IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received October 21, 2018, accepted January 10, 2019, date of publication January 17, 2019, date of current version February 6, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2893209

sOrTES: A Supportive Tool for Stochastic Scheduling of Manual Integration Test Cases

SAHAR TAHVILI[1], RITA PIMENTEL[1], WASIF AFZAL[2], MARCUS AHLBERG[3], ERIC FORNANDER[3], AND MARKUS BOHLIN[1]
[1]RISE SICS Västerås, 722 12 Västerås, Sweden
[2]School of Innovation, Design and Engineering, Mälardalen University, 722 20 Västerås, Sweden
[3]KTH Royal Institute of Technology, 114 28 Stockholm, Sweden

Corresponding author: Sahar Tahvili (sahar.tahvili@ri.se)

This work was supported in part by ECSEL & VINNOVA through projects XIVT, TESTOMAT, and MegaM@RT2, in part by the Swedish Knowledge Foundation through projects TOCSYC and TESTMINE, and in part by ERCIM "Alain Bensoussan" Fellowship Programme.

ABSTRACT The main goal of software testing is to detect as many hidden bugs as possible in the final software product before release. Generally, a software product is tested by executing a set of test cases, which can be performed manually or automatically. The number of test cases which are required to test a software product depends on several parameters such as the product type, size, and complexity. Executing all test cases with no particular order can lead to waste of time and resources. Test optimization can provide a partial solution for saving time and resources which can lead to the final software product being released earlier. In this regard, test case selection, prioritization, and scheduling can be considered as possible solutions for test optimization. Most of the companies do not provide direct support for ranking test cases on their own servers. In this paper, we introduce, apply, and evaluate sOrTES as our decision support system for manual integration of test scheduling. sOrTES is a Python-based supportive tool which schedules manual integration test cases which are written in a natural language text. The feasibility of sOrTES is studied by an empirical evaluation which has been performed on a railway use-case at Bombardier Transportation, Sweden. The empirical evaluation indicates that around 40% of testing failure can be avoided by using the proposed execution schedules by sOrTES, which leads to an increase in the requirements coverage of up to 9.6%.

INDEX TERMS Software testing, integration testing, test optimization, decision support systems, stochastic test scheduling, manual testing, scheduler algorithm, dependency.

I. INTRODUCTION

The crucial role of software testing in a sustainable software development cannot be ignored. To make a more effective and efficient testing process, several factors should be considered. One of the most important factors is estimating the total required cost for testing a software product, which can be a step toward justifying any software testing initiative. The software testing costs can be classified as fixed costs (including test team salaries, tester training, testing environment and automated testing tools) or variable costs, which deals with the troubleshooting and re-execution efforts [1].

Reducing the fixed testing cost is more related to the organization's policies and procedures, whereas minimizing the testing variable cost is an optimization problem directly impacted by the test efficiency. In an inefficient testing process, a wide range of redundant test cases is created and thereby a large number of redundant executions can occur during the testing phase. In recent years, utilizing different techniques for test case selection, prioritization and test suit minimization has received much attention. By using the above mentioned aspects, we can also address other testing issues such as earlier fault detection [2] and faster release of the software products [3].

Sequencing and scheduling are a form of dynamic and continuous decision making with an industrial applicability. The dynamic decision-making approach fits the environment that changes over time and the previous decision might affect the new decision [4]. The test scheduling problem can be considered as a dynamic decision-making problem which can be a practical solution for minimizing the redundant execution at industry. Monitoring the test records (results) of the previously executed test cases can affect the results of the new test cases, especially in the integration testing level, where test cases are more interdependent. Knowing the test results (pass or fail) for each test case can help testers to select a right test candidate for execution. However, monitoring and decision-making process on a large set of test cases is both a challenging and cost consuming process.

Through analyzing several industrial case studies, seen in [2], [5], and [6], it has been proven that the problem of test optimization is a multi-criteria decision making problem, which can be applied to the dynamic test scheduling problem as well. Measuring the effect of several criteria on the test cases and also dynamically scheduling them is a challenging task which is addressed in this paper. In the present work, we introduce, apply and evaluate sOrTES (Stochastic Optimizing TEstcase Scheduling), as an automated decision support system for test scheduling. sOrTES is a multi-criteria decision support system with a fast performance, which makes continuous execution decisions for manual integration of test cases. Furthermore, the feasibility of sOrTES is evaluated on a railway domain at Bombardier Transportation (BT) in Sweden.

This paper makes the following contributions: (i) detecting the dependencies between manual integration test cases, (ii) dynamically scheduling test cases through analyzing their execution results, (iii) increasing the requirements coverage up to 9.6% and (iv) decreasing the total required troubleshooting time to 40%.

The organization of this paper is laid out as follows: Section II provides a background of the initial problem and also an overview of research on test optimization, Section III describes the proposed approach. The structure of sOrTES is depicted in Section IV. An industrial case study has been designed in Section V. Section VI compares the performance between sOrTES, BT and a history-based test case prioritization approach. Threats to validity and delimitations are discussed in Section VII. Section VIII clarifies some points of future directions of the present work and finally Section IX concludes the paper.

II. BACKGROUND AND RELATED WORK

The concept of test optimization has received much attention in the past decade, which can be performed through several approaches such as test automation, test suite minimization, test case selection, prioritization and also test scheduling. However, all mentioned approaches are not applicable on all testing environments within industry. The tradeoff between the test optimization effort and the expected gain should be considered in an early stage of a testing process. For instance, changing the test procedure from manual to automated requires a huge effort and sometimes is not a proper decision in terms of fault detection [7]. Moreover, manual testing is a popular approach for testing the safety critical systems where the human judgment and supervision are superior to machines [8]. Among the mentioned test optimization aspects, test case selection, prioritization and scheduling can be applied almost in all industries where the testing process can be optimized through minimizing the required costs and effort for running the proposed approaches. Selecting a subset of generated test cases or ranking them for execution can lead to a more efficient usage of the allocated testing resources. The generated test cases for testing a software product have different quality attributes and therefore do not have the same values for execution. Identifying the properties of the test cases and measuring their value for execution can be considered a master key to solving the test optimization problem.

As explained in Section I, the test optimization problem is a multi-criteria and multi-objective decision making problem, where the properties of the test cases (e.g. execution time, requirement coverage) are the criteria and the targeted objectives are to maximize requirement coverage and detect faults earlier. Determining the critical criteria and the desirable objectives depends on several factors such as the test cases's size, complexity, diversity and also the testing procedure. For instance, the lines of code (LOC) can be considered as a metric to measure the size of a test script, which is not a valid criteria in a manual testing procedure. Furthermore, test-satisfying objectives can be changed during the testing process based on different test optimization aspects. In this section, we briefly discuss three different aspects of test optimization.

A. TEST CASE SELECTION

Generally, test case selection deals with choosing a subset of designed test cases for execution. Test case selection techniques can be used in exploratory testing, where the testers are involved in minimum planning and maximum test execution [9]. The problem of test case selection is formulated as follow by Yoo and Harman [10]:

Definition 1:

Given: The program, P, the modified version of P, P', and a test suite, T.

Problem: To find a subset of T, T', with which to test P'.

In other words, not all generated test cases need to be executed, as they also can be tested in some levels of testing such as acceptance testing, where all test cases are already executed at least once and just some test cases need to be selected (randomly) for execution. Test case selection can also be utilized for test automation, where a subset of test cases can be selected as good candidates for automation in the manual testing approach.

B. TEST CASE PRIORITIZATION

Ranking all designed test cases for execution is called test case prioritization, which can be applied in all testing levels such as unit testing, regression and system integration testing. The main goal of test prioritization is to give a higher priority to those test cases which have a better-quality attribute for execution. The following definition of test case prioritization is proposed by Yoo and Harman [10]:

Definition 2:

Given: A test suite, T, the set of permutations of T, PT and a function from PT to real numbers, $f : PT \to \mathbb{R}$.

Problem: To find a $T' \in PT$ that maximizes f.

Several objectives such as total testing time minimization and earlier fault detection can be satisfied through applying the test case prioritization techniques. Moreover, test cases can be prioritized for test automation in the manual testing approach, where the most critical manual test cases can be top ranked among all manual test cases.

C. TEST CASE STOCHASTIC SCHEDULING

Selecting a subset of test cases or prioritizing test cases for execution is usually performed offline and it is not a daily task during a testing process. However, optimizing test cases for execution without monitoring the test results (after execution) is not the most efficient approach in terms of test optimization. The results (pass or fail) of previously executed test cases can influence on the execution results of the new test cases. This problem can be seen clearly in the integration testing level, when the interactions between software modules are tested. There is a strong interdependency between integration test cases, which directly impacts the execution results of each other [6], [11]. This level of interdependency also influences on the process of the test optimization, where selecting and prioritizing test cases should satisfy the dependency constraints. Considering the above-mentioned issues, we opted to use a stochastic scheduling model for ranking and sequencing test cases for execution. The stochastic scheduling is a subset of an optimization problem, where the processing time of tasks are modeled as random variables [12], therefore a job's processing time is not known until it is completed [13]. In the test scheduling problem, a new execution decision should be made based on the results of the previously executed test cases. We proposed the following definition to the problem of test case stochastic scheduling:

Definition 3:

Given: A test suite, T. For all subset of T, $A \subseteq T$, the set of all permutations A, SPA. For all $B \subseteq T$, the set of all possible outputs after execution of the test cases in B, R. For each $r \in R$, the function $f_r : SPA \to \mathbb{R}$.

Problem: To find a prioritized set of T, T', considering the function $f_\emptyset : PT \to \mathbb{R}$, where PT is the set of permutations of T. To execute the test cases in T' until the first failure (if any).

To update the previous procedure for $T - T_p$, considering the function f_{r_e}, until $T_p = T$, where T_p is the set of passed test cases and r_e is the output of the executed test cases, respectively.

In other words, the executed test cases need to be saved in r_e and the prioritizing process should be continued until all generated test cases are executed at least one time.

Note that the main difference between Definition 2 and Definition 3 is monitoring the results of the test executions, which leads to a dynamic test optimization process. If no failures occur after the first execution then we only need to prioritize test cases once, according to Definition 2.

D. RELATED WORK

Test prioritization tries to optimally order a set of test cases for execution, typically by balancing criteria such as detecting faults as early as possible with minimal cost, which is dependent on execution time. However, in the most cases a selective subset of possible test cases is utilized for test prioritization.

Several test optimization techniques have been proposed in literature [10], while more and more techniques that utilize multi-objective and multi-criteria techniques are being proposed [14]. Walcott *et al.* [15], present time-aware test prioritization that balances execution time and code coverage. A similar approach is also presented in [16]. Wang *et al.* [17], introduce resource-aware multi-objective test prioritization where one cost measure (total time) and three effectiveness measures (prioritization density, test resource usage, fault detection capability) were defined and formulated into a fitness function. Although the Greedy algorithm may produce a suboptimal result, it has received much attention to be utilized for regression test case prioritization. Li *et al.* [18] are empirically investigated metaheuristic algorithms for the regression test case prioritization. Strandberg *et al.* [19] present a multiple factor automated system level regression test prioritization approach that combines multiple properties of tests such as test duration, previous fault detection success, interval since last executed and modifications to the code tested.

System-level test case prioritization has also been investigated in [20] where requirements coverage and/or volatility can be considered as one important prioritization criterion. It is interesting to note that during integration testing, dependencies between components and functions of the system under test becomes a critical criterion. Few studies have investigated test case prioritization based on such dependencies. Caliebe *et al.* [21] utilized a system graph based on the component dependency model and on path searching methods through the graph, where test cases were selected based on dependencies between the components in the system. Similarly, Haidry and Miller [22] prioritized functionally-dependent test cases based on different forms of the graph coverage values. In order to detect functional dependencies among integration test cases at an early stage, our earlier work [23] proposed, using natural language processing (NLP), to analyze multiple related artefacts (test specification, software requirement specification and relevant signaling information between functions under test).

Nahas and Bautista-Quintero [25] introduced Scheduler Test Case (STC) as a new technique which provides a systematic procedure for documenting and testing. STC supplies a black box tool which predicts the behavior of implementation sets of the real time scheduling algorithms. The STC uses several scheduling example tasks and also a subset of test cases in order to examine the expected behavior output of time trigged co-operative architectures.

The problem of test case prioritization is identified as a single objective optimization problem by Wong *et al.* [28], where all test cases are ranked based on their increasing cost per additional coverage. Srivastava and Thiagarajan [24] measured the changes that had been made to the program and prioritized test cases based on that. Moreover, the branch coverage has been identified several times as the most critical criteria by Do *et al.* [29], Elbaum *et al.* [30], [31], and

TABLE 1. Summary of related work.

