SN Computer Science (2022) 3:373
https://doi.org/10.1007/s42979-022-01255-1

ORIGINAL RESEARCH

A Learning Classifier System for Automated Test Case Prioritization and Selection

Lukas Rosenbauer[1] · David Pätzel[2] · Anthony Stein[3] · Jörg Hähner[2]

Received: 30 July 2021 / Accepted: 20 June 2022
© The Author(s), under exclusive licence to Springer Nature Singapore Pte Ltd 2022, corrected publication 2022

Abstract
For many everyday devices, each newly released model contains more functionality. This technological advance relies heavily on software solutions of increasing complexity which results in novel challenges in the domain of software testing. Most prominently, while an ever higher number of test cases is required to meet quality demands, performing a large number of test cases frequently amounts to a significant increase in development time and costs. In order to overcome this issue, agile development methods such as continuous integration usually only execute a subset of important test cases to meet both time and testing demands. One way of selecting such a subset of important test cases is to assign priorities to all the available test cases and then greedily pick the ones with the highest priority until the available time budget is spent. For this, in a previous work, we presented a new machine learning approach based on a learning classifier system (LCS). In the present article, we summarize our earlier findings (which are spread over several publications) and provide insights about the most recent adaptations we made to the method. We also provide an extended experimental analysis that outlines more in detail how it compares to a state of the art artificial neural network. It can be observed that the performance of our LCS-based approach is often much higher than the one of the network. Since our work has already been deployed by a major company, we give an overview of the resulting product as well as several of its in-production quality attributes.

Keywords Evolutionary machine learning · Software testing · Continuous integration · XCSF classifier system

Introduction

In many areas of everyday life, newly developed products rely on more software-based solutions than ever before. The increasing amount of underlying program code that needs to be continually extended and maintained results more often in complex software projects. For these, a key challenge is integrating the code written by individual engineers into the overall project [7]; in order to prevent problems arising from merging large, diverged code bases, the common practice of continuous integration (CI) has been developed [36]. CI means to much more frequently ("continuously"—e.g., daily) combine source code of individual programmers into the central code base.

A CI process usually consists of more or less four basic steps: downloading the source code (the proposed integration), building the software and then testing and deploying it. The sequence of these steps is called the project's (CI) pipeline each run of which is referred to as one CI cycle. A schematic of this basic pipeline can be seen in Fig. 1. CI pipelines are usually run on dedicated build servers by software such as Jenkins [31].

Software testing is a vital part of most CI pipelines; it has a major economic impact on a project's lifetime and cost [8, 9, 41]. One of the central goals of CI is speed, i. e., getting insights into the current state of the software quickly in order

This article is part of the topical collection "Computational Intelligence" guest edited by Kurosh Madani, Kevin Warwick, Juan Julian Merelo, Thomas Bäck and Anna Kononova.

✉ Lukas Rosenbauer
lukas.rosenbauer@bshg.com

David Pätzel
david.paetzel@uni-a.de

Anthony Stein
anthony.stein@uni-hohenheim.de

Jörg Hähner
joerg.haehner@uni-a.de

1 BSH Hausgeräte GmbH, Regensburg, Germany

2 University of Augsburg, Augsburg, Germany

3 University of Hohenheim, Stuttgart, Germany

Published online: 15 July 2022

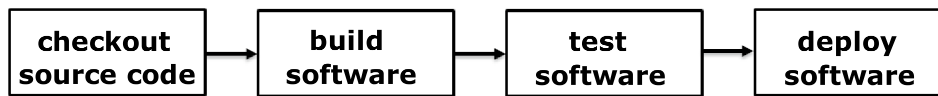


Fig. 1 Example of a CI pipeline (adapted from [29])

to be able to react accordingly. However, as software projects get larger, as does the number of test cases and executing all test cases often becomes infeasible within CI time scales (e. g., new code may be merged in on a daily basis but running all test cases may require more than 24 h). One solution to this problem is to only execute a subset of test cases; ideally, the most relevant or crucial ones [7].

A recent approach to determine test case importance, and based on that select a subset of test cases, is to employ machine learning (ML). This is grounded on the observation that the testing history, that is, test case running durations and their likelihood to fail, can be exploited to prioritize test cases: Short test cases which have a high probability to fail can be considered to be important whilst long test cases which simply pass do not need to be run as urgently. One of the first successful implementations of such an ML approach has been presented by Spieker et al. [32] who employed an artificial neural network (NN). The quality of results of their approach has motivated companies such as Netflix to integrate the methodology in practice [12].

We were the first to use a learning classifier system (LCS) for this task [28]. LCSs are a powerful family of evolutionary ML algorithms which have found their way into various applications such as smart cameras [35] or traffic control [21]. These successes motivated us to examine LCSs on this use case as well. One major milestone was a solution based on the XCSF classifier system (XCSF) which enabled us to use real-valued test case priorities [29]. We were able to show experimentally that our XCSF-based approach is not only comparable to the NN-based one of Spieker et al. [32] but actually superior in many cases.

In a series of publications, we improved the performance of our approach step by step [25, 28, 30]. This work summarizes our earlier findings as well as provide insights about the most recent adaptations we made to the method. Further, our work provides the following new contributions:

– We give a deeper and extended analysis of the developed method. While we only conducted a temporal analysis in earlier work, we now also employ a distributional analysis based on percentiles—this delivers more smooth and human-interpretable results and can be seen as an orthogonal extension.
– Our previous work focused on the question if LCSs are better in terms of performance. Here, we also introduce a percentile-based quality measure to show by how much LCSs are better.
– In order to further widen our analysis we compare our methods against a non-ML and non-evolutionary approach: a pure random selection.
– Our LCS variant relies on a wide-spread ML mechanism: experience replay (ER). While our earlier work [25, 30] merely followed the ER approach of Spieker et al. [32], we now extend our experiments and add a critical analysis of whether other ER variants perform better.
– Finally, we move beyond the experiments traditionally conducted. We think that, in order to have a practical impact, algorithmic approaches like ours need to be delivered to industry. We provide insight into how we did that for BSH Hausgeräte GmbH which is Europe's biggest producer of home appliances, outlining that our approach is easy to use and reporting on the first real-world results our solution achieved.

We continue this work with a formal description of the problem we consider ("Problem description") followed by a discussion of related work ("Related work"). After that, we explain how the problem can be approached using ML ("Machine learning approach") and introduce the LCS we employ ("Employed Learning Classifier System) as well as its parametrization ("Parametrization"). In "Evaluation", we show and evaluate the results of the experiments we conducted. This is followed by a brief overview of the full-stack solution that we deployed for BSH Hausgeräte GmbH ("An evolutionary computation application"). The article ends with an overview of future work and a conclusion.

Problem Description

We assume that, in every CI cycle i, there is a fixed, planned time budget C available for testing and that this time budget does not suffice to run all available test cases. We write $\mathcal{T}_i$ for the set of all test cases available in CI cycle i. The estimated execution time of test case T is $d(T)$. The goal is to assign to each test case T an optimal priority in form of a rank $\text{rk}_i(T)$. These ranks are not necessarily unique (i. e., two or more test cases may be assigned the same rank) and are computed at the start of every CI cycle. The computed ranking is meant to be used to perform test case selection, that is, to compile a test suite greedily whilst considering the time budget C as follows (thereby we follow Spieker et al. [32]):

1. The test cases at hand are sorted according to their ranks (in descending order).

2. Test cases are taken repeatedly from the head of that sorted list and added to the current CI cycle's test suite $\mathcal{TS}_i$ which is initially empty. We stop this process when adding the next test case in the sorted list to the test suite would lead to the sum of the execution times of test cases in the test suite exceeding the time budget C.
3. If there are several test cases with the same rank at the head of the sorted list but there is not enough time budget left to add all of them to the test suite, test cases from that rank are repeatedly chosen uniformly at random until the time budget is expended.

The resulting test suite $\mathcal{TS}_i$ can be represented as a list; we write $l_i(T)$ for the index of test case T in that list. Note that indexing starts from 1.

