Test Case Prioritization for Acceptance Testing of Cyber Physical Systems: A Multi-objective Search-Based Approach

Seung Yeob Shin
University of Luxembourg
Luxembourg
shin@svv.lu

Shiva Nejati
University of Luxembourg
Luxembourg
nejati@svv.lu

Mehrdad Sabetzadeh
University of Luxembourg
Luxembourg
sabetzadeh@svv.lu

Lionel C. Briand
University of Luxembourg
Luxembourg
briand@svv.lu

Frank Zimmer
SES Techcom
Luxembourg
frank.zimmer@ses.com

ABSTRACT

Acceptance testing validates that a system meets its requirements and determines whether it can be sufficiently trusted and put into operation. For cyber physical systems (CPS), acceptance testing is a hardware-in-the-loop process conducted in a (near-)operational environment. Acceptance testing of a CPS often necessitates that the test cases be prioritized, as there are usually too many scenarios to consider given time constraints. CPS acceptance testing is further complicated by the uncertainty in the environment and the impact of testing on hardware. We propose an automated test case prioritization approach for CPS acceptance testing, accounting for time budget constraints, uncertainty, and hardware damage risks. Our approach is based on multi-objective search, combined with a test case minimization algorithm that eliminates redundant operations from an ordered sequence of test cases. We evaluate our approach on a representative case study from the satellite domain. The results indicate that, compared to test cases that are prioritized manually by satellite engineers, our automated approach more than doubles the number of test cases that fit into a given time frame, while reducing to less than one third the number of operations that entail the risk of damage to key hardware components.

CCS CONCEPTS

• **Software and its engineering → Search-based software engineering**;

KEYWORDS

Acceptance Testing, Test Case Prioritization, Search-based Software Engineering, Multi-objective Optimization, Cyber Physical Systems.

ISSTA'18, July 16–21, 2018, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5699-2/18/07…$15.00
https://doi.org/10.1145/3213846.3213852

ACM Reference Format:
Seung Yeob Shin, Shiva Nejati, Mehrdad Sabetzadeh, Lionel C. Briand, and Frank Zimmer. 2018. Test Case Prioritization for Acceptance Testing of Cyber Physical Systems: A Multi-objective Search-Based Approach. In *Proceedings of 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'18).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3213846.3213852

1 INTRODUCTION

Many cyber physical systems (CPS), e.g., those used in aerospace, automotive and healthcare, are subject to extensive testing at different stages of development. *Acceptance testing*, which is aimed at ensuring that a completed system as a whole meets its specified requirements [1], is the last phase of testing, and takes place following the completion and integration of all the software and hardware components of a system. While most software testing activities are focused on *verification*, acceptance testing is a *validation* activity. It aims to determine whether the system does what it is expected to do, and it involves users or independent parties with strong domain knowledge [1].

Acceptance testing often involves prioritization of test cases because there are usually too many test cases to consider given time constraints. This paper investigates the use of specific test case minimization and prioritization techniques to build test suites that exercise the most critical system functions within the time window allotted to acceptance testing. In addition to time constraints, in the context of CPS acceptance testing, engineers need to contend with several considerations and tradeoffs: *First*, as is common practice in all levels of testing, each acceptance test case is structured into an *initialization* step, a *test scenario* step, and a *teardown* step [9]. The initialization step brings the system to a desired state before running the test scenario; the test scenario exercises a (system-level) requirement; and the teardown step brings the system to a default (pristine) state after the test scenario is run. The initialization and teardown steps do not contribute to fault detection. Yet, these steps can take a substantial amount of time to run, as they usually involve complex interactions with hardware components and potentially moving these components from one place or state to another. The engineers need to minimize the amount of initialization and teardown, so that a larger proportion of the acceptance testing time window will be left to running actual test scenarios.

Second, the running time of the test cases is impacted by the uncertainty in the environment. For example, a test case may

take longer to run when the hardware components need to be re-calibrated during test execution, e.g., to adjust to the ambient temperature. Unless the engineers take this uncertainty into account, they will be unable to estimate in a reliable way how long a test suite takes to run. *Third,* since acceptance testing is performed on actual hardware, the engineers need to be conscious of the impact of testing on hardware components. For example, every time a hardware switch is being flipped, there is the risk that the switch may get stuck during the flip operation. A test suite that flips hardware switches hundreds or thousands of times may cause irreversible damage.

Test case prioritization is well-studied, but primarily for software testing activities focused on verification [42]. A variety of prioritization schemes have been developed, aimed at detecting as many faults as early as possible [35]. In contrast, acceptance testing is driven by different objectives and, instead of maximizing the fault-revealing rate during earlier-stage testing, aims to prioritize test cases exercising the most important system functions according to domain experts. Further, little consideration has been given to objectives such as test time budget and hardware impact, which inevitably arise in CPS acceptance testing. Some recent strands of work take steps to address these limitations, for example by defining explicit objectives on the duration of testing and the resources utilized, including hardware resources [46, 51, 52]. The approach that we present in this paper provides two major advances over prior work: First, prior work concentrates exclusively on optimization via test case permutations. Our analysis indicates that achieving major gains, particularly with respect to the impact of testing on hardware, is not possible by test case permutations alone. Alongside permutations, our approach employs an additional optimization mechanism, which enables the removal of unnecessary initialization and teardown operations from test cases. These operations are both time-consuming and increase the risk of hardware damage. Second, prior work treats time and resources as *deterministic* notions. For CPS, these notions may not be expressible as exact functions due to the uncertainty in the environments where CPS are deployed. Our approach accounts for such uncertainty by using *probabilistic* notions for time and hardware impact.

Contributions. Our contributions are as follows:

1) A multi-objective approach for prioritizing acceptance test cases in CPS. The main elements of our approach are: (a) a lightweight formalism for characterizing CPS acceptance testing concepts, including the uncertainty in test execution times and the risk of damage to hardware, (b) an algorithm for removing redundant initialization and teardown operations from test case sequences while preserving the behavior of individual test cases, and (c) fitness functions for the quality of a given test case sequence based on execution time, criticality, and hardware damage risks.

2) Industrial case study. We apply our approach to a representative satellite system. Our results show that test case prioritization via multi-objective search, when compared to doing so via Random search, leads to a 61% increase in the number of critical test cases (41% increase in the total number of test cases) that can execute in a given time frame. Augmenting multi-objective test case prioritization with the removal of unnecessary initialization and teardown operations brings about another 55% increase in the number critical test cases covered (41% increase in the total number of test cases covered). Further, compared to applying multi-objective test prioritization alone, the removal of unnecessary operations reduces to one third the number of operations that can cause damage to key hardware components. Finally, compared to test cases that are prioritized manually by satellite engineers, our approach more than doubles the number of test cases that fit into a given time frame, while at the same time reducing to less than one third the number of operations that entail potential damage to key hardware components.

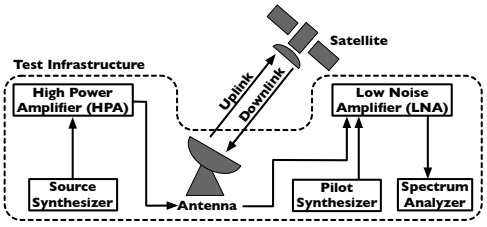


Figure 1: A simplified and partial view of the acceptance testing setup for a satellite after launch.

2 MOTIVATING CASE STUDY

We motivate our work using a real-world case study from the satellite domain. After launch and once in orbit, a satellite is subject to acceptance testing, often referred to as in-orbit testing. Fig. 1 shows a *simplified* setup for the acceptance testing of a new satellite. In addition to the satellite, a number of test instruments are involved in the setup. Such instruments are commonly used during the testing of CPS for injecting inputs and reading outputs. Collectively, the test instruments (and their connections) constitute the test infrastructure. The test infrastructure in our case study notably includes an antenna for communication with the satellite. There are several other test instruments which we do not list here. The exact instruments that are used vary across different test cases. We describe the function of some of these instruments below, as we illustrate test cases in our context.

The main steps of a test case, as noted earlier, are: *initialization*, *test scenario* and *teardown*. *Initialization* brings to a *ready* state the satellite as well as any test instruments used. This is achieved by performing an *init* operation on these components. We provide in Section 3.1 a precise formalism for the abstract states of the components of a CPS and the operations performed on the components. In our case study, initialization may involve, among other tasks, (re)calibrating the test instruments to ensure their accuracy under the environmental conditions at the time of testing.