Reference	Purpose of paper	Drawback
Wang et al. [17]	Uses total time, prioritization density, test resource usage, fault detection capability for test prioritization	Static prioritization which does not check the execution results of test cases
Strandberg et al. [19]	Utilizes a set of test cases' properties for regression test prioritization	Not efficient in terms of minimizing test failures
Caliebe et al. [21]	Selects a subset of test cases for execution based on component dependency models	The component dependency model might not be available in all testing environments
Srivastava and Thiagarajan [24]	Uses the changes that have been made to the program for test prioritization	Use of changes make it unfit for prioritization in an early stage of testing
Nahas and Bautista-Quintero [25]	Identifies a set of representative implementation classes for a TTC scheduler	Designing an appropriate set of test cases is a challenging task
Elbaum et al. [26]	Uses modules's history for test case prioritization	Focus on version specific test cases
Kim and Porter [27]	Ranks test cases based on their execution histories	Historical data of the test cases required

Rothermel *et al.* [26] where test cases are ranked based on a single criteria. History-based test prioritization is another single objective test optimization problem which is proposed by Kim and Porter [27].

Table 1 presents a summary of the related work. We propose a multi-criteria decision approach for scheduling test cases for execution, where most of the proposed approaches are a single objective approach. Furthermore, our proposed approach monitors the execution results of test cases and a new decision would be made after each failure.

III. PROPOSED APPROACH

In the present work, we introduce sOrTES as a supportive tool for stochastic test scheduling. sOrTES measures the interdependency between integration test cases and ranks them for execution based on their requirement coverage and execution time. A new schedule is proposed after each execution for the remaining test cases. As outlined earlier, there is a different number of critical criteria which influences the test cases. The criteria in the testing concept can be interpreted as a property for each test case, which creates a difference between test cases. The following criteria are utilized by sOrTES for the integration of test case scheduling:

- **Functional dependency between test cases:** test cases TC_1 and TC_2 are functionally dependent if they are designed to test different parts of function F_1 or if they are testing the interaction between functions F_1 and F_2. For instance, given two functions F_1 and F_2 of the same system, let the function F_2 be allowed to execute if its required conditions are already enabled by function F_1. Thus, function F_2 is dependent on function F_1. Consequently, all test cases which are designed to test F_2 should be executed any time after the assigned test cases for testing F_1. Detecting functional dependencies between test cases can lead to a more efficient use of testing resources by means of [1]:
  - avoiding redundant execution,
  - parallel execution for independent test cases,
  - simultaneous execution of test cases that test the same functionality,
  - any combination of the previous options.
- **Execution time:** is the total required time that each test case is allowed to take for execution. Test case's execution time can differ from one test to another. Knowing the execution time of test cases before execution can help test managers to divide test cases between several testers. Moreover, estimating the required time for each test case can provide a better overview of the total required time for testing a software product.
- **Requirement coverage:** as the title implies, shows the number of requirements which have been fulfilled by a test case. The coverage of requirements is a fundamental need throughout the software life cycle. Sometimes, a test case can test more than one requirement and sometimes several test cases are designed to test just one requirement.

Identifying and measuring the influences of the mentioned criteria on each test case requires a close collaboration with the testing experts at industries, which consumes time and resources. On the other hand, human judgment suffers from uncertainty but eliminating them from experiments would impact the results. One of the main objectives of designing sOrTES as a supportive tool is automatic measurement of the testing critical criteria, where the human judgment will be reduced and thereby a more trustable result will be produced. However, in some testing processes, there is no available information about the dependencies between test cases. In the case of a lack of a requirement traceability matrix in a testing level, the requirement coverage cannot be measured automatically. Moreover, the execution time for test cases, in most companies is only available after execution. We need to consider that, to have an efficient testing schedule, the effect of the mentioned criteria on the test cases should be measured in an early stage of a testing process and even before the first execution. The criteria measurement process can become even more complicated in a manual testing procedure. Analyzing a wide range of test specifications which are written by human with a variance in language and testing skills makes the problem more complex.

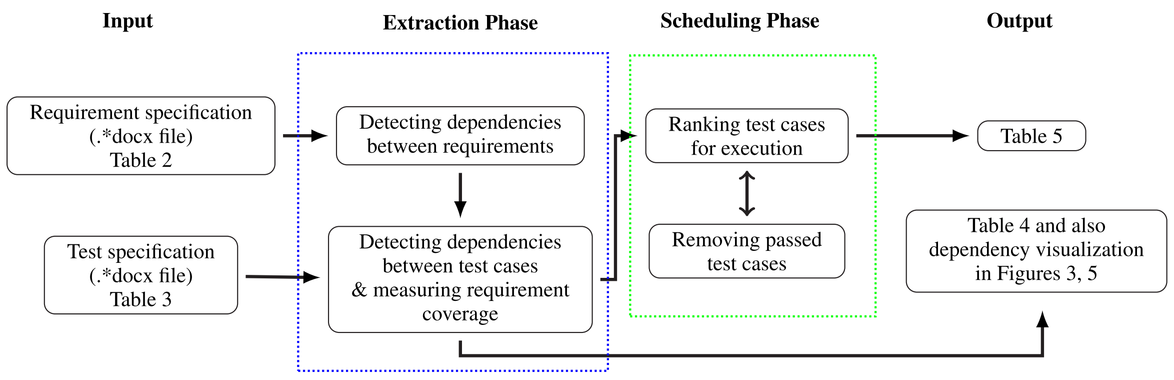

Input
Extraction Phase
Scheduling Phase
Output
Requirement specification (.*docx file) Table 2
Test specification (.*docx file) Table 3
Detecting dependencies between requirements
Detecting dependencies between test cases & measuring requirement coverage
Ranking test cases for execution
Removing passed test cases
Table 5
Table 4 and also dependency visualization in Figures 3, 5

FIGURE 1. The input-output and phases of sOrTES.

The problem of dependency detection between manual integration test cases has previously been solved through proposing several approaches such as a questionnaire based study, deep learning, natural language processing (NLP) and machine learning [6], [23], [32]. We also proposed an aiding tool called ESPRET[1] for estimating the execution time for manual integration test cases before the first execution [33]. Previously, test cases have been prioritized and ranked by adapting some ranking algorithms such as AHP[2] in [5] and [6] and TOPSIS[3] in [2]. Since test cases need to be scheduled for execution more than one time (daily scheduling), running the manual methods is not an optimal approach. The second main goal of designing sOrTES as a supportive tool is the need of fast and daily scheduling. Note that the test scheduling problem is a dynamic task, meaning the results of scheduled test cases in the first cycle (first day) can imply the scheduling plan for the second day due to the following two reasons:

1) The dependency structure between test cases is changing after a successful execution. In other words, when an independent test case is passed, the next dependent test case (to this independent test case) can be considered as an independent test case. Since the dependencies between test cases is changing continuously during a testing process, a new schedule based on the current dependencies status need to be proposed.
2) The passed test cases need to be removed from the testing cycle (testing pool) and the failed test cases need to be re-scheduled for the second-round execution. The failed test cases must be troubleshooted first and the time needed to accomplish this needs to be considered when scheduling the remaining test cases for execution.

IV. SORTES- STOCHASTIC OPTIMIZING TEST CASE SCHEDULING

sOrTES is a Python-based automated decision support system which consists of two separate phases: 1- extractor and 2-scheduler, that dynamically schedules manual integration test cases for execution based on three main criteria: (i) dependencies between test cases, (ii) test cases execution time and (iii) the requirement coverage. sOrTES reads the requirement and test specifications as inputs and provides a list of ranked test cases for execution as an output. In the extraction phase, the functional dependencies and the requirement coverage for each test case are computed and in the scheduler phase, the test cases are sorted for execution. The execution time for each test case is predicted by ESPRET (our proposed tool for execution time prediction [33]) which can be added to the test cases manually, or the same time value can be assumed for all test cases. Figure 1 shows the required inputs, expected outputs and also the embedded phases inside of sOrTES, which is exemplified through analyzing an industrial case study later in this paper. The following paragraphs describe the mentioned phases.

A. THE EXTRACTION PHASE

To get a clearer picture of the required inputs for running sOrTES, we provide some examples of software requirement specification (SRS) and test case specification, extracted from DOORS[4] database at BT. A typical SRS at BT consists of different pieces of information including the signal information and standards, which are described textually. The SRSs are written by the requirement engineering teams' members as early as the needed input is available to the project. The requirement adjustments are performed continuously during the project life cycle. Each requirement from the SRS is assigned to a sub-level functional group (SLFG). The requirement is then implemented as a part of one module within the SLFG, or as a part of several modules within the same SLFG. Table 2a and Table 2b represent two requirement specification examples for the brake system and line voltage respectively. During the project, some of the SRSs might be removed, merged or new SRSs might be added to the project.

1) FUNCTIONAL DEPENDENCIES DETECTION

In the extraction phase, the functional dependencies between test cases are detected through analyzing the internal signal

[1]EStimation and PRediction of Execution Time
[2]Analytic Hierarchy Process
[3]Technique for Order of Preference by Similarity to Ideal Solution
[4]Dynamic Object-Oriented Requirements System

TABLE 2. Software requirement specification examples - Bombardier Transportation. (a) Brake system (SLFG). (b) Line voltage (SLFG).

Requirement ID	BTI/SFC Requirement	Interface
SRS-BHH-Brake system 768	Emergency brake loop is not de-energized without order.	input: Internal Signal 44-A34.X11.4.DI3 output: MVB SDr$_3$EmRel
SRS-BHH-Brake system 241	Emergency Brake Not (26-K23) is not active.	input: Internal Signal 44-A34.X11.4.DI3 output: Internal Signal 95-B27.X01.5.PI10
SRS-BHH-Brake system 98	The activation of the emergency brake shall be shown on the HMI	input: Internal Signal 95-B27.X01.5.PI10 output: MCB 44-F34

(a)

Requirement ID	BTI/SFC Requirement	Interface
SRS-BHH-Line Voltage 327	Display the line voltage in AC-mode and the rail voltage in DC-mode .	input: 44-A23.X11.4.DI2 output: Internal Signal 95-B27.X01.5.PI10
SRS-BHH-Line Voltage 243	TCMS shall indicate the AC line voltage to the driver via HMI .	input: Internal Signal 44-A37.X11.1.DI5 output: Internal Signal 44-A34.X11.4.DI3
SRS-BHH-Line Voltage 398	TCMS shall suppress values between 0.1 and 2.0 kV.	input: I/O iDcuPd$_{128}$ output: Internal Signal 95-B27.X01.5.PI10

(b)

communications between the software modules, which is described textually in the SRSs. In order to illustrate the signal communications between the software modules, a traceability graph for the train-borne distributed control system (in summary TCMS) at BT is provided in Figure 2.

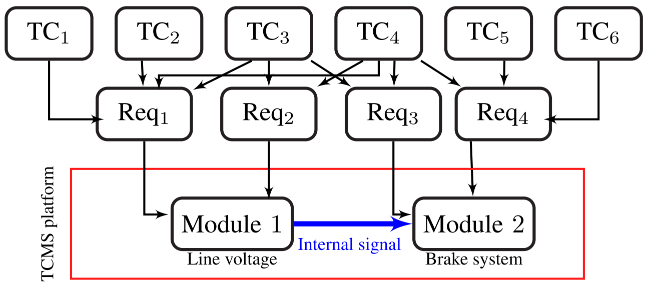


FIGURE 2. The traceability graph for TCMS.

Figure 2 shows how two software modules from two different SLFGs (line voltage and brake system) are communicating by sending and receiving an internal signal to each other. As depicted in Figure 2, module 1 sends an internal signal to module 2, which builds a dependency relation between those software modules. Since module 2 is functionally dependent on module 1, it should be tested after it. Thereby, all assigned requirements (e.g. the requirements in Table 2a) and test cases for module 2 are functionally dependent on the assigned requirements (e.g. the requirements in Table 2b) and test cases for module 1. According to Figure 2, requirements (Req$_1$ and Req$_2$) describe how module 1 should be tested and four test cases TC$_1$, TC$_2$, TC$_3$ and TC$_4$ are also designed to test module 1 based on Req$_1$ and Req$_2$. The test cases that are designed to test module 1 (i.e. TC$_1$, TC$_2$, TC$_3$ and TC$_4$) should ideally be top ranked for execution. TC$_3$ and TC$_4$ are designed to test both modules, but they need to be tested with the assigned test cases for module 1 in the first testing cycle.

