Software Qual J (2018) 24:1421–1449
https://doi.org/10.1007/s11219-017-9401-7

# Similarity-based prioritization of test case automation

**Daniel Flemström[1] · Pasqualina Potena[2] · Daniel Sundmark[1] · Wasif Afzal[1] · Markus Bohlin[2]**

Published online: 8 March 2018
© The Author(s) 2018. This article is an open access publication

**Abstract** The importance of efficient software testing procedures is driven by an ever increasing system complexity as well as global competition. In the particular case of manual test cases at the system integration level, where thousands of test cases may be executed before release, time must be well spent in order to test the system as completely and as efficiently as possible. Automating a subset of the manual test cases, i.e, translating the manual instructions to automatically executable code, is one way of decreasing the test effort. It is further common that test cases exhibit similarities, which can be exploited through reuse when automating a test suite. In this paper, we investigate the potential for reducing test effort by ordering the test cases before such automation, given that we can reuse already automated parts of test cases. In our analysis, we investigate several approaches for prioritization in a case study at a large Swedish vehicular manufacturer. The study analyzes the effects with respect to test effort, on four projects with a total of 3919 integration test cases constituting 35,180 test steps, written in natural language. The results show that for the four projects considered, the difference in expected manual effort between the best and the worst order found is on average 12 percentage points. The results also show that our proposed prioritization method is nearly as good as more resource demanding meta-heuristic approaches

✉ Daniel Flemström
daniel.flemstrom@mdh.se

Pasqualina Potena
pasqualina.potena@ri.se

Daniel Sundmark
daniel.sundmark@mdh.se

Wasif Afzal
wasif.afzal@mdh.se

Markus Bohlin
markus.bohlin@ri.se

[1] Mälardalen University, Västerås, Sweden

[2] RISE SICS Västerås, Västerås, Sweden

at a fraction of the computational time. Based on our results, we conclude that the order of automation is important when the set of test cases contain similar steps (instructions) that cannot be removed, but are possible to reuse. More precisely, the order is important with respect to how quickly the manual test execution effort decreases for a set of test cases that are being automated.

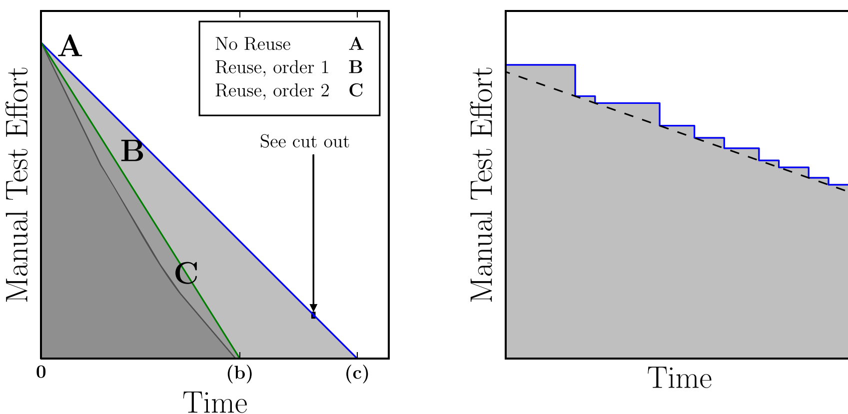
**Keywords** Software-testing · Test-case automation · Prioritization · Reuse · Effort

## 1 Introduction

Several studies, e.g., Asaithambi and Jarzabek (2013), Engström and Runeson (2013), and Greiler et al. (2012a), have shown the presence of similarities or overlaps in test cases. Even though similarities, both within and among test cases, can be identified, our previous study (Flemström et al. 2015) showed that these overlaps can be scattered all over the set of test cases and could not be trivially removed. Further, the study indicated that test cases seldom overlap entirely. If these test cases are expressed in natural language and at some point in time translated to automatically executable code, the work of automating the similar parts *again* is possibly wasteful.

Figure 1a illustrates how the manual test effort is reduced over time due to test case automation (ignoring the effort of executing the automated test cases). The line $A$ illustrates how the manual effort would decrease over time if the test cases were automated in a steady pace, considering each part of the test cases in isolation, i.e., not considering any reuse. At point ($c$) in time, the automation work is completed and no more manual effort is required for a system test. Given that we can efficiently reuse already automated parts of the test cases, using a technique such as the one suggested in Flemström et al. (2016) to identify reusable parts, the automation work could be finished earlier. In Fig. 1a, this is illustrated as the point in time ($b$).

We argue that, by combining similarity-based reuse with a more efficient ordering strategy, an earlier reduction of the manual test execution effort can be achieved. This is




(a) Effect of different automation strategies on test effort

(b) Effort reduction is gained when a test case is completely automated, resulting in a step wise reduction

**Fig. 1** Manual test execution effort reduction over time due to test case automation

illustrated in Fig. 1a by the slopes $B$ and $C$ that represent two different automation orders, whereas $C$ gives the earliest manual effort reduction. In the collaboration with our industrial partners, we have observed that test cases are performed manually if they are not entirely translated to executable code. This is illustrated in Fig. 1b.

In this paper, we present a strategy for decreasing manual test execution effort by ordering the test cases for automation, using knowledge on test similarities. We further present an industrial evaluation of our approach for four large-scale systems in the vehicular embedded software domain. The main contributions are as follows:

- A formal definition of the test case automation problem.
- A fast, time deterministic, and easy-to-implement automation ordering heuristic for test case automation prioritization.
- A large industrial case study with four industrial projects in the vehicular embedded software domain.
- A thorough computational evaluation of the approach, in which seven prioritization methods (five heuristic and two meta-heuristic) are compared.

The results of the evaluation show that for the four projects considered and random initial ordering of test cases, the best and the worst orders found differ on average 12 percentage point in expected manual test effort.

The results also show that our proposed prioritization method is nearly as good as more resource demanding meta-heuristic approaches at a fraction of the computational time.

The paper is organized as follows. In Section 2, we present a selection of previous work and some preliminaries about test cases. In Section 3, we present our approach, followed by an industrial case study in Section 4. The results of this industrial case study is presented in Section 5. The threats to validity are discussed in Section 6, followed by conclusions and future work in Section 7.

## 2 Background

In this section, we outline the previous work that has an direct or indirect bearing on the work described in this paper, and discuss the preliminaries and context of the work. Finally, we give an overview of the problem we address.

### 2.1 Previous work

Several papers in the past dealt with test case automation from different perspectives, see e.g., Anand et al. (2013), that surveys techniques for automatic generation of software test cases. Further, Ramler and Wolfmaier (2006) analyze test automation from a cost-benefit perspective. Another class of related papers deals with decision-support to determine the optimal degree of automation adoption, such as a simulation model (Garousi and Pfahl 2015) or using a search-based approach (Amannejad et al. 2014). Our work focuses on the industrial test case automation (scripting) order problem based on reuse; therefore, hereafter, we review works appearing in the literature dealing with (i) *similarity-based approaches for test cases* and (ii) *test case prioritization*.

*(i) Similarity-based approaches for test cases:* Several research efforts have been devoted to find and group similarities in test-related artifacts using different methods, e.g., test similarity exists in several dimensions in the context of software product lines (Engström and Runeson 2013). One approach is by using clone detection in test cases (see, e.g., Asaithambi

and Jarzabek (2013) and Basit and Jarzabek (2009)). Another technique, used to make the similarity comparison easier, is to record and analyze the execution trace of test cases rather the test cases themselves (Greiler et al. 2012b). The major drawbacks of this method are the instrumentation of the production code and the difficulty to get reliable traces from manual test cases (an extensive overview of finding software clones can be found in Roy et al. (2014)).

Similarity-based algorithms have also been applied to regression test case prioritization, based on the distances between pair-wise test cases (Wang et al. 2015), using historical failure data (Noor and Hemmati 2015) or topics models (Thomas et al. 2014). Research effort have also been devoted to similarity-based selection techniques, in particular, in the model-based domain (Hemmati et al. 2013). Another class of related papers represents the opposite direction of test similarity research and measures the diversity between two test sets (Shi et al. 2016; Feldt et al. 2015). These techniques may be used as an inverse indicator of test case similarity.

*(ii) Test case prioritization*. Several papers in the past dealt with regression testing (an extensive list can be found in the survey (Yoo and Harman 2012)). Considerable research effort has been devoted to prioritize test cases with test execution information, that includes the total coverage of code components, the coverage of code components not previously covered, or the ability of test cases to reveal faults in the code components that they cover (see Rothermel et al. (2001) for a comparison of these techniques).

Most existing prioritization approaches are based on single-objective optimization models and are focused on greedy algorithms. However, some recent papers focus on multi-objective optimization, e.g., (i) with resource-awareness (Wang et al. 2016) and (ii) minimizing coverage-based criteria, such as the average percentage of coverage achieved, average percentage of coverage of changed code, and average percentage of past fault coverage (Epitropakis et al. 2015; Strandberg et al. 2016). Evolutionary algorithms have also been applied to test case prioritization with single-objective optimization (Li et al. 2007), as well as with multi-objective optimization (Wang et al. 2016; Epitropakis et al. 2015) (for which swarm algorithms and hybrid ones have also been used (Wang et al. 2016)). Few approaches have addressed time constraints in the context of test case prioritization (Marijan 2015). Some research effort has been spent to deal with the test execution time, adopting for example a genetic algorithm to reorder test suites in light of testing time constraints (under the assumption that all the tests have the same execution time) (Walcott et al. 2006). Other approaches are integer linear programming (Zhang et al. 2009) and knapsack solvers (Alspaugh et al. 2007). The effect of constraints (on time to setup testing, time to identify and repair obsolete tests, and human time to inspect results) on test prioritization has been analyzed in Do et al. (2010) and Do and Rothermel (2006). Moreover, in Do et al. (2010), Do and Rothermel (2006), and Tahvili et al. (2016), it has also been defined an economic model to evaluate costs and benefits of prioritization techniques—in terms of the cost of applying the methodologies and how much revenue they help organizations obtain.

Our approach can be used complementary to other ones aimed to reduce human involvement in writing test oracles and evaluating testing outcomes (Barr et al. 2015).

The focus of this paper is at system integration testing level, for which, especially in the case of manually performed test cases, there is no or little industrial data or evidence to show test similarities and how these can be used for reducing test effort. The paper allows practitioners to exploit knowledge on test similarities for decreasing test case automation as well as manual test execution effort. Furthermore, we provide empirical evidence of the usage of our approach in a real-word context.