A test scenario exercises an end-to-end behavior of the satellite to check the satisfaction of a system requirement. An example test scenario follows; the test instruments in this scenario are the ones illustrated in Fig. 1: (1) The Source Synthesizer generates a signal at a specified frequency for transmission to the satellite. (2) The High Power Amplifier (HPA) boosts the signal so that it is strong enough to reach the satellite. (3) The antenna transmits the (amplified) signal to the satellite via the uplink. (4) The signal that the satellite sends back to earth in response is received on the downlink. (5) The Low Noise Amplifier (LNA) amplifies the received signal without substantially degrading its signal-to-noise ratio. This often involves the use of a Pilot Synthesizer which generates a reference signal for comparison against the signal received from the satellite. (6) The properties

Table 1: Components involved in our illustrative test scenario.

c_1	c_2	c_3	c_4	c_5	c_6	c_7
Source Synthesizer	High Power Amplifier (HPA)	Antenna	Satellite	Pilot Synthesizer	Low Noise Amplifier (LNA)	Spectrum Analyzer

of the received signal are measured using the Spectrum Analyzer in order to verify that the satellite is working as intended.

Teardown brings the satellite and the test instruments to a *standby* state by performing cleanup operations on them (see Section 3.1). In our case study, cleanup can, for example, result in shutting down certain parts of the satellite or the test instruments to save energy, and muting instruments such as the HPA to ensure that no errant signal is accidentally sent to the satellite.

Table 1 lists the components involved in our illustrative test scenario. In Fig. 2, we show three test cases: tc_1, tc_2 and tc_3. These test cases are variant realizations of our illustrative test scenario described above, targeting different transmission functions of a satellite. Test cases tc_1 and tc_2 use all the seven components listed in Table 1, while tc_3 bypasses the uplink path in Fig. 1 and uses only the components on the downlink transmission path, i.e., c_3 to c_7.

As shown in Fig. 2, the test cases are parameterized by *chFreq*, indicating the specific channel frequency that is being tested. A satellite typically has tens of channels. There are various other parameters, which we do not show for simplicity. To create an acceptance test suite for a given satellite, the engineers choose the test cases that apply to the satellite. They then vary the parameters of the test cases, e.g. *chFreq*, to obtain test cases for execution. The resulting test suites are typically large, containing *hundreds* of test cases.

Below, we illustrate in more detail the considerations and trade-offs that we outlined in Section 1:

Initialization and teardown overhead. Test cases tc_1 and tc_2 work with the same components and share the same initialization and teardown steps. For a given channel frequency, if tc_2 runs immediately after tc_1, one can skip the teardown step of tc_1 and some of the operations in the initialization step of tc_2. Excluding unnecessary initialization and teardown operations minimizes testing time.

Execution time uncertainty. The execution time of test cases is influenced by several factors. For example, the time it takes for the antenna to be pointed to the satellite under test depends on where the antenna was previously pointing. Another influencing factor is that some test instruments need to be re-calibrated periodically or in response to changes in the environment, e.g., temperature changes. These re-calibrations prolong the execution of test cases. Since, at the time of developing test suites, it is impossible to come up with exact values for how long the test cases will take to run, one needs to account for variations in execution times.

Test budget constraints. Satellite engineers would ideally like to have their proposed test suites run in their entirety. This may however be infeasible due to the limited time imposed by delivery deadlines and the fact that in-orbit testing can take place only during certain times when the satellite is in a suitable orbital position and when the risk of interference with neighboring satellites is low. Hence, the engineers prioritize the test cases to ensure that the most important ones are run first and thus not left unexercised should the test suite need to be truncated. For instance, testing the core functions of a satellite are given priority over testing backup functions.

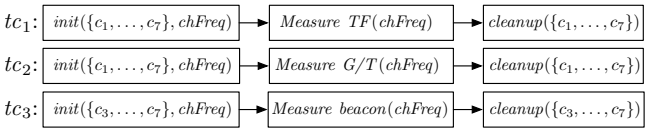


Figure 2: Three test cases for in-orbit testing parameterized by the channel frequency (*chFreq*).

Risk of hardware damage. Manipulating hardware components often poses a damage risk. For example, changing the configuration of a satellite involves flipping coaxial or waveguide switches to different positions. Like any mechanical switch, these switches may get stuck in between two positions during the flip operation. Should this happen, the signal path will be permanently out of service. Consequently, during acceptance testing, it is important to minimize operations that pose a risk of hardware damage.

The approach that we present next provides a generalizable solution for automatically prioritizing acceptance test cases in a way that takes into account all the concerns illustrated above.

3 APPROACH

In this section, we address the following problem: Given a set of test cases, recommend test suites that (1) include the most critical test cases, (2) minimize the risk of damage to the hardware components involved in testing, and (3) fit with high certainty into a specified time budget. Our solution incorporates a mechanism for excluding redundant initialization and teardown operations; test suite execution times are estimated with such redundancies removed.

3.1 Acceptance Testing Concepts

We use three main abstractions for characterizing acceptance testing: *components*, *test cases*, and *test suites*. We formalize these abstractions below. Informal descriptions and illustrations for the abstractions were already provided in Sections 1 and 2.

Components. A component can be either the system under test (e.g., the satellite in Fig. 1) or a test instrument (e.g., the synthesizers or the spectrum analyzer in Fig. 1). Since acceptance testing is black-box, we limit visibility into the internals of components. Specifically and as we explain below, we restrict knowledge about components to component variables and an abstract behavioral interface.

Let C be the set of components involved in acceptance testing. Each component $c \in C$ has a set V_c of variables. We denote by $c.v$ a variable $v \in V_c$. For instance, in Fig. 1, we have: $V_{\text{satellite}} = \{longitude, latitude\}$ indicating the satellite position, and $V_{\text{antenna}} = \{elevation, azimuth\}$ indicating the antenna position.

A variable $c.v$ can be of type float, integer or enumeration. Each variable of type integer or float has a range, specified by a minimum and a maximum value. A variable assumes a special value, denoted unknown, when the actual value of the variable can be determined only at the time of testing. In particular, one can use unknown for variables that depend on uncertain environmental factors such as temperature. To illustrate, consider again the setup in Fig. 1. Here, the frequency of the signal that the source synthesizer should generate is known and set prior to test execution. In contrast, the gain of the HPA can be known only at test execution time. This is because the gain depends on atmospheric conditions. Hence, when prioritizing test cases, the *gain* variable of the HPA is set to unknown.

We assume that each component implements the *abstract* behavioral interface shown in Fig. 3. A component is (1) on *standby* when

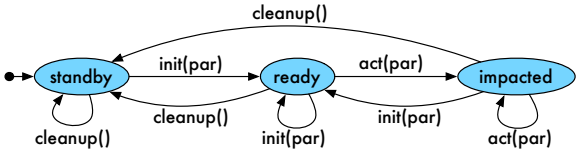


Figure 3: Abstract interface for components. Concrete component implementations behaviorally refine this interface.

the component is not in use and further ensured not to be interfering with the operations of other components; (2) *ready* when the component is ready for use; and (3) *impacted* when the component has been altered from its *ready* state during test execution.

The interface in Fig. 3 requires each component to implement three methods: *init*, *act* and *cleanup*. The *init* method takes a component from any state to *ready*. The *cleanup* method takes a component from any state to *standby*. The *act* method takes a component from *ready* to *impacted*. Note that, in component implementations, the *act* method is in fact a placeholder for a group of methods that change the component states from *ready* to *impacted*. The *init* and *cleanup* methods of a component are assumed to always be invoked respectively by the initialization and teardown steps of a test case (the steps of test cases were discussed in Sections 1 and 2). As for *act*, we assume that the method is invoked during the test scenario step of a test case, but *only* when the scenario has an impact on the component in question. The rationale for such handling of *act* is as follows: depending on how a test scenario unfolds, a participating component may be used without being impacted or be left unused. In either case, we want the component to remain in its *ready* state. For example, in our case study context, the bandwidth and frequency of a spectrum analyzer often remain unchanged during test case executions.

The *init* and *act* methods in the interface of Fig. 3 are parameterized. The parameter values for these methods are provided by the test cases that use the components. This was illustrated in Fig. 2 for the channel frequency (*chFreq*) parameter. In contrast, the *cleanup* method has no parameters, meaning that the method has no data dependency to the test cases. This treatment is justified by the fact that putting a component on standby is typically a generic process that is defined by the component's manufacturer.

We assume that the test cases in a test suite are executed sequentially; all the components required by a test case can thus be considered as being in (exclusive) use by that test case over the course of its execution. We note that the abstract interface of Fig. 3 is *not* meant to help parallelize test case executions by capturing whether a component is in use at a given time. Parallel execution of test cases is orthogonal to our purposes in this paper.

For each component c, we define four functions as follows: *init-time(c)*, *init-risk(c)*, *cleanup-time(c)*, and *cleanup-risk(c)*. The *init-time(c)* and *init-risk(c)* functions respectively denote the execution time and the hardware damage risk posed by executing the *init* method of c with different input parameters in varying environment conditions. The *cleanup-time(c)* and *cleanup-risk(c)* functions do the same for the *cleanup* method of c. These four functions are typically described as distributions or value ranges. In Section 4.2, we describe how we develop these functions for our case study.