The required information for creating Figure 2 is described textually in the requirement specification at BT. sOrTES reads the requirement specifications in the format of an excel sheet (see Table 2) and finds the matched internal input-output signals. As we can see in Table 2a, the internal signal `44-A34.X11.4.D13` is described as an input for the brake voltage module which is described by the requirement SRS-BHH-brake system 768 (see column Interface in Table 2a). However, the same signal (Internal signal `44-A34.X11.4.D13`) is described as an output signal in the interface column for the brake module, assigned to the requirement SRS-BHH-Line Voltage 243 in Table 2b.

Moreover, Table 3 represents an example of a manual test case for a safety critical system at BT, which consists of a test case description, an expected test result, test steps, a test case ID, corresponding requirements, etc. In this example, two requirements (`SRS-BHH-LineVolt1707` and `SRS-BHH-Speed2051`) are assigned to the test case example in Table 3. Once the dependencies between requirements are detected, sOrTES searches for the dependencies between test cases through mapping the assigned test cases to the corresponding requirements. Since both requirement and test specifications are written in a natural text, several library packages are used, such as `xlrd` for reading excel (.xls) documents (the requirement and test specifications) and `vis.js`, which is a dynamic browser based visualization library, for visualization of the dependencies. Moreover, some specific algorithms were implemented in Python to extract the required information from the requirement and test specifications. Since the implementation details (packages, libraries and pseudocodes) are already described in length in [32], thus the Python implementation details are omitted from this paper. The extraction phase is currently embedded in sOrTES. Furthermore, in a large testing project, with a wide range of requirements and test cases, the dependency between requirements and test cases is very complex. Figure 3 displays a part of the dependency relations between the requirements together with the test cases in an industrial testing project at BT. The blue and red nodes in Figure 3 represent the requirements and the test cases, respectively. Note that all red nodes are connected to only blue nodes, even if it might seem otherwise due to dense visualization. As mentioned earlier, there is a strong and complicated dependency

TABLE 3. A test case specification example from the safety-critical train control management system at bombardier transportation.

Test case name:	Auxiliary Compressor Control	**Date:**	2018-01-20
Test case ID	**Test level (s)**	**Test Result**	**Comments**
3EST001845-2032 - RCM (v.1)	Sw/Hw Integration		
Test configuration			
TCMS baseline: TCMS 1.2.3.0 Test rig: VCS Release 1.16.5 VCS Platform 3.24.0			
Requirement(s)			
SRS-BHH-Line Voltage 1707 SRS-BHH-Speed 2051			
Tester ID			
BR−1211			
Initial State			
No active cab			
Step	**Action**	**Reaction**	**Pass / Fail**
1	Lock and set Auxiliary reservoir pressure $<$ 5.5 bar	Signal Command auxiliary compressor	
2	Activate cab $A2$ lock and set signal braking mode from ATP to 109	Signal braking mode to IDU is set to 109	
3	Lock and set Auxiliary reservoir pressure $>$ 5.5 bar	Signal Auxiliary compressor is running to IDU is set to FALSE	
4	Wait 20 seconds		
5	Reset dynamic brake in the train for 5 seconds	IDU in $B1$ car as On	
6	Set Auxiliary reservoir pressure $<$ 5.5 bar	Signal Auxiliary compressor is running to IDU is set to FALSE	
7	Clean up		

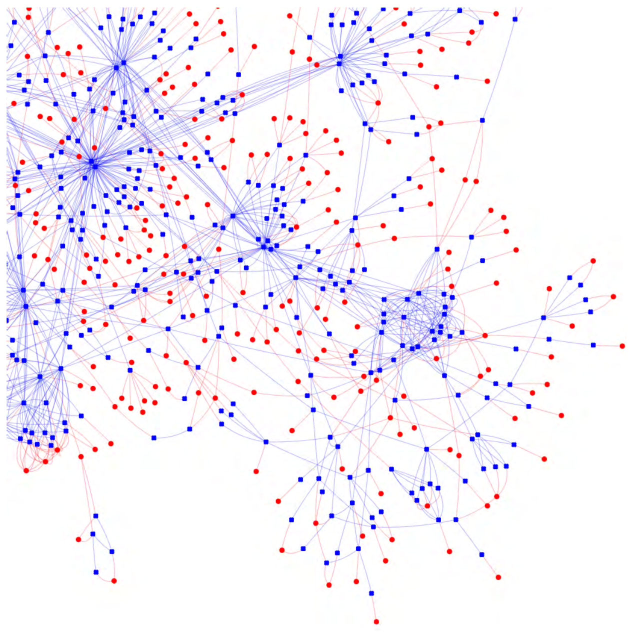


FIGURE 3. The dependency between the requirements (blue nodes) and the test cases (red nodes).

between requirements and test cases in the integration testing level. As one can see in Figure 3, often more than one test case (red nodes) is assigned to test a requirement. However, in some testing scenarios, several requirements (blue nodes) are meant to be tested by just one test case. Visualizing the dependency relationships between the requirements together with test cases, and also showing the test diversity data, can assist testers and test managers to improve the testing process in terms of test case selection and prioritization. For example, the minimum number of required test cases for testing a certain set of requirements can easily be extracted from Figure 3.

2) REQUIREMENT COVERAGE MEASUREMENT

Computing the number of assigned requirements for each test case provides the requirement coverage information. sOrTES captures the inserted information in each test specification (e.g. Table 3) and provides a number acting as the requirement coverage to each test case. The requirement coverage is equal to 2 for the provided test specification example in Table 3. However, Table 4 shows an output example provided by the extraction phase in sOrTES for dependencies and requirement coverage for each test case. In Table 4 we can see how many test cases need to be executed before testing another particular test case. The independent test cases have a 0 value in the *Dependent on* column in Table 4. Moreover, the number of test cases that can be tested after each test case is inserted in the *Output* column. The requirement coverage (the number of assigned requirements to a single test case) is calculated and stored in the *Requirement coverage* column in Table 4. For instance, test case number 1 is an independent test case, which does not require execution of any other test cases before it. Furthermore, a total of 38 requirements are assigned to this test case. The testers can plan to prioritize execution of this test case first given its requirements coverage. In certain industrial contexts, the requirement coverage is assumed as the most important criterion for test case selection and

TABLE 4. Independent and dependent test cases and the requirement coverage for each test case.

Nr.	Test case ID	Requirement coverage	Dependent on	Output
1	IVVS-ATP-IVV-51	38	0	0
2	IVVS-Battery-IVV-96	24	0	0
3	IVVS-Linevoltage-IVV-5	21	1	0
4	IVVS-Drive-IVV-27	20	0	0
5	IVVS-Trainradio-IVV-02	6	13	1
6	IVVS-Linevoltage-IVV-3	46	12	0
7	IVVS-Braketest-IVV-21	3	9	0
8	IVVS-Speed-IVV-04	2	0	46
9	IVVS-Speed-IVV-06	5	0	45
10	IVVS-TC-IVV-07	4	0	36

prioritization [6]. The tester can also test this test case at any time due to its independent nature. On the other hand, test case number 5 is dependent on 13 other test cases with a requirements coverage equal to 6 and just one test case being dependent on it. Thus, test case number 5 is not a good candidate for first cycle execution given its dependency on 13 other test cases.

However, after a successful execution of test case number 8, a total of 46 test cases will be available for execution (as independent test cases) and therefore this test case can also be considered for early execution in the testing cycle. As we can see in Table 4, test cases number 9, 10 are two independent test cases, where a total of 45 and 36 test cases are dependent on these two test cases, respectively. The names of some test cases which are dependent on test case numbers 9 and 10, are shown in Table 4. However, the inserted information in Table 4 can be used as an input in any other decision support system for ranking algorithms or can even be utilized manually for test case selection and prioritization (one-time ranking).

B. THE SCHEDULING PHASE

The information provided in Table 4 can help us to schedule test cases for dynamic execution. The test execution results of dependent test cases directly impact the other test cases. On the other hand, maximizing the requirement coverage, or execution of as many test cases as possible in a limited period of time, is always demanded by industry. The proposed optimization solution for test scheduling in this work can be divided into two main parts: (i) finding a feasible set and (ii) maximizing the requirement coverage for each testing cycle. Finding a feasible set of test cases for execution deals directly with interdependence, interaction, and relationships between test cases. First, a dependency graph (directed graph) should be built up for the dependent test cases, which shows our constraints. The objective function is to maximize the requirement coverage.

Ignoring the dependencies can lead to redundant failures, thereby increasing the troubleshooting cost and the total testing time [1]. Additionally, maximizing the requirement coverage can help testers to finish the testing process faster which might lead to earlier release of the final software product. Hence the main constraint of this optimization problem is dependency; we are not able to rank test cases just based on their requirement coverage (or any other criteria) for execution. For instance, the inserted test case in line 6 in Table 4 has the highest value for the requirement coverage (46), but this test case is a multi-dependent test case and cannot be executed as a first test case in a testing cycle because 12 other test cases should be executed successfully before reaching this one. Choosing the test candidate for execution from the feasible set might minimize the risk of unnecessary failure.

Generally, test cases can fail based on the following causes: (i) there is a mismatch between test case and requirement, (ii) the testing environment is not ready for testing, (iii) bugs in the system and (iv) paying no attention to the dependencies between test cases. A failed test case needs to go through troubleshooting, which consumes testing resources. Thus, minimizing the redundant executions and thereby unnecessary failures (based on the dependency causes) can lead to a decrease in troubleshooting efforts. Note that the proposed approach in this paper just deals with the unnecessary failure based on the dependent test cases as we are not able to avoid any other failure causes.

1) MODEL ASSUMPTIONS AND PROBLEM DESCRIPTION

Let us consider a subset of n test cases, designated by TC_1, TC_2, ... TC_n, that will be tested in a testing cycle, which are chosen from the testing pool among all test cases. Each test case, TC_i, is characterized by its execution time, $t_i > 0$, the required time for performing the troubleshooting process in case of failure, $b_i > 0$, and the requirement coverage, $s_i \in \mathbb{N}$. After each execution, a testing result such as *pass*, *fail*, *not run* and *partly pass* is recorded for the executed test cases. According to the testing policy at BT, all failed, partly passed and not run test cases must be executed again. Following their procedure, we consider all results different than *pass* as *fail*. For test case TC_i, we denote its testing result as r_i, which we represent either by 1 for *fail* or 0 for *pass*. In fact, r_i is a realization of a random variable R_i, given that we do not know what the result will be beforehand, i.e.

$$R_i = \begin{cases} 1, & \text{if test case } \mathrm{TC}_i \text{ fails} \\ 0, & \text{if test case } \mathrm{TC}_i \text{ passes.} \end{cases}$$

where, $R_i \sim Bernoulli(p_i)$ with $p_i = \mathbb{P}(\mathrm{TC}_i \text{ fails})$.

Our main goal is to define an optimal execution order to test all test cases within each testing cycle. In our study, we assume that (i) the n test cases are all tested in each testing cycle only once; (ii) when a failed test case is sent for troubleshooting, it will be tested again in another testing cycle (not in the current one). Note that these are acceptable assumptions by our industrial partner. In general, for n test cases, there are $n!$ ways of ordering the test cases. However, the order should be chosen in a way to avoid unnecessary failure based on dependency and thereby redundant troubleshooting. As it was explained before, some of these failure reasons are unavoidable, due to the failure based on dependency which is avoidable.

Let us consider that each test case in a testing cycle is a node (red nods in Figure 3) in a complete directed graph. Thus, there are bi-directed edges connecting every pair of nodes (but there are no edges from a node to itself). In fact, we want to solve a stochastic traveling salesman problem (TSP) [34] as:

Definition 4:

Given: The n test cases and their troubleshooting times.

Problem: To find the best execution order to test each test case, minimizing the total troubleshooting time.

To formalize the TSP, we define:

$$\min \mathbb{E}\left[\sum_{i=1}^{n}\sum_{\substack{j=1\\ j\neq i}}^{n} x_{ij}\, b_i\, R_i\right]$$
$$\text{s.t. } x_{ij} \in \{0,1\}, \quad \forall\, i,j = 1,\ldots,n$$
$$\sum_{\substack{j=1\\ j\neq i}}^{n} x_{ij} = 1, \quad \forall\, i = 1,\ldots,n$$
$$\sum_{\substack{i=1\\ i\neq j}}^{n} x_{ij} = 1, \quad \forall\, j = 1,\ldots,n$$
$$\sum_{i\in V}\sum_{\substack{j\in V\\ j\neq i}} x_{ij} \le |V|-1, \quad \forall\, V \subsetneq \{1,2,\ldots,n\},\ V\neq\emptyset \qquad (1)$$

The solution of Equation 1 gives us the best order to execute the n test cases in such a way that the troubleshooting time is minimized at each testing cycle, which can be written as an n-dimension vector:

$$\left(\mathrm{TC}_{\cdot,1}, \mathrm{TC}_{\cdot,2}, \cdots, \mathrm{TC}_{\cdot,n}\right).$$

In our notation $\mathrm{TC}_{\cdot,k}$ represents the k^{th} test case that should be tested, i.e. the second index stands for the execution order where the first index indicates the test case number. For instance, $\mathrm{TC}_{5,2}$ represents that TC_5 should be executed in second place in a testing cycle.