After having built a test suite as described, the test cases it contains are executed during the testing phase of the CI pipeline and their results retrieved and evaluated. We denote by $\mathcal{TS}_i^{\mathrm{f}}$ the subset of the test suite $\mathcal{TS}_i$ that contains all the failed test cases. In contrast, we write $\mathcal{T}_i^{\mathrm{f}}$ for the set of failed test cases if all available test cases would have been executed. Obviously, $\mathcal{TS}_i^{\mathrm{f}} \subset \mathcal{T}_i^{\mathrm{f}}$.

Based on $\mathcal{TS}_i^{\mathrm{f}}$ and $\mathcal{T}_i^{\mathrm{f}}$ we could measure the percentage of failures found p_i as follows:

$$p_i = \frac{|\mathcal{TS}_i^{\mathrm{f}}|}{|\mathcal{T}_i^{\mathrm{f}}|} \tag{1}$$

while p_i may be used to measure the quality of the compiled test suite, it cannot be used to assess the quality of the chosen prioritization as it leaves out the test cases' ordering. A commonly-used metric for evaluating the quality of test suites generated based on test case prioritizations is the normalized average percentage of faults detected (NAPFD) [22][1]:

$$\mathrm{NAPFD}(\mathcal{TS}_i) = \begin{cases} 1, & |\mathcal{T}_i^{\mathrm{f}}| = 0 \\ p_i - \frac{\sum_{T \in \mathcal{TS}_i^{\mathrm{f}}} l_i(T)}{|\mathcal{T}_i^{\mathrm{f}}| \cdot |\mathcal{T}_i|} + \frac{p_i}{2|\mathcal{T}_i|}, & \text{otherwise} \end{cases} \tag{2}$$

This metric's values lie in [0, 1] with high values being desired: Prioritizations that rank many short, failing test cases as important receive high scores whereas ones that recommend the execution of long, passing test cases are rated low. NAPFD is further equal to one for the edge case that all available tests would be passing if executed (this means that no subset of the available tests can identify a fault).

A brief look at NAPFD's definition in (13) reveals that knowledge of p_i and, in turn, $\mathcal{T}_i^{\mathrm{f}}$ is necessary in order to compute it. This means, that while NAPFD captures the quality of prioritizations well, it cannot be used directly in practice as the very problem to be solved by test selection is to avoid running all test cases. However, NAPFD can nevertheless be used for evaluating approaches to computing prioritizations since these usually are performed on simulations where that information is available to the experiment's observer. This is also the case for the experiments that we conducted and whose results are presented in "Evaluation".

At this point, we are able to concisely define the problem that priority-based test case selection approaches try to solve. It is termed the adaptive test case selection problem (ATCS) and is defined as follows [32]:

> Based on the set of available test cases $\mathcal{T}_i$, use prioritizations to build a test suite $\mathcal{TS}_i$ whose execution time is not greater than C while still maximizing NAPFD.

This can be written formally like so:

$$\begin{aligned} &\max \mathrm{NAPFD}(\mathcal{TS}_i) \\ &\text{subject to} \quad \sum_{T \in \mathcal{TS}_i} d(T) \le C \\ &\qquad\qquad \mathcal{TS}_i \subseteq \mathcal{T}_i \end{aligned} \tag{3}$$

Related Work

The general task of generating a test suite, that is, selecting a subset of the available test cases, is commonly known as test selection. This section gives an overview of the different existing approaches to doing that. Further we mention several other evolutionary approaches to software testing problems.

Test Selection in General

The survey by Yoo and Harman [40] documents that there are various forms that methods for test selection can take on: In general, test selection methods can be divided into whether they reduce the number of available tests, whether they build the test suite directly or—as is the case for the method we propose in the present article—compute first a test case prioritization (i. e., assign a priority to each test case) to then choose test cases accordingly. An example of a method that permanently reduces the number of available tests is specification-based filtering whereas direct test case selection methods may for example be based on data flow analysis. Test selection methods can be divided further into white box and black box methods, the first taking into account the source code of the software being tested while the latter does not. Since the method we analyse solely relies

[1] Note that, despite its name, NAPFD actually measures test case failures and not faults. Faults are system errors caused by bugs etc. and each fault may result in multiple test cases failing.

on CI metadata such as previous test outcomes and runtimes, it classifies as a black box method.

There is currently no consensus on what exactly makes a certain test case important. Existing research focuses on coverage criteria, failure revealing capabilities or use case–specific objectives which can also be considered in combination by employing multi-objective optimization algorithms. For example, Lachmann et al. [14] and Arrieta et al. [2] identify six and five such criteria, respectively. It is worth mentioning that some of these criteria require a certain maintenance effort (e.g., identification or update of business relevance).

Evolutionary Computing and Software Testing

Employing evolutionary techniques to testing problems such as test selection is an active field of research. For example, the aforementioned works of Lachmann et al. [14] and Arrieta et al. [2] employ a non-dominated sorting genetic algorithm II (NSGA-II) to optimize test suites. Both show that the test suites created outperform random selection. However, computing test suites of decent quality using their approach is too computationally expensive to be feasible for the CI use case (keep in mind CI is all about speed and getting insights fast). We determined this in one of our previous studies where we proposed artificial immune systems (AIS) for multi-objective test selection use cases where the test suite computation time is not critical [24, 27][2]. Note that our approach does not share this problem; LCSs have a more limited computation time and have been used in real-time critical applications [37] such as traffic control [17] in the past.

We further want to mention a few other applications of evolutionary computation (EC) to testing problems. The survey of Anand et al. [1] gives an overview of test case generation techniques based on EC. Such methodologies usually involve a search heuristic which tries to construct test cases which cover some abstract code representation such as the control graph. It is also worth mentioning that sometimes programming language specifics must be taken into account such as for Python [16] or Java [1].

Another approach to test case generation is mutation testing where the focus lies on identifying weak tests as well as paths in the software that are not covered by tests [18]. In order to do so, changes or bugs (mutations) are introduced to the software automatedly and an evaluation of whether existing tests are capable of identifying them is performed. Papadakis et al. [18] give an overview of research into EC-based approaches to this. Overall, generating a small, permanent test suite of crucial test cases via mutation testing may be seen as a valid alternative to test case selection. However, it requires knowledge of the underlying code base (and is thus a white-box approach) whereas the present work considers black box methods. We decided to use a black box testing method as the test levels our industrial partner is interested is are usually black box ones (such as system level testing).

Another field of testing application for EC is random testing which focuses on generating optimally distributed test inputs [10]. At that, an input distribution's fitness is based on its capabilities to reveal failures. One drawback of this approach is that detailed knowledge of the possible inputs (domains) of the available test cases is required which may lead to a considerable maintenance effort if the set of available tests is large and heterogeneous; since this is exactly the case for the test cases of our industrial partner (i. e., heterogeneous tests whose domains have not been documented with enough scrutiny in the past), a solution based on random testing could be ruled out early.

Machine Learning Approaches to Test Case Selection

The black box approach of Spieker et al. [32] for test selection has already been mentioned in the introduction. It is rather minimal in terms of data requirements as it only requires CI metadata that usually gets recorded anyway (e. g. by CI software such as Jenkins [31]). Spieker et al.'s system tries to find a test suite of short test cases which are failing which is very close to Dijkstra's well-known testing objective which states that the goal of testing is to reveal bugs and not show their absence [4]. While our approach follows this objective as well and has the same requirements towards the testing environment, our system differs from Spieker et al.'s in that it employs an LCS instead of an NN.

Our approach makes use of the XCSF classifier system which was originally introduced by Wilson [38] as an extension of the earlier XCS classifier system (XCS) [39] to the domain of function approximation. The system we built also takes advantage of ER, a well-studied mechanism for NNs [5] that, quite recently, saw inclusion into LCSs. Stein et al. [33] performed a first study on combining XCS with ER showing that, under certain circumstances, ER is beneficial for XCS. In another study, we underlined this for XCSF and the task of test case selection [30]. We also introduced and evaluated a simplistic form of transfer learning for our method [25].