Test Cases. We use the notation *tc* to refer to the *specification* of a test case. Each *tc* has a set C_{tc} of components. This set contains from the set C of all components those whose variables are accessed or modified by *tc*. Each *tc* further has a set *par* of input parameters. To refer to a test case (execution) based on specific value assignments to the input parameters in *par*, we use the notation *tc*(*par*). Let $C_{tc} = \{c_1, \ldots, c_p\}$. A test case *tc*(*par*) is composed of the following sequence: (1) the initialization of c_1 to c_p using the components' respective *init* methods, (2) a test scenario *test-sc*, (3) the cleanup of c_1 to c_p using the components' respective *cleanup* methods. The first two activities, namely the component initialization and the test scenario, may depend on the parameter values in *par*. We assume that the order of components in the initialization and cleanup activities is specified by the engineers. Test cases were exemplified in Fig. 2.

For each *tc*, we define three functions *time*(*tc*), *risk*(*tc*) and *crt*(*tc*). These functions respectively denote the execution time of *tc*, the risk of hardware damage posed by *tc*, and the criticality of *tc*. We characterize *time*(*tc*) and *risk*(*tc*) using probabilistic distributions. This enables us to express execution time variations caused by both uncertainty in the environment and also the different input parameter values. In contrast, we capture *crt* using numeric point values. The rationale for using point values is that criticality primarily has to do with how important the functionality targeted by *tc* is in the system under test. In our case study in Section 2, as we discuss in Section 4.2, we approximate *time*(*tc*) and *risk*(*tc*) based on historical data (log files), and *crt(tc)* based on expert knowledge.

Test Suites. Given a set *TC* of test cases, a *test suite* π is a sequence (permutation) $tc_1(par_1) \cdot \ldots \cdot tc_l(par_l)$ of all or a subset of the test cases in *TC*. We refer to a test suite as *full* when it includes all the test cases in *TC*; otherwise, we refer to it as *partial*. Further, for an i less than or equal to the length of a test suite π, we denote by π^i the prefix containing the first i elements of π. For example, $\pi^2 = tc_1(par_1) \cdot tc_2(par_2)$.

We lift the functions *time*(*tc*), *risk*(*tc*) and *crt*(*tc*) to the level of test suites. Let π be a test suite and l be its length. We define $f(\pi) = \sum_{i=1}^{l} f(tc_i(par_i))$ where $f \in \{time, risk, crt\}$. Similarly, for $j < l$ and $f \in \{time, risk, crt\}$, we define $f(\pi^j) = \sum_{i=1}^{j} f(tc_i(par_i))$.

For function $f \in \{time, risk\}$, we are typically interested in f's value at some confidence level *cl*. This is the value v such that the probability of f being less than or equal to v is *cl*. We denote the value of f at confidence level *cl* by $f(\pi, cl)$.

3.2 Removing Redundancies from Test Suites

We notice that the *init* and *cleanup* methods of the components are repeatedly called over the course of a test suite execution. If one considers the test cases in a test suite individually, all the *init* and *cleanup* calls are necessary: when a test case starts, it cannot make assumptions about the context in which it has been called and thus the state of the components involved. In a similar vein and not knowing what system activity will follow, a test case is obligated to return to a default state all the components it uses.

Several *init* and *cleanup* methods become redundant when a test suite is executed as a whole without being interleaved with other system activities during acceptance testing. In Fig. 4, we provide an algorithm, ReduceTCS (Reduce Test Case Sequences), for removing redundant *init* and *cleanup* invocations from test case sequences (i.e., test suites). The algorithm receives as input a test suite π and returns a test suite π' that retains all the test scenarios of π, but with the redundant *init* and *cleanup* invocations in π removed.

Algorithm ReduceTCS
Input: - A test suite $\pi = tc_1(par_1) \cdot \ldots \cdot tc_l(par_l)$
Output: - A test suite $\pi' = tc'_1(par_1) \cdot \ldots \cdot tc'_l(par_l)$ which removes from π redundant *init* and *cleanup* invocations

1. **for** $i = 1$ to l : /* l is the number of test cases in π and π' */
2. **for** $c \in C_{tc_i} \cap C_{tc_{i+1}}$:
3. remove *cleanup* of c from tc_i
4. **for** $j = 1$ to $|V_c|$:
/* V_c is treated as a sequence $c.v_1, \cdots, c.v_{|V_c|}$ of variables. */
5. Let S be the state of $c.v_j$ after executing *test-sc* of $tc_i(par_i)$, and let S' be the state of $c.v_j$ after executing *init* of $tc_{i+1}(par_{i+1})$
6. **if** S = unknown or $S \neq S'$:
7. **break**
8. **if** $j = |V_c|$:
9. remove *init* of c from tc_{i+1}

Figure 4: Algorithm for removing redundant *init* and *cleanup* invocations over the components in a test suite.

The algorithm works as follows: For each successive pair $tc_i(par_i) \cdot tc_{i+1}(par_{i+1})$ of test cases in π, their common components are computed, i.e., $C_{tc_i} \cap C_{tc_{i+1}}$ (line 2). Given the overall structure of test cases defined in Section 3.1, for every $c \in C_{tc_i} \cap C_{tc_{i+1}}$, there is a *cleanup* method invocation over c in the teardown step of tc_i, and an *init* method invocation over c in the initialization step of tc_{i+1}. Here, the *cleanup* method of c is redundant and can be removed from tc_i (line 3) because the *init* method of c in tc_{i+1} ensures to take c to its *ready* state (see the interface of Fig. 3). In addition, we can remove the *init* method of c from tc_{i+1} if, after executing the body of tc_i (i.e., *test-sc*), component c is still at its *ready* state and the values of its variables are consistent with the input parameters par_{i+1} of tc_{i+1} (lines 4-9). Specifically, for every variable $c.v_j$, we record the state of $c.v_j$ after the test scenario step of $tc_i(par_i)$ in variable S, and the state of $c.v_j$ after the initialization step of $tc_{i+1}(par_{i+1})$ in variable S'. If for any $c.v_j$, state of $c.v_j$ after the test scenario step of tc_i (i.e., S) happens to be unknown or different from its state after running the initialization step of tc_{i+1} (i.e., S'), the algorithm moves on to the next component in $C_{tc_i} \cap C_{tc_{i+1}}$ (line 7). Otherwise, the *init* invocation over c is removed from tc_{i+1} (line 9). To implement ReduceTCS, we use JavaPathFinder [50] to check assertions $assert(S \neq \text{unknown})$ and $assert(S = S')$ for each pair of consecutive test cases to determine whether or not *init* methods can be eliminated.

To illustrate, consider test cases tc_1, tc_2 and tc_3 in Fig. 2. Suppose we execute tc_1, tc_2 and tc_3 with the same value for *chFreq*. Further, suppose c_2, c_4 and c_6 are in their *ready* state after running *Measure TF* (i.e., their *act* methods are not called by $tc_1(chFreq)$), and c_4 and c_5 also remain in their *ready* state after running *Measure G/T*, but none of the c_3 to c_7 remain in their *ready* states after running *Measure beacon*. Consider two alternative test suites "$tc_1(chFreq) \cdot tc_2(chFreq) \cdot tc_3(chFreq)$" and "$tc_1(chFreq) \cdot tc_3(chFreq) \cdot tc_2(chFreq)$". The results of applying ReduceTCS to these test suites are shown in Figs. 5(a) and (b), respectively. The test suite in Fig. 5(a) has five less *init* and *cleanup* method calls compared to that in Fig. 5(b), and hence, executes faster and poses less risk of hardware damage.

We need to make the following remarks about the ReduceTCS algorithm: First, as shown by our example of Fig. 5, the order of the test cases in a test suite has a direct role in determining which component initialization and cleanup method invocations are eliminated. Maximizing the benefit of the reductions made by

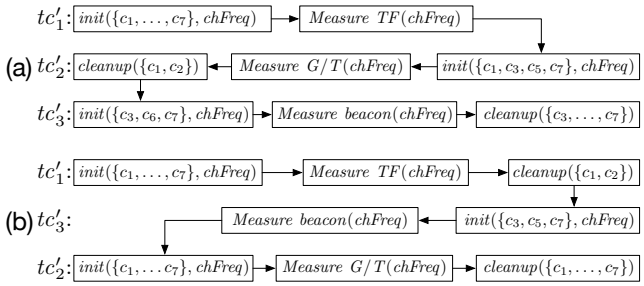


Figure 5: Example application of ReduceTCS; the original test cases tc_1, tc_2 and tc_3 were shown in Fig. 2.

ReduceTCS through ordering the test cases is an important part of our optimization solution in Section 3.3.

Second, ReduceTCS does not process a component c in tc_i, unless c is also used by tc_{i+1}. In particular, tc_i, after being processed by the algorithm, will still invoke the *cleanup* method for any component c that is absent from tc_{i+1}. This is to ensure that any such component is put on *standby* and thus does not interfere with the execution of the subsequent test cases, including tc_{i+1}.