## 2.2 Preliminaries

The focus of this paper is system integration level testing. This kind of testing aims to ensure that a combined (integrated) set of subsystems behaves according to the system specification. In our context, we regard a test case as a series of test *steps* that should be executed in a specific order. A test step consists of a *stimulus* and an expected *reaction*. A stimulus consists of one or more manual instructions, such as “Press the button A.”. The purpose of the stimulus is to eventually put the system under test into a state where it is feasible to verify the associated requirements. The verification is performed as per the instructions in the corresponding reaction. If the observable output of the system matches the specified reaction, the test step is considered “passed,” if not, it is marked “failed” in a testing protocol and a trouble report is written.

In our context, such system integration test cases are either expressed as manual or automated test cases. The manual test cases comprise a set of instructions, targeting a human that should perform some actions and verify the response of the system according to the instructions. In contrast, the automated test cases do not require manual intervention. We use the term *test case automation* for the process of translating each such individual stimulus to an automatically executable script, that requires no human intervention.

An example of a system integration test case is shown in Table 1. The manual instructions are presented in the column “Stimuli” whereas the corresponding automated code is presented in the rightmost column. The purpose of the column “Test step ID” is to facilitate identification of similar test steps with respect to stimuli. If the ID is the same, the stimuli of the test steps are considered the same. An example of this are the step numbers 3 and 6, that have the same ID. Looking at the stimuli column, we can see that they indeed express the same thing, although slightly reworded. Although we use the technique originally described in our previous work (Flemström et al. 2016) to identify such similar test steps, any similarity function that returns the same numerical value for each two similar test steps can be used. For this paper, we have only considered the stimulus of the test steps. The approach, however, is still the same: first neutralize some of the natural language issues, such as formatting, stop words (e.g., this, the) and word order for each step. Each resulting distinct test step is then assigned a unique test step ID as shown in the *Test step ID* column of Table 1. The whole test cases can then be expressed as a series of test step IDs. The example in Table 1 would result in the sequence [1024, 2033, **2311**, 2517, 4211, **2311**, 3210]. The test steps in bold denote the above mentioned similarities. The column *Automated code*

**Table 1** Obfuscated example test case, with automated test steps

| Step | Stimuli (reaction omitted) | Test step ID | Automated code |
|---|---|---|---|
| 1 | Initial state: Train is ready to drive | 1024 | `train_init(READY)` |
| 2 | Run the train at speed 1234 km/h | 2033 | `speed_set(1234)` |
| 3 | Status set to faulty for sensors *XYZ* for the motor car axle x | **2311** | **`fault_inject(XYZ,x)`** |
| 4 | Set *XYZ* status to non faulty | 2517 | `fault_clear(XYZ)` |
| 5 | Set slide *XYZ* status as active | 4211 | `slide_status(XYZ,1)` |
| 6 | Set status to faulty for sensors *XYZ* for motor car axle *x* | **2311** | `fault_inject(XYZ,x)` |
| 7 | Clean up.... | 3210 | `reset()` |

This test case would be encoded as [1024, 2033, **2311**, 2517, 4211, **2311**, 3210]

illustrates how the automatically executable code could look like for the corresponding test steps. Given that the similarity between steps #3 and #6 was known, the automated code for step #3 could have been reused when automating step #6. In this paper, we denote such reuse as *similarity-based reuse*, or just reuse for short. Such similarities are, however, not restricted to those occurring in the same case, as in the example, but in general, span several test cases.

### 2.3 Problem statement

In some of our partner organizations, the system integration testing is performed manually during the first release cycles of the developed software system. When the system has reached an acceptable level of completeness and stability, a subset of the test cases, $T$, is selected and automated with the purpose of reducing manual test execution effort. Such automation work is performed over time with a subset of the testing crew, while the normal testing activities in the project must continue in parallel. Thus, there is a high probability that the system needs to be tested before all the test cases have been automated. Further, due to the many people involved, as well as regulatory issues, there exists a certain amount of similarities (or overlaps) in the test cases. Such similarities occur often due to copy-paste reuse of the manual test cases. Our intuition is that by reusing the automated versions of similar test steps, not only can the automation effort be reduced, but by carefully planning the automation work order, it should be possible to reduce the manual test execution effort in a quicker pace.

> **Problem Statement:** *The problem is to find an optimal automation order with respect to early reduction of remaining manual effort during the work of automating a set of test cases, considering similarity-based in the bubble.*

## 3 Reducing manual test execution effort using similarity-based reuse and automation order prioritization

In this section, we present the underlying assumptions and definitions that finally leads to our suggested approach to automation order prioritization. In Section 3.1, we present the assumptions that the rest of the work relies on. Based on these assumptions, we define test case automation in Section 3.2 . In Sections 3.3 and 3.4, we explain the theoretical framework on how to measure manual effort and how to compare the effect of different automation strategies. This theory is further exemplified in Section 3.5. A formal problem formulation is given in Section 3.6, and lastly, we present a heuristic simple, fast, and deterministic ordering algorithm in Section 3.7. Table 2 summarizes the symbols used throughout the paper.

### 3.1 Assumptions

Our approach makes the following explicit assumptions and simplifications:

- All test cases are independent.
- Manual test cases are written in natural language.
- The automation work is performed sequentially and at a rate, $\lambda$ test steps / hour. This implies that all test steps are considered equal with respect to automation time.

**Table 2** Main notation adopted in the paper

| Symbol | Description |
|---|---|
| $T$ | The set of manual test cases that we want to automate. |
| $\|\tau\|$ | Number of test steps in the particular test case $\tau$. |
| $\|\|T\|\|$ | Number of test steps accumulated for all test cases in $T$. |
| $\sigma$ | Automation strategy, resulting in an automation order as an ordered list of the indices in $T$ such that $i, j \in \sigma, i \neq j, \tau_i \in T, \tau_j \in T$. Practically we see $\sigma = \{reuse/no-reuse, orderingalgorithm\} \longrightarrow \sigma_1, ..., \sigma_n$ |
| $\tau_{\sigma_i}$ | The $i$:th test case index in the order of $\sigma$ |
| $\tau_i$ | The test case in $T$ with index $i$. $\tau_i \in T, i = 1..\|T\|$ |
| $\gamma$ | Average manual test execution rate (test steps / hour). |
| $\lambda$ | Average automation rate (test steps / hour). |
| $t_0$ | The point in time where we start automating the test cases in $T$. |
| $t$ | Point in time when a system test is performed. |

- The rate $\lambda$ is constant until all test cases have been automated.
- A test case is considered to be a manual test case until all its steps have been scheduled for automation and automated.
- The effort of executing an automated test case is not considered.
- The effort of verifying and adapting a reused test step is not considered.
- During a system test, the execution of remaining manual test cases is performed sequentially and at a rate, $\gamma$ test steps / hour.
- The rate $\gamma$ is constant until all manual test cases have been executed.
- We approximate the effort for reusing an automated step to zero for the reason that this effort is substantially less than searching for the corresponding commands and signal names in the test automation framework manuals.

In this paper, for sake of simplicity, we have only considered the test stimuli. However, the approach is trivially extended to include the expected response part of a test step as well. This can be done without essentially changing the overall approach/model structure, but with the side effect of increasing the solution's complexity.

### 3.2 Test case automation

Automating a set of manual test cases involves translating the instructions into their automatically executable counterpart (automation code). While these manual instructions are written in natural language, the automation code is typically expressed in some form of script or visual programming language. This automation code can sometimes be reused if we can keep track of similar manual test steps that have been automated before.

For such automation work, we envision a tool where the tester can automate one manual test case at a time, step by step, having the manual instructions to the left and the corresponding script code attached to the right as illustrated in Table 1. If the user encounters a test step that is similar to a test step that has been automated before, the corresponding script code is presented to the user as a suggestion.

How fast the automation work is completed depends on many factors. In this paper, we shift the focus from the *automation rate* ($\gamma$) in favor of the *automation order* ($\sigma_1, .., \sigma_n$)

given a *decision to reuse*. The automation order describes how we schedule the test cases for the automation work, i.e., the sequential order of the test cases. With reuse, in this context, we mean similarity-based reuse, where the automation code of similar test steps is reused.

A well-chosen automation order is an order where as much of the test cases as possible gets automated, as early as possible, and with the least required effort. This automation order can be determined manually, or by an algorithm. An example of such an algorithm is given in Section 3.7. Other algorithms search for an optimal solution using an objective function that measures how good a given automation order is. An example of an objective function is given in Section 3.6.

We denote this process of creating an automation order the *automation strategy* ($\sigma$). The concept of automation strategy may be extended to include more aspects such as sub system prioritization and selection of test cases. In this paper, however, we primarily focus on the automation order and the necessity of similarity-based reuse. The effect of different automation strategies will be further discussed in Section 3.3.

The context given above leads us to the following observations: The set of manual test cases $T$ is sequentially automated over time, using some automation strategy $\sigma$. Given that we can regard the same test case as either manual or automated, this means that $T$ can always be divided into two partitions: The remaining (not wholly automated) manual test cases $T_{\mathrm{M}}$ and the *wholly automated* test cases $T_{\mathrm{A}}$.

**Definition 1** A test case is *wholly automated* when (i) the test case has been scheduled for automation according to the decided automation order, (ii) all its steps have been translated to script code, and (iii) all reused step scripts have been verified and adapted until they are suitable for the current test case.

From Definition 1, it follows that a test case is not considered to be wholly automated even if *all* of its steps are similar to steps that has been automated before. According to the above definition, such a test case is still considered to be a manual test case until the test case has been scheduled for automation, and all steps have been properly validated and adapted to the current test case. However, in our model, such validation and adaptation effort is approximated to zero, since it is considered to be much less than the automation effort. This has the effect that such a test case will be "immediately" automated as soon as it is scheduled for automation, but not before that.

Let us consider a practical example of the reasoning above: The test cases $T_x$ and $T_y$ contain one test step each. These test steps are similar to each other. At some point in time, we wholly automate $T_x$. As a consequence, the test step in $T_y$ is already automated. However, this test case is scheduled to be automated later, so it will continue to be considered as a manual test case. Whenever we reach $T_y$ in the automation order, $T_y$ is immediately considered wholly automated. Notably, a successful prioritization strategy would place $T_y$ close to $T_x$.