Since our ultimate aim is a full-stack solution for test case selection in CI environments, we also designed a system architecture which encapsulates the LCS-based method presented herein [26]. At that, we employ techniques and ideas from the systems engineering discipline Organic Computing [17].

[2] In those publications we report on the AIS approach performing equal to or better than the NSGA-II of Lachmann et al. [14]. The NSGA-II used by Arrieta et al. [2] differs from the one used by Lachmann et al. only in the employed crossover operator.

Machine Learning Approach

Many ML problems can be seen as a form of function approximation. This is also the case for our approach: We try to find an optimal mapping $\mathrm{V}(\cdot)$ from test cases to their respective value (in a sense similar to the reinforcement learning semantics of a state's or an action's value). Given such a mapping, we follow a simple heuristic to assignment priorities to test cases: a test case valued highly should have a high priority, we thus set the priority $\pi(T)$ of test case T to the estimated value of that test case $\mathrm{V}(T)$.

The domain of $V(\cdot)$ is the test case state space S whereas the codomain is $\mathbb{R}$; test case values (and thus priorities) may thus be continuous. In the following we describe both S and several possible choices for $V(\cdot)$.

In the ML model for ATCS of Spieker et al. [32][3], each test case T corresponds to one state s. That state is a vector comprising the test case's

- Relative execution time which is expressed relative to the overall execution time of all available test cases (and thus is a number between 0 and 1).
- Testing history, a snapshot of the test case's result in its k last executions. This is represented as a binary vector of length k (0 indicating failed, 1 indicating passed) which, if there have not yet been k executions, we pad to length k by filling it with zeros.
- Last execution time which is the normalized index of the most recent CI cycle during which the test case was run (normalized by dividing by the overall number of CI cycles that occurred so far).

Thus the state space S can be defined formally as follows:

$$S := [0, C] \times \{0, 1\}^k \times [0, 1] \tag{4}$$

and has dimensionality $k + 2$.

In order to ease the understanding of the state space we give an explicit example of a state vector. Assume there are test cases whose execution times sum up to 1000 minutes and that there have already been 100 CI cycles. Suppose we have a history length of $k = 4$ and a test case T at hand that lasts 10 minutes. T's relative execution time as denoted in the state vector is thus $\frac{10}{1000} = 0.01$. Assume that the test case failed the last two times it was executed and before that it passed; this results in a history component of [0, 0, 1, 1]. Further suppose the last time it was run was in CI cycle 90, its last execution value is thus $1 - \frac{90}{100} = 0.1$. In summary, the state corresponding to T is $s = [0.01, 0, 0, 1, 1, 0.1]$.

[3] Spieker et al. [32] originally formulated the use case as a reinforcement learning one. We intend to provide a more high-level machine learning view.

Finally we have to define the already mentioned value function V. A feasible thought would be trying to use the NAPFD metric. However, as stated earlier, NAPFD requires knowledge of the outcomes of all the available test cases which renders it unfeasible to compute in practice—and thus cannot be used as the value function either. Spieker et al. instead propose to use one of the following three (approximate) value functions as drop-in replacements for NAPFD [32]:

- Failure count

$$\mathrm{V}_i^{\mathrm{fc}}(T) = |\mathcal{TS}_i^{\mathrm{f}}| \tag{5}$$

- Test failure

$$\mathrm{V}_i^{\mathrm{tcf}}(T) = \begin{cases} 1 - v_i(T), & T \in \mathcal{TS}_i \\ 0, & \text{otherwise} \end{cases} \tag{6}$$

- Time ranked

$$\mathrm{V}_i^{\mathrm{trk}}(T) = |\mathcal{TS}_i^{\mathrm{f}}| - v_i(T) \cdot \sum_{\substack{T' \in \mathcal{TS}_i^{\mathrm{f}}, \\ \mathrm{rk}(T) < \mathrm{rk}(T')}} 1, \tag{7}$$

where $v_i(T)$ is the binary verdict of test case T for CI cycle i. A 0 indicates that the test case failed whereas a 1 means either that it passed or that there was not enough time to execute it.

The failure count function is rather coarse-grained as the same value (the number of failed test cases found) is assigned to all test cases encountered during a CI cycle. The test failure function, on the other hand, is more fine-grained as it values test cases individually based on their outcome. Finally, the time ranked function takes it another step further as it also considers the prioritization itself by punishing high-ranked passing test cases using the number of lower-ranked failing test cases in the test suite currently considered.

Note that, since the software being tested as well as the available test cases may change between two CI cycles, each of the value functions defines a non-stationary learning task. We deem it to be non-stationary due to the fact that, between two cycles, the software to be tested or the available tests may change. Thus the learning environment can be vastly different.

Employed Learning Classifier System

LCSs belong to the field of evolutionary ML algorithms. While there are many different LCS algorithms for different domains, the most-used and -investigated LCS so far is the XCS classifier system [37, 39]. In this work we use one of its derivatives, the XCSF classifier system, or simply XCSF, which is an extension of XCS to the problem of function

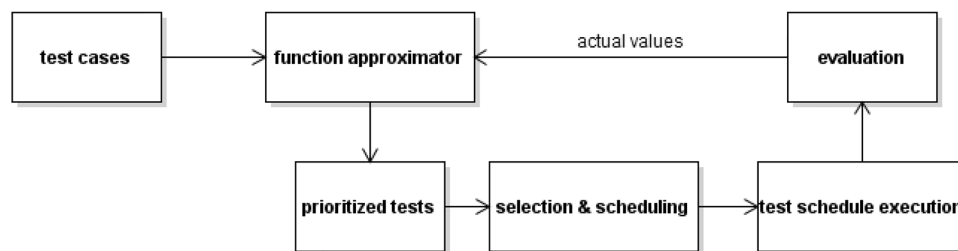


Fig. 2 Workflow for solving ATCS using ML

approximation [38]. In a previous publication it has already been shown that XCSF outperforms XCS on the ATCS [29].

Before we jump into the details of LCS we intend to give an overview of its role in the testing workflow (Fig. 2). The process begins by retrieving the test cases at hand and computing the corresponding states. This information is given to the function approximator (in our case an LCS and for Spieker et al. [32] an NN) which in turn computes the prioritization. Thereafter the test cases are selected accordingly and executed. Based on the outcomes, the actual value $V(T)$ of each test T that was run is retrieved and given to the approximator which uses that data to refine the value function estimate.

We further want to point out that this workflow implicitly introduces a time axis for the test cases as these are given in a certain ordering to the approximator. We denote this time as t and the corresponding state and priority as s_t and π_t but only in case of ambiguity to keep the notation uncluttered. Our later experiments reuse the source code of Spieker et al. [32] and sort the test cases by their numerical ID. Further, it is worth stressing that this time is not resetted from CI cycle to CI cycle but instead further incremented.

Overview of XCSF

We loosely follow Wilson [38] and Urbanowicz and Browne [37] for the introduction of XCSF.

XCSF evolves a population P of rules which are historically called classifiers (a rule is denoted in the following formal expressions by cl). The two main components of a rule are

- A condition that is essentially a predicate $S \rightarrow \mathbb{B}$ (i. e., a function from the input space to the Boolean domain ($\{true, false\}$) and
- A local model (i. e., a function $\mathrm{cl}.p : S \rightarrow \mathbb{R}$) which serves as a local function approximation.

Essentially, a rule cl can be seen as a partial function that is only defined on $\{s \in S \mid \text{cl's condition matchess}\}$. In addition to conditions and local models, rules contain bookkeeping parameters which play a role when performing learning updates or making decisions. Examples are how often a rule has been applied or how accurate it is (in XCSF, the latter is known as the rule's fitness, $\mathrm{cl}.F$). XCSF has a rule limit N which is periodically checked and if the population grows too big then classifiers are removed. We call this mechanism pruning.