Third, by reducing the amount of initialization and cleanup performed over the components, ReduceTCS impacts the test cases' *time* and *risk* functions (defined in Section 3.1). The impact of ReduceTCS on *time* and *risk* can nevertheless be quantified. This is because the distributions for the execution time and damage risk associated with the initialization and cleanup of individual components is also known and given by the *init-time(c)*, *init-risk(c)*, *cleanup-time(c)* and *cleanup-risk(c)* functions (defined in Section 3.1). For example, let tc' be tc with the *init* method call of some component c removed. For $f \in \{time, risk\}$, we have $f(tc') = f(tc) - init\text{-}f(c)$. Similarly, we have $f(tc') = f(tc) - cleanup\text{-}f(c)$ if tc' is tc with the *cleanup* method call of c removed.

3.3 Test Suite Optimization

In this section, we present our approach for optimizing acceptance test suites. Let TC be the set of all test cases that can be exercised. We first aim to find a full test suite, i.e., a test suite consisting of all the test cases in TC, such that the following two objectives are met: (1) test cases with higher levels of criticality appear as early as possible in the test suite, and (2) the damage risk posed by the test cases that appear early in the test suite is minimized. Once the engineers have a test suite π optimized according to the above objectives, they can select, based on the time budget available for acceptance testing, a prefix π^i of π to run on the system under test. We first present our solution for identifying optimal full test suites, before describing in precise terms how a test suite prefix is chosen.

We cast our solution into a *bi-objective search optimization problem* [34]. Following standard practice for expressing multi-objective search problems [16], we define the *representation*, the *fitness function*, and the *computational search algorithm* underlying the solution. For the remainder of this section, we recall from Section 3.1 that the execution time and risk functions, denoted *time* and *risk* are probabilistic distributions. Given a test suite π, the function $time(\pi, cl)$ (respectively, $risk(\pi, cl)$) yields a single value v such that, with a probability of cl, the execution time (respectively, the damage risk) of π does not exceed v.

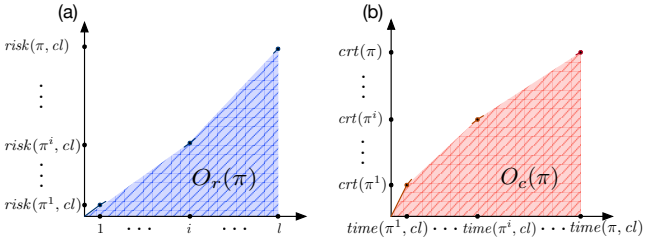


Figure 6: Fitness functions for test suite optimization.

Representation. Given a set TC of test cases, a feasible solution is a full test suite for TC, i.e., a permutation of all the test cases in TC.
Fitnesses. We define two quantitative fitness functions $O_r(\pi)$ and $O_c(\pi)$. The function $O_r(\pi)$ aims to ensure that the risk posed by test cases appearing early in π is minimized. We define $O_r(\pi)$ as the *area under the curve* of the *risk* function applied to successive prefixes of π, going from π^1 to π. More precisely, let $l = |TC|$. The function $O_r(\pi)$ is the area under the curve created by connecting the following points sequentially: $\big(1, risk(\pi^1, cl)\big), \ldots, \big(i, risk(\pi^i, cl)\big), \ldots, \big(l, risk(\pi, cl)\big)$. Fig. 6(a) illustrates $O_r(\pi)$. By lowering the value of this function, we favor test suites that lead to late and slow increases in risk. Hence, to minimize the risk incurred over the test cases appearing early in π, we need to minimize $O_r(\pi)$.

Our second fitness function, $O_c(\pi)$, ensures that as many critical test cases as possible appear as early as possible in a test suite. We define $O_c(\pi)$ as the *area under the curve* of the *crt* function applied to successive prefixes of π. For $O_r(\pi)$, as illustrated in Fig. 6(a), we used unit increments on the X-axis. For $O_c(\pi)$, in contrast, we scale the X-axis based on the execution time of the test suite prefixes. More precisely, $O_c(\pi)$ is the area under the curve created by connecting the following points sequentially: $\big(time(\pi^1, cl), crt(\pi^1)\big), \ldots, \big(time(\pi^i, cl), crt(\pi^i)\big), \ldots, \big(time(\pi, cl), crt(\pi)\big)$. Fig. 6(b) illustrates $O_c(\pi)$. The higher the value of $O_c(\pi)$, the more critical are the test cases appearing early in π. Further, by scaling the X-axis based on execution time, we reward early appearance of test cases that are not only critical but also *fast*. By doing so, under time budget constraints when we can only run a prefix of π, we ensure that both the *crt* of the prefix and the number of test cases in the prefix are optimized.
Computational Search. We use the Non-dominated Sorting Genetic Algorithm version 2 (NSGAII) algorithm [12]. NSGAII has been applied to several software engineering problems involving optimization. The output of NSGAII is a non-dominating (equally viable) set of solutions, representing the best tradeoffs found among the given objectives. This is referred to as a *Pareto nondominated front* [28], where the dominance relation over solutions is defined as follows: A solution x dominates another solution x' if x is not worse than x' in all fitnesses, and x is strictly better than x' in at least one fitness.

In our adaptation of NSGAII, we apply the ReduceTCS algorithm (see Fig. 4) to each individual test suite π generated during the search before computing fitness functions $O_c(\pi)$ and $O_r(\pi)$. That is, we first remove redundant *init* and *cleanup* methods from each candidate solution before computing the fitness functions.

To tailor NSGAII to our problem, we use the following standard genetic operators described in [44] that have been applied to many problems [37, 38]: (1) *Selection.* We use a binary tournament selection based on non-domination ranking and crowding distance. This selector has been used in the original implementation of NSGAII [12]. (2) *Crossover.* We use partially mapped crossover (PMX) [20], which ensures that the offsprings are valid permutations. (3) *Mutation.* We use the permutation swap. This mutation strategy interchanges two randomly-selected test cases based on a given mutation rate.

To summarize, our optimization solution receives as input a set of test cases, and generates a set of test suites that are equally viable and optimized based on our two fitness functions, while also free of redundant *init* and *cleanup* methods.
Running test suites under time budget constraints. Acceptance testing is typically subject to time budget (duration) constraints. Due to the way our fitness functions $O_r(\pi)$ and $O_c(\pi)$ are defined, an optimal test suite π produced by our solution has the test cases with the largest risk as late as possible in the sequence, while it has the most critical test cases as early as possible in the sequence. For a given test budget T, the best partial test suite is therefore a prefix π^i of π such that π^i fits into T with a certain level of confidence. Specifically, since execution time is probabilistic, the engineers should pick the prefix π^i such that $time(\pi^i, cl) < T$ for a given confidence level cl. In Section 4, we examine the quality of both the full test suites generated by our solution and prefixes of these test suites that fit into selected time budgets.

4 EMPIRICAL EVALUATION

In this section, we empirically evaluate our test case prioritization approach through a real case study from the satellite domain. Our (sanitized) case study data is available online [43].

4.1 Research Questions (RQs)

RQ1 (sanity check): *How does our (NSGAII-based) test suite generation approach fare against random search?* This RQ is an important "sanity check" for search-based solutions [2, 23]. A search-based solution is expected to significantly outperform naive random search.
RQ2 (impact of redundancy removal): *How does our test suite generation approach perform with and without the removal of redundant init and cleanup method calls?* With this RQ, we evaluate the impact of the ReduceTCS algorithm (Fig. 4). As illustrated in Section 3.2 through an example, given a fixed set of test cases, ReduceTCS may eliminate more *init* and *cleanup* method calls for some permutations than others. RQ2 evaluates the reductions brought about by ReduceTCS in execution time and hardware damage risks for different permutations generated via search.
RQ3 (usefulness): *How do test suites generated by our approach compare with test suites built manually by expert engineers?* For our approach to be useful, the test suites it generates must present an advantage over those developed manually by engineers with advanced domain knowledge. RQ3 examines the quality of test suites generated by our approach against a manually-constructed test suite by satellite engineers.

4.2 Industrial Study Subject

We evaluate our approach by applying it to the in-orbit testing case study. As discussed in Section 2, in-orbit testing is a CPS acceptance testing process. Our evaluation is based on seven representative test case specifications for in-orbit testing developed by SES Techcom.

These seven test case specifications use 15 different components, a subset of which were listed in Table 1. The components used have between three to 20 variables of types float and enumeration. Each test case specification has five input parameters of type float.

The engineers involved in our study instantiated each test case specification with varying parameter values. The goal was to test a total of 40 satellite channels with different frequencies (in the 11.70–12.50GHz range) and different polarizations. The size of the full test suite defined by the engineers, i.e., the number of test cases, is 242. The test cases in the test suite were ordered manually by the engineers. We denote this reference test suite by R.