We now formally define the set of wholly automated test cases at any particular time $t$, given an automation rate of $\lambda$ test steps / hour and an automation order $\sigma$, defined by a sequence of test cases $(\tau_{\sigma_1}, .., \tau_{\sigma_n})$ in the set of test cases $T$. From Definition 1 and the assumption that verification and adaptation time is zero, the set $T_{\mathrm{A}}(T, t, \lambda, \sigma)$ of wholly automated test cases at time $t$ contains exactly the elements in the longest prefix $(\tau_{\sigma_1}, .., \tau_{\sigma_i})$ for which the number of unique test steps therein is less than $\lambda t$. From the assumptions and Definition 1, we have that $k(i)$, the total number of automated test steps at position $i$ in $\sigma$, is

$$k(i) = \left| \cup_{j=1}^{i} \tau_{\sigma_j} \right|. \tag{1}$$

Using (1), we can then simply define $T_A(T, t, \lambda, \sigma)$ as follows.

**Definition 2** The set $T_A(T, t, \lambda, \sigma)$ of wholly automated test cases up to time $t$ is

$$T_A(T, t, \lambda, \sigma) = \{\tau_{\sigma_1}, \ldots, \tau_{\sigma_i}\}, \tag{2}$$

where $i = \text{argmax}_i(k(i) \mid k(i) \leq \lambda t)$.

### 3.3 Measuring manual execution effort

To understand how the choice of automation strategy affects manual test execution effort, we first consider the examples in Fig. 2. The examples illustrate how test case automation reduces the manual effort over time for a set of test cases $T$, given a constant automation rate ($\lambda$). The *y*-axis shows how much of the initial manual effort is left, expressed in number of test steps, where $||T||$ is our notation for the total numbers of test steps in $T$. The *x*-axis shows the time, from the start of automation.

In the first example, "No Reuse, Any Order" ($\sigma$), the automation strategy is to automate all steps of the test cases in isolation, in any order. The time it takes to automate a test case is not dependent on the test cases we have automated before. Thus, how quickly the manual effort is reduced is determined by the automation rate ($\lambda$), the choice of "no reuse", but not the automation order.

In contrast, when considering similarity-based reuse, the automation work may be finished quicker than in the case of not considering reuse, despite the same automation rate ($\lambda$). The reason for this is that fewer test steps will need automation. This in turn results in a much quicker reduction of the manual effort.

The order affects how fast the effort is reduced because the time it takes to automate a test case now depends on the test cases we have automated before, i.e., how many already automated test steps we can reuse. Theoretically, it should be possible to find an order so we reduce the manual effort as early as possible. This is illustrated in the example ($\sigma'$) in Fig. 2. Given the same $\lambda$, different automation orders would result in different slopes. Some orders

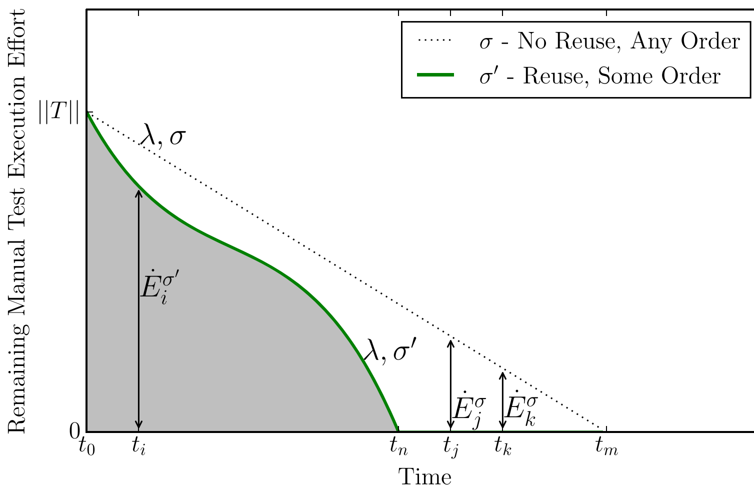



**Fig. 2** Manual test execution effort reduction over time, using different automation strategies

will be more beneficial than others with respect to early manual execution effort reduction or total effort reduction.

If we perform a system test before the automation work has even started (i.e., before $t_0$), the required manual execution effort is clearly equal to the effort of executing all the test cases in $T$ manually. If we can wait to test the system until all test cases have been automated, there would be no manual execution effort required at all. At $t_j$ and $t_k$ in the same figure, the required manual effort to test the system would be $\dot{E}^{\sigma}_{j}$ and $\dot{E}^{\sigma}_{k}$, respectively. In the remainder of this section, we will elaborate on how to estimate that effort.

**Definition 3** The manual test execution effort for a set of test cases, $T$, measured in number of test steps is defined as follows:

$$e_{\mathrm{m}}(T) = \sum_{i=1}^{|T|} |\tau_i|, \tag{3}$$

where $T$ is a set of test cases to be manually executed.

The goal of this section is to present a measure of the manual test execution effort if we perform a system test before the automation work has finished, say at time $t$. Using Definition 3, this effort corresponds to the effort of executing the set of test cases that has not yet been wholly automated. Since the set of wholly automated test cases can be found using Definition 2, we can measure the effort as follows:

**Definition 4** The effort of executing all remaining test cases at time $t$ is as follows:

$$\dot{E}(T, t, \lambda, \sigma) = e_{\mathrm{m}}(T - T_{\mathrm{A}}(T, t, \lambda, \sigma)), \tag{4}$$

where $t$ denotes the point in time where the test starts. $T_{\mathrm{A}}(T, t, \lambda, \sigma)$ is the set of wholly automated test cases up to that point in time ($t$), given the automation strategy $\sigma$ and the automation rate $\lambda$. $T$ is the original set of manual test cases that should be automated.

### 3.4 Comparing approaches

As the manual effort reduction from the set of automated test cases $T_{\mathrm{A}}(T, t, \lambda, \sigma)$ varies over time, we base our comparisons on the manual test execution effort we can expect *on average* over the duration of the test automation process. To illustrate the principle, we could take some samples of the manual effort in Fig. 2 according to Definition 4 at the points $t_j$ and $t_k$. This yields an average of $\bar{E} = \left(\dot{E}^{\sigma}_{j} + \dot{E}^{\sigma}_{k}\right)/2$.

Intuitively, the more samples, the closer $\bar{E}$ will be to the true mean. Since an effort reduction can only take place whenever a test step has been automated, we sample after each time a test step has been automated in order to get a true mean of the manual effort. Sampling more often would not yield a different answer since we divide by the number of sample points and the required effort does not change during automation of a single step.

We further define a constant automation rate, approximated to $\lambda = 1$ step per time unit, which gives us that the automation effort as well as the manual test execution effort for one test step is equal to one time unit. Thus, by using the Definition 4 and sampling at each finished test step $t = \{0, .., ||T||\}$, where $T$ is the set of test cases that are subject to automation, we argue that this will capture (i) that a fully automated test case does not contribute to effort reduction until its fully automated and (ii) that an early effort

reduction (with respect to manual test execution effort) will contribute more to the total effort reduction since reduction will cumulatively contribute to each subsequent system test.

**Definition 5** The average expected manual test execution effort (in % of original manual effort of $T$) is as follows:

$$\bar{E}(T, \lambda, \sigma) = \frac{100}{||T||^2} \sum_{t \in \{1,..||T||\}} \dot{E}(T, t, \lambda, \sigma). \qquad (5)$$

$||T||$ is the total number of test steps in the set of test cases $T$ and $\dot{E}(T, t, \lambda, \sigma)$ is the required manual test execution effort at time $t$, given the automation rate $\lambda$ and automation strategy $\sigma$ according to Definition 4.

### 3.5 An example of similarity-based reuse

To understand in more detail how similarity-based reuse and automation order contributes to the overall effort reduction, let us examine the example in Table 3 from an automation effort perspective and from a manual test effort perspective. For the sake of simplicity, we set both the automation rate and the execution rate to 1 ($\lambda = 1$, $\gamma = 1$) in this example.

Table 3 describes three imaginary test cases $A$, $B$, and $C$. The characters in the attached sequences, e.g., [**a**, **b**, **c**] and [**x**, **y**, **z**], symbolizes *test steps*. Each such test step corresponds to some manual action, e.g., “Turn Ignition Key to II,” that is subject to be automated.

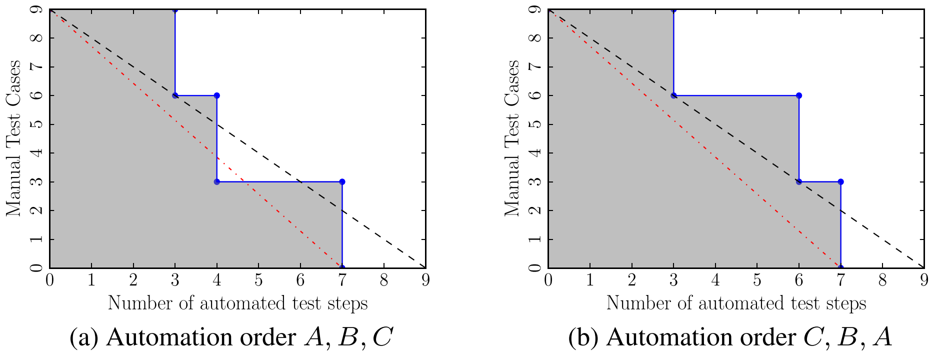
Given these prerequisites and a traditional approach, where each test case is automated in isolation, a total of nine test steps would require automation. However, if we examine the test sequences, some stimuli are repeated trough out the set of test cases. The stimuli *a* occurs in $A$, but also in $B$. It should thus be possible to save *automation* effort if we keep track of similar test steps in such a way that we can reuse the corresponding automated script. The automation effort for the given case can thus (theoretically) be reduced to the set of *distinct* test steps $S = [\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, \mathbf{x}, \mathbf{y}, \mathbf{z}]$ and the automation effort is thus $|S| = 7$ steps. The total automation effort will always be constant with respect to the automation order.

To better understand how the manual test *execution* effort varies depending on the automation order, we go ahead and automate the test cases in Table 3, first in the order $A, B, C$ and then in the order $C, B, A$ .

The work is illustrated in Fig. 3. On the *y*-axis, we find the required manual test execution effort over time. This effort is reduced over time, as the proportion of automated test cases grows. The test effort reduction for an automated test case occurs when it has been scheduled for automation and wholly automated, which gives the staircase shape of the slope. The *x*-axis describes the time spent on automating test steps with the resolution of one automated test step. The dotted lines are visual aids to facilitate comparing the resulting effort reduction slope for different automation orders. Further, the gray shadow under the manual effort is for also for clarity.