In the present work we are preoccupied with minimizing the needless troubleshooting time caused by interdependent integration test cases. In other words, if TC_j depends on TC_i, then it is very unlikely that TC_j passes if TC_i was not tested before, then we consider that:

$$\mathbb{P}(\mathrm{TC}_j \text{ passes}|\mathrm{TC}_i \text{ was not tested}) = 0. \qquad (2)$$

Thus, if we choose an order where TC_j is tested before TC_i then we will have $R_j = 1$ almost certainly (and a troubleshooting time will be summed up). On the other hand, if all tests that TC_j is dependent on have already passed, then TC_j will behave as an independent test case. For instance, let us consider that TC_j only depends on TC_i and TC_k, then we have:

$$\mathbb{P}(\mathrm{TC}_j \text{ passes}|\mathrm{TC}_i \text{ and } \mathrm{TC}_k \text{ already passed}) = \mathbb{P}(\mathrm{TC}_j \text{ passes}).$$

In order to avoid adding unnecessary troubleshooting time before testing a test case with dependencies, we should first test all test cases that it is dependent on. However, to accomplish this, an embedded digraph of dependencies should be described first. This is one of the main contributions of sOrTES in the extraction phase.

To clarify the explanation, let us consider a dummy example,[5] where we only have 5 test cases: TC_1, TC_2, TC_3, TC_4 and TC_5. We have $5! = 120$ different execution orders of testing the 5 test cases. However, let us assume that we are able to describe the following embedded digraph of dependencies:

[5]It is a dummy in the sense that the number of test cases is very small compared with real industry cases.

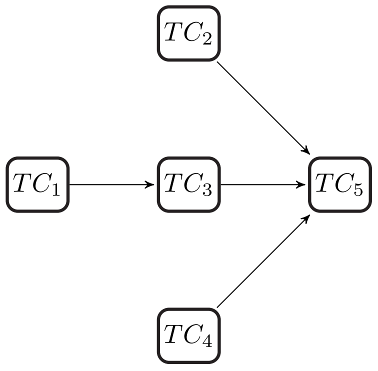

$$\begin{tikzcd} & TC_2 \arrow[dr] & \\ TC_1 \arrow[r] & TC_3 \arrow[r] & TC_5 \\ & TC_4 \arrow[ur] & \end{tikzcd}$$

FIGURE 4. Example of an embedded digraph of dependencies for 5 test cases.

According to Figure 4, TC_1, TC_2 and TC_4 are independent test cases, TC_3 depends on TC_1, TC_5 directly depends on TC_2, TC_3 and TC_4, where we call them as the precedents, and indirectly depends on TC_1. Knowing this grid of dependencies and taking into account Equation (2), we realize that there are only 12 feasible choices to execute those 5 test cases (always with TC_5 in the last place and TC_3 after TC_1).

To formalize the set of the feasible choices, for each test case TC_i, we consider the set of all precedents, P_i, i.e. the set of test cases that TC_i directly depends on. For instance, for the presented example in Figure 4, $P_1 = P_2 = P_4 = \emptyset$, $P_3 = \{\mathrm{TC}_1\}$ and $P_5 = \{\mathrm{TC}_2, \mathrm{TC}_3, \mathrm{TC}_4\}$.

In order to avoid unnecessary troubleshooting time, let us call $\mathcal{F}$ the set of all possible ways to test the n test cases, which takes into account the interdependencies. We call it the feasible set and it is defined as follows:

$$\mathcal{F} = \Big\{\left(\mathrm{TC}_{\cdot,1}, \mathrm{TC}_{\cdot,2}, \cdots, \mathrm{TC}_{\cdot,n}\right) : P_{\cdot,1} = \emptyset,$$
$$\forall\, i = 2,\cdots,n,\ P_{\cdot,i} \subsetneqq \left\{\mathrm{TC}_{\cdot,1}, \cdots, \mathrm{TC}_{\cdot,i-1}\right\}\Big\}.$$

Considering again the example in Figure 4, if we have $\left(\mathrm{TC}_{\cdot,1}, \mathrm{TC}_{\cdot,2}, \mathrm{TC}_{\cdot,3}, \mathrm{TC}_{\cdot,4}, \mathrm{TC}_{\cdot,5}\right) \in \mathcal{F}$, the total expected time for troubleshooting is $b_1\, p_1 + b_2\, p_2 + b_3\, p_3 + b_4\, p_4 + b_5\, p_5$. On the other hand, if $\left(\mathrm{TC}_{\cdot,1}, \mathrm{TC}_{\cdot,2}, \mathrm{TC}_{\cdot,3}, \mathrm{TC}_{\cdot,4}, \mathrm{TC}_{\cdot,5}\right) \notin \mathcal{F}$, the total expected time for troubleshooting is either $b_1\, p_1 + b_2\, p_2 + b_3\, p_3 + b_4\, p_4 + b_5$ or $b_1\, p_1 + b_2\, p_2 + b_3 + b_4\, p_4 + b_5\, p_5$ or $b_1\, p_1 + b_2\, p_2 + b_3 + b_4\, p_4 + b_5$. This means that the total expected time for troubleshooting for elements in the feasible set $\mathcal{F}$ is always lower than the corresponding expected time for elements that do not belong to $\mathcal{F}$. In other words, any solution not included in $\mathcal{F}$ is sub–optimal to minimize the total expected time for troubleshooting. This implies that from now on we only consider elements in $\mathcal{F}$.

Currently, sOrTES considers all elements in the feasible set $\mathcal{F}$ and orders them by requirement coverage, i.e. it wants to achieve the highest requirement coverage possible, as early as possible. We can formalize it in the following way:

We want to choose $\left(\mathrm{TC}_{\cdot,1}, \mathrm{TC}_{\cdot,2}, \cdots, \mathrm{TC}_{\cdot,n}\right) \in \mathcal{F}$, with respective requirements coverage $s_{\cdot,1}, s_{\cdot,2}, \ldots, s_{\cdot,n}$, such that, if we consider another order $\left(\mathrm{TC}'_{\cdot,1}, \mathrm{TC}'_{\cdot,2}, \cdots, \mathrm{TC}'_{\cdot,n}\right) \in \mathcal{F}$ with respective requirements coverage $s'_{\cdot,1}, s'_{\cdot,2}, \ldots, s'_{\cdot,n}$, for all $i = 1, 2, \ldots, n$, the following inequality holds

$$\sum_{j=1}^{i} s_{\cdot,j} \geq \sum_{j=1}^{i} s'_{\cdot,j}. \tag{3}$$

V. EMPIRICAL EVALUATION

In order to analyze the feasibility of sOrTES, we designed an industrial case study at Bombardier Transportation (BT) in Sweden, by following the proposed guidelines of Runeson and Höst [35] and also Engström *et al.* [36]. BT provides various levels of testing in both manual and automated approaches, where the integration testing is performed completely manually. The number of required test cases for testing a train product at BT is rather large and the testing process is performed in several testing cycles.

A. UNIT OF ANALYSIS AND PROCEDURE

The units of analysis in the case under study are manual test cases at the level of integration testing for a safety-critical train control subsystem at BT. sOrTES is however not restricted to the testing level or methods and can be applied to other testing procedures in other levels of testing (e.g. unit, regression and system level in both manual and automated procedures) in other domains. The case study is performed in several steps:

- An Commuter train for Hamburg called BR project[6] is selected as a case under study.
- A total number of 3938 requirements specifications (SRSs) from 17 different sub-level function groups are extracted from the DOORS database at BT for the BR project.
- The dependencies between requirements are detected for 3201 SRSs while 737 SRSs are detected as being independent.
- A total number of 1748 test specifications are extracted from DOORS and analyzed for dependency detection.
- The results of dependency between test cases are presented to the BR project team members.
- Three different test execution cycles from the beginning, middle and end of the BR project are selected to be compared with the proposed execution schedules by sOrTES.
- The testers' and engineers' opinions about the number of failed test cases based on the dependencies are collected and analyzed.

[6]The BR series is an electric rail car specifically for the S-Bahn Hamburg GmbH network in production at Bombardier Hennigsdorf facility started in June 2013. Bombardier will deliver 60 new single and dual-voltage commuter trains with innovative technology, desirable energy consumption and low maintenance costs. The heating system in the BR project is designed to use waste heat from the traction equipment system to heat the passenger compartments [37].

B. CASE STUDY REPORT

As mentioned before, the main goal of proposing sOrTES is dynamic test scheduling. The results of executed test cases can impact how the dependencies are shaped between test cases. Therefore making a correct decision for test execution is directly related to the test cases' dependencies. sOrTES first detects the dependencies between the assigned requirements to the BR project and thereby the dependencies between corresponding test cases are detected. In the next level, the requirement coverage is computed for each test case and thereafter test cases are ranked based on their dependencies and requirement coverage.

TABLE 5. An execution schedule example, proposed by sOrTES for the BR project at Bombardier, RC stands for the requirement coverage and Time represents the execution time.

Schedule · Project: BR	Toggle Graph			
Test case ID	**Priority**	**RC**	**Time**	**Result**
IVVS-ATP-S-IVV-051	1	38	1	▶
IVVS-Battery-IVV-096	2	24	1	▶
IVVS-LineVoltage-S-IVV-015	3	27	1	▶
IVVS-Drive-IVV-027	4	20	1	▶
IVVS-S-ConfItems-IVV-013	5	17	1	▶
IVVS-Fire-S-IVV-036	6	17	1	▶
IVVS-LineVoltage-S-IVV-018	7	21	1	▶
IVVS-Drive-IVV-004	8	20	1	▶
IVVS-Drive-IVV-008	9	16	1	▶
IVVS-Fire-S-IVV-034	10	16	1	▶

Table 5 represents the user interface of sOrTES with an execution schedule example, where the dependent test cases are highlighted in red. Moreover, the execution time and requirement coverage (RC column in Table 5) are available for each test case. Since the execution time for test cases is captured by another tool (ESPRET), we assume the same time value for all test cases in the present work. Note that by changing the execution time value in Table 5, another execution schedule will be proposed which might be different than the current one. In other words, test cases are ranked for execution based on their dependencies, requirement coverage and the execution time, where the user is able to specify that a lower or higher execution time per test case is demanded.

The priority column in Table 5 shows the execution order for each test case. After each execution, the testers can insert the test results for those test cases which are *passed* by clicking on the play circle icon in Table 5. Thus, the *passed* test cases are removed automatically from the proposed schedule and a new execution schedule will appear in Table 5. Using the red color for the dependent test cases can help testers to have a better overview of the dependencies between test cases. By clicking on the *Toggle Graph* button (embedded in sOrTES user interface seen in Table 5), Figure 5 is shown, which represents the dependencies between test cases (the red lines in Table 5) for the BR project. Moreover, a new graph is created after removing the *passed* test cases from Table 5. We need to consider that, Figure 5 shows a part of the dependencies graph between test cases (the complete dependency graph contains 1748 test cases).