The actual test cases that SES Techcom intends to run during in-orbit testing have been implemented in Java. Naturally, we could not have our optimization approach work directly over these (hardware-in-the-loop) test cases – doing so would have defeated our purpose altogether. To realize our optimization approach, we *simulate* test case executions, and monitor the executions to keep track of how the components change states. To perform such simulation, we specified the test cases in a textual domain-specific language (DSL), which we designed in collaboration with SES Techcom. We submit alongside the paper anonymized DSL implementations of our test cases. Space restrictions prevent us from elaborating the DSL itself.

As we explained in Section 3.1, for each test case specification tc, we have three functions $time(tc)$, $risk(tc)$ and $crt(tc)$. These functions, again as explained in Section 3.1, induce functions $time(\pi)$, $risk(\pi)$ and $crt(\pi)$ for a partial or full test suite π. Below, we describe how we compute these functions in our case study.

For a test specification tc, we define $time(tc)$ based on log files from real-world executions of tc in previous in-orbit testing campaigns performed on satellites and infrastructures similar to ours. Specifically, the log files were obtained based on components that were identical or near-identical to our case study components, the same satellite orbital characteristics, and the same ground station for communicating with the satellite.

We extract from the log files a vector of execution time values (one value per prior execution of tc). The log files further provide information about the execution time of the individual operations in tc. From this information, we extract, for every component c used by tc, a vector of values for *init-time*(c) and a vector of values for *cleanup-time(c)*. Based on these functions, we are able to compute for any reduced test case tc' generated by the ReduceTCS algoritm a vector of values for *time(tc′)*.

To compute $time(\pi)$ for a test suite π, we use the simple Monte Carlo process [41] shown in the algorithm of Fig. 7. For $\pi = tc_1(par_1) \cdot \ldots \cdot tc_l(par_l)$, we have as input a vector $[v_1^i, \ldots, v_k^i]$ of execution time values for each tc_i $(1 \leq i \leq l)$. In each iteration, the algorithm randomly selects a value from each of the vectors. The algorithm then sums up the selected values. Given a number of iterations M, the resulting sums form a vector of values for $time(\pi)$. Note that the algorithm of Fig. 7 also applies to a reduced test suite π' (generated by ReduceTCS) when the input vectors are for the reduced test cases comprising π'.

Determining $risk(tc)$, and by extension, $risk(\pi)$ is a domain-specific task. In our case study, the engineers were specifically concerned with the impact of testing on the following: (1) the antenna's moving parts and (2) the mechanical switches. We thus

Algorithm. TestSuiteMonteCarlo
input: - A set $\{time(tc_1), \ldots, time(tc_l)\}$ such that $time(tc_i) = [v_1^i, \ldots, v_k^i]$ for $1 \leq i \leq l$
Output: - A distribution vector $time(\pi)$

1. **for** j = 1 to M **do**
2. **for** i = 1 to l **do**
3. randomly select some v^i from each $time(tc_i)$
4. $x_j = \sum_{i=1}^{l} v^i$
5. $time(\pi) = [x_1, \ldots, x_M]$

Figure 7: Monte Carlo process for creating $time(\pi)$.

elect to capture $risk(tc)$ by counting the number of commands that flip some switch or move the antenna. These counts can in principle differ across different executions of the same test case due to different parameter values and different environmental conditions. Upon an inspection of the source code for our seven test case specifications, we noticed that all commands that manipulate the switches or the antenna's position are outside conditional code segments. The engineers further confirmed that, as far as the switches and the antenna's moving parts were concerned, little influence was anticipated from environmental factors. Hence, we do not need to express *risk* as a probabilistic function in our case study, although our formalism in Section 3.1 provides the ability to do so.

For each test case specification tc in our case study, we thus obtain for *risk(tc)* a single value. Consequently, lifting *risk* to test suites is done by simple summation, be these regular test suites or test suites reduced by ReduceTCS. As we show in our case study results, reduced test suites considerably decrease the number of operations that could damage the switches or the antenna.

Finally, the $crt(tc)$ function was provided by the engineers. Specifically, based on expert knowledge, the engineers assigned a value of 1 (one) to *critical* test cases and a value of 0 (zero) to *non-critical* test cases. The reasoning that the engineers followed when assigning these values was that test case specifications related to the core satellite functions are critical, whereas those related to the backup functions are not. Among the seven test case specifications in our case study, five were deemed critical and two non-critical.

4.3 Multi-objective Evaluation Metrics

To assess the results of multi-objective search algorithms, we use the three quality indicators (QI) listed below based on existing guidelines in the literature [53]. To compute the QIs, we create a reference Pareto front as the union of all the non-dominated solutions obtained from all runs of the algorithms under comparison.

– *Generational Distance (GD)* measures the Euclidean distance between each point on a Pareto front output of a search algorithm and the nearest solutions on a reference Pareto front [49]. The *lower* the GD values, the more optimal the search outputs are.

– *Spread (SP)* measures the extent of spread among the points on a Pareto front output of a search algorithm [12]. The *lower* the SP values, the better spread out the search outputs are.

– *Hypervolume (HV)* represents the size of the space covered by members of a Pareto front generated by a search algorithm [6]. The *higher* the HV values, the better the Pareto front outputs are.

Following existing recommendations [2], we use Mann-Whitney U-test [36], and Vargha and Delaney's $\hat{A}_{12}$ effect size [47] to statistically compare search algorithms and to measure probabilistic superiority (effect size) between search algorithms, respectively. The level of significance (α) is set to 0.05. Two algorithms are considered to be equivalent when the value of $\hat{A}_{12}$ is 0.5.

4.4 Parameter Tuning and Setting

Our experiments involve two groups of parameters: *parameters of the fitness functions* and *parameters of the NSGAII algorithm*.

Parameters of the fitness functions. Recall from Section 4.2 that in our case study only *time* is a probabilistic distribution, and *risk* can be estimated as a deterministic function. To compute the *time* function for test suites, we generate vectors with 1000 values (i.e., $M = 1000$ in Fig. 7). With a vector size of 1000, we obtain a 95% confidence interval (CI) precision [18] of $1\% \times \bar{x}$ where $\bar{x}$ is the mean of the values in the *time* vectors. In our work, we consider this level of precision acceptable. For example, the *time* function of the reference test suite (R) with 1000 values yields the 95% CI of $[51.95h, 52.24h]$ (i.e., precision of 8.7 min).

The second parameter required to compute our fitness functions in Fig. 6 is the confidence level variable, i.e., cl, used in $time(\pi, cl)$. We performed our experiments for cl = %100 (the most conservative estimate) and cl = %50 (the median estimate). We present our experiment results for cl = %100. We note that the results obtained based on cl = %50 do not change the answers to the research questions.

Parameters of NSGAII. For the NSGAII search parameters, we set the population size to 100, the mutation rate to 0.2 and the crossover rate to 0.8. These values are consistent with the guidelines in the literature [3]. Note that the parameters can be further tuned to improve the performance of our approach. However, since with our current setting, we were able to convincingly and clearly support our analysis, we do not report further experiments on tuning population size, and crossover and mutation rates.

To determine the total number of fitness evaluations, we performed an initial experiment. We ran our NSGAII algorithm for 50 times and each time for 50,000 fitness evaluations. Our results showed that, after around 25,000 fitness computations, the GD values reach a plateau (i.e., there is no notable difference in GD values after 25,000 fitness computations). Hence, in our experiments, we ran our algorithms for 25,000 fitness computations.

4.5 Experiments and Implementation

To answer the RQs in Section 4.1, we compared the following algorithms: (1) *NR*: NSGAII with ReduceTCS (our approach in Section 3.3); (2) *RR*: Random search with ReduceTCS; (3) *NX*: NSGAII without ReduceTCS; and (4) *RX*: Random search without ReduceTCS. We executed each of these four algorithms for 25,000 fitness computations. To account for randomness, we repeated each run of these algorithms for 50 times.

We implemented our NSGAII-based approach of Section 3.3 using the jMetal multi-objective optimization framework [13]. We ran our experiments over the high performance computing cluster [48] at the University of Luxembourg. Each run of our algorithms was executed on a different node of the cluster, and took about 10 hours. The running time is acceptable as our optimization approach can be executed offline in practice.