**Table 3** Example test case set $T$

| Case ID | Length | Stimuli sequence |
|---|---|---|
| $A$ | 3 | [**a**, **b**, **c**] |
| $B$ | 3 | [**a**, **b**, **d**] |
| $C$ | 3 | [**x**, **y**, **z**] |
| Total | 9 | |




(a) Automation order $A, B, C$ (b) Automation order $C, B, A$

**Fig. 3** Different automation orders give different slopes

Before we have automated anything ($t = 0$ in Fig. 3), the test effort is always equal to the effort of manually executing all test steps of all test cases: $|A| + |B| + |C| = 9$ steps in total. Starting with order $A, B, C$ (Fig. 3a), we get an effort reduction of $|A| = 3$ steps when all steps in $A$ have been automated. So between $t = 3$ and $t = 4$, the remaining manual effort is $9 - 3 = 6$ steps. Next up is test case $B$. Since the steps $a$ and $b$ are already automated, the test case is already complete after automating test step $d$. The last test case, $C$, contains three new test steps and thus takes three time units to completely automate. If we sample the expected manual test execution effort after each finished test step we would thus get $\bar{E} = 100 \cdot (9+9+9+6+3+3+3+0+0)/9^2 = 52\%$. The practical meaning is that, a full system test at a random point in time between $t = 0$ and $t = 9$, would on average require 52% of the initial manual effort.

Figure 3b illustrates the manual effort reduction over time when automating the test cases in the order $C, B, A$. Here, the information on overlaps cannot be used until the very last test case, which gives $\bar{E} = 100 \cdot (9+9+9+6+6+6+3+0+0)/9^2 = 59\%$.

### 3.6 Formal problem definition

The problem considered in this paper is to find the automation strategy for a set of test cases $T$, that gives the lowest average expected manual test execution effort ($\bar{E}$). The automation strategy is fixed to similarity-based reuse where the sought parameter represents the ordered sequence of test cases (that are subject to automation), denoted as $\sigma$. The automation of each test case follows Definition 5. The approach developed in this paper solves a restricted version of the full problem, since we assume that the initial effort of automating a test case, as well as the reuse effort for any step, is zero. Further, we assume that all test steps take exactly one unit of work time to execute and to automate.

**Definition 6** Let $T$ be a set of test cases that should be automated over time. Find the sequence $\sigma$ which minimizes the fitness function (derived from Definition 5).

$$\text{Fit}(T, \lambda, \sigma) = \frac{||T||^2}{100} \bar{E}(T, \lambda, \sigma) = \sum_{t \in \{1,..||T||\}} \dot{E}(T, t, \lambda, \sigma) \tag{6}$$

### 3.7 Potential/effort prioritization

There are several off-the-shelf algorithms that, given an objective function as in Definition 6, are capable of returning an optimized order. However, we expect these to be too time-consuming and complex to use in an on-line tool situation. In a situation where several persons are performing the automation work in parallel, the prioritization algorithm must be re-run each time a new test case is to be picked for automation to make the automation work appear as sequential to the algorithm, since the algorithm does not take parallel automation into account. Therefore, we propose a simple and time-efficient heuristic algorithm, detailed in Algorithm 2, that attempts to arrange the test cases in such a way that the automation effort is spent in order to give as much manual effort reduction as early as possible.

The input to this algorithm is a set of test cases that should be automated. The output is the order $\sigma$ that they should be automated in to give the assumed largest and earliest expected manual test execution effort reduction.

The basic idea is to successively rank all remaining test cases in the set and pick out the highest ranked one until the set is empty. The rank is constructed for each test case in the set using two factors. The first factor is its potential of reducing the number of manual test steps, which is at least its own length, but can be more if the contained test steps are repeated in other test cases. The second factor is the required automation effort for the test case. If many of its steps have already been automated, the effort will be less. The ranking factor is thus the ratio potential/effort score. In more detail, the potential/effort calculation function, depicted in Algorithm 1, estimates the possible potential for a particular test case $\tau$ as the gain of automating $\tau$ plus potential future gain, considering the set of already implemented test steps as well as the set of remaining manual test steps. The information on future gain is derived from the occurrence map $O$ that associates each distinct step with its number of, not yet, automated occurrences in the total set of test cases $T$. With $O[s] = 3$, we thus mean that the step $s$ occurs three times in $T$. The practical implication is that if we automate $s$, we will get a future potential gain of 3. If the count is zero, i.e., $O[s] == 0$, the step $s$ has already been automated. The effort required for reusing an already automated step, e.g., validating it for usage in this particular test case, is considered to be substantially less than automating the test step again. Therefore, the validation effort is neglected. Given the above, we can describe the set of test steps that has to be automated as $S_{\text{A}} := s \in \tau : O[s] > 0$. Each of these steps will each give a potential future gain of $O[s]$ when automated. If all test steps in $\tau$ have already been automated $|S_{\text{A}}|$ will be zero. To avoid division by zero, we use a sufficiently small number here in such cases. A small number will result in a high ranking of test cases that require no effort, making sure that we schedule “effortless” test cases as early as possible.

**Definition 7** The potential/effort (P/E) value for a test case is given by the following:

$$\text{P/E}(\tau) = \frac{\text{gain} + \text{potential}}{\text{effort}} = \frac{|\tau| + \sum\limits_{s \in S_{\text{A}}} O[s]}{\max(|S_{\text{A}}|, 1/\infty)}$$

$$S_{\text{A}} := s \in \tau : O[s] > 0,$$

where $O$ is the step occurrence count map for $T$, TC is the current test case, and $S_{\text{A}}$ is the set of test steps that requires automation effort.

Given the following example:

$$t_0 = (aabcd), O[s] = \{a : 5, b : 0, c : 0, \ldots, d : 0\},$$

we get $S_R = \{a\}$ the gain $t_0 = 5$ and the potential 5. The effort is 1 since only one test step has to be automated. The P/E value is thus 10.

Going back to the ordering of test case problem, Algorithm 1 has three internal structures of interest. $T_c$ is the set of test cases that has not yet been scheduled for automation. $T_o$ is the ordered list of test cases scheduled for automation. $O$ is a map, keeping track of each unique test step and the number of time this particular test step occurs in the original set of test cases $T$. If $O[s]$ is zero, this indicates that the step $s$ already has been included in the test cases of $T_o$.

The algorithm starts with counting the occurrences of each unique test step into a occurrence map $O$. Next, it iterates through all unprocessed cases and calculates the P/E score for each test case, and the test case with the highest P/E value, $t_{best}$, is chosen. This test case is added to the list of automated test cases and also removed from set of unprocessed test cases. Lastly, all steps in the chosen test case are considered to be implemented and marked with a zero in the occurrence map to signal that they have been implemented. The process continues until all test cases have been ordered, i.e., the set of unprocessed test cases is empty.

**Algorithm 1** $PE_RATIO(t, O)$

1: $gain \leftarrow |t|$
2: $potential \leftarrow 0$
3: $effort \leftarrow 0$
4: **for** $step \in set(t)$ **do**
5: $\quad potential \xleftarrow{+} O[step]$
6: $\quad$ **if** $O[step] > 0$ **then**
7: $\quad\quad effort \xleftarrow{+} 1$
8: $\quad$ **end if**
9: **end for**
10: **return** $(gain + potential)/\max(effort, 0.00000001)$

**Algorithm 2** $ORDER(T)$

1: $T_c \leftarrow T$ {Remaining Test cases}
2: $T_o \leftarrow \epsilon$ {Optimal order}
3: $O \leftarrow \epsilon$ {Occurrence map}

4: **for** $step \in T$ **do**
5: $\quad O[step] \xleftarrow{+} 1$
6: **end for**

7: **while** $T_c \neq \epsilon$ **do**
8: $\quad t_{best} = \underset{t \in T_c}{\text{argmax}}\, \mathbf{PE_RATIO(t, O)}$
9: $\quad T_c = T_c - t_{best}$
10: $\quad \texttt{append}(T_o, t_{best})$
11: $\quad$ **for** $step \in set(t_{best})$ **do**
12: $\quad\quad O[step] \leftarrow 0$
13: $\quad$ **end for**
14: **end while**
15: **return** $T_o$

## 4 Industrial case study

We have conducted an exploratory industrial case study, following the guidelines by Runeson and Höst (2009). The objective of the case study is to evaluate to what extent the automaton order is a viable criterion when prioritizing test cases. We evaluate the average expected manual effort, as per Definition 5, using optimized automation orders from a number of algorithms. The result is compared to automating test cases without considering similarity-based reuse, where the order does not matter. The considered algorithms are random order, shortest first, longest first, P/E prioritization, stochastic hill climbing, and a standard genetic algorithm.

In particular, we investigate the following research questions:

- RQ1: What effect does similarity-based reuse have on the test effort for real-world projects, not considering a particular automation order?
- RQ2: How much does the order, in which we automate test cases, affect the test effort?
- RQ3: How do different ordering approaches (random order, shortest first, longest first, P/E prioritization, stochastic hill climbing and a standard genetic algorithm) perform with respect to affecting test effort?

### 4.1 Case study context

The studied case organization is active within the vehicular embedded systems domain. The project development work is distributed among a set of international sites and departments, each responsible for a subset of the functionality that is finally integrated and tested in our case organization. The integration test cases are primarily written in natural language for manual test execution. Whenever the system is considered mature enough by the testing steering group, a subset of the test cases is automated. The reason for waiting with the automation work is that their integration process is performed in a number of incremental (internal) release cycles, where new sub systems may be added in each release cycle. During this process, the system gradually grows but also gets more and more stable. The practitioners express that during the early cycles, faults occur in unexpected places in their system, and that such faults are hard to catch using automated test cases. Thus, the automation work may be postponed until the system has reach a certain level of stability.

### 4.2 Units of analysis

The units of analysis in our case study are the test cases of four different projects (**P1**, **P3**, **P4**, **P5**) at the case organization. The project **P2**, not included in this study, is well known to be very similar to **P3**, and was only used for verifying that the similarity function and the results of the tested approaches were consistent.

