mu_BO* significantly outperformed the other mutation operators for 75% (6 out of 8) cases, i.e., *Mu_BO* achieved significantly better performance than *Mu_TC* in terms of the UAV and Cruise control case studies and *Mu_TS* for all four case studies. This interesting observation denotes that performing the mutation at both levels can be more effective towards generating optimal test cases than mutating at each separate level.

Figure 10 also depicts running times for different test generation strategies. On average, except in the DC engine case study, *RS* was the slowest algorithm. As for the DC engine subject, from the figure, we can observe that there is no practical difference in terms of the running time. Except DC engine case study, it can be observed that the strategy showing the best *HV* value was the fastest in generating test cases.

Furthermore, Table III shows the average improvement of each mutation operator when compared to *RS*. In addition, it can be highlighted that for the most complex case studies (i.e., UAV and cruise control), our approach largely improved the requirements coverage with on average 86.21% and 37.77%. In these case studies test execution was not always improved by our approach ant thus their improvement is reported negatively in the table, meaning that RS improves the average test execution time in those percentages (e.g., 142.5%) as compared with our approach. As for the simplest case studies (i.e., DC engine and Tank), the improvement on requirements coverage was not that large whereas the test execution time was largely reduced by our approach with on average 24.73% and 82.11% as compared with *RS*.

TABLE III: Average improvement of each objective as compared to RS for each study subject

Mutation Strategy	Study Subject	RC (%)	TET (%)	Sim (%)
Mu_BO	DC Engine	8.14	24.73	87.54
	UAV	86.21	-67.28	52.99
	Cruise Control	37.77	15.49	64.19
	Tank	4.94	82.11	95.12
	Average	34.26	13.76	74.96
Mu_TC	DC Engine	8.2	12.3	98.15
	UAV	61.3	-142.52	56.2
	Cruise Control	23.9	13.04	60.74
	Tank	3.32	78.73	87.82
	Average	24.18	-9.61	75.73
Mu_TS	DC Engine	1.04	-32.54	19.79
	UAV	69.13	-239.29	24.09
	Cruise Control	14.05	-44.5	46.53
	Tank	3.69	19.11	40.37
	Average	21.98	-74.30	32.69

D. Discussion of the Results

Mu_TS <> RS: 8~22.8%
Mu_TC <> RS: 10~36%
Mu_BO <> RS: 28~45.7%
HV indicator

The results of RQ1 indicate that our approach along with the crossover operator and three designed mutation operators significantly outperformed *RS* in all four case studies. Based on the results, we can conclude that the test case generation problem is not trivial to deal with, requiring an efficient approach. The first designed mutation operator at the test suite level (i.e., *Mu_TS*), improved *RS* on average between 8% and 22.8% in terms of the *HV* indicator. The second mutation strategy, which referred to mutating solutions at the test case level (i.e., *Mu_TC*), improved *RS* on average between 10% and 36% in terms of the *HV* indicator. Lastly, the strategy of mutating solutions at both levels (i.e., *Mu_BO*) improved *RS* on average between 28% and 45.7% in terms of the *HV* indicator.

Mu_TC in DC engine and tank, ≤4 ss ⇒ simple + few ss use Mu_TC ==>
Mu_BO in UAV and CC, >4ss ⇒ complex + many ss use Mu_BC green

Regarding the three mutation operators (RQ2), the results showed that performing mutation at both test suite and test case levels (i.e., *Mu_BO*) significantly outperformed the other two mutation operators in an average of 75% cases. However, we also noticed that the mutation operator *Mu_TC* achieved the best results for the DC engine and the tank case studies. It is worth mentioning that the DC engine and the tank case studies are less complex than the other two case studies (i.e., UAV and Cruise Control) in terms of the number of blocks and stimuli signals. We also studied the characteristics of the four case studies (Table I) and observed that for the case studies involving 4 or less stimuli signals (simple CPSs), the mutation operator *Mu_TC* performed best while the mutation operator *Mu_BO* achieved the best performance for the case studies including more than 4 stimuli signals (complex CPSs). Therefore, we conclude that for the CPSs with less complexity (e.g., few test stimuli signals, few blocks) our approach along with *Mu_TC* is recommended in terms of generating optimal reactive test cases whereas our approach together with *Mu_BO* is recommended for dealing with complex CPSs. However, if it is practically challenging to determine the complexity of a particular CPS, we recommend our approach with *Mu_BO* for test case generation based on the results obtained from the experiments.

Indicate requirement coverage
UAV & CC are complex => HV low < 0.6 and 0.36
DC & Tank are easy => HV high > 0.65 and 0.6

The average values of the *HV* varied a lot between case studies. The UAV as well as the cruise control (which is the industrial case study), are the most complex case studies, and as a result, their *HV* average is lower. This might be due to the fact that achieving a high requirements coverage in these case studies is more difficult (as shown in Table III, where the improvement of our approach in the UAV and cruise control case studies was considerably higher than in the DC engine and tank case studies when compared to *RS*).