4.6 Experiment Results

RQ1. Fig. 8 shows the best Pareto front solutions obtained by 50 runs of *NR*, *RR*, *NX*, and *RX* after 25,000 fitness computations. To answer **RQ1**, we compare the algorithms that build on NSGAII search with those that build on Random search (i.e., *NR* with *RR*, and *NX* with *RX*). Fig. 8 indicates that *NR* and *NX* obtain significantly

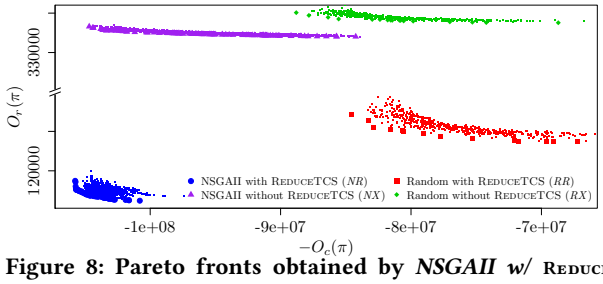


Figure 8: Pareto fronts obtained by *NSGAII w/* ReduceTCS *(NR), Random w/* ReduceTCS *(RR), NSGAII w/o* ReduceTCS *(NX),* and *Random w/o* ReduceTCS *(RX).*

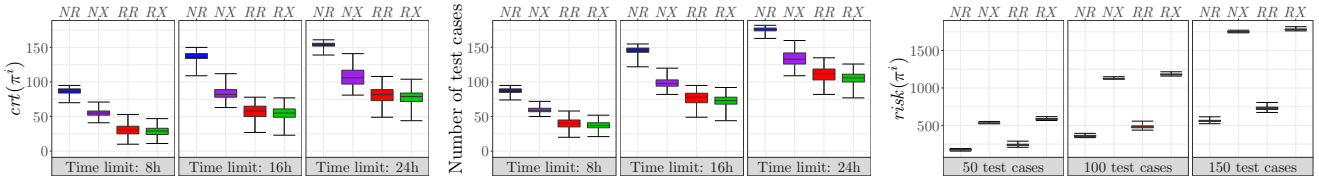
Table 2: Comparing *NR*, *NX*, *RR*, and *RX* using quality indicators (GD, SP, and HV).

Pair A vs. B	QI	p-value	$\hat{A}_{12}$ [95% CI]	Mean QI A	Mean QI B
NR vs. *RR*	GD	0.00	0.01 [0.00,0.07]	**0.06**	1.96
	SP	0.03	0.38 [0.27,0.49]	**0.88**	0.91
	HV	0.00	1.00 [0.97,1.00]	**0.37**	0.00
NX vs. *RX*	GD	0.00	0.01 [0.00,0.07]	**0.01**	0.49
	SP	0.00	0.01 [0.00,0.07]	**0.69**	0.88
	HV	0.00	1.00 [0.97,1.00]	**0.60**	0.00
NR vs. *NX*	GD	0.00	0.01 [0.00, 0.07]	**0.06**	3.09
	SP	0.00	0.03 [0.00,0.09]	**0.88**	0.98
	HV	0.00	1.00 [0.97,1.00]	**0.37**	0.00

better Pareto front solutions compared to *RR* and *RX*, respectively. Further, as shown in Table 2, the GD, SP and HV values computed based on the outputs of *NR* and *NX* are significantly better than those computed based on the outputs of *RR* and *RX*, respectively. For all comparisons, the p-values are less than 0.05, and the $\hat{A}_{12}$ [95% CI] values show large effect sizes.

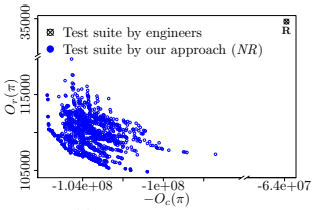
Fig. 8 and Table 2 compare *full* test suite solutions. However, as discussed in Section 3.3, in practice, engineers often execute a prefix of a full test suite depending on the available time budget for testing. We consider two ways to truncate *full* test suites: (1) based on *time limits*, and (2) based on *size limits*. Specifically, we consider T_1 = 8h, T_2 =16h and T_3 = 24h as three typical time limits from past satellite in-orbit testing projects. We further consider the following three size limits for test suites: 50, 100 and 150. These size limits reflect the number of test cases that engineers can typically run within the time limits T_1, T_2 and T_3. For each full test suite π generated by *NR*, *NX*, *RX* and *RR* across their 50 different runs, we computed three partial test suites π^{i_1}, π^{i_2} and π^{i_3} based on the time limits T_1, T_2, and T_3, respectively. Specifically, for each test suite π^{i_k}, $k \in \{1, 2, 3\}$, we have $time(\pi^{i_k}, cl) < T_k$ where cl = %100. Further, we computed partial test suites π^{50}, π^{100} and π^{150} based on the size limits 50, 100 and 150, respectively, by truncating full test suites generated by 50 runs of *NR*, *NX*, *RX* and *RR*.

We compare partial test suites obtained based on time limits with respect to the following two metrics: (1) the test suite criticality, i.e., $crt(\pi^{i_k})$; and (2) the number of test cases in π^{i_k}, i.e., $|\pi^{i_k}|$. These metrics capture our objectives to increase the number of critical test cases as well as the total number of test cases we can run within a given time limit. Note that as discussed in Section 4.2, in our case study, *crt* is a binary function assigning zero to non-critical and one to critical test cases. Hence, in our results, $crt(\pi^{i_k})$ is the number of *critical* test cases in a partial test suite π^{i_k}. Figs. 9(a) and 9(b)



(a) $crt(\pi^i)$ (b) $|\pi^i|$ (c) $risk(\pi^i)$

Figure 9: Comparing the *NR*, *RR*, *NX*, and *RX* algorithms based on (a) $crt(\pi^i)$, (b) $|\pi^i|$, and (c) $risk(\pi^i)$.



(a) Fitness values

Metric	Limit	*NR* (mean)	Practice
$crt(\pi^i)$	time:8h	86.80	20.00
	time:16h	137.28	40.00
	time:24h	153.81	60.00
$\|\pi^i\|$	time:8h	87.51	29.00
	time:16h	145.28	59.00
	time:24h	175.60	89.00
$risk(\pi^i)$	size:50	175.67	594.00
	size:100	359.49	1192.00
	size:150	560.76	1792.00

(b) $crt(\pi^i)$, $|\pi^i|$, and $risk(\pi^i)$

Figure 10: Comparing test suites prioritized by *NR* against the test suite manually prioritized by the engineers: (a) fitness values for full test suites, (b) $crt(\pi^i)$, $|\pi^i|$, and $risk(\pi^i)$ values for partial test suites under time and size limits.

respectively show the number of critical test cases (which, in our case study, coincides with $crt(\pi^{i_k})$) and the total number of test cases ($|\pi^{i_k}|$) obtained for different time limits. Since our approach selects mostly critical test cases within time limits, the diagrams in Figs. 9(a) and 9(b), in particular for the time limit 8h, look similar.

To compare the algorithms with respect to *risk* values, we use partial test suites obtained based on size limits. This is because, in our analysis of damage risks, we do not want to provide any advantage to algorithms that run more test cases. In other words, we intend to know whether an algorithm is better at minimizing damage risks, independently of test suite size. Fig. 9(c) compares the values of the *risk* function for partial test suites π^{50}, π^{100} and π^{150}.

As indicated by Figs. 9(a)–9(c), for both *crt* and length metrics, *NR* outperforms *RR*, and *NX* outperforms *RX*. Further, the partial test suites generated by *NR* and *NX* pose less *risk* than those generated by *RR* and *RX*, respectively. On average *NR*, when compared to *RR*, increases test suite lengths and average *crt* values by 89.3% and 138.6%, respectively, while it decreases *risk* by 25.3%. Similarly, *NX*, when compared to *RX*, on average increases test suite lengths and average *crt* values by 41.1% and 61.4%, respectively, while it decreases *risk* by 5.4%.

The comparison results for other time and size limits are similar to those shown in Figs. 9(a)–9(c), and hence omitted due to space.

The answer to **RQ1** *is that* NSGAII outperforms Random search in prioritizing full test suites. Further, under time and size limits, NSGAII generates partial test suites which, compared to those generated by Random search, contain on average *at least* 41.1% more test cases and 61.4% more critical test cases, and pose 5.4% less risk of hardware damage.

RQ2. We compare *NR* (NSGAII *with* ReduceTCS) against *NX* (NSGAII *without* ReduceTCS). As shown in Fig. 8, *NR* outperforms *NX* for both O_c and O_r, but more significantly for O_r. Further, comparing the quality indicators in Table 2 shows that *NR* solutions are significantly better than *NX* solutions with respect to all GD, SP and HV. Additionally, the results in Figs. 9(a)–9(c) show that *NR*, when compared to *NX*, on average improves test suite lengths and average *crt* values by 41.5% and 54.9%, respectively, while it decreases *risk* by 67.6%.

The answer to **RQ2** *is that* ReduceTCS combined with NSGAII outperforms NSGAII in prioritizing full test suites. Further, under time and size limits, ReduceTCS combined with NSGAII generates partial test suites which, compared to those generated by NSGAII, contain on average 41.5% more test cases and 54.9% more critical test cases, and pose 67.6% less risk of hardware damage.

RQ3. We compare test suites prioritized by *NR* against the reference test suite *R* from SES Techcom (see Section 4.2). Fig. 10(a) depicts, with respect to our search fitnesses, *R* alongside the best full test suite solutions computed by *NR*. As can be seen from the figure, *NR* outputs outperform *R* for both fitnesses (O_c and O_r).