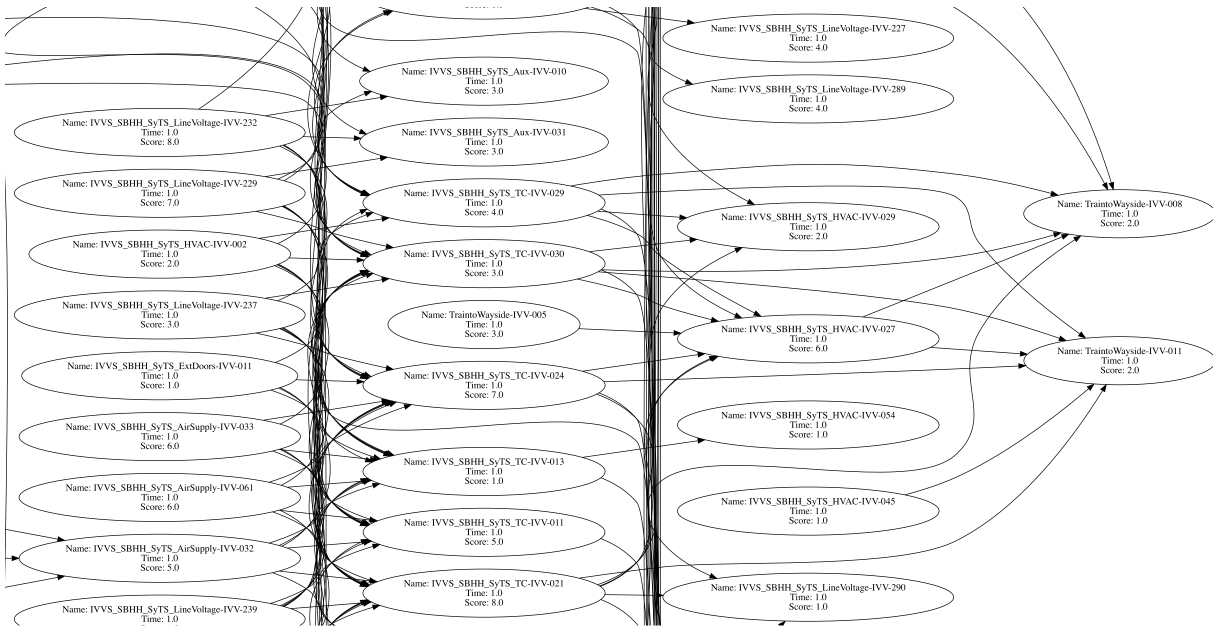

Name: IVVS_SBHH_SyTS_LineVoltage-IVV-227 Time: 1.0 Score: 4.0
Name: IVVS_SBHH_SyTS_Aux-IVV-010 Time: 1.0 Score: 3.0
Name: IVVS_SBHH_SyTS_LineVoltage-IVV-289 Time: 1.0 Score: 4.0
Name: IVVS_SBHH_SyTS_LineVoltage-IVV-232 Time: 1.0 Score: 8.0
Name: IVVS_SBHH_SyTS_Aux-IVV-031 Time: 1.0 Score: 3.0
Name: IVVS_SBHH_SyTS_LineVoltage-IVV-229 Time: 1.0 Score: 7.0
Name: IVVS_SBHH_SyTS_TC-IVV-029 Time: 1.0 Score: 4.0
Name: TraintoWayside-IVV-008 Time: 1.0 Score: 2.0
Name: IVVS_SBHH_SyTS_HVAC-IVV-029 Time: 1.0 Score: 2.0
Name: IVVS_SBHH_SyTS_HVAC-IVV-002 Time: 1.0 Score: 2.0
Name: IVVS_SBHH_SyTS_TC-IVV-030 Time: 1.0 Score: 3.0
Name: IVVS_SBHH_SyTS_LineVoltage-IVV-237 Time: 1.0 Score: 3.0
Name: TraintoWayside-IVV-005 Time: 1.0 Score: 3.0
Name: IVVS_SBHH_SyTS_HVAC-IVV-027 Time: 1.0 Score: 6.0
Name: TraintoWayside-IVV-011 Time: 1.0 Score: 2.0
Name: IVVS_SBHH_SyTS_ExtDoors-IVV-011 Time: 1.0 Score: 1.0
Name: IVVS_SBHH_SyTS_TC-IVV-024 Time: 1.0 Score: 7.0
Name: IVVS_SBHH_SyTS_HVAC-IVV-054 Time: 1.0 Score: 1.0
Name: IVVS_SBHH_SyTS_AirSupply-IVV-033 Time: 1.0 Score: 6.0
Name: IVVS_SBHH_SyTS_TC-IVV-013 Time: 1.0 Score: 1.0
Name: IVVS_SBHH_SyTS_AirSupply-IVV-061 Time: 1.0 Score: 6.0
Name: IVVS_SBHH_SyTS_HVAC-IVV-045 Time: 1.0 Score: 1.0
Name: IVVS_SBHH_SyTS_TC-IVV-011 Time: 1.0 Score: 5.0
Name: IVVS_SBHH_SyTS_AirSupply-IVV-032 Time: 1.0 Score: 5.0
Name: IVVS_SBHH_SyTS_TC-IVV-021 Time: 1.0 Score: 8.0
Name: IVVS_SBHH_SyTS_LineVoltage-IVV-290 Time: 1.0 Score: 1.0
Name: IVVS_SBHH_SyTS_LineVoltage-IVV-239 Time: 1.0

FIGURE 5. Partial dependency graph between the BR project's test cases at BT, using sOrTES.

VI. PERFORMANCE EVALUATION

In this section we evaluate the performance of sOrTES through comparing the proposed execution schedules by sOrTES with three different execution orders at BT. The purpose of the performance evaluation is to adequately compare the amount of fulfillment requirements and troubleshooting efforts in the different testing cycles during the project.

A. PERFORMANCE COMPARISON BETWEEN SORTES AND BOMBARDIER

For each set of test cases, in each testing cycle, sOrTES and BT provide an execution schedule, ordering the test cases based on their own criteria. Following the notation already introduced, we represent the scheduling order given by sOrTES and BT, respectively by:

$$\mathrm{TC}_{\cdot,1^S}, \mathrm{TC}_{\cdot,2^S}, \ldots, \mathrm{TC}_{\cdot,m^S}$$

and

$$\mathrm{TC}_{\cdot,1^B}, \mathrm{TC}_{\cdot,2^B}, \ldots, \mathrm{TC}_{\cdot,m^B}.$$

where $\mathrm{TC}_{\cdot,1^S}$ represents the first test case in sOrTES proposed execution schedule. For instance, $\mathrm{TC}_{5,1^S}$ means that TC_5 is the first test case that needs to be executed, according to sOrTES. Suppose that at BT the TC_5 is tested in 12nd place, then we have $\mathrm{TC}_{5,12^B}$. From now on, whenever it is necessary, we use the S to associate with sOrTES and B to associate with BT. Our goal is to compare the performance of both of the provided schedules. The presented test cases in Table 4 are arranged in alphabetical order for the 1st testing cycle in Table 6.

As Table 6 shows, TC_1 is scheduled by sOrTES as the first test candidate for execution, however, the TC_1 has been executed by BT as the 942th test case in the 1st testing cycle. To compare the planned schedules, we consider a variable which represents the sum of the requirements coverage fulfilled after the execution of the k^{th} test case, namely:

TABLE 6. Execution order in the 1st testing cycle, for a subset of 10 test cases (among 1462), according to sOrTES and BT.

TC_i	Test case ID	sOrTES	BT
TC_1	IVVS_SBHH_SyTS_ATP_S-IVV-051	$\mathrm{TC}_{1,1^S}$	$\mathrm{TC}_{1,942^B}$
TC_2	IVVS_SBHH_SyTS_Battery-IVV-096	$\mathrm{TC}_{2,2^S}$	$\mathrm{TC}_{2,119^B}$
TC_3	IVVS_SBHH_SyTS_Braketest-IVV-021	$\mathrm{TC}_{3,1191^S}$	$\mathrm{TC}_{3,224^B}$
TC_4	IVVS_SBHH_SyTS_Drive-IVV-027	$\mathrm{TC}_{4,4^S}$	$\mathrm{TC}_{4,387^B}$
TC_5	IVVS_SBHH_SyTS_Linevoltage-IVV-003	$\mathrm{TC}_{5,237^S}$	$\mathrm{TC}_{5,560^B}$
TC_6	IVVS_SBHH_SyTS_Linevoltage_S-IVV-015	$\mathrm{TC}_{6,3^S}$	$\mathrm{TC}_{6,1340^B}$
TC_7	IVVS_SBHH_SyTS_Speed-IVV-004	$\mathrm{TC}_{7,1399^S}$	$\mathrm{TC}_{7,784^B}$
TC_8	IVVS_SBHH_SyTS_Speed-IVV-006	$\mathrm{TC}_{8,108^S}$	$\mathrm{TC}_{8,785^B}$
TC_9	IVVS_SBHH_SyTS_TC-IVV-007	$\mathrm{TC}_{9,173^S}$	$\mathrm{TC}_{9,894^B}$
TC_{10}	IVVS_SBHH_SyTS_Trainradio-IVV-002	$\mathrm{TC}_{10,1199^S}$	$\mathrm{TC}_{10,888^B}$

$$CS^S(k) = \sum_{j=1}^{k} s_{\cdot,j^S} \quad \text{and } CS^B(k) = \sum_{j=1}^{k} s_{\cdot,j^B}.$$

Indeed, $CS^S(k)$ and $CS^B(k)$ are the cumulative requirements coverage achieved by sOrTES and BT, respectively.

To compare the schedules after execution, just as before, we consider a variable that gives us the cumulative requirement coverage. However, now we assume a penalty for every time a test case gets a *failed* result. We don't sum the requirement coverage from the failed test cases, i.e. we have instead:

$$CSA^S(k) = \sum_{i=1}^{k} s_{\cdot,i^S} \ \left(1 - r_{\cdot,i^S}\right)$$

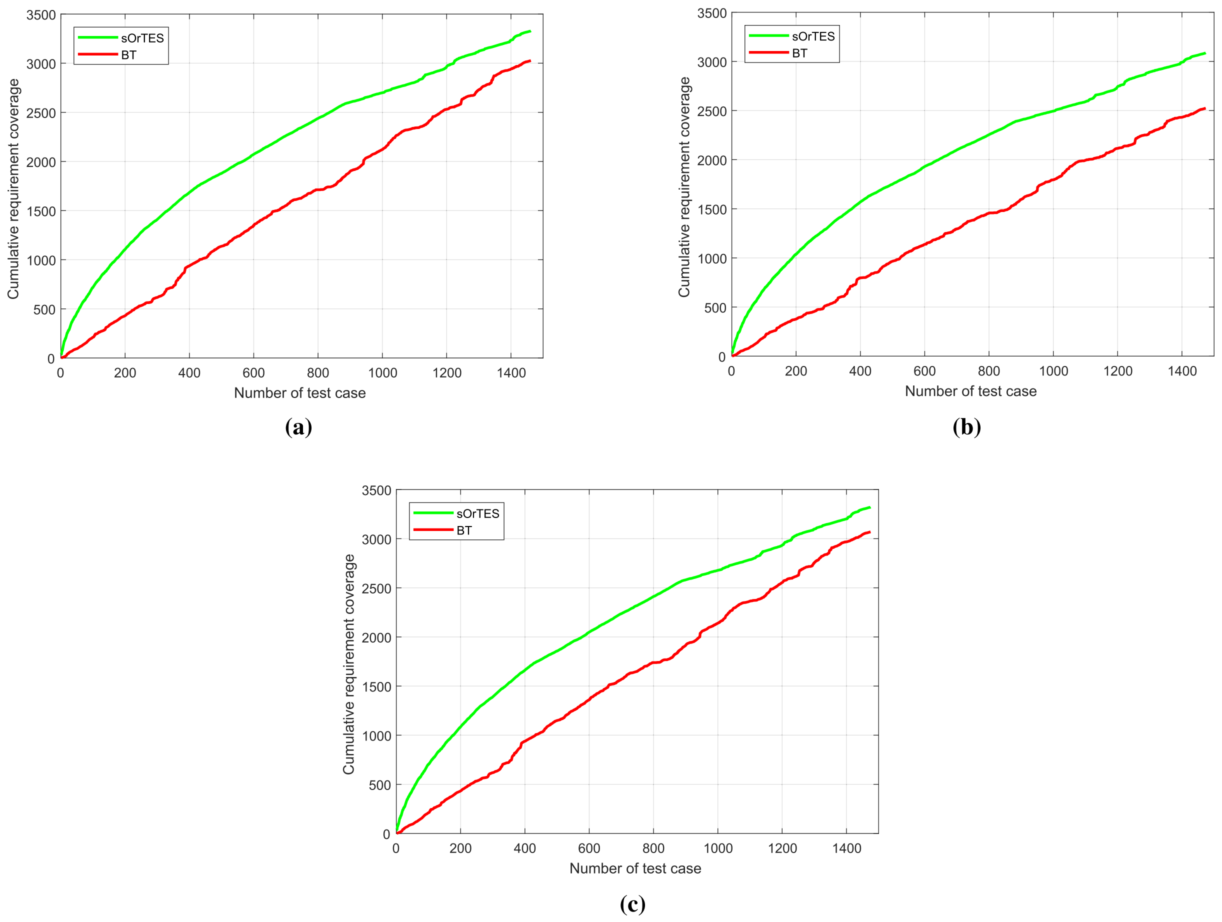


FIGURE 6. **Comparing the scheduling execution results at BT with the proposed execution schedule by sOrTES for the BR project. (a) Cumulative requirement coverage after the 1st execution cycle. (b) Cumulative requirement coverage after the 2nd execution cycle. (c) Cumulative requirement coverage after the 3rd execution cycle.**

and

$$CSA^B(k) = \sum_{i=1}^{k} s_{\cdot,i^B} \left(1 - r_{\cdot,i^B}\right).$$

It is also worthy to consider a variable regarding the troubleshooting time. This variable represents the sum of the number of labor hours, regarding to troubleshooting, after the execution of the k^{th} test case. In fact, when a test case gets a *fail* result, we add its troubleshooting time, i.e.

$$CTA^S(k) = \sum_{i=1}^{k} \left[b_{\cdot,i^S}\, r_{\cdot,i^S}\right]$$

and

$$CTA^B(k) = \sum_{i=1}^{k} \left[b_{\cdot,i^B}\, r_{\cdot,i^B}\right].$$

In other words, *CSA* is the cumulative requirement coverage after execution for the *passed* test cases and *CTA* represents troubleshooting time for the failed test cases.