The details of the different projects are presented in Table 4. Projects **P1** and **P5** are two large projects, whereas **P3** and **P4** are substantially smaller. The potential automation reduction for each project is given in the column *Reduction potential*. This is the fraction of possibly redundant test steps. With redundant, we mean that if a test step has already been automated, the work of automating it again would be redundant work. The *Distinct Recurrence* column describes the probability of a test step from the group of distinct test steps to occur more than once. Intuitively, this column describes how big fraction of the distinct test steps are repeated more than once in order to sum up to the *Reduction potential* (e.g., 37% in **P1**). The last column (which is an interpretation of Fig. 4) describes the probability of a test

**Table 4** General metrics of the units of analysis

| Project | Test cases | Test steps | Distinct steps | Reduction potential (%) | Distinct recurrence (%) | Short test probability |
|---|---|---|---|---|---|---|
| **P1** | 1327 | 11,665 | 7331 | 37 | 17 | Low |
| **P3** | 285 | 1812 | 1362 | 24 | 11 | High |
| **P4** | 337 | 2365 | 1657 | 29 | 18 | High |
| **P5** | 1684 | 17,372 | 11,485 | 33 | 12 | Low |
| **SUM** | 3919 | 35,011 | 23,180 | 31.5 (Avg) | | |

case to be short (up to five test steps). Even though all projects have a rather high probability, **P1** and **P5** show a lower probability and thus define the LO level. Correspondingly, the HI level is defined by **P3** and **P4** (all in Fig. 4). This probability distribution of test length of the test cases is detailed in Fig. 4 as pair combinations of the four projects. Notable is that there are two groups of projects that are similar with respect to test case length: **P1** and

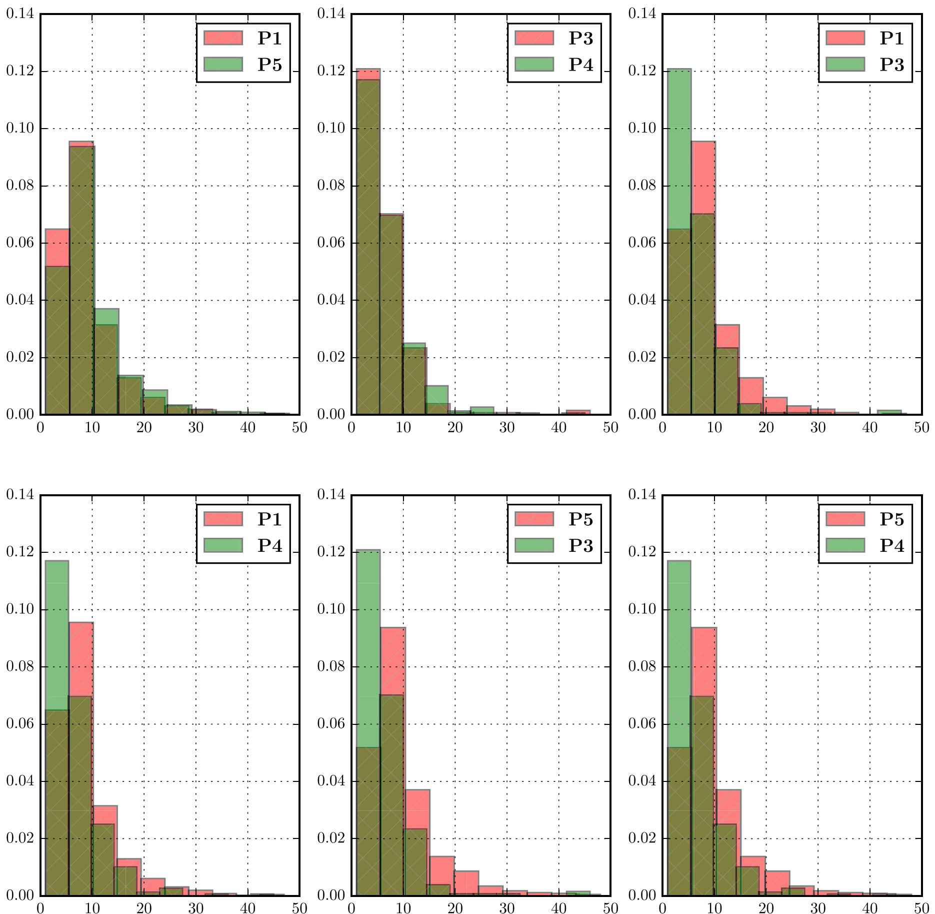



**Fig. 4** Probability distributions with respect to number of test steps. The darker area is where the bars overlap

**P5** have a low fraction of really short test cases, whereas **P3** and **P4** show a larger fraction of really short test cases (less than ten test steps). The distribution of individual step lengths is roughly the same for all the projects.

Another, possibly, important feature is that in **P1** and **P5**, the preparatory[1] steps of the test cases have been more standardized than in **P3** and **P4**.

### 4.3 Variables and measurements

The independent variable recorded in this study is as follows:

– Applied automation strategy.

The following dependent variables are recorded:

– The average expected manual execution effort ($\bar{E}$) as per Definition 5.
– Execution time of the prioritization algorithm (given by the automation strategy). The execution time is measured (in wall-clock seconds) from the point in time the algorithm is given the list of test cases in random order as input until the list of prioritized test cases has been returned.

### 4.4 Assumptions and simplifications

To be consistent, we apply the same assumptions on the industrial evaluation as in Section 3.1.

### 4.5 Experimental settings

This section describes the experimental settings used in the industrial evaluation. Again, with reuse, we always mean similarity-based reuse of automation code.

1. **R0 —*No reuse and random order***
   The purpose of this experiment is to simulate the case where we do not reuse the automation code of similar test steps and we further do not automate the test cases in any particular order. This should yield an $\bar{E}$ of 50% and serves as the base comparison.
2. **R1 —*Reuse with random order***
   The automation code of similar test steps is reused. The automation order is random.
3. **SF —*Reuse and shortest first***
   The automation code of similar test steps is reused. The *shortest* test case is automated first and if two test cases are of the same length, the one that occurred first in the initial order is picked.
4. **LF —*Reuse and longest first***
   The automation code of similar test steps is reused. The *longest* test case is automated first and if two test cases are of the same length, the one that occurred first in the initial order is picked.
5. **P/E —*Reuse and P/E prioritization***
   The automation code of similar test steps is reused. The automation order is decided by the P/E algorithm, as described in Section 3. It orders the test cases by assigning a P/E score to each test case. If two test cases have the same score, the one that occurred first in the initial order is picked.

[1] Preparatory test steps prepare the system under test (SUT) to be tested.

6. **SH —*Reuse and stochastic hill climbing***
   The automation code of similar test steps is reused. The automation order is decided by the stochastic hill climbing algorithm. We use the implementation in the JAMES[2] framework for solving the traveling salesman problem.
   In this work, we have chosen the parameters as follows:

   - Objective function—the function $\text{Fit}(T, \sigma)$ as defined in Definition 6.
   - Time limit—60 s
   - Evaluation limit—250,000,000 to ensure that the time limit is always used.

   These values were chosen by observations of several initial experiments that showed that a longer run time ($> 60$ s) did not significantly improve the result.
7. **GA - *Reuse and genetic algorithm (GA)***
   The automation code of similar test steps is reused. The automation order is decided by a standard genetic algorithm. We use the implementation in the jMetal[3] framework for solving the traveling salesman problem. More specifically, the framework class `GeneticAlgorithmBuilder` together with the class `PermutationProblem` is used. This implementation can only use the `PermutationSwapMutation` and `PMXCrossover` operators. The used selection operator is `BinaryTournament` with the comparison operator `RankingAndCrowdingDistanceComparator`. The other parameters stem from the results of a sensitivity analysis with population = {50, 80, 100}, mutation probability= {0.5, 0.7, 0.9}, and crossover probability = {0.001, 0.008, 0.07, 0.1}. The sensitivity analysis was run with the largest set of test cases ($P1$). The difference between the highest and lowest average (of ten executions of each parameter combination) was $\sim 7\%$. Since an increased population size is expected to contribute to a better result at larger problems (Li et al. 2007), the parameters were kept also for the smaller test case sets ($P2$, $P3$). The detailed set of parameters are as follows:

   - Individual—one permutation of the test case indices.
   - Objective—$\text{Fit}(T, \sigma)$ as defined in Definition 6.
   - Crossover probability—0.7.
   - Mutation probability—0.1.
   - Population size—50 individuals.
   - Max evaluations—25,000.

The input to all algorithms in experiments 1–9 is a set test cases $T$ and an initial order $\sigma_1, ..., \sigma_n$. The goal of each algorithm is to return an order $\sigma'_1, ...\sigma'_n$ with the lowest possible value $\text{Fit}(T, \sigma)$ as defined in Definition 6. The initial order $\sigma_1, ..., \sigma_n$ is always randomized before being fed into the algorithms to prevent any bias from the initial order. The algorithms in the list above were implemented from scratch in Java, where nothing else stated.

### 4.6 Execution and data collection procedures

The data from the DOORS data base was extracted into a sandbox environment, similar to Flemström et al. (2016). After extraction, the major preparation steps included feature

[2]JAMES framework 5.2-SNAPSHOT (Nov 2016). http://www.jamesframework.org/

[3]Metal framework 5.1. http://jmetal.github.io/jMetal/

extraction, removal of stop words as well as formatting characters and lastly encoding into word vectors. Each distinct word vector was assigned an ID and each test case were encoded accordingly. In this paper, however, we have only considered the stimuli part of the test steps, thus ignoring the reaction part (for an example of an encoded test case, see Table 1). Each step in a test case is now represented by a stimuli ID. If the IDs are equal, the steps are considered to be similar.

When prioritizing test cases with the heuristic approaches, a priority value is assigned to each test case. In the shortest first and longest first approaches, this priority value is the number of steps in the test case.

There is always a risk that two or more test cases are assigned the same value (aka ties). In such cases, the first encountered test case is always picked. To mitigate any bias due to such effects, each of the algorithms was run with random start orders 100 times on each unit of analysis (Arcuri and Briand 2014). The meta-heuristic algorithms also suffer from randomness and thus, they were also run 100 times per unit of analysis. The dependent and independent variables were recorded for each experiment.

## 5 Industrial case study results

The data from the studied projects were prepared as described in Section 4.6, resulting in four sets of test cases. In summary, each test case was encoded as a series of numbers, each number representing a distinct test step. If two numbers are the same, we suggest that the automated code for that particular test step can be reused with substantially less effort than automating that step again. Since it will take less time to automate the test cases, the proportion of manual test cases will reduce quicker.

To compare how different automation orders affect this reduction, the $\bar{E}$ measure, introduced in Section 3.3 was used. From a practical point of view, the $\bar{E}$ value allows us to reason about the chance that a randomly chosen test case at a randomly chosen point in time has to be carried out manually. Intuitively, the more of the available test steps have been automated, the less the chance of ending up with manually executing a test case.

Each project was studied in isolation to avoid bias from previously automated projects. For a real setting, it would most likely be beneficial to consider all existing projects as potential sources for reuse.

As recommended by Arcuri and Briand (2014), each experiment was executed 100 times for each project, using a random automation order as start value. The compared algorithms were no reuse with random order (**R0**), random order (**R1**), shortest first (**SF**), longest first (**LF**), P/E prioritization (**P/E**), stochastic hill climbing (**SH**), and genetic algorithm (**GA**).