Fig. 10(b) shows the average *crt*, length and *risk* values obtained from *NR* outputs under time and size limits. In the figure, we compare these values against current practice, i.e., the corresponding values obtained from *R*. More precisely, we compare against the *crt* and length values of *R* when it is truncated under time limits T_1 = 8h, T_2 =16h and T_3 = 24h. Similarly, we compare against the *risk* values of *R* when it is truncated to 50, 100 and 150 test cases. The results in Fig. 10(b) indicate that *NR*, on average and compared to the reference test suite *R*, increases the lengths and the *crt* values by 148.4% and 244.5%, respectively, and decreases *risk* by 69.7%.

The answer to **RQ3** *is that* prioritizing the full test suite using our approach yields results that are better than the reference test suite defined by expert engineers. Further, under time and size limits, our approach generates partial test suites which, compared to manually-defined ones, contain on average 148.4% more test cases and 244.5% more critical test cases, and pose 69.7% less risk of hardware damage.

4.7 Threats to Validity

Conclusion validity: The main threats to conclusion validity arise from not accounting for random variation and inappropriate use of statistics. We mitigate these threats by following standard guidelines in search-based software engineering [3]. We ran each search algorithm 50 times. We sampled 1000 values to generate the probabilistic *time* function and showed that this sample size achieves high precision (see Section 4.4). Statistical comparisons are based on Mann-Whitney U-test, and Vargha and Delaney's $\hat{A}_{12}$.

Internal validity: To mitigate risks posed by confounding factors, we compared different algorithms under identical parameter settings. We present all the underlying parameters, and provide along

all our experimental data to facilitate reproducibility. We mitigate potential biases and errors in our experimental data by drawing on real test cases and on real historical data for execution times.

Construct validity: The main threat to construct validity is posed by unsuitable or ill-defined metrics. To this end, we note that we use standard quality indicators (GD, SP and HV) and Pareto-front analysis methods for answering our research questions. Further, our metrics for execution time, criticality, and risk of hardware damage are defined based on common engineering practices.

External validity: The main threat to external validity is that our results may not generalize to other contexts. We distinguish two dimensions for external validity: (1) the applicability of our approach beyond our case study system, and (2) obtaining the same level of benefits as observed in our case study. As for the first dimension, we note that we provide in Section 3.1 a precise formalization of the context upon which we build. Our approach applies to any CPS whose acceptance testing context is expressible using our formalism. The formalism is generic and based on simple assumptions that can be accommodated by many systems. With respect to the second dimension, while our case study was performed in a representative industrial setting, additional case studies remain necessary to further validate our approach.

5 RELATED WORK

In this section, we discuss and compare with different strands of related research in the area of system and software testing.

Acceptance testing has been primarily studied in the context of agile software development methodologies (e.g., XP programming and scrum) and based on user acceptance criteria [10, 11, 17, 25, 26, 30–33, 45, 54, 56]. There is little work on test case prioritization for acceptance testing. An exception is the work of Srikanth et al. [45] that relies on field failure information extracted from historical data to prioritize test cases for service-oriented and cloud-based applications. Unlike this approach, we study acceptance testing in a CPS context, accounting for such factors as constraints on the duration of testing, uncertainty in the environment, and the impact of testing on hardware.

Test case minimization has been used to refer to both removing redundant test cases from test suites [5, 15, 19, 24, 55] and removing (redundant) operations from individual test cases [21, 29, 40]. In our work, test case minimization means the latter. Broadly, techniques in the latter group rely on white-box information extracted from software code (e.g., fault detection or structural code coverage criteria) to minimize individual test cases in order to improve readability, debugging and manual effort of test oracles [4]. In contrast, the test case minimization component of our approach (i.e., ReduceTCS) relies on CPS acceptance testing criteria only. Further, test case minimization in our work is motivated by reducing execution time and hardware damage risks.

Some prior studies attempt to minimize the setup cost for testing. In particular, in the context of software regression testing, Hsu and Orso [27] develop MINTS that minimizes *manual* setup efforts (man-hour) required to build emulators. In contrast to MINTS, in our work, setup is an *automated* sequence repeated at the beginning of each test case and brings hardware to a desired state. Raghavan et al. [39] minimizes test cases by removing test setup commonalities. In contrast to Raghavan et al. [39], we automatically remove redundant setup/teardown operations by keeping track of the impact of test scenarios on component states. We further account for hardware damage risks and uncertainty. Finally, we evaluate our work in the context of a real CPS case study.

Test case prioritization is widely studied for software regression testing [7, 8, 14, 22, 55]. The research threads that most closely relate to our work are those concerned with prioritizing test cases with respect to resource constraints. Most such techniques consider only test execution time as a resource, with the remaining prioritization objectives being based on the source code of the software under test. For example, alongside test execution time, Walcott et al. [51] use a (code-based) fault detection objective, and Zhang et al. [57] and Turner et al. [46] use objectives defined based on structural code coverage. Since these techniques require observability into the source code of the system under test, they are not readily applicable in the context of CPS acceptance testing, as targeted by our work. Recently, Wang et al. [52] have considered in their prioritization objectives the utilization of hardware resources. None of the above techniques, however, consider hardware damage. Minimizing the risk of such damage during testing is an essential consideration for many CPS. In addition and at a more conceptual level, our work improves on the existing strands of work in the following ways: First, our approach has built into it a minimization algorithm for removing redundant initialization and teardown operations. Second, through a lightweight formalism, we develop a generalizable characterization of test case prioritization in the context of acceptance testing. Finally, we provide direct support for handling uncertainty in test case prioritization. This is important for coping with CPS characteristics, and may further be useful in contexts other than that of our current work.

6 CONCLUSIONS

This paper was concerned with optimizing acceptance testing for cyber physical systems, taking into account four key factors: test time budget, uncertainty in the environment, test case criticality, and hardware damage risks. We provided a precise characterization of acceptance testing in the context of cyber physical systems. Drawing on this characterization, we devised a multi-objective search-based solution for test case prioritization. This solution has built into it a mechanism for removing redundant operations from test case sequences. Our empirical results obtained over a real case study from the satellite domain indicate that, compared to test cases that are prioritized manually by expert engineers, our approach more than doubles the number of test cases that fit into a given time frame, while at the same time reducing to less than one third the operations with an entailed risk of damage to important hardware components.

ACKNOWLEDGMENTS

This project has received funding from the Luxembourg National Research Fund under the grant C-16PPP/IS/11270448 and the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 694277).

REFERENCES

[1] Paul Ammann and Jeff Offutt. 2016. *Introduction to Software Testing* (2 ed.). Cambridge University Press.

[2] Andrea Arcuri and Lionel Briand. 2014. A Hitchhiker's Guide to Statistical Tests for Assessing Randomized Algorithms in Software Engineering. *Software Testing Verification and Reliability* 24, 3 (2014), 219–250.

[3] Andrea Arcuri and Gordon Fraser. 2011. On Parameter Tuning in Search Based Software Engineering. In *Proceedings of the 3rd International Conference on Search Based Software Engineering (SSBSE'11)*. 33–47.

[4] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* 41 (2015), 507–525.

[5] Jennifer Black, Emanuel Melachrinoudis, and David Kaeli. 2004. Bi-Criteria Models for All-Uses Test Suite Reduction. In *Proceedings of the 26th International Conference on Software Engineering (ICSE'04)*. 106–115.

[6] Dimo Brockhoff, Tobias Friedrich, and Frank Neumann. 2008. Analyzing Hypervolume Indicator Based Algorithms. In *Proceedings of the 10th International Conference on Parallel Problem Solving from Nature (PPSN X)*. 651–660.

[7] Benjamin Busjaeger and Tao Xie. 2016. Learning for Test Prioritization: An Industrial Case Study. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'16)*. 975–980.

[8] Cagatay Catal and Deepti Mishra. 2013. Test Case Prioritization: A Systematic Mapping Study. *Software Quality Journal* 21, 3 (2013), 445–478.

[9] Lee Copeland. 2003. *A Practitioner's Guide to Software Test Design*. Artech House.

[10] Lisa Crispin, Tip House, and Carol Wade. 2001. The Need for Speed: Automating Acceptance Testing in an Extreme Programming Environment. In *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP'01)*. 96–104.

[11] Fred D. Davis and Viswanath Venkatesh. 2004. Toward Preprototype User Acceptance Testing of New Information Systems: Implications for Software Project Management. *IEEE Transactions on Engineering Management* 51 (2004), 31–46. Issue 1.

[12] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. 2002. A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation* 6, 2 (2002), 182–197.

[13] Juan J. Durillo and Antonio J. Nebro. 2011. jMetal: A Java framework for multi-objective optimization. *Advances in Engineering Software* 42 (2011), 760–771.

[14] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. 2002. Test Case Prioritization: A Family of Empirical Studies. *IEEE Transactions on Software Engineering* 28, 2 (2002), 159–182.