Figure 6 illustrates the gained cumulative requirements coverage (CSA) for three different execution cycles at BT, where sOrTES maximizes the number of requirements in each execution cycle, compared with BT. According to Figure 6a in the 1st execution cycle at BT, the number of 2000 requirements are tested through executing 880 test cases with the proposed execution order by the testers. However, the same number of requirements could be covered by executing 580 test cases if the testers follow the proposed execution schedule by sOrTES. Since we are using the cumulative requirement coverage in the end of each testing cycle, the total number of tested requirements would be the same by both proposed execution schedules as long as there are no failures based on dependency between test cases. However, given that sOrTES has less failures, it can cover more requirements. Moreover, comparing the tested requirements in Figure 6 indicates that, in all proposed execution schedules by BT, fewer requirements have been tested compared with sOrTES. In some testing companies testing as requirements as possible in a limited period of time is demanded. Therefore prioritizing test cases based on their requirement coverage is a well known and acceptable approach. Note that the proposed execution schedule by sOrTES in Figures 6 is not only based on the requirement coverage but is also based on the dependencies between test cases. In other words, we avoid the risk of failure based on dependencies between test cases by using the execution schedules proposed by sOrTES. Hence sOrTES ranks test cases for execution based on their requirement coverage (after dependency detection), those test cases which are covering more requirements will be ranked higher for execution. As seen in Table 5, the first test candidate for execution covers 38 requirements. The average of executed test cases in each testing cycle at BR project is around 1500, which leads to a very low probability of executing the test cases with a high number of requirements coverage in the early stages of a testing cycle. According to Figure 6 using the proposed execution schedules by sOrTES leads to increase the average of requirement coverage up to 9.6% . In Figure 7 we compared the scheduled cumulative requirement coverage with 10000 random test execution orders. As can be seen, 2000 requirements are fulfilled by sOrTES through executing 400 test cases, whereas in the 10000 random execution orders for more than 1400 test cases, the minimum number of test cases is 700 to obtain the same 2000 requirements.[7]

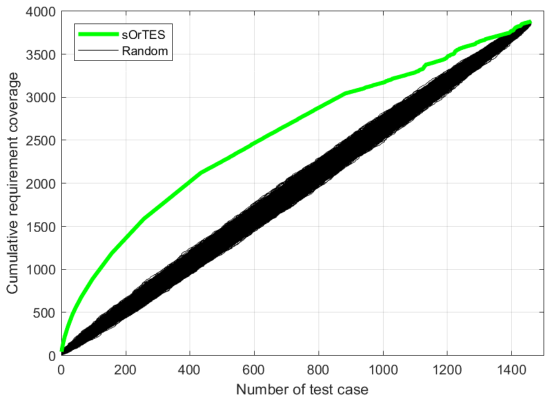


FIGURE 7. Cumulative requirement coverage for random execution orders.

As highlighted in Section III, one of the main advantages of using sOrTES is the ability to avoid unnecessary failures based on previous failures (fail based on dependency). All failed test cases at BT are labeled with a tag such as *software change request (SCR)*. SCRs are usually used when a test case fails based on an existing bug in the system or there is a mismatch between a requirement and a test case. For analyzing the performance of sOrTES in the troubleshooting time perspective, we assumed all test cases which are tagged with SCR (and failed in each testing cycle) would result in a failure with the proposed execution order by sOrTES as well. Since, the dependencies between test cases are detected by sOrTES, we are able to monitor the execution results for dependent test cases after each execution cycle. In fact, we have searched for "failed based on failed" test cases among all failed test cases. To exhibit the differences between the proposed execution schedule by sOrTES and BT's execution orders, we consider those test cases that have resulted in a *fail* based on dependencies as a *pass* instead.

In all three execution cycles at BT, a test case which depends on any test case is failed when: (i) the independent test case was already tested, and the result was *failed* or (ii) the independent test case was not tested before the dependent one (wrong execution order). As highlighted before, we need to consider that using the proposed execution order by sOrTES does not avoid any other causes of failure (e.g. the mismatching between test cases and requirements), as those kinds of failures would most likely occur with sOrTES schedules. As previous study [1] shows, fails based on dependency are the most common cause of failure in the integration testing level for the Zefiro[8] project at BT.

Our analysis shows that 20.79% of the executed test cases in the first execution cycle at BR project have failed, where 40.79% of the failures have occurred based on the dependencies between test cases. The occurrence percentage is 44.20% and 36.18%, out of a failure percentage given by 33.92% and 19.86%, for the second and third execution cycle respectively. The total troubleshooting time for each test case is estimated as 16 labor hours with a collaboration of the testing experts at BT, meaning a failed test case takes two days to cycle back to testing again. Figure 8 illustrates the troubleshooting time for the failed test cases in the three testing cycles at BT. As can be seen in Figure 8a, the total required troubleshooting time in the first execution cycle at BT is close to 5000 labor hours, where using the proposed schedule by sOrTES can lead to less than 3000 labor hours. Furthermore, in the second execution cycle (see Figure 8b), we observe that the troubleshooting time has increased even more, reaching 8000 labor hours, which is almost two times higher than the required troubleshooting time by sOrTES.

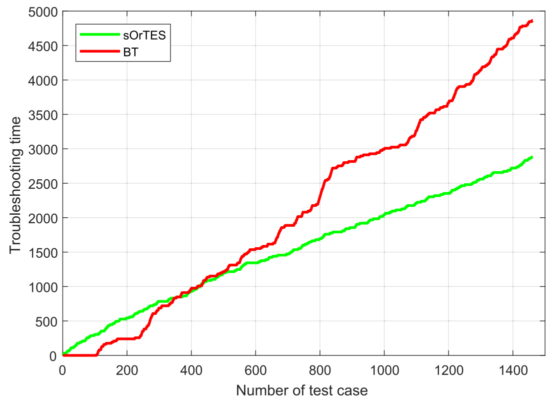
According to Figure 8, the proposed execution schedule by BT has less failure in the beginning of each testing cycle. In the first cycle, after executing 100 test cases, a significant increase in troubleshooting can be observed (see Figure 8a), which indicates the fail based on fail problem. In the second execution cycle, failure starts from the beginning, illustrated in Figure 8b. Moreover, failure starts after almost 100 executions, displayed in Figure 8c, for the third execution cycle.

As stated earlier in this section, 44.20% of the executed test cases failed in the second execution cycle at BT. This led to a longer troubleshooting time compared to the first and third execution cycle (See Figure 8b).

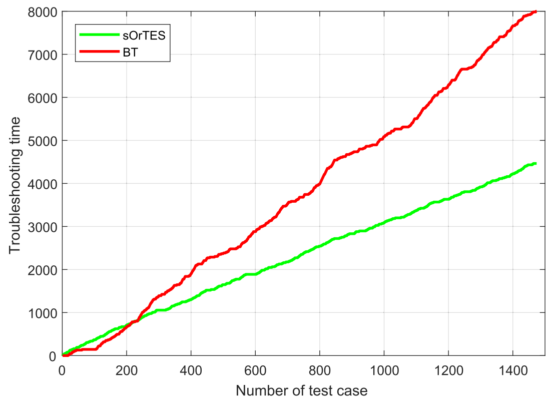
The main reason that the execution order at BT in Figure 8a and Figure 8c has a lower failure in the beginning is that the testers are usually executing the simple test cases in the beginning of each testing cycle. Those type of test cases, which are not complicated and generally cover one or two requirements, have a higher chance for a successful execution.

[7]Note that Figure 7 shows the first execution schedule plan (before the first execution), i.e. CS^S defined before. For this reason Figure 7 does not coincide with Figure 6a.

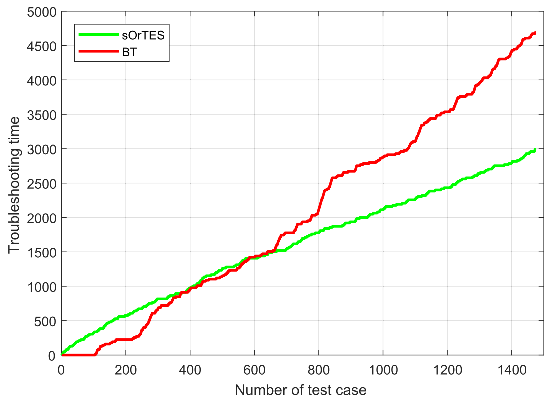
[8]Bombardier Zefiro is a family of very high-speed passenger trains designed by Bombardier Transportation whose variants have top operating speeds of between 250 km/h (155 mph) and 380 km/h (236 mph).



(a)



(b)



(c)

FIGURE 8. Troubleshooting time comparison at BT with the proposed execution schedules by sOrTES for the BR project. (a) Cumulative requirement coverage after the 1st execution cycle. (b) Troubleshooting time after the 2nd execution cycle. (c) Troubleshooting time after the 3rd execution cycle.

In the proposed execution schedule by sOrTES, those test cases which cover a high number of requirements need to be tested as soon as possible. However, postponing more complex test cases for execution is not an optimal decision. First of all, ranking test cases based on their requirement coverage (and the execution time) leads to testing the software product faster. Secondly, if a complex test case fails in the end of the testing process, it would be necessary to cycle the test case back through testing, which would take more overall time and could ultimately require a deadline extension for the testing project.

B. PERFORMANCE COMPARISON INCLUDING A HISTORY-BASED TEST CASE PRIORITIZATION APPROACH

As reviewed earlier in Section II-D, several approaches have been proposed for test case selection and prioritization in the state of the art. Among the existing methods, we have opted to compare the performance of sOrTES with a *history-based prioritization* approach in this subsection. The history-based approaches utilize information from test case's history data [38] for selecting and prioritizing test cases for the next execution. Strandberg *et al.* [19] provide a list of critical factors for test case prioritization, where the test case's history is recognized as the most impact. Generally, in a history-based test prioritization, the failed test cases in each testing cycle will be high ranked for execution in the next test cycle [39]. In some cases, the history-based prioritization improves the test efficiency compared to other existing test case prioritization techniques [40]. Furthermore, the history-based test prioritization has been considered as a common and popular approach in terms of reducing the risk of failure between those test cases which are failed once before. In the previous subsection, the performance of sOrTES is compared with the proposed test scheduling approach at BT within three test execution cycles, in terms of requirement coverage measurement and unnecessary failure reduction. Since the execution results for the BR project's test cases are available, we are able to schedule them one more time, by using the history-based prioritization method, where the performance of all three approaches (sOrTES, BT and history-based approach) can be compared with each other. Toward this goal, in the history-based prioritization, we high rank the failed test cases in the first execution cycle, to be executed in the second cycle.