The results of these executions are presented in Table 5. The table shows four row-groups, collecting the results from the experiments, for each project. The sub-rows in each group are min, max, standard deviation, and mean for $\bar{E}$ followed by the algorithm mean execution time in milliseconds. Since the frameworks we used measure time as wall-clock time, values less than 1 ms is marked with "*". The results for the different algorithms are presented in corresponding column.

From Table 5 we observe that when similar test steps are not reused (**R0**), the manual effort ($\bar{E}$) is steadily around 50% for all the projects. This is expected since, if we do not reuse any steps, *all* test steps are automated *always*. This in turn leads to each test case having the same automation effort. The slight deviation from 50.00% is due to Definition 1 of wholly automated test cases, which gives a staircase-formed curve.

**Table 5** Expected manual effort ($\bar{E}$) and average execution for the experiments

| Project | | **R0** | **R1** | **SF** | **LF** | **P/E** | **SH** | **GA** |
|---|---|---|---|---|---|---|---|---|
| **P1** | Mean | 50.05 | 33.09 | 31.70 | 32.79 | 26.47 | 26.32 | 30.41 |
| | std | 0.00 | 0.24 | 0.01 | 0.01 | 0.00 | 0.01 | 0.12 |
| | Min | 50.05 | 32.55 | 31.68 | 32.77 | 26.47 | 26.30 | 30.14 |
| | Max | 50.05 | 33.68 | 31.72 | 32.82 | 26.48 | 26.37 | 30.66 |
| | Exec | 2 | 2 | 2 | 2 | 399 | 60,041 | 32,608 |
| **P3** | Mean | 50.28 | 39.19 | 39.49 | 37.67 | 33.45 | 33.19 | 34.88 |
| | std | 0.00 | 0.44 | 0.03 | 0.02 | 0.00 | 0.02 | 0.13 |
| | Min | 50.28 | 37.86 | 39.42 | 37.61 | 33.44 | 33.16 | 34.60 |
| | Max | 50.28 | 40.20 | 39.57 | 37.73 | 33.46 | 33.27 | 35.16 |
| | Exec | * | * | * | * | 9 | 63,313 | 5138 |
| **P4** | Mean | 50.32 | 37.26 | 36.82 | 35.38 | 30.47 | 30.29 | 32.50 |
| | std | 0.00 | 0.55 | 0.02 | 0.02 | 0.00 | 0.03 | 0.15 |
| | Min | 50.32 | 35.82 | 36.76 | 35.32 | 30.46 | 30.23 | 32.14 |
| | Max | 50.32 | 38.32 | 36.87 | 35.43 | 30.48 | 30.36 | 32.87 |
| | Exec | * | * | * | * | 15 | 60,812 | 7066 |
| **P5** | Mean | 50.05 | 34.07 | 32.01 | 35.24 | 28.54 | 28.46 | 31.88 |
| | std | 0.00 | 0.21 | 0.00 | 0.00 | 0.00 | 0.01 | 0.10 |
| | Min | 50.05 | 33.54 | 32.00 | 35.22 | 28.53 | 28.44 | 31.70 |
| | Max | 50.05 | 34.54 | 32.02 | 35.25 | 28.54 | 28.49 | 32.12 |
| | Exec | 1 | 2 | 3 | 2 | 840 | 60,039 | 46,954 |
| **Proj. avg.** | Mean | 50.18 | 35.90 | 35.01 | 35.27 | 29.73 | 29.57 | 32.42 |
| | Exec | * | 1 | 1 | 1 | 316 | 61,051 | 22,941 |

*Means execution time less than 1ms

When reusing the automation code of already automated test steps, the automation work should finish quicker. Intuitively, this would reduce the expected manual effort ($\bar{E}$). Table 5 confirms that the $\bar{E}$ value is less for all experiments considering similarity-based reuse (**R1**, **SF**, **LF**, **P/E**, **SH**, **GA**).

> *Answer to RQ1: Our case study results show that when doing similarity-based reuse, the expected manual effort ($\bar{E}$) is less than when similar test steps are not reused. When similar test steps are not reused, the manual effort ($\bar{E}$) is ∼50% for all projects.*

Further, from the same Table 5, it is evident that the stochastic hill climbing yields the best result (returning an automation order with a low $\bar{E}$). By running the algorithms with negated $\text{Fit}(T, \sigma)$, we find that the stochastic hill climbing also yields the worst order of the algorithms (included in Fig. 6 but not in Table 5). The difference between the best $\bar{E}$ result and the worst result with the negated $\text{Fit}(T, \sigma)$ indicates the maximum effect we can expect from the automation order, given the algorithms at hand. For the given projects and algorithms, it was on average 12 percentage points in difference.

*Answer to RQ2: We argue that the difference between the best $\bar{E}$ result and the worst $\bar{E}$ result indicates the maximum effect we can expect from varying the automation order, given the algorithms at hand. Our case study results show that when considering similarity-based reuse, and given the selected set of algorithms, the difference between the best and worst expected effort ($\bar{E}$) was on average 12 percentage points.*

Figure 5 is a graphical representation of Table 5 with to the purpose of facilitating the comparison between the algorithms. The *y*-axis shows the average expected manual effort ($\bar{E}$). The lower this value, the better performance of the algorithm. On the *x*-axis, we find the experiments grouped by project. The vertical dotted lines are there for readability. The green dots for each algorithm and project show the mean value of $\bar{E}$ from 100 executions of the algorithm with random start orderings. The vertical thick lines in Fig. 5 are drawn vertically from the min to the max value of each algorithm. Where the difference between min and max is very small, these vertical lines are not visible. The worst case executions (negated fitness function) of the algorithms are not considered in the figure.

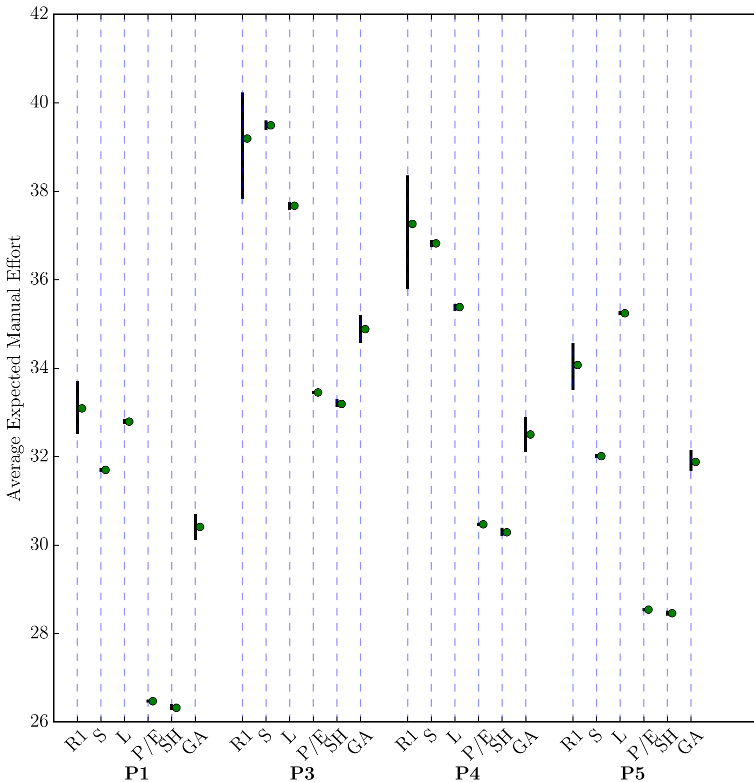



**Fig. 5** Min, max and average of expected effort ($\bar{E}$) of 100 executions of each algorithm with random start orderings. The algorithms are denoted random (**R1**), shortest first (**SF**), longest first (**LF**), P/E prioritization (**P/E**), stochastic hill climbing (**SH**), and genetic algorithm (**GA**)

*Answer to RQ3: For the four projects, the performance of P/E prioritization is well comparable to the stochastic hill climbing algorithm. The Stochastic hill climbing algorithm performed slightly better than the P/E approach (on average* $\sim 0.17$ *percentage points). The other algorithms performed significantly worse. For an online tool implementation, the P/E prioritization is most suitable due to the shorter execution time.*

From the Fig. 5, we see that the stochastic hill climbing (**SH**) always outperforms the others, tightly followed by the P/E prioritization (**P/E**). The genetic algorithm performs significantly worse than the stochastic hill climbing (**GA**) but still better than the remaining algorithms.

Notable in Fig. 5 is that, even though the standard deviation is extremely small for non random algorithms, the algorithms perform slightly different in different projects. To explain the differences in performance with respect to the projects, we collect some features from the projects and the algorithm results in Table 6. The columns represent the projects. The first row "**SF** vs **LF**" shows whether the shortest first or longest first was the best one of the two. Since P/E prioritization (**P/E**) and stochastic hill climbing (**SH**) yielded a slightly better result for two of the projects, these projects were marked with "H" in the rows "**P/E** Performance" and "**SH** performance." Correspondingly, an L on these two rows means that the result was slightly worse for the marked projects. The "Short TCs" row is, for convenience, the same as the "Short test probability" column in Table 4. The "Distinct O" is an interpretation of the "Distinct recurrence" in Table 4. The two highest were assigned the value H, and the two lowest the value L. The last row "Size" is a similar interpretation of the number of test steps in Table 4.

A closer examination of Table 6 and Fig. 4 reveals that **P1** and **P5** have roughly the same proportions of short test cases. **P3** and **P4** also have roughly the same proportions, but have much more short test cases. Since the gain of an automated test case is counted earliest when the whole test case is fully automated, a higher ratio of short test cases will contribute to an earlier effort reduction, which may partly explain why shortest first gives more reduction in **P3** and **P4**.