[15] Emelie Engström, Per Runeson, and Mats Skoglund. 2010. A Systematic Review on Regression Test Selection Techniques. *Information and Software Technology* 52, 1 (2010), 14–30.

[16] Filomena Ferrucci, Mark Harman, Jian Ren, and Federica Sarro. 2013. Not Going to Take This Anymore: Multi-objective Overtime Planning for Software Engineering Projects. In *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)*. 462–471.

[17] Malte Finsterwalder. 2001. Automating Acceptance Tests for GUI Applications in an Extreme Programming Environment. In *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP'01)*. 114–117.

[18] Ronald A. Fisher. 1959. *Statistical Methods and Scientific Inference*. Oliver & Boyd.

[19] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Practical Regression Test Selection with Dynamic File Dependencies. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA'15)*. 211–222.

[20] David E. Goldberg and Robert Lingle Jr. 1985. Alleles, Loci and the Traveling Salesman Problem. In *Proceedings of the 1st International Conference on Genetic Algorithms (ICGA'85)*. 154–159.

[21] Alex Groce, Mohammad Amin Alipour, Chaoqiang Zhang, Yang Chen, and John Regehr. 2016. Cause Reduction: Delta Debugging, Even Without Bugs. *Software Testing, Verification & Reliability* 26, 1 (2016), 40–68.

[22] Dan Hao, Lu Zhang, and Hong Mei. 2016. Test-case prioritization: achievements and challenges. *Frontiers of Computer Science* 10, 5 (2016), 769–777.

[23] Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. 2012. Search-based Software Engineering: Trends, Techniques and Applications. *Comput. Surveys* 45, 1, Article 11 (2012), 61 pages.

[24] M. Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. 1993. A Methodology for Controlling the Size of a Test Suite. *ACM Transactions on Software Engineering and Methodology* 2, 3 (1993), 270–285.

[25] Børge Haugset and Geir Kjetil Hanssen. 2008. Automated Acceptance Testing: A Literature Review and an Industrial Case Study. In *Proceedings of the Agile 2008 (AGILE'08)*. 27–38.

[26] Børge Haugset and Tor Stålhane. 2012. Automated Acceptance Testing as an Agile Requirements Engineering Practice. In *Proceedings of the 45th Hawaii International Conference on System Science (HICSS'12)*. 5289–5298.

[27] Hwa-You Hsu and Alessandro Orso. 2009. MINTS: A general framework and tool for supporting test-suite minimization. In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)*. 419–429.

[28] Joshua D. Knowles and David W. Corne. 2000. Approximating the Nondominated Front Using the Pareto Archived Evolution Strategy. *Evolutionary Computation* 8, 2 (2000), 149–172.

[29] Andreas Leitner, Manuel Oriol, Andreas Zeller, Ilinca Ciupa, and Bertrand Meyer. 2007. Efficient Unit Test Case Minimization. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07)*. 417–420.

[30] Hareton K. N. Leung and Peter W. L. Wong. 1997. A Study of User Acceptance Tests. *Software Quality Journal* 6, 2 (1997), 137–149.

[31] Grischa Liebel, Emil Alégroth, and Robert Feldt. 2013. State-of-Practice in GUI-based System and Acceptance Testing: An Industrial Multiple-Case Study. In *Proceedings of the 39th Euromicro Conference on Software Engineering and Advanced Applications (SEAA'13)*. 17–24.

[32] Olga Liskin, Christoph Herrmann, Eric Knauss, Thomas Kurpick, Bernhard Rumpe, and Kurt Schneider. 2012. Supporting Acceptance Testing in Distributed Software Projects with Integrated Feedback Systems: Experiences and requirements. In *Proceedings of the 7th International Conference on Global Software Engineering*. 84–93.

[33] Renate Löffler, Baris Güldali, and Silke Geisen. 2010. Towards Model-based Acceptance Testing for Scrum. *Softwaretechnik-Trends* 30, 3 (2010).

[34] Sean Luke. 2013. *Essentials of Metaheuristics* (second ed.). Lulu. Available for free at http://cs.gmu.edu/~sean/book/metaheuristics/.

[35] Qi Luo, Kevin Moran, and Denys Poshyvanyk. 2016. A Large-scale Empirical Comparison of Static and Dynamic Test Case Prioritization Techniques. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'16)*. 559–570.

[36] Henry B. Mann and Donald R. Whitney. 1947. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *Annals of Mathematical Statistics* 18, 1 (1947), 50–60.

[37] Alessandro Marchetto, Md. Mahfuzul Islam, Waseem Asghar, Angelo Susi, and Giuseppe Scanniello. 2016. A Multi-objective Technique to Prioritize Test Cases. *IEEE Transactions on Software Engineering* 42, 10 (2016), 918–940.

[38] Dario Di Nucci, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. 2015. Hypervolume-based Search for Test Case Prioritization. In *Proceedings of the 7th International Symposium on Search-Based Software Engineering (SSBSE'15)*. 157–172.

[39] Vijaya Raghavan, Mojdeh Shakeri, and Krishna Pattipati. 1999. Optimal and Near-Optimal Test Sequencing Algorithms with Realistic Test Models. *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans* 29 (1999), 11–26. Issue 1.

[40] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case Reduction for C Compiler Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'12)*. 335–346.

[41] Christian P. Robert and George Casella. 2005. *Monte Carlo Statistical Methods*. Springer.

[42] Gregg Rothermel, Roland J. Untch, and Chengyun Chu. 2001. Prioritizing Test Cases For Regression Testing. *IEEE Transactions on Software Engineering* 27, 10 (2001), 929–948.

[43] Seung Yeob Shin, Shiva Nejati, Mehrdad Sabetzadeh, Lionel C. Briand, and Frank Zimmer. 2018. [Case study data] Test Case Prioritization for Acceptance Testing of Cyber Physical Systems: A Multi-objective Search-Based Approach. https://github.com/seungyeob/issta2018-mosis.

[44] S. N. Sivanandam and S. N. Deepa. 2007. *Introduction to Genetic Algorithms*. Springer.

[45] Hema Srikanth, Mikaela Cashman, and Myra B. Cohen. 2016. Test case prioritization of build acceptance tests for an enterprise cloud application: An industrial case study. *Journal of Systems and Software* 119 (2016), 122–135.

[46] Andrew J. Turner, David R. White, and John H. Drake. 2016. Multi-objective Regression Test Suite Minimisation for Mockito. In *Proceedings of the 8th International Symposium on Search-Based Software Engineering (SSBSE'16)*. 244–249.

[47] András Vargha and Harold D. Delaney. 2000. A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132.

[48] S. Varrette, P. Bouvry, H. Cartiaux, and F. Georgatos. 2014. Management of an Academic HPC Cluster: The UL Experience. In *Proceedings of the 2014 International Conference on High Performance Computing & Simulation (HPCS'14)*. 959–967.

[49] David A. Van Veldhuizen and Gary B. Lamont. 1998. *Multiobjective Evolutionary Algorithm Research: A History and Analysis*. Technical Report TR-98-03. Air Force Institute of Technology, Wright-Patterson AFB, OH.

[50] Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Lerda. 2003. Model Checking Programs. *Automated Software Engineering* 10, 2 (2003), 203–232.

[51] Kristen R. Walcott, Mary Lou Soffa, Gregory M. Kapfhammer, and Robert S. Roos. 2006. Time-aware Test Suite Prioritization. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis (ISSTA'06)*. 1–12.

[52] Shuai Wang, Shaukat Ali, Tao Yue, Øyvind Bakkeli, and Marius Liaaen. 2016. Enhancing Test Case Prioritization in an Industrial Setting with Resource Awareness and Multi-objective Search. In *Proceedings of the 38th International Conference on*

Software Engineering Companion (ICSE'16). 182–191.

[53] Shuai Wang, Shaukat Ali, Tao Yue, Yan Li, and Marius Liaaen. 2016. 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 (ICSE'16)*. 631–642.

[54] Johannes Weiss, Alexander Schill, Ingo Richter, and Peter Mandl. 2016. Literature Review of Empirical Research Studies within the Domain of Acceptance Testing. In *Proceedings of the 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA'16)*. 181–188.

[55] Shin Yoo and Mark Harman. 2012. Regression Testing Minimization, Selection and Prioritization: A Survey. *Software Testing, Verification & Reliability* 22, 2 (2012), 67–120.

[56] Yi Yu and Fangmei Wu. 1999. A Software Acceptance Testing Technique Based on Knowledge Accumulation. In *Proceedings of the 9th Great Lakes Symposium on VLSI*.

[57] Lu Zhang, Shan-Shan Hou, Chao Guo, Tao Xie, and Hong Mei. 2009. Time-aware Test-case Prioritization Using Integer Linear Programming. In *Proceedings of the 2009 International Symposium on Software Testing and Analysis (ISSTA'09)*. 213–224.