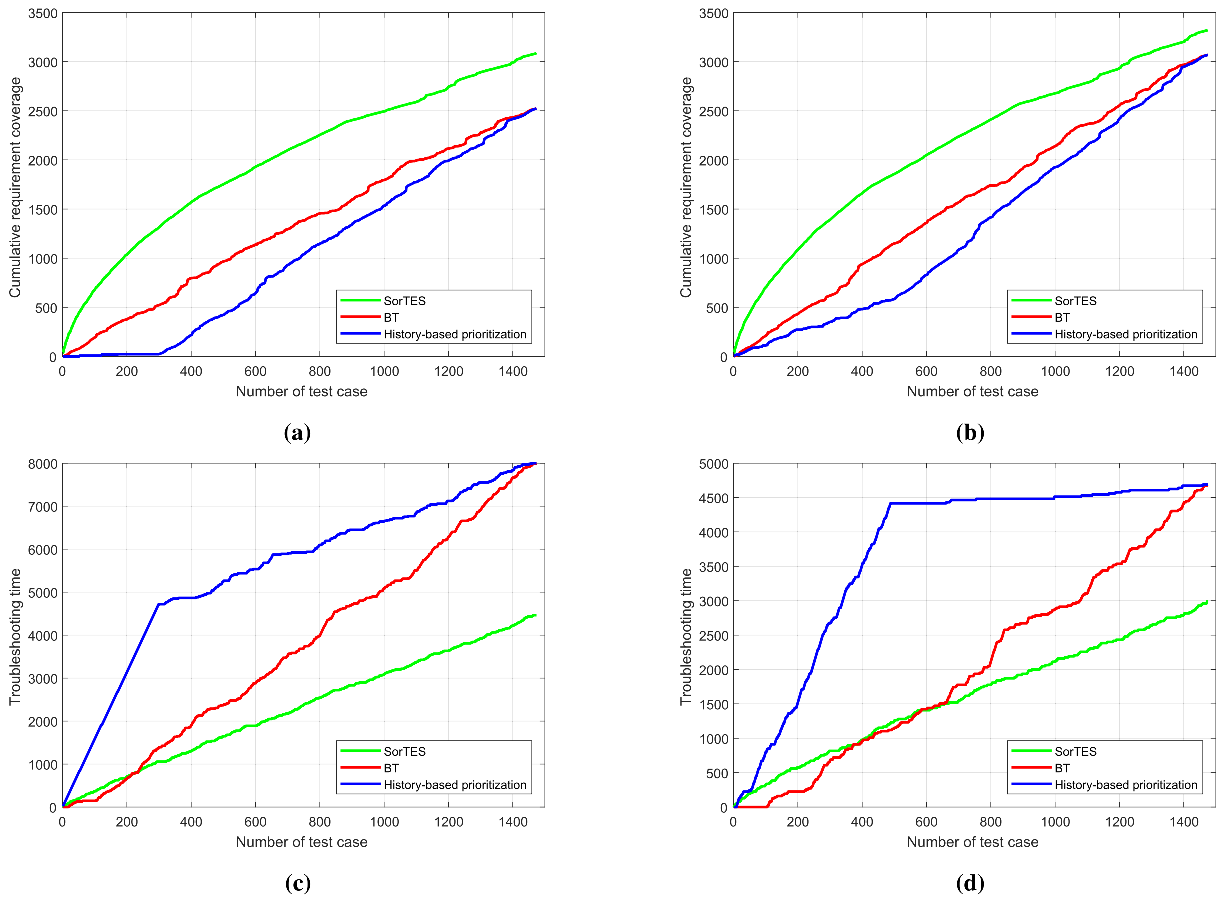


FIGURE 9. Comparing the scheduling execution results at BT with the proposed execution schedule by sOrTES together with a historical-based test case prioritization approach for the BR project. (a) Cumulative requirement coverage after the 2nd execution cycle. (b) Cumulative requirement coverage after the 3rd execution cycle. (c) Troubleshooting time after the 2nd execution cycle. (d) Troubleshooting time after the 3rd execution cycle.

TABLE 7. Test execution failure analysis in each testing cycle for the BR project at BT.

	1st Execution cycle	2nd Execution cycle	3rd Execution cycle
Percentage of failed test cases	20.79%	33.92%	19.86%
Percentage of re-tested test cases	–	93.36%	97.60%
Percentage of re-failed test cases	–	98.66%	56.56%
Percentage of re-failed test cases based on dependency	–	43.39%	30.43%

Table 7 shows the percentage of test failures and the amount of re-failed test cases together with re-failed test cases based on dependency in each testing cycle. As one can see, 20.79% of test cases are failed in the first execution cycle, where 93.36% of those are considered for re-execution in the second execution cycle at BT. This indicates that, around 6.64% of the failed test cases (in the 1st execution cycle) are omitted in the 2nd execution cycle. Using the history-based test prioritization technique, the failed test cases in the 1st execution cycle are high ranked in the 2nd execution cycle. For ranking other test cases (which are not failed in the 1st execution cycle), the proposed execution approach by BT is utilized. As Table 7 presents, 98.66% of the failed test cases in the 1st execution cycle, are failed again in the 2nd execution cycle, where 43.39% of the failures are occurred due to the dependencies between test cases. A similar interpretation can be done for the 3rd execution cycle.

Moreover, figure 9 shows the gained cumulative requirements coverage and also the troubleshooting time for the failed test cases in the two testing cycles at BT. As we can see in Figures 9a and 9b, sOrTES maximizes the number of requirements in both execution cycles, compared with the history-based prioritization approach. As mentioned earlier, the first failed test case in the 1st execution cycle is assumed be to executed as a first test case in the 2nd execution cycle (see Figures 9a and 9c). In this regard, all failed test cases in the 2nd execution cycle are high ranked for execution in the 3rd execution cycle (see Figures 9b and 9d). As Table 7 shows, the dependency between test cases is an important cause for failure in each testing cycle and therefore the troubleshooting value in Figures 9a and 9b is higher than sOrTES and BT.

The cumulative requirement coverage in the 2nd and 3rd (see Figures 9a and 9b) execution cycles indicates that the proposed test scheduling approaches by BT, is led to covering more requirements, compared with the history-based approach in the both execution cycles. This shows the intuitions and testing skills of BT's testers for ranking test cases for execution. Moreover, one of the main reasons behind the achieved results is the unnecessary failures which occurred between the failed test cases (again in the next execution cycle) due to their dependencies (see the last row in Table 7). The empirical results presented in Figure 9 indicate that sOrTES outperforms the historical-based prioritization approach in terms of requirement coverage and unnecessary failure between integration test cases, which leads to reducing the troubleshooting time. However, the history-based prioritization technique can be an efficient approach in any other testing environment, where there is no complex dependency between test cases.

VII. THREATS TO VALIDITY

The threats to the validity, limitations and the challenges faced in conducting the present study are discussed in this section.

A. CONSTRUCT VALIDITY

is one of three main types of validity evidence which makes sure the study measures what it intended to measure [41]. The major construct validity threat in the present study is the way that the functional dependencies between test cases is detected. Utilizing the *internal signals* information of the software modules for dependency detection may not be attainable in other testing processes. Generally, the information about the signals is provided in the design level of a software product and it might be hard (or not possible) to capture this information in the testing level. On the other side, communication between different departments in an organization in order to capture and share information may require more time and effort than the amount of time saved by test scheduling. Moreover, scheduling test cases for execution based on other types of dependencies (e.g. temporal dependency) might be considered as a more efficient way for scheduling. The execution sequence is an important factor in a real-time system. The functionally dependent test cases can be tested after each other at any time during the testing cycles. However, the temporal dependent test cases should be executed directly after each other over a specific time period.

B. INTERNAL VALIDITY

concerns the conclusions of the study [35]. In order to reduce the threats to internal validity, the obtained results in this study are compared with the execution results of the test cases from three different testing cycles at BT. Furthermore, we assumed that those test cases which had failed due to other testing failure causes (e.g. mismatching between requirement and test cases) might be failed in the proposed execution schedule by sOrTES. Therefore only the failed test cases based on dependencies are assumed to be *passed* by following our execution order. In the present work, multiple test artifacts such as test specification, requirement specification and test records are analyzed and the experimental knowledge of testing experts is also considered. However, the structure of the software requirement specification and the test specification at our use case company provider can be considered a threat to this study. sOrTES was performed on a set of well-defined semi-natural language SRSs and test specifications which can be decomposed and analyzed quickly. In a more complex case with a different structure of SRS and test cases, sOrTES might (or might not) perform accordingly, which can influence the accuracy of dependencies and thereby the proposed execution schedules.

C. EXTERNAL VALIDITY

addresses the generalization of the proposed approach and findings [42]. sOrTES has been applied on just one industrial testing project in the railway domain, however, it should be applicable to other similar contexts in other testing domains. Primarily, the functional dependencies between requirements (and thereafter test cases) can be detected through analyzing the internal signal communications between the software modules in many different contexts. The extraction phase in sOrTES is currently designed based on the input structure of the requirements and test cases specifications. Secondly, scheduling test cases for execution based on several criteria and also their execution results are also applicable in other testing environments. Since the scheduling phases are designed based on a stochastic traveling salesman problem, sOrTES can also be applied for solving other queueing or stochastic traffic problems. Moreover, the context information has been provided in Section III to enable comparison with other proposed approaches in the testing domain.

D. CONCLUSION VALIDITY

deals with the factors which can impact the observations and can lead to an inaccurate conclusion [43]. Generally, an inadequate data analysis can yield conclusions that a proper analysis of the data would not have supported [44]. Utilizing the human's judgment, decision and supervision might decrease this threat. Since sOrTES is designed and developed for an industrial usage at BT, a close collaboration and dialogue with the domain experts is established in order to ensure the industry's requisite and needs. The study presented in this paper is started in parallel with the integration testing phase of the BR project and the initial results (dependency detection) have been presented in a few BR project meetings. Moreover, a workshop is conducted by us at BT for the members of BR and also C30,[9] which is a parallel ongoing project at BT. The testers's and engineers' opinions about the

[9]MOVIA C30 is a metro cars project ordered by the Stockholm Public Transport.

proposed schedules by sOrTES have been gathered, resulting in necessary modifications to sOrTES.

E. RELIABILITY

is the repeatability and consistency of the study [35]. The extraction phase of sOrTES has been tested for accuracy of results [32]. As outlined in section V, a total of 737 requirements are identified as independent requirements, which means that sOrTES could not find any matches for them. In consultation with the testers and engineers at BT, a set of wrongly spelled meanings were found in the requirements and test case specification documents. Thus, the data in the DOORS database sometimes contains ambiguity, uncertainty and spelling issues. sOrTES searches for exact names for $input - output$ signals for detecting dependencies. In the case that no *output signal* matches are found for an internal *input signal*, the corresponding requirement is counted as an independent requirement. Indeed, by missing one letter in the name of a signal, no signal matches will be found, even if the signal enters (or exits) to several requirements. This issue can directly impact the proposed execution schedules as well. In addition, most of the language parsing techniques have some performance issues when a large set of data needs to be processed. There are demerits in the available tools for natural language processing.

VIII. DISCUSSION AND FUTURE WORK

The main goal of this study is to design, implement and evaluate an automated decision support system which schedules manual integration test cases for execution. To this end, we make the following contributions:

- We have proposed an NLP-based approach to detect the dependencies between manual integration test cases automatically. The dependencies have been extracted by analyzing multiple test process artifacts such as the software requirement specification and test case specification (the extraction phase). A stochastic approach for optimal scheduling of test execution has been proposed. The travelling salesman problem (TSP) was utilized for identifying a feasible set among the test cases (the scheduling phase). The mentioned phases are integrated in a Python based tool called sOrTES.
- The evaluation of sOrTES was performed through applying it on an industrial testing project (BR) in a safety critical train control management subsystem at Bombardier Transportation in Sweden. Moreover, the execution schedules proposed by sOrTES was compared with three different execution orders, which have been previously performed by BT.
- The performance analysis of sOrTES indicates that the number of fulfilled requirements increased by 9.6% compared to the execution orders at BT.
- The total troubleshooting time is reduced up to 40% through avoiding redundant test executions based on dependencies.

Scheduling test cases for execution can provide an opportunity for using the testing resources in a more efficient way. Decreasing the redundant execution directly impacts the testing quality and the troubleshooting cost. Utilizing sOrTES in an early stage of a testing process can help testers to have a more clear overview about the dependencies between the requirements. This information can also be used for designing more effective test cases. The provided information about the test cases' properties (dependency and the requirement coverage) in Table 4 can be utilized for test case selection and prioritization purposes. In some testing levels, a subset of test cases just needs to be executed once. For this purpose, the inserted test cases in Table 4 can be ranked for execution. However, the problem of dependencies between test cases does not exist in all testing environments, where test cases can be prioritized based on a single or multiple criterion, such as their requirement coverage or execution time. As we discussed in Section VI, around 40% of failure that had occurred due to the dependencies between test cases, which can be eliminated through detecting the dependencies between test cases before the execution. Moreover, maximizing the requirement coverage in each execution is another optimization aspect of the proposed approach in this paper. Developing sOrTES as a more powerful tool which can handle even larger sets of requirements and test specifications is one of the future directions of the present work. Moreover, merging ESPRET for execution time prediction with sOrTES is also another considered research direction for us. In the future, one more step in the extraction phase will be added to sOrTES, which estimates a time value for each test case as the maximum required time for the execution. Dealing with time as a constraint for optimizing the scheduling problem can drive us toward using the traveling salesman problem with time windows that supports this intuition.