Regarding the running time, we noticed that except for the DC engine case study, the result of the *HV* showed a correlation with the test generation time (i.e., the higher the *HV*, the lower the test generation time). Except for the DC engine case study, *RS* was the slowest algorithm. In contrast, the fastest algorithm was the NSGA-II following the mutation operator *Mu_BO* for the UAV and cruise control case studies,

Explanation why Mu_BO + NSGA-II < Mu_TC + NSGA-II in Tank: bc better solution -> # TC + # States ít lại => chạy nhanh hơn

whereas for the tank case study the NSGA-II along with the mutation operator *Mu_TC* was the fastest algorithm. A possible explanation for this could be that as the results of the solutions are better, the number of test cases as well as the states that the test cases have can be lower, and thus it takes less time to calculate the objective functions. In addition, it can be highlighted that the standard deviation of the test generation time for the NSGA-II algorithm with the *Mu_TC* mutation operator was significantly higher than the deviation of the test generation time of the rest of algorithms.

1) Answers to the Research Questions: RQ 1 refers to the comparison between our approach along with three mutation operators and *RS*. The results for the 100 algorithm runs together with a rigorous statistical analysis showed that the proposed approach significantly outperformed *RS*, which means that the problem to solve is non-trivial. RQ 2 refers to the comparison among the different mutation strategies with the NSGA-II algorithm. The strategy that mutates solutions at the test case level performed best with those non-complex case studies, whereas the strategy that mutates solutions at both the test case and the test suite level is the best in complex case studies.

E. Threats to Validity

This section summarizes the identified threats that could invalidate our empirical evaluation. One of the *internal validity* threats lies on the parameter configurations of search algorithms (e.g., population size, number of generation, crossover rate) since different parameter settings may lead to different performance of algorithms [31]. To reduce this threat, we selected the settings suggested by the guide provided by Arcuri and Briand [10] as well as the default settings of jMetal [28]. An *external validity* threat could be related to the generalization of the results. To deal with this issue we employed four independent case studies from different application domains and with different complexities for evaluating our approach. Moreover, one of the case studies was a real-world industrial case study. Regarding *conclusion validity* threats, one could be the random variations produced by search algorithms. To reduce this threat, we repeated the executions of the algorithms 100 times and employed statistical analysis as recommended in [10]. A *construct validity* threat might be that the measures used are not comparable across the selected algorithms. To reduce this threat we used the same stopping criterion for all the algorithms, i.e., the number of fitness evaluations is set to 100,000 to seek the best solutions for test generation.

ngoài DC, UAV, tank có thể còn những cái khác chưa test

VII. Conclusion and Future Work

This paper aims at dealing with the problems of systematically generating cost-effective reactive test cases for testing CPSs. We proposed a multi-objective search approach that generates cost-effective reactive test cases guided by three objectives (i.e., test execution time, requirements coverage and test similarity). In addition, we proposed one crossover operator and three mutation operators from test suite and test case level. An empirical evaluation with four case studies

showed that our approach is cost-effective when compared with random search and managed to improve on average 34% for the requirements coverage, 75% for the test similarity, and improved the test execution time by 14%. As for the three mutation operators, *Mu_BO* achieved the best performance in general especially for the complex CPSs while *MU_TC* performed best for the simple CPSs in terms of generating reactive test cases. Moreover, we also present tool support with a test generator based on the proposed approach.

In the future, we plan to evaluate more multi-objective search algorithms (e.g., *SPEA2* [32]) along with more case studies. We would also like to use simulation models to test the obtained solutions with mutation testing.

ACKNOWLEDGMENT

This work has been developed by the embedded systems group supported by the Department of Education, Universities and Research of the Basque Government. Shuai Wang is jointly supported by the Research Council of Norway (RCN) funded Certus SFI, RFF Hovedstaden funded MBE-CR project and COST Action CA15140 (ImAppNIO).

REFERENCES

[1] P. Derler, E. A. Lee, and A. Sangiovanni-Vincentelli, "Modeling cyber-physical systems," *Proceedings of the IEEE (special issue on CPS)*, vol. 100, no. 1, pp. 13 – 28, January 2011.

[2] A. Arrieta, S. Wang, G. Sagardui, and L. Etxeberria, "Test case prioritization of configurable cyber-physical systems with weight-based search algorithms," in *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, ser. GECCO '16. New York, NY, USA: ACM, 2016, pp. 1053–1060.

[3] L. Briand, S. Nejati, M. Sabetzadeh, and D. Bianculli, "Testing the untestable: Model testing of complex software-intensive systems," in *Proceedings of the 38th International Conference on Software Engineering Companion*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 789–792.

[4] R. Matinnejad, S. Nejati, L. C. Briand, and T. Bruckmann, "Automated test suite generation for time-continuous simulink models," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 595–606.

[5] A. Arrieta, G. Sagardui, L. Etxeberria, and J. Zander, "Automatic generation of test system instances for configurable cyber-physical systems," *Software Quality Journal*, pp. 1–43, 2016.

[6] R. Matinnejad, S. Nejati, L. Briand, T. Bruckmann, and C. Poull, "Search-based automated testing of continuous controllers: Framework, tool support, and case studies," *Information and Software Technology*, vol. 57, pp. 705 – 722, 2015.