The individual rules are fit to the inputs for which their conditions evaluate to true (i. e., the inputs that the rules match) by a locally acting learning mechanism that trains each rule independently of the other rules. In addition to that, a genetic algorithm (GA) optimizes the rules' conditions, that is, the (in general, non-disjoint) segmentation of the input space described by the set of the conditions of all rules in the population.

For the rules' local models $\mathrm{cl}.p : S \rightarrow \mathbb{R}$, we follow the original XCSF formulation [38] which uses linear models:

$$\mathrm{cl}.p(s) = w_0 + \sum_{i=1}^{k+2} w_i \cdot s_i \tag{8}$$

where the w_i are real-valued weights which are initialized randomly, $k + 2$ denotes the state space's dimensionality as before and the additional weight w_0 is used to fit the intercept.

Given an input s, XCSF first computes the set M of matching rules (called match set). If M holds too few classifiers then new ones are constructed randomly and added to M (at that, new rules created at this stage always match s); this process is called covering. Since rules may overlap, M usually contains more than one rule which means that, next, multiple rules have to be mixed in order to get an overall system prediction. This is done by performing a fitness-weighted sum of the matching rules' predictions:

$$P(s) = \frac{\sum_{\mathrm{cl} \in M} \mathrm{cl}.p(s) \cdot \mathrm{cl}.F}{\sum_{\mathrm{cl} \in M} \mathrm{cl}.F} \tag{9}$$

At that, $\mathrm{cl}.F$ is the aforementioned fitness value of rule cl, that is, a heuristic approximation of its accuracy.

The classifier keeps track of several other magnitudes except fitness which are worth mentioning:

- $\mathrm{cl}.exp$: Its experience which counts the number of times the rule has been used.
- $\mathrm{cl}.\epsilon$: estimates the error of the predictions made.

- cl.t: the last time the classifier interacted with the GA.
- cl.ms: an estimation of the size of match sets that use cl.

Learning Mechanism

XCSF is fitted to training data in a supervised but iterative manner. This means that for each example (s, v) (i. e. state, value), the prediction functions of all classifiers matching s (i. e., all classifiers in the match set M computed for s) are updated using a gradient descent–based approach which is known as the modified delta rule:

$$\Delta w_i = \frac{\eta}{\|\tilde{s}\|^2}(v - \text{cl}.p(s))\tilde{s}_i \tag{10}$$

where $\|\cdot\|$ is the Euclidean norm, η is a learning rate and $\tilde{s}$ is the state vector s augmented with an additional entry containing 1 at the front:

$$\tilde{s} = \begin{bmatrix} 1 \\ s \end{bmatrix} \tag{11}$$

This augmentation of s fulfills the following two purposes:

- The intercept (the additional weight w_0 of cl.$p(\cdot)$) can be fitted.
- The numerical stability of the modified delta rule is ensured as, this way, the Euclidean norm is always at least 1. The norm of s itself may be very small (consider a frequently used, short test that always fails) which could result in arithmetic overflows.

The weights of cl.$p(\cdot)$ are updated for each training example by simply adding the corresponding Δw_i to their current value.

Whenever XCSF updates the weights it also refines other classifier attributes such as the match set size and the prediction error. This also takes the match set corresponding to the state along with the value used for the gradient descent into account. This is done by adding a fraction of the deviation of the actual value. The fraction depends on the experience of the classifier encountered. For inexperienced rules this fraction is usually bigger and for experienced smaller. This is controlled by the variable β in Algorithm 1. Note that the experience is also incremented in this procedure.

Algorithm 1: Learning mechanism of a single rule based on [4].

```
   input: classifier cl, match set M, value r, state s
1  cl.exp = cl.exp + 1
2  update weights of cl.p(·)
3  if cl.exp < 1/β then
4  |  cl.ε = cl.ε + (|r − cl.p(s)| − cl.ε) / cl.exp
5  |  cl.ms = cl.ms + (|M| − cl.ms) / cl.exp
6  else
7  |  cl.ε = cl.ε + β(|r − cl.p(s)| − cl.ε)
8  |  cl.ms = cl.ms + β(|M| − cl.ms)
9  end
```

Algorithm 2: Fitness update based on [4].

```
    input: match set M
1   accuracy_sum = 0
2   initialize accuracy dictionary κ
3   for cl in M do
4   |  if cl.ε < ε0 then
5   |  |  κ(cl) = 1
6   |  else
7   |  |  κ(cl) = α(cl.ε/ε0)^(−ν)
8   |  end
9   |  accuracy_sum = accuracy_sum + κ(cl)
10  end
11  for cl in M do
12  |  cl.F = cl.F + β(κ(cl) / accuracy_sum − cl.F)
13  end
```

After the match set's classifiers have been adjusted using Algorithm 1 the fitness is recomputed. Here, fitness is an accuracy-based metric which takes the classifiers' prediction error and an error level ϵ_0 into account. First an accuracy value is computed for each rule. If cl.ϵ is below the error level then the maximum accuracy of 1 is assigned (these rules are considered to be accurate). If not, a fraction α of the error ratio $\frac{\mathrm{cl}.\epsilon}{\epsilon_0}$ exponated by $-\nu$ is used. Note that $\nu > 0$ and thus the accuracy is smaller than 1. The fitness is updated by the classifier's normalized accuracy in a similar fashion as the metrics in the previous algorithm. We summarized this methodology in Algorithm 2.

Classifier Generation

In Sect. 5.1 we focused on a rather high-level description and only briefly mentioned the two mechanisms XCSF uses to create new classifiers: covering and GA. Here we give a more precise insight about covering and how the GA we chose works.

The GA optimizes the set of rule conditions whereas covering generates entirely new rules (and thus, conditions). In order to be able to describe their functioning we now first describe how rule conditions are encoded. As was seen in Sect. 2, the state space S (which is the input space to our XCSF-based regression) is a mix of binary and continuous components. We thus chose to represent the classifiers' conditions as disjunctions of interval (for the continuous parts, i. e. everything but the state's testing history; see, e. g., [38]) and ternary subconditions (for the binary parts, i. e. the state's testing history; see, e. g., [39]). An interval subcondition evaluates to true if and only if the given value lies within the subcondition's interval. A ternary subcondition may take on one of three possible values: 0, 1 and #. At that, # is a don't care symbol (true for either of 0 and 1), whereas 0 and 1 are only true if the corresponding element of the state is 0 or 1, respectively (i. e. they only allow an exact match). A rule condition thus consists of two interval subconditions and k ternary subconditions.

Covering creates a new classifier cl for a given state s_t. It assigns the cl an experience of zero and t for cl.t. Further, ϵ_I and F_I are assigned as default values for cl.ϵ, cl.F. Novel ternary subconditions are created by either taking the corresponding value from s_t or by setting it to # with a probability of $P_\#$. For example if the last test result of s_t was failed ($= 0$), then we set the first ternary subcondition to either 0 or #. For new interval subconditions we simply create a random interval that contains the corresponding state value. Finally, the new classifier's prediction function's weights are drawn uniformly at random from a predefined interval.

GAs are iterative, population-based metaheuristics that, in each iteration, typically perform the following computational steps:

- *Selection*: This operation selects two classifiers from the population. These are called parents.
- *Crossover*: This operator combines the two parents to create two new classifiers (the so called offspring). Note that we use the crossover operator only with a probability χ (Similar to the well-known XCS classifier system [39]).
- *Mutation*: Here the offspring might be randomly changed in order to perform exploration.

It is important to note that XCSF's GA does not work on the entire classifier population, but only on the current match set M and state s_t.

Our GA employs a fitness-proportionate selection which means that the probability of any one classifier being chosen as a parent is directly proportional to that classifier's fitness (also known as roulette-wheel selection).

We use two different crossover operators. For the interval-based subconditions we use an arithmetic crossover. Suppose $[l_1, u_1)$ and $[l_2, u_2)$ are the subconditions of the two parents referring to the last execution time. The first offspring's lower bound is $\zeta l_1 + (1-\zeta)l_2$, the second offspring's lower bound is $(1-\zeta)l_1 + \zeta l_2$. Analogously for the upper bounds of the offspring. Note that $\zeta \in (0, 1)$.