IX. CONCLUSION

Test optimization plays a critical role in the software development life cycle, which can be performed through test case selection, prioritization and scheduling. In this paper, we introduced, applied and evaluated our proposed approach and tool, sOrTES, for scheduling manual integration test cases for execution. sOrTES takes software requirement specifications and test specifications as input and provides the dependencies and requirement coverage for each test case as output. First, a feasible set of dependent test cases is identified by sOrTES and secondly, test cases are ranked for execution based on their requirement coverage. Our empirical evaluations at Bombardier Transportation and analysis of the results of one industrial project shows that sOrTES is an applicable tool for scheduling test cases for execution. sOrTES is also able to handle a large set of manual requirements and test specifications. sOrTES assigns a higher rank for those test cases which have a lower dependency and cover more requirements. By removing the *passed* test cases from each testing schedule, a new execution schedule will be proposed for the remaining test cases. This process will be continued until there are no test cases left in the testing cycle. Continuous monitoring of the test cases' execution results minimizes the risk of redundant execution and thereafter the troubleshooting efforts. Moreover, utilizing sOrTES leads to maximizing the number of fulfilled requirements per execution and thereby allowing for a faster release of the final software product. Having these two optimization aspects at the same time results in achieving a more efficient testing process and a higher quality software product.

ACKNOWLDEGMENT

The authors would like to thank Ola Sellin and Per Lindberg at Bombardier Transportation, Västerås, Sweden, for their support during the project.

REFERENCES

[1] S. Tahvili, M. Bohlin, M. Saadatmand, S. Larsson, W. Afzal, and D. Sundmark, "Cost-benefit analysis of using dependency knowledge at integration testing," in *Proc. Int. Conf. Product-Focused Softw. Process Improvement*. Norway: Springer, 2016, pp. 268–284.

[2] S. Tahvili, W. Afzal, M. Saadatmand, M. Bohlin, D. Sundmark, and S. Larsson, "Towards earlier fault detection by value-driven prioritization of test cases using fuzzy TOPSIS," in *Proc. 13th Int. Conf. Inf. Technol., New Gener.*, 2016, pp. 745–759.

[3] G. Kumar and P. K. Bhatia, "Software testing optimization through test suite reduction using fuzzy clustering," *CSI Trans. ICT*, vol. 1, no. 3, pp. 253–260, 2013.

[4] C. Gonzalez, J. F. Lerch, and C. Lebiere, "Instance-based learning in dynamic decision making," *Cognit. Sci.*, vol. 27, pp. 591–635, Jul./Aug. 2003.

[5] S. Tahvili, M. Saadatmand, and M. Bohlin, "Multi-criteria test case prioritization using fuzzy analytic hierarchy process," in *Proc. 10th Int. Conf. Softw. Eng. Adv.*, 2015, pp. 290–296.

[6] S. Tahvili, M. Saadatmand, S. Larsson, W. Afzal, M. Bohlin, and D. Sundmark, "Dynamic integration test selection based on test case dependencies," in *Proc. 11th Workshop Test., Academia-Ind. Collaboration, Pract. Res. Techn.*, Apr. 2016, pp. 277–286.

[7] E. Enoiu, D. Sundmark, A. Čaušević, and P. Pettersson, "A comparative study of manual and automated testing for industrial control software," in *Proc. IEEE Int. Conf. Softw. Test., Verification Validation*, Mar. 2017, pp. 412–417.

[8] R. Ramler, T. Wetzlmaier, and C. Klammer, "An empirical study on the application of mutation testing for a safety-critical industrial software system," in *Proc. Symp. Appl. Comput. (SAC)*, New York, NY, USA, 2017, pp. 1401–1408.

[9] J. Itkonen and M. Mäntylä, "Are test cases needed? Replicated comparison between exploratory and test-case-based software testing," *Empirical Softw. Eng.*, vol. 19, no. 2, pp. 303–342, 2014.

[10] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Softw., Test. Verification Rel.*, vol. 22, no. 2, pp. 67–120, 2012.

[11] S. Arlt, T. Morciniec, A. Podelski, and S. Wagner, "If a fails, can b still succeed? Inferring dependencies between test results in automotive system testing," in *Proc. IEEE 8th Int. Conf. Softw. Test., Verification Validation (ICST)*, Apr. 2015, pp. 1–10.

[12] X. Cai, X. Wu, and X. Zhou, in *Optimal Stochastic Scheduling* (International Series in Operations Research & Management Science). New York, NY, USA: Springer, 2014.

[13] J. Niño-Mora, "Stochastic scheduling," in *Encyclopedia of Optimization*, C. A. Floudas and P. M. Pardalos, Eds., 2nd ed. New York, NY, USA: Springer, 2009, pp. 3818–3824.

[14] M. Harman, "Making the case for MORTO: Multi objective regression test optimization," in *Proc. IEEE 4th Int. Conf. Softw. Test., Verification Validation Workshops*, Mar. 2011, pp. 111–114.

[15] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos, "Timeaware test suite prioritization," in *Proc. Int. Symp. Softw. Test. Anal.*, New York, NY, USA, 2006, pp. 1–12.

[16] L. Zhang, S.-S. Hou, C. Guo, T. Xie, and H. Mei, "Time-aware test-case prioritization using integer linear programming," in *Proc. 18th Int. Symp. Softw. Test. Anal.*, New York, NY, USA, 2009, pp. 213–224.

[17] S. Wang, S. Ali, T. Yue, O. Bakkeli, and M. Liaaen, "Enhancing test case prioritization in an industrial setting with resource awareness and multi-objective search," in *Proc. 38th Int. Conf. Softw. Eng. Companion*, New York, NY, USA, May 2016, pp. 182–191.

[18] Z. Li, M. Harman, and R. M. Hierons, "Search algorithms for regression test case prioritization," *IEEE Trans. Softw. Eng.*, vol. 33, no. 4, pp. 225–237, Apr. 2007.

[19] P. E. Strandberg, D. Sundmark, W. Afzal, T. J. Ostrand, and E. J. Weyuker, "Experience report: Automated system level regression test prioritization using multiple factors," in *Proc. IEEE 27th Int. Symp. Softw. Rel. Eng. (ISSRE)*, Oct. 2016, pp. 12–23.

[20] H. Srikanth, L. Williams, and J. Osborne, "System test case prioritization of new and regression test cases," in *Proc. Int. Symp. Empirical Softw. Eng.*, Nov. 2005, p. 10.

[21] P. Caliebe, T. Herpel, and R. German, "Dependency-based test case selection and prioritization in embedded systems," in *Proc. IEEE 5th Int. Conf. Softw. Test., Verification Validation*, Apr. 2012, pp. 731–735.

[22] S. E.-Z. Haidry and T. Miller, "Using dependency structures for prioritization of functional test suites," *IEEE Trans. Softw. Eng.*, vol. 39, no. 2, pp. 258–275, Feb. 2013.

[23] S. Tahvili, L. Hatvani, M. Felderer, W. Afzal, M. Saadatmand, and M. Bohlin, "Cluster-based test scheduling strategies using semantic relationships between test specifications," in *Proc. 5th Int. Workshop Requirements Eng. Test.*, 2018, pp. 1–4.

[24] A. Srivastava and J. Thiagarajan, "Effectively prioritizing tests in development environment," *SIGSOFT Softw. Eng. Notes*, vol. 27, no. 4, pp. 97–106, 2002.

[25] M. Nahas and R. Bautista-Quintero, "Developing scheduler test cases to verify scheduler implementations in time-triggered embedded systems," *Int. J. Embedded Syst. Appl.*, vol. 6, nos. 1–2, pp. 1–20, Jun. 2016.

[26] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE Trans. Softw. Eng.*, vol. 27, no. 10, pp. 929–948, Oct. 2001.

[27] J. M. Kim and A. Porter, "A history-based test prioritization technique for regression testing in resource constrained environments," in *Proc. 24th Int. Conf. Softw. Eng. (ICSE)*, New York, NY, USA, 2002, pp. 119–129.

[28] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal, "A study of effective regression testing in practice," in *Proc. Int. 8th Symp. Softw. Rel. Eng.*, Nov. 1997, pp. 264–274.

[29] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Softw. Eng.*, vol. 10, no. 4, pp. 405–435, 2005.

[30] S. Elbaum, A. Malishevsky, and G. Rothermel, "Incorporating varying test costs and fault severities into test case prioritization," in *Proc. 23rd Int. Conf. Softw. Eng. (ICSE)*, May 2001, pp. 329–338.

[31] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," *IEEE Trans. Softw. Eng.*, vol. 28, no. 2, pp. 159–182, Feb. 2002.

[32] S. Tahvili *et al.*, "Functional dependency detection for integration test cases," in *Proc. 18th IEEE Int. Conf. Softw. Qual., Rel. Secur.*, Jul. 2018, pp. 207–214.

[33] S. Tahvili, W. Afzal, M. Saadatmand, M. Bohlin, and S. H. Ameerjan, "ESPRET: A tool for execution time estimation of manual test cases," *J. Syst. Softw.*, vol. 146, pp. 26–41, Dec. 2018.

[34] J. B. Odili and M. N. M. Kahar, "Solving the traveling salesman's problem using the African buffalo optimization," *Comput. Intell. Neurosci.*, vol. 2016, Aug. 2016, Art. no. 1510256.

[35] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Softw. Eng.*, vol. 14, no. 2, p. 131, 2008.

[36] E. Engström, P. Runeson, and A. Ljung, "Improving regression testing transparency and efficiency with history-based prioritization—An industrial case study," in *Proc. 4th IEEE Int. Conf. Softw. Test., Verification Validation*, Mar. 2011, pp. 367–376.

[37] Hamburg, Germany. *Electric Multiple Unit Class 490*. Accessed: Feb. 12, 2018. [Online]. Available: https://www.bombardier.com/en/transportation/projects/project.ET-490-Hamburg-Germany.html

[38] Y. Cho, J. Kim, and E. Lee, "History-based test case prioritization for failure information," in *Proc. 23rd Asia–Pacific Softw. Eng. Conf. (APSEC)*, Dec. 2016, pp. 385–388.

[39] T. B. Noor and H. Hemmati, "A similarity-based approach for test case prioritization using historical failure data," in *Proc. IEEE 26th Int. Symp. Softw. Rel. Eng. (ISSRE)*, Nov. 2015, pp. 58–68.

[40] X. Wang and H. Zeng, "History-based dynamic test case prioritization for requirement properties in regression testing," in *Proc. Int. Workshop Continuous Softw. Evol. Del. (CSED)*, New York, NY, USA, May 2016, pp. 41–47.

[41] C. Robson, *Real World Research: A Resource for Users of Social Research Methods in Applied Settings*, 3rd ed. Chichester, U.K.: Wiley, 2011.

[42] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering: An Introduction*. Norwell, MA, USA: Kluwer, 2000.

[43] P. C. Cozby, *Methods in Behavioural Research*. New York, NY, USA: McGraw-Hill, 2012.

[44] E. A. Drost, "Validity and reliability in social science research," *Educ., Res. Perspect.*, vol. 38, no. 1, p. 105, 2011.

SAHAR TAHVILI received the master's degree in applied mathematics and the Ph.D. degree in software engineering from Mälardalen University. Her Ph.D. Thesis was on multi-criteria optimization of system integration testing. She is currently a Researcher with RISE SICS Västerås and also a member of the Software Testing Laboratory, Mälardalen University. His research interests include the advanced methods for testing complex software-intensive systems and designing the decision support systems.

RITA PIMENTEL received the master's degree in mathematical finance and the Ph.D. degree in statistics and stochastic processes. She is currently a Researcher with RISE SICS Västerås. She also works in mathematical modeling, mainly to solve optimization problems. She also uses statistical learning for prediction. She is involved in projects with applications in energy and software testing.

WASIF AFZAL is currently an Associate Professor with Mälardalen University, Sweden. He likes to do empirical research within software engineering in general and within software verification and validation in particular. His research interests include (not exhaustive): software testing and quality assurance, prediction and estimation in software engineering, the application of artificial intelligence techniques in software engineering (including search-based software engineering), decision-making based on software analytics, software metrics, and evidential assessment of software engineering literature.

MARCUS AHLBERG received the M.Sc. degree in applied mathematics from the KTH Royal Institute of Technology, in 2018.

ERIC FORNANDER received the M.Sc. degree in applied mathematics from the KTH Royal Institute of Technology, in 2018.

MARKUS BOHLIN received the Ph.D. degree with Mälardalen University, in 2009, where he was appointed as an Associate Professor (Docent) in computer science, in 2013. He is currently the Leader of the industrial efficiency with RISE SICS Västerås. His research interests include the real-world application of optimization methods and operations research to industrial processes.