[7] R. Feldt, S. M. Poulding, D. Clark, and S. Yoo, "Test set diameter: Quantifying the diversity of sets of test cases," in *2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016, Chicago, IL, USA, April 11-15, 2016*, 2016, pp. 223–233.

[8] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *IEEE transactions on evolutionary computation*, vol. 6, no. 2, pp. 182–197, 2002.

[9] R. Ben Abdessalem, S. Nejati, L. C. Briand, and T. Stifter, "Testing advanced driver assistance systems using multi-objective search and neural networks," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: ACM, 2016, pp. 63–74.

[10] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *Software Engineering (ICSE), 2011 33rd International Conference on*. IEEE, 2011, pp. 1–10.

[11] T. E. J. Vos, F. F. Lindlar, B. Wilmes, A. Windisch, A. I. Baars, P. M. Kruse, H. Gross, and J. Wegener, "Evolutionary functional black-box testing in an industrial setting," *Software Quality Journal*, vol. 21, no. 2, pp. 259–288, 2013.

[12] J. Zander-Nowicka, "Model-based testing of real-time embedded systems in the automotive domain," Ph.D. dissertation, Technical University Berlin, 2008.

[13] A. Mjeda, "Standard-compliant testing for safety-related automotive software," Ph.D. dissertation, University of Limeric, 2013.

[14] E. Lehmann, "Time partition testing: A method for testing dynamic functional behaviour," in *Proceedings of the European Software Test Congress*, 2000.

[15] A. Arrieta, S. Wang, G. Sagardui, and L. Etxeberria, "Search-based test case selection of cyber-physical system product lines for simulation-based validation," in *Proceedings of the 20th International Systems and Software Product Line Conference*, ser. SPLC '16. New York, NY, USA: ACM, 2016, pp. 297–306.

[16] P. J. Mosterman and J. Zander, "Cyber-physical systems challenges: a needs analysis for collaborating embedded software systems," *Software & Systems Modeling*, vol. 15, no. ISSN 1619-1366, pp. 5–16, February 2016.

[17] P. J. Mosterman, D. E. Sanabria, E. Bilgin, K. Zhang, and J. Zander, "Automating humanitarian missions with a heterogeneous fleet of vehicles," *Annual Reviews in Control*, vol. 38, no. 2, pp. 259–270, 2014.

[18] W. Mueller, M. Becker, A. Elfeky, and A. DiPasquale, "Virtual prototyping of cyber-physical systems," in *Asia and South Pacific Design Automation Conference*, 2012, pp. 219 – 226.

[19] H. Shokry and M. Hinchey, "Model-based verification of embedded software," *Computer*, vol. 42, no. 4, pp. 53 – 59, 2009.

[20] J. Zander-Nowicka, "Reactive testing and test control of hybrid embedded software," in *Proceedings of the 5th Workshop on System Testing and Validation*, 2007, pp. 45–62.

[21] A. Mjeda and M. Hinchey, "Requirement-centric reactive testing for safety-related automotive software," in *2015 IEEE/ACM 2nd International Workshop on Requirements Engineering and Testing*, Florence, Italy, 2015.

[22] E. A. Lee, "Cyber physical systems: Design challenges," in *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*. IEEE, 2008, pp. 363–369.

[23] J. Shi, J. Wan, H. Yan, and H. Suo, "A survey of cyber-physical systems," in *Proc. of the Int. Conf. on Wireless Communications and Signal Processing*, 2011, pp. 1–6.

[24] H. Hemmati, A. Arcuri, and L. Briand, "Achieving scalable model-based testing through test case diversity," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 22, no. 1, pp. 6:1–6:42, 2013.

[25] F. Sarro, A. Petrozziello, and M. Harman, "Multi-objective software effort estimation," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 619–630.

[26] T. Thuem, C. Kastner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich, "Featureide: An extensible framework for feature-oriented software development," *Science of Computer Programming*, vol. 79, pp. 70 – 85, 2014.

[27] S. Wang, S. Ali, and A. Gotlieb, "Cost-effective test suite minimization in product lines using search techniques," *Journal of Systems and Software*, vol. 103, no. 0, pp. 370 – 391, 2015.

[28] J. J. Durillo and A. J. Nebro, "jmetal: A java framework for multi-objective optimization," *Advances in Engineering Software*, vol. 42, no. 10, pp. 760 – 771, 2011.

[29] S. Wang, S. Ali, T. Yue, Y. Li, and M. Liaaen, "A practical guide to select quality indicators for assessing pareto-based search algorithms in search-based software engineering," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 631–642.

[30] A. Vargha and H. D. Delaney, "A critique and improvement of the cl common language effect size statistics of mcgraw and wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.

[31] A. Arcuri and G. Fraser, "Parameter tuning or default values? an empirical investigation in search-based software engineering," *Empirical Software Engineering*, 2013.

[32] E. Zitzler, M. Laumanns, L. Thiele *et al.*, "Spea2: Improving the strength pareto evolutionary algorithm," in *Eurogen*, vol. 3242, no. 103, 2001, pp. 95–100.