The testing histories (ternary strings of length k) are recombined using two-point crossover with crossover points c_1, c_2 chosen uniformly at random from $\{1, 2, \ldots, k\}$. Suppose $c_1 < c_2$. The first offspring receives the first $c_1 - 1$ and the last $k - c_2 - 1$ ternary subconditions from the first parent. The middle ternary subconditions (c_1 to $c_2 - 1$) are taken from the second parent. For the second offspring the roles of the parents are reversed.

We further use a creep mutation: We iterate over all subconditions and mutate each independently at random with a fixed probability of μ. If a ternary conditions is to be mutated, we either change it to # if it had previously not been a # or we use the concrete value of the current state s_t. in order to mutate an interval condition we simply assign a new random interval that fits the corresponding value of the current state s.

Our GA further sets the offspring's fitness and error values to a fraction of the respective mean values of the parents. This is meant to model the uncertainty attributed to the new combinations and mutations not having been tried out yet (unlike their parents).

For XCSF [38] the GA can further be used to optimize the prediction function cl.$p(\cdot)$ which we do at the same time as optimizing the conditions (i. e., we work on the same offspring, here). We employ the same crossover and mutation operators as for the conditions[4].

When to run XCSF's GA depends on the match set's size as well as how long ago the match set's classifiers took part in the GA the last time. Therefore each classifier tracks the time of its last participation in the GA as cl.t (already mentioned earlier). The execution condition can be formulated as follows:

$$t - \frac{\sum_{\text{cl} \in M} \text{cl}.t}{|M|} > \theta_{GA}, \tag{12}$$

whereas θ_{GA} is a hyperparameter. In the literature [38, 39] this condition is checked every time a match set is computed.

Population Pruning

As mentioned earlier, XCSF's population has a fixed capacity N which is periodically checked for being exceeded and if it is, classifiers are removed. The selection of the classifiers to be deleted is roulette wheel–based similar to the GA. However, it is not using fitness as a metric, but the so-called deletion vote of the classifiers which is computed as follows:

$$\text{Deletion vote(cl)} = \begin{cases} \text{cl}.ms \frac{\sum_{\text{cl} \in P} \text{cl}.F}{\text{cl}.F |P|}, & \text{cl}.exp > \theta_{del} \wedge cl.F < \delta \frac{\sum_{\text{cl} \in M} \text{cl}.F}{|P|} \\ \text{cl}.ms, & \text{otherwise}, \end{cases} \tag{13}$$

where θ_{del} is a deletion threshold and δ is a real number controlling the influence of the average population fitness.

A closer look at the formula reveals that experienced classifiers with a relatively low fitness receive a much higher vote as cl.ms is multiplied by a number bigger than one (the factor is the average population fitness divided by the classifier's fitness). This makes it much more likely for an ill-performing rule to be deleted.

The match set size is also included. If a rule is often a part of large match sets then it is more likely to be removed. This introduces an evolutionary pressure towards an even distribution of the rules across the state space.

Extending XCSF with a Replay Memory

While, in a previous study [29], we used the information from all states and values encountered during one CI cycle to update the rules' parameters, we later found out that experience replay (ER) is better suited [30]. ER is a technique widely used for NNs, especially for deep Q-learning [5], which just recently got into the focus of LCS research: Stein et al. [33] performed a first study on its effects on LCS performance and outlined that for use cases such as ours the technique is beneficial in terms of performance. Our implementation of ER saves past experiences of the form (s, v) (i. e. state, value) in a first in, first out (FIFO) buffer B of fixed capacity; whenever the model's parameters are to be updated, a batch of training examples is sampled from this buffer.

Algorithmically, the main functionality of our XCSF-ER is split into two parts: Choosing a priority and remembering the state in the buffer (see Algorithm 3) as well as an update procedure (Algorithm 4). During the update, the observed states are linked with their respective observed values; the resulting pairs of the form (s, v) are then inserted into the buffer. To perform the actual ER-based update, we draw a sample of a fixed size from the buffer. We then compute for each pair (s, v) in this sample the match set M for s and run updates on the local models of all the classifiers in M (i. e., update both quality and bookkeeping parameters [38]). It is worth mentioning that we also run the GA there (Lines 9 and 10 of Algorithm 4).

It is also worth mentioning that the ER-based update is not performed after every CI cycle in order to avoid overfitting (note the condition in Line 4 of Algorithm 4).

Algorithm 3: Priority computation of XCSF-ER

input : state s_t
output: priority π_t
1 M = matching(population, s_t)
2 $\pi_t = \hat{\text{V}}(\text{s}) = \frac{\sum_{\text{cl} \in M} \text{cl}.p(s_t) \cdot \text{cl}.F}{\sum_{\text{cl} \in M} \text{cl}.F}$
3 states.append(s_t)
4 return π_t

[4] Of course they are adapted to numbers. Mutating a number translates to drawing a new random number. Crossover consists of first performing an arithmetic crossover (for two numbers x, y, this corresponds to $\zeta x + (1 - \zeta)y$ and $\zeta y + (1 - \zeta)x$) and then the same two-point crossover as used for the ternary subconditions.

Algorithm 4: ER and buffer update for XCSF-ER

```
   input: values of CI cycle, encountered_states
1  for j = 1 to length(values) do
2  |  insert (states[j], values[j]) into B
3  end
4  if ER should run then
5  |  draw batch b from B
6  |  for (s, v) in b do
7  |  |  match_set = matching(population, s)
8  |  |  update match set using v
9  |  |  if GA should run then
10 |  |  |  run one GA iteration on match_set
11 |  end
12 states = {}
13 prune population
```

Our algorithmic description yet lacks the vital information of how we draw the random batch from the buffer. For this mechanism we examine three different options:

- *Uniform*: All data points have the same probability to be drawn:

$$\frac{1}{|B|} \tag{14}$$

- *Buffer index prioritized*: The probability to choose the data point at position j in the buffer is

$$\frac{j}{\sum_{i=1}^{|B|} i}. \tag{15}$$

- *CI cycle prioritized*: The probability to draw a data point injected within CI cycle i is

$$\frac{i - \gamma + 1}{\mathcal{CI}_i \sum_{l=1}^{\Gamma-\gamma} l}, \tag{16}$$

  where γ is the smallest CI cycle which still has samples in the buffer and Γ the latest[5]. Further, $\mathcal{CI}_i$ denotes the number of data points in the buffer that were collected during CI cycle i.

We selected uniform ER as it was one of the first versions described by Lin [15] and we see it as a default option. The other strategies both use a form of sampling prioritization. The first (buffer index priorization) was used by Spieker et al. [32] for their ML approach towards ATCS. Empirically it showed good results which motivated us to consider it as well. However, it treats the samples of one CI cycle differently as the first test case of a cycle receives a lower probability then the last of the same cycle. Note that the ordering of samples in the buffer is determined by the way the test cases are passed to the ML algorithm (consider our discussion at the start of Chapter 5) which might be problematic as it could introduce some form of bias. The other prioritized ER version that we consider is a refinement of the buffer index prioritization. It still prefers to draw samples from newer CI cycles, but treats the samples of one cycle equally.

Overall, we coin the resulting XCSF adaptation *XCSF-ER*.

Reusing Classifier Populations

Due to the use of XCSF, it is possible to reuse an already-trained model (i. e., a classifier population) for novel software projects by using a straightforward population transformation. This is a form of transfer learning.

First of all we intend to keep the prediction functions as they encapsulate how the local prediction is computed. Thus we also keep the rules' conditions as they also hold the information when we should use the local prediction functions. In 5.4 we underlined that the match set size holds the information on how the classifiers are distributed among the state space which is an important information we intend to conserve.

We further mentioned in 5.4 the importance of fitness values as they have an impact on which classifiers are kept and which are removed. However, fitness is an accuracy-based metric which was computed for the old software project (for our use case) and not the novel one. The same is valid for the classifier's prediction error cl.ϵ which is the basis to compute the aforementioned accuracy. Additionally the old rules are unexperienced for the novel problems. Hence we do not want

[5] Note that the normalization by γ and Γ is necessary to ensure that the result is indeed a probability distribution.

to reuse those values as they might not be suitable for the new project.