The P/E prioritization performs slightly better in **P1** and **P4**. Examining Table 6, P/E does not seem to be that sensitive to the proportion of short and long test cases. Further examining the occurrence distribution for the individual test steps, the probability that a test step occur more than one time in project **P1** and **P4** is slightly higher than in project **P3** and **P5**. Since test steps that occur often are prioritized by the approach, a larger gain may be "claimed" early, which possibly contributes to the better performance in **P1** and **P4**. As presented in Table 6, the chance of finding test steps that occur more than twice is

**Table 6** Comparing experiment result with project project features

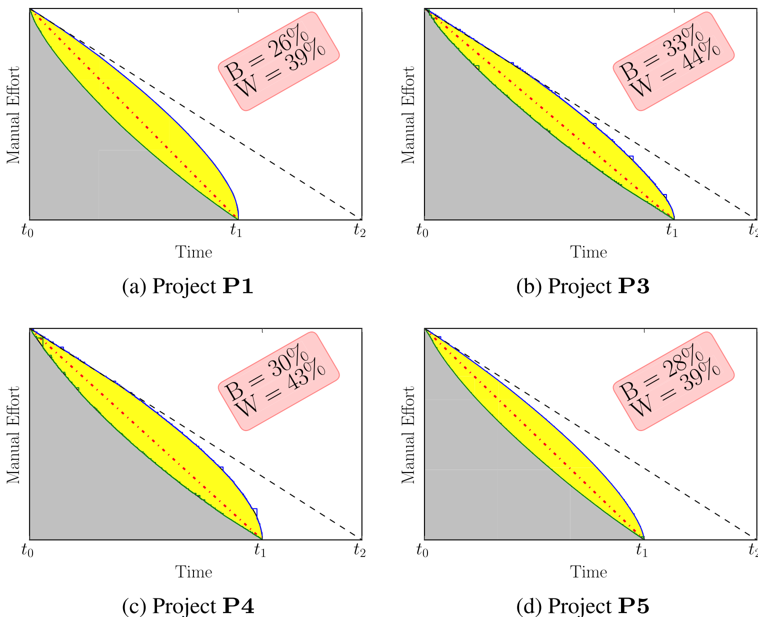
| | **P1** | **P3** | **P4** | **P5** |
|---|---|---|---|---|
| **SF** vs **LF** | S | L | L | S |
| **P/E** performance | H | L | H | L |
| **SH** performance | H | L | H | L |
| Short TCs | L | H | H | L |
| Distinct O | H | L | H | L |
| Size | L | S | S | L |

slightly less in project **P3** and **P5** than in **P1** and **P4**; thus, there is less gain to find for those projects, which should explain the slight decrease in performance for the P/E prioritization. The stochastic hill climbing follows the same pattern and is probably also rewarded by a high portion of recurring test steps. The size of the project seems not to matter since the hill climbing performed well in one small and in one large project.

While the $\bar{E}$ value is useful for comparing one ordering algorithm with another, it does not tell much about how the manual effort declines over time due to the on-going automation work. To better understand how the best and worst automation orders affect this effort reduction over time, we have reported the resulting order from the stochastic hill climbing for the best and worst $\bar{E}$ results in Fig. 6. The figure has four sub-figures, one for each project. On the $y$-axis we find the remaining manual effort (i.e., the not yet automated test cases). On the $x$-axis, we find the time from start of automation work, until the automation work would have finished if we had not considered reuse.

Continuing with Fig. 6, the lower curved line, ending at $t_1$ illustrates the best case with $\bar{E}$ = 26% (for **P1**). The upper curved line, ending at $t_1$ illustrates the worst case with $\bar{E}$ = 39% (for **P1**). Due to Definition 1 of wholly automated test cases, small "spikes" appear on the lines, like a staircase. Due to the relative scale, this is only visible for the small projects **P3** and **P4**. The yellow area between the curved lines shows the span between the best and the worst case. The shadowed area under the best case curve is for visual purposes only. For the sake of comparison, there are two more lines plotted in the figure. The dotted lines ending at $t_2$ illustrate the case where we do not consider reuse. In this case, the corresponding $\bar{E}$ is always approximately 50%. Lastly, the dotted red lines, ending at $t_1$, are added as a visual reference to facilitate the comparison.

From the figure, it is evident that the **SH** algorithm yields a consistent result for the different projects with an average of 29% compared to non reuse that would yield 50%. The




(a) Project **P1** (b) Project **P3**

(c) Project **P4** (d) Project **P5**

**Fig. 6** Remaining manual execution effort over time for **SH** (B) and **SH**+negated Fit$(T, \sigma)$ (W)

difference between the best and worst order is on average 12 percentage points. The slope from the **P/E** algorithm tightly followed the slope from the **SH** algorithm. It was hardly possible to distinguish them from each other. From the slopes in the figure, we also observe that both optimizing the $\bar{E}$ value and **P/E** prioritization give the earliest reduction since any other order would result in a slope above the best case in the figure.

### 5.1 Execution performance of the prioritization algorithms

Finding the most efficient automation order is an important criterion when choosing the algorithm for prioritizing the automation work. However, given that the algorithms should be used in an on-line tool, it is also important to consider the computational efficiency. In this section, we compare the two best algorithms, SH and P/E, and compare their performance with respect to computational efficiency.

The execution time reported in Table 5 shows a relevant value for the P/E algorithm, while the time reported for the hill climbing algorithm is tainted by the timeout. Since the stochastic hill climbing algorithm in reality successively improves its solution, the successive development of the result over time is more appropriate.

To find out how long time it takes for the hill climbing algorithm to reach the same result as the P/E algorithm, we instrumented the stochastic hill climbing algorithm to emit the current best $\bar{E}$ and the time stamp each time a new solution was found. To make sure that we would not get a substantially better value if the stochastic hill climbing was left running for a longer period of time, we doubled the time limit.

Figure 7 describes the successive development of the result for the P/E Algorithm and the stochastic hill climbing algorithm. On the $y$-axis, we see the $\bar{E}$ value that would be the result if the algorithm was stopped at that point in time. The time is shown on the $x$-axis in milliseconds. The horizontal dotted red line shows the result of the P/E prioritization algorithm. Before $t_a$, there is no result available. Since the algorithm is deterministic and only returns one result once it has finished, it will not give better or other results, even if run multiple times. Thus, the line is vertical. The black lines show the successive development for ten executions of the stochastic hill climbing on project **P1**. The algorithm performs rather consistently and reaches the same $\bar{E}$ value as P/E at $t_b$, at approximately 1 min. Notably, the P/E algorithm is magnitudes faster than the hill climbing, which is an advantage if used in an on-line tool.

## 6 Threats to validity

There are several threats to the validity of the industrial evaluation. One threat to the construct validity of the method is that we have simplified the effort estimation of the individual test steps. This could be a threat if the test steps differ substantially in length or the similarities do not stem from copy-paste operations. Given more machine interpretable details on the required effort for *(i)* automating and *(ii)* executing the test steps, we would have got more accurate results. Concerning the internal validity, it is obvious that the resulting effort reduction is dependent on the skills of the testers and the quality of the underlying test automation framework. To mitigate this, we have used all test cases we could access. This is a threat itself to the external validly, as in a real-world situation, only a subset of the manual test cases are subject to automation. However, we prioritized the internal validity over the external validity in this case. Also even though we studied industrial projects that are real-world and real-sized, the results are only generalizable to situations with

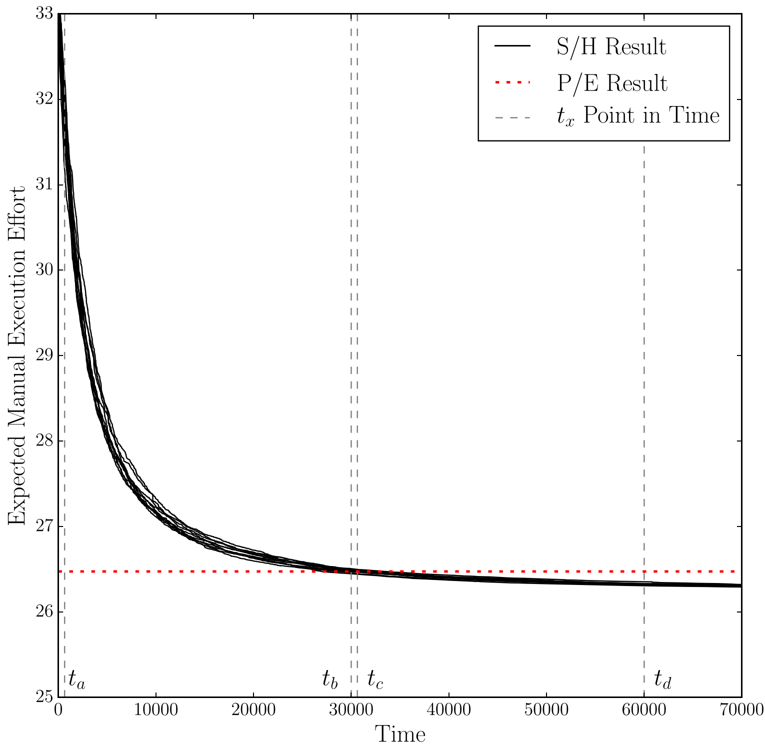



**Fig. 7** Solution development over time for the stochastic hill climbing together with P/E prioritization result. The scale of the $y$-axis is % of initial manual effort and the time scale on the $x$-axis is milliseconds

similar characteristics with respect to similarities and test step lengths (referred to as analytical generalization (Runeson and Höst 2009)).

The resulting effort reduction is, of course, dependent on both which test cases were picked out for automation, as well as the number and distribution of system tests during the period. Other threats would be if the test steps differ substantially in length or the similarities do not stem from copy-paste operations. In case there is substantial variability in the length of the test steps, one would like to know how bad the different curves would get. Given the assumption that the length of the test steps is an indirect indication of test complexity, another possibility is to help the test automation engineer by some visualization technique once a variation in length is detected.

## 7 Conclusions and future work

When automating a set of test cases $T$ without considering reuse, the order does not matter to the average manual test execution effort. However, if it is possible to reuse the automation code for similar test steps, order *has* an impact. Avoiding automating the set of test cases in a totally random order decreases the average manual effort required if the system is tested any time before the work of automating $T$ is finished. Any of the studied approaches is better than the random.

This paper presents a fast, simple, and industrially scalable approach to *prioritize* the test cases to be automated with the purpose of getting a big and early reduction of remaining manual test effort until all test cases have been automated. Evidently, the suggested P/E prioritization approach is comparable to the more standard stochastic hill climbing. The simplicity of the P/E approach makes it a good candidate for integration in the existing industrial testing frameworks. Another major advantage, in many industrial settings, is that our approach does not require any manual input for the prioritization or the similarity tracking. Only the available manual steps (instructions) in the selected set of test cases are required, together with the set of already automated test cases. These can be retrieved automatically from the test case repositories. Since the approach is modular, the set of already automated test cases may include all test cases in all available projects in order to increase the level of possible reuse.

The technology transfer to companies take place in different steps (Gorschek et al. 2006). The results of this study should be seen as one candidate solution. Once our company commits to automate all of their manual test cases, a dynamic validation of the results provided in this paper would yield more tangible benefits in future.

In this work, we have only considered the automation effort and future effort reduction potential based on the number of steps in the test cases. The cost function can be further developed to take into account the length of the test steps and possibly the complexity. Even other criteria can be considered such as the code coverage and the ability of test cases to reveal faults. These can then become a basis for different prioritization methods.

We also believe that there are other factors, equal or even more important, to fulfill when automating test cases. Examples of such information are, as mentioned before, functional dependencies, subsystem maturity, and level of importance. We argue that there is a need to investigate how the known factors that can be automatically retrieved can contribute to a more accurate model. Ultimately, we would like to know whether such model additions improve the prioritization or not.

In summary, ordering of test cases for automation has shown to be one interesting piece in the test effort reduction puzzle.

**Funding information** This work was supported by the Swedish Innovation Agency (VINNOVA) through grant 2014-03397 (IMPRINT) and the Knowledge Foundation (KKS) through the projects 20130085 (TOCSYC) and 20160139 (TESTMINE).

**Open Access** This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

## References

Alspaugh, S., Walcott, K.R., Belanich, M., Kapfhammer G.M., Soffa, M.L. (2007). Efficient time-aware prioritization with knapsack solvers. In *Proceedings of the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies: Held in Conjunction with the 22Nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007, ACM, WEASELTech '07,* (pp. 13-18).

Amannejad, Y., Garousi, V., Irving, R., Sahaf, Z. (2014). A search-based approach for cost-effective software test automation decision support and an industrial case study. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops (ICSTW), IEEE* (pp. 302–311).

Anand, S., Burke, E. K., Chen, T. Y., Clark, J. A., Cohen, M. B., Grieskamp, W., Harman, M., Harrold, M. J., McMinn, P. (2013). An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, *86*(8), 1978–2001.

Arcuri, A., & Briand, L. (2014). A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. Software Testing. *Verification and Reliability*, *24*(3), 219–250.

Asaithambi, S.P.R., & Jarzabek, S. (2013). Towards test case reuse: a study of redundancies in android platform test libraries. In *International Conference on Software Reuse, Springer,* (pp. 49–64).

Barr, E. T., Harman, M., McMinn, P., Shahbaz, M., Yoo, S. (2015). The oracle problem in software testing: a survey. *IEEE Trans Software Eng*, *41*(5), 507–525.

Basit, H., & Jarzabek, S. (2009). A data mining approach for detecting higher-level clones in software. *IEEE Transactions on Software Engineering*, *35*(4), 497–514.

Do, H., & Rothermel, G. (2006). An empirical study of regression testing techniques incorporating context and lifetime factors and improved cost-benefit models. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2006, Portland, Oregon, USA, November 5-11, 2006, ACM,* (pp. 141–151).

Do, H., Mirarab, S., Tahvildari, L., Rothermel, G. (2010). The effects of time constraints on test case prioritization: a series of controlled experiments. *IEEE Trans Software Eng*, *36*(5), 593–617.

Engström, E., & Runeson, P. (2013). Test overlay in an emerging software product line—an industrial case study. *Information and Software Technology*, *55*(3), 581–594.

Epitropakis, M.G., Yoo, S., Harman, M., Burke, E.K. (2015). Empirical evaluation of Pareto efficient multi-objective regression test case prioritisation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015, ACM,* (pp. 234–245).

Feldt, R., Poulding, S. M., Clark, D., Yoo, S. (2015). Test set diameter: quantifying the diversity of sets of test cases. arXiv:1506.03482.

Flemström, D., Sundmark, D., Afzal, W. (2015). Vertical test reuse for embedded systems: A systematic mapping study. In *Proceedings of the 41st Euromicro Conference on Software Engineering and Advanced Applications (SEAA'15).*

Flemström, D., Afzal, W., Sundmark, D. (2016). Exploring test overlap in system integration: An industrial case study. In *Proceedings of the 42nd Euromicro Conference on Software Engineering and Advanced Applications (SEAA'16).*

Garousi, V., & Pfahl, D. (2015). When to automate software testing: a decision-support approach based on process simulation. Journal of Software: Evolution and Process.

Gorschek, T., Garre, P., Larsson, S., Wohlin, C. (2006). A model for technology transfer in practice. *IEEE software*, *23*(6), 88–95.

Greiler, M., van Deursen, A., Zaidman, A. (2012a). Measuring test case similarity to support test suite understanding. In *Proceedings of the 50th International Conference on Objects, Models, Components, Patterns (TOOLS'12), Springer-Verlag.*

Greiler, M., van Deursen, A., Zaidman, A. (2012b). Measuring test case similarity to support test suite understanding. In *Proceedings of the 50th International Conference on Objects, Models, Components, Patterns, TOOLS'12,* (pp. 91–107).

Hemmati, H., Arcuri, A., Briand, L. (2013). Achieving scalable model-based testing through test case diversity. *ACM Transactions on Software Engineering Methodology*, *22*(1), 6:1–6:42.

Li, Z., Harman, M., Hierons, R. M. (2007). Search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering*, *33*(4), 225–237.

Marijan, D. (2015). Multi-perspective regression test prioritization for time-constrained environments. In *2015 IEEE International Conference on Software Quality, Reliability and Security, QRS 2015, Vancouver, BC, Canada, August 3-5, 2015, IEEE,* (pp. 157–162).

Noor, T.B., & Hemmati, H. (2015). A similarity-based approach for test case prioritization using historical failure data. In *26th IEEE International Symposium on Software Reliability Engineering, ISSRE 2015, IEEE Computer Society,* (pp. 58–68).

Ramler, R., & Wolfmaier, K. (2006). Economic perspectives in test automation: balancing automated and manual testing with opportunity cost. In *Proceedings of the 2006 international workshop on Automation of software test, ACM,* (pp. 85–91).

Rothermel, G., Untch, R. H., Chu, C., Harrold, M. J. (2001). Prioritizing test cases for regression testing. *IEEE Trans Software Eng*, *27*(10), 929–948.

Roy, C. K., Zibran, M. F., Koschke, R. (2014). The vision of software clone management: past, present, and future (Keynote paper). In *Softw Maintenance, Reengineering Reverse Eng (CSMR-WCRE), 2014 Softw Evol Week - IEEE Conf* (pp. 18–33). https://doi.org/10.1109/CSMR-WCRE.2014.6747168.

Runeson, P., & Höst, M. (2009). Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, *14*(2), 131–164.

Shi, Q., Chen, Z., Fang, C., Feng, Y., Xu, B. (2016). Measuring the diversity of a test set with distance entropy. *IEEE Transactions on Reliability*, *65*(1), 19–27.

Strandberg, P.E., Sundmark, D., Afzal, W., Ostrand, T.J., Weyuker, E.J. (2016). Experience report: automated system level regression test prioritization using multiple factors. In *27th IEEE International Symposium on Software Reliability Engineering (ISSRE'16)*.

Tahvili, S., Bohlin, M., Saadatmand, M., Larsson, S., Afzal, W., Sundmark, D. (2016). *Cost-benefit analysis of using dependency knowledge at integration testing*. Berlin: Springer International Publishing.

Thomas, S. W., Hemmati, H., Hassan, A. E., Blostein, D. (2014). Static test case prioritization using topic models. *Empirical Software Engineering*, *19*(1), 182–212.

Walcott, K.R., Soffa, M.L., Kapfhammer, G.M., Roos, R.S. (2006). Timeaware test suite prioritization. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine, USA, July 17-20, 2006, ACM,* (pp. 1–12).

Wang, R., Jiang, S., Chen, D. (2015). Similarity-based regression test case prioritization. In *The 27th International Conference on Software Engineering and Knowledge Engineering, SEKE 2015, KSI Research Inc. and Knowledge Systems Institute Graduate School,* (pp. 358–363).

Wang, S., Ali, S., Yue, T., Bakkeli, Ø., Liaaen, M. (2016). Enhancing test case prioritization in an industrial setting with resource awareness and multi-objective search. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume, ACM,* (pp. 182–191).

Yoo, S., & Harman, M. (2012). Regression testing minimization, selection and prioritization: a survey. Softw Test. *Verif Reliab*, *22*(2), 67–120.

Zhang, L., Ho, S.S., Guo, C., Xie, T., Mei, H. (2009). Time-aware test-case prioritization using integer linear programming. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ACM, ISSTA '09,* (pp. 213–224).

**Daniel Flemström** is a researcher at the SICS and PhD student at the Mälardalen University. Before his employment at the SICS, Daniel has been the project manager for, and participated in, many industrial research projects at different levels at the Mälardalen University.

During 2005–2014, he has also built a secure industrial research lab together with Ericsson AB and ABB, enabling industrial application of academic results within an academic environment. Daniel has given several courses in 800xA usage and maintenance as well as authored several programming courses for the 800xA core system for the ABB University.

**Pasqualina Potena** is a Senior Researcher with RISE SICS Västerås AB, Sweden. She received the graduation degree in computer science from the University of L'Aquila and the Ph.D. degree in sciences from the University "G. D'Annunzio" Chieti e Pescara, Italy. She was a research fellow with the University of L'Aquila, Politecnico di Milano, and University of Bergamo, Italy. She works closely with large scale and SMEs companies – e.g., she was an EU Marie Curie Industry-Academia Partnerships and Pathways Experienced Researcher. Her research interests include optimal decision frameworks and software quality (reliability, availability, performance, cost, ...), software testing of large scale industrial software systems, self-adaptive systems with uncertainties, and Search Based Software Engineering (SBSE).

**Daniel Sundmark** is a professor of computer science at the Mälardalen University. His research interests include software testing, empirical software engineering, and software and systems engineering. Sundmark received a PhD in computer science from the Mälardalen University. Contact him at daniel.sundmark@mdh.se.

**Wasif Afzal** is a senior lecturer at the Mälardalen University, with the Software Testing Laboratory. His research interests include software testing, empirical software engineering, and decision-support tools for software verification and validation. Afzal received a PhD in software engineering from the Blekinge Institute of Technology. Contact him at wasif.afzal@mdh.se.

**Markus Bohlin** is the research leader in industrial efficiency and automation at RISE SICS Västerås AB. He defended his doctoral thesis in 2009 and was appointed as associate professor (docent) in 2013, both at the Mälardalen University. In 2014, he was appointed as adjunct professor in analysis of rail traffic systems at the KTH Royal Institute of Technology. Between 2012 and 2016, he served as president of the Swedish Operations Research Association. He has acted as program chair and main organizer for several conferences and workshops, including the bi-annual Swedish Operations Research Conference (twice) and the annual Industrial Efficiency conference (four times). He serve as reviewer and PC member for several scientific journals and conferences. His work has been presented in more than 50 peer-reviewed papers in international journals, conferences, and workshops. He serve on several national and international research boards and councils. He has extensive experience from research leadership, having planned and led more than 20 individual research projects involving up to 20 people, and currently supervises five Ph.D. students.