The transformation we devised simply resets the aforementioned magnitudes to the default values (as used during covering) whilst the match set sizes, prediction functions, and conditions are kept as they are.

This mechanism follows a simple notion: a reused classifier that performs well for a new project regains high quality values. On the other hand a badly performing one is deleted earlier. As we additionally keep the match set sizes the pruning mechanism can combine them with the new fitness values. Thereby the subspaces lacking well-performing rules and locations containing a too high number of good rules can be identified faster. This further supports in the decision where to sort out rules in order to achieve a decent performance throughout the entire state space.

Further, we set the hyperparameter β that determines the update fraction to 0.1. The last LCS parameter is the deletion threshold θ_{del} which equals 20.

The only parameter purely related to ATCS is the history length k which we set to 6, a value that we determined in hyperparameter studies in our previous works [29, 30].

XCSF-ER shares the aforementioned configurations with XCSF. Hence it only differs by the fact that it uses ER and XCSF does not. Thereby a ceteris paribus environment is created and we can later observe its effects in an isolated way. The used ER buffer has a maximum capacity of 12, 000 and we draw batches of size 2000. Further, updates using ER are performed every third CI cycle.

Spieker et al. [32] employ a rather small NN using one hidden layer containing 12 nodes and the RELU activation function. They use the well-known ADAM optimizer to

Algorithm 5: Classifier conversion for transfer learning.

```
input  : classifier cl
output: transformed classifier
1 cl.experience = 0
2 cl.t = 1
3 cl.fitness = F_I
4 cl.ε = ε_I
5 return cl
```

We summarized the transformation of a single classifier (which simply is applied to all classifiers in the population) in Algorithm 5 and refer interested readers to [25] for more details. We reuse the entire population.

Parametrization

We adopted the hyperparameter and evolutionary operators from our previous works [29, 30]. Here we describe their concrete values starting with XCSF. For its GA we set the arithmetic crossover parameter ζ to 0.6. The crossover probability χ equals 0.75 and the mutation probability μ is 0.015. Its execution threshold θ_{GA} is 25.

Overall the classifier population has a maximum capacity of $N = 2000$. We draw new prediction function weights from $[-10, 10]$ and use the learning rate $\eta = 0.1$ to train them. When we create new covering classifiers we use the initial fitness $F_I = 0$ and error $\epsilon = 0$ as default values. Further, whilst creating new ternary conditions we switch them to # with a probability $P_{\#} = 0.33$ (also during covering).

For the accuracy computation used for updating the fitness values we employ $\alpha = 0.15$, $\nu = 5$, and $\epsilon_0 = 0.01$.

train the weights with a learning rate of 0.05. They employ a history length k of 4 and use buffer index prioritized ER with a buffer size of 10, 000 and a batch size of 1000. ER is performed every fifth CI cycle and after the first.

Our initial version of XCSF-ER [30] had the same buffer mechanism as the NN of Spieker et al. [32] (buffer index prioritized). For the majority of our experiments we rely on that approach. However, in a dedicated experiment we benchmark said approach against others due to the possible issues of the buffer index prioritization (as discussed in Sect. 5.5).

To make our study repeatable and increase reproducibility, we publish our implementation as open source[6].

Evaluation

We evaluate our approach using three open source data sets[7] that were collected during development of three industrial software projects (two by a Scandinavian robot company

[6] Our code is available here: https://github.com/LagLukas/transfer_learning.

[7] The data sets can be downloaded here: https://bitbucket.org/HelgeS/atcs-data/src/master/.

Table 1 Summary of the three examined data sets (paint control, IOF/ROL, GSDTSR). Note that the number of test executions is not a multiple of the number of test cases as the latter increased over each project's lifetime

	Paint control	IOF/ROL	GSDTSR
Company	ABB	ABB	Google
CI cycles	312	320	336
Test cases	114	2 086	5 555
Test executions	25 594	30 319	1 260 617
Failed (%)	19.36	28.43	0.25

called ABB and one by Google). Each of the data sets contains the outcomes of all available test cases for every CI cycle. This enables us to simulate the use case (i. e., selecting and executing only a subset of test cases) but still evaluate the performance of the ML methods considered using the NAPFD metric (which requires all test results to be computed). An overview of the data sets' sizes and additional statistics is given in Table 1.

We compare our approaches (both XCSF without ER [29] and XCSF-ER) with the NN-based solution of Spieker et al. [32]. The latter can be seen as the current state of the art not only with respect to using reinforcement learning for ATCS but also more in general, as it has been shown to be superior to a Q-learning-based approach as well as several traditional techniques such as a greedy selection [32]. In addition, as mentioned before, it has been adopted by major software companies such as Netflix [12].

The data sets we chose for performing the evaluation are exactly the ones used by Spieker et al. [32] which makes it possible to have a fair comparison with our approach. We further rely on Spieker et al.'s original implementation of their NN which is available open source[8] and, instead of relying on the results reported in their paper, repeated all the necessary experiments.

We perform 30 independent and identically distributed runs of each experiment. We examine the following five different issues:

Temporal behaviour Here we concentrate on the evolution of the performance over time with the intention to find out which approach is better.

Performance distribution We switch from a temporal to a percentile-based axis, trading the temporal information for deeper insights into the NAPFD distributions.

Buffer mechanism In our original work [30] we merely followed the ER approach of Spieker et al. [32]. In an additional deeper analysis we consider the two other ER mechanisms introduced in Sect. 5.5 and compare them with each other.

Transfer learning An evaluation of the benefits of our transfer learning method for XCSF-ER.

Random selection We verify our methodology against this approach since we explicitly wanted to benchmark against an algorithm outside of the EC and ML world. Thus we widen our view by comparing against a standard method.

All visualizations have been created using *matplotlib* [11].

Temporal Behaviour

The averaged (over the 30 runs performed), smoothed[9] temporal behaviour of the examined ML methods is shown in Fig. 3. In each row, the used value function is kept fixed, whereas each column deals with one of the three data sets. It is worth mentioning that there is still a high total variation in the averaged NAPFD values due to the nature of software development (new bugs are introduced and old ones fixed).

For the paint control data set (first column), we can observe that only the test case failure value function is beneficial for the NN; for the other two value functions, the performance declines over time. We see something similar for XCSF with the time ranked value function. Visually, both LCSs seem to be comparable with the NN as long as appropriate value functions are chosen.

The trend lines for the IOF/ROL data set (second column) all have a very similar slope (very small or visually close to zero). For the LCSs, the intercepts seems to be higher and keep outperforming the NN over time. However, we cannot identify major differences between the two LCSs solely based on the graphs.

The most striking differences can be seen for the GSDTSR experiments (last column). There is a considerable performance gap between the LCSs and the NN. The usage of ER can further correct bad behaviour, for example, for the time rank value function (XCSF's performance with respect to NAPFD declines whilst XCSF-ER is able to uphold values). Furthermore, the best performance on GSDTSR can be achieved if XCSF-ER is combined with either the failure count or the time ranked value function.

The previous observations were purely made on visual insights and trend lines. This bears some risks as follows: For one, trend lines can be disturbed by outliers. Secondly, instead of the raw results, we only examined averages which can be similarly misleading. In order to more critically

[8] Spieker et al.'s implementation of their NN-based approach can be found here https://bitbucket.org/HelgeS/retecs.

[9] We take the average over three succeeding values. We consider disjoint CI cycle sets with indexes $\{3k, 3k+1, 3k+2\}$.

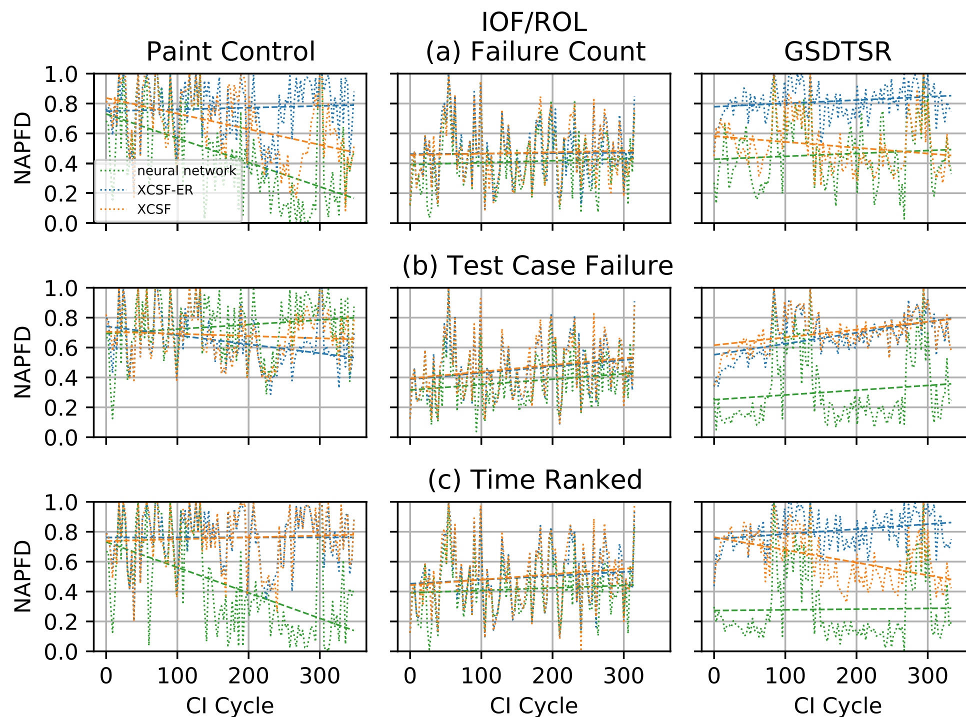


Fig. 3 Time series and trend lines of the averaged (30 runs) and smoothed[8] NAPFDs of the considered approaches. This is an improved version of the graph shown in [30]

evaluate our observations, we thus perform a series of statistical tests. In particular, we want to be able to make a recommendation as to which value function is best suited for usage with XCSF-ER. Based on Fig. 3, we can deduce that the failure count and time ranked value are the two more interesting options. Thus we now statistically compare the performance of XCSF-ER when using these with the two other approaches (XCSF and NN); at that, we combine the latter with any of the value functions. In doing so, we rely on a one-sided Student's t tests with a significance level of 0.05. Student's t tests have the precondition of normally distributed data; we examined that property for each test conducted using Shapiro–Wilk tests.

We show the p values of the one-sided Student's t tests in Table 2. A definite improvement of XCSF-ER over XCSF can be seen for GSDTSR as the p values are basically zero. A similar observation can be made for two out of three cases on the paint control data set. On the IOF/ROL data set, however, we cannot determine which approach is better. When comparing XCSF-ER with the NN, the p values indicate that XCSF-ER is superior in all but one case. Note, however, that a non-significant test does not imply that the NN performs better. In general, the statistical evaluation is in line with the visual one performed above. Based on that, we can make a recommendation for the combination of ML method and value function as folows: XCSF-ER with either the time ranked or failure count value function (which is something that we cannot do for XCSF and the NN).

Performance Distribution

While the previous analysis established that XCSF-ER is often better than the other two approaches, it did not measure by how much it is better. This is hard to examine based solely on the temporal data because of the high variation in the observed NAPFD values, which is due to the aforementioned non-stationary nature of the problem (software and test cases change frequently between CI cycles). We thus continue with analyzing the distributions of the set of NAPFD values achieved at any time in any of the 30 runs we performed. At that, we rely on empirical quantiles (the percentiles) which express cut points dividing the (sorted) observations into intervals with certain relative frequencies. For example, the 50th percentile indicates the lowest observed value that is greater than 50% of the observed values. In our case, high percentile values are desired as this translates to many CI cycles where a good NAPFD score has been achieved. Instead of only investigating selected quantiles, we will plot the overall NAPFD distribution and with that visualize all quantiles.

Before diving into the explorative data analysis, we briefly want to outline why we do not use a more sophisticated visualization technique such as box plots. A box

Table 2 p values for H_0: "Method M_A with value function $V_A(\cdot)$ performs better on data set d than method M_B with value function $V_B(\cdot)$"

			M_B	XCSF-ER	XCSF-ER
			$V_B(\cdot)$	Time ranked	Failure count
M_A	$V_A(\cdot)$	d			
XCSF	Failure count	Paint control		**0.00681**	**0.00269**
		IOF/ROL		0.15045	0.52413
		GSDTSR		**7.31e–58**	**6.35e–63**
	Test failure	Paint control		**1.99e–05**	**5.224e-06**
		IOF/ROL		0.08591	0.40465
		GSDTSR		**5.41e–15**	**2.81e-18**
	Time ranked	Paint control		0.47241	0.34184
		IOF/ROL		0.55662	0.88968
		GSDTSR		**1.11e–29**	**1.56e–33**
NN	Failure count	Paint control		**1.47e–33**	**4.19e–34**
		IOF/ROL		**0.00072**	**0.01779**
		GSDTSR		**1.41e–66**	**5.68e–71**
	Test failure	Paint control		0.2248	0.14179
		IOF/ROL		**7.73e–07**	**0.0001**
		GSDTSR		**1.60e–100**	**2.71e–105**
	Time ranked	Paint control		**1.62e–34**	**2.35e–35**
		IOF/ROL		**0.00121**	**0.02691**
		GSDTSR		**1.21e–106**	**1.75e–111**

Significant entries in bold

plot displays the median, quartiles as well as the so-called whiskers (the quartiles ± the interquartile range). Sample points beyond the whiskers are seen as outliers and visualized as that. However, due to the limited NAPFD range and the non-stationary nature of the problem (which leads to a high interquartile range), there are no outliers in the box plot sense. Our visualization is thus more rich than corresponding box plots since all the information found in a box plot of our data can be easily accessed (e. g., quartiles are the 25th, 50th and 75th percentile which are readily available) while at the same time we display the complete distribution instead of only isolated points of the distribution.

Figure 4 shows plots of the NAPFD percentiles of the different combinations of ML methodology, value functions (rows) and data sets (columns). Each point on one of the graphs relates a percentage of measured NAPFD values (X axis) to the NAPFD value that is greater than all of them (Y axis). Most of the graphs show plateaus at the left and right side which we will call the 0- and 1-plateau respectively based on the NAPFD values they are located at. Having a longer 1-plateau is desirable as it corresponds to the function approximator being able to hit a NAPFD value of 1 more often, which is the best possible value for this metric. 0-plateaus, on the other hand, should be as short as possible as they indicate the number of times that the approach completely failed at its task (the minimum NAPFD value is 0). Overall, a first look at the percentile-based perspective already establishes that the visual evaluation of these is easier than of the temporal plots as the plots are much smoother.

The distribution plots reveal that the workflow used in a software project has an impact on an ML algorithm's NAPFD scores. For example all three methods have a harder time on the IOF/ROL data set than on the GSDTSR data set. A key difference between these projects is the rate at which testing was performed. Upon inspection of the data sets, it can be seen that the IOF/ROL project did not perform CI cycles daily whereas the GSDTSR project sometimes even performed several CI cycles within one day. Thus, the differences between temporally close CI cycles are smaller for the GSDTSR software which can be seen as an indicator of why the ML approaches perform better. Furthermore, the LCS percentiles are consistently higher in the range of the 20th to the 80th percentile which corresponds to more than 60% of the samples. The 1-plateaus are very similar for all three methodologies.

For the GSDTSR data set, we can see that XCSF-ER has almost no NAPFD breakdowns (and if using one of the recommended value functions, i. e., failure count or time ranked value, actually none). Furthermore the 1-plateau of both LCSs is longer than the one of the NN. For every value function, the NN is inferior to XCSF-ER. Positive effects of using ER in XCSF can be seen clearly in the experiments using the failure count and the time ranked value functions: ER pushes the performance across the entire bandwidth. Finally, we can see that XCSF-ER is by far the best choice for this data set.

The plots of the paint control data set show great differences as well. These become evident when using the time ranked or the failure count value function where both LCSs are superior. For the latter we can observe ER once more having a positive effect; on the former, however, we cannot visually see any effect of using ER in XCSF. Again, we notice that the 0-plateau of XCSF and XCSF-ER is shorter than on the IOF/ROL data set, indicating far fewer performance breakdowns. Additionally we can see that almost 40 percent of the samples have the highest NAPFD value (for the respective most suitable value function). This differs from the behaviour on GSDTSR where almost all the percentiles are higher but the 1-plateau is shorter. Also, the distribution-based view allows us to more closely examine the non-significant difference between the NN (with test case

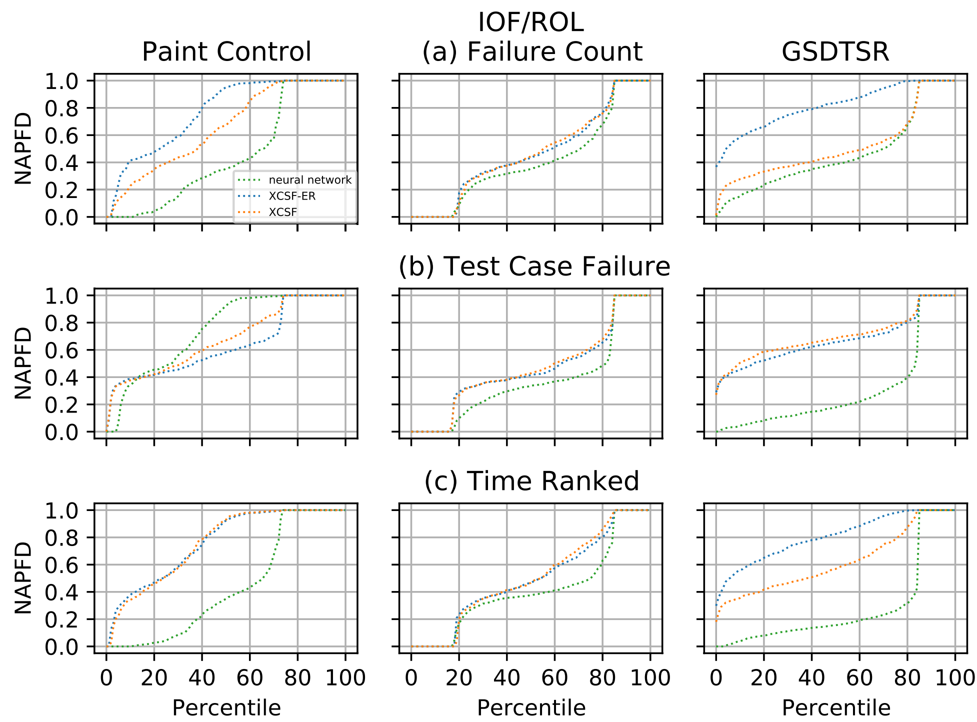


Fig. 4 Percentiles of the considered ML methods on the three data sets (columns) for each of the three value functions (rows). The kth percentile of an ordered sample $x_1 \leq x_2 \leq \cdots \leq x_n$ is defined as $x_{\lceil \frac{kn}{100} \rceil}$, i. e., the kth percentile is the value for which k % of values in the sample are smaller and $(100 - k)$ % are larger

failure value function) and XCSF-ER (with either the failure count or the time ranked value functions) that was identified in the previous section: Their percentile curves have the same shape and magnitude.

We further defined a key performance indicator (KPI) to compress the empirical percentiles into a single number. At that, we rely on calculus and compute the integrals of the NAPFD distributions. We define $p(\mathcal{M}, V, x)$ as the percentile function of an ML method $\mathcal{M}$ using a value function V. The input x is the desired percentile, e. g., 50 for the median. We define the KPI as follows:

$$\text{NAPFD ratio}(\mathcal{M}, V) = \frac{\int_0^{100} p(\mathcal{M}, V, x)dx}{\int_0^{100} p(\text{NN}, V_i^{\text{tcf}}, x)dx}, \tag{17}$$

which we coin the NAPFD ratio. Thus we compare the relation of the percentile area of one ML method with the state-of-the-art NN proposed by Spieker et al. [32] (using the value function where it performed best). Values higher than 1 indicate that $\mathcal{M}$ outperforms the NN-based solution. It is worth mentioning that we have to approximate the integrals as we lack the actual percentile functions; to do so, we rely on the well-known trapezoidal rule and use a mesh of 100.

The NAPFD ratios of XCSF and XCSF-ER for each value function and data set are given in Table 3. We can see that

Table 3 NAPFD ratio for XCSF and XCSF-ER for the different data sets and value functions

Data set	Value function	XCSF-ER	XCSF
Paint control	Failure count	**1.032**	0.876
	Test case failure	0.852	0.911
	Time ranked	**1.022**	1.018
IOF/ROL	Failure count	1.258	1.266
	Test case failure	1.229	1.252
	Time ranked	**1.335**	**1.356**
GSDTSR	Failure count	**2.740**	1.726
	Test case failure	2.260	2.356
	Time ranked	**2.708**	2.083

The two best values per data set are marked bold

for every data set, a combination of LCS and value function exists which outperforms the NN (i. e., which have a NAPFD ratio higher than 1). On the paint control data set these differences are rather minor (they should probably be regarded as equivalent), but on the other two data sets they become more distinctive. For IOF/ROL, we can observe up to 35% more performance and on GSDTSR, XCSF-ER even achieves up to 2.7 times the output of the NN. We can once more see that XCSF-ER achieves good results even if we select the value

function a priori and keep it fixed. Further, XCSF-ER performs equally or better than XCSF on the data sets considered.

In summary, we extended our temporal evaluation by taking a special focus on the distribution of the NAPFD results achieved. Thereby we could see a large amount of CI cycles where the LCS-based methods (especially XCSF-ER) outperformed the NN. For a considerable range of percentiles we could observe optimal results. Generally we could observe that the workflow used by a software project (i. e., the number of CI cycles per day) has an impact on the performance of all the approaches considered. For example, on IOF/ROL, there were several CI cycles where all techniques had performance breakdowns. We could additionally compare the ML algorithms with each other using a KPI that takes the NN as a baseline. Based on that, we were able to conclude that LCSs are in most cases not only equivalent but superior to the NN.

Buffer Mechanism

In our previous experiments we merely followed the ER mechanism of Spieker et al. [32] (as stated in Chapter 6). We already mentioned that the buffer index prioritization performance might be influenced by the test case order which induces the buffer index. Therefore, we empirically analyse if this is indeed the case by comparing the approach with the two alternatives stated in Chapter 5.5 (uniform and CI cycle prioritized). We reuse the buffer and batch size described in Chapter 6.

Figure 5 once more consists of has nine plots with columns corresponding to data sets and rows corresponding to value functions. For the following analysis, we found boxplots to be more suitable than temporal behaviour or percentile graphs. We decided to do so as the percentile graphs were heavily overlapping and thus visually hard to examine. Further, we wanted to measure the overall impact of the buffer mechanism throughout the projects' lifetime and not at individual time points (thus we decided against a temporal analysis).

For the time-ranked value function (third row) we can observe no visual difference between the individual buffer mechanisms considered. All have similar quartiles, whiskers, and medians. Thus the chosen batch drawing method has no impact. For the failure count value function (first row) we can see nearly the identical behaviour, except that uniform ER leads to one negative outlier that the other two ER mechanisms do not have.

Visual differences can only be seen if we consider the test case failure value function (second row). On the Paint control task, the CI prioritized method yields the best results while the other two perform equally well. On the IOF/ROL task, similar medians can be observed whilst the buffer index prioritization has a better third quartile. Further, buffer index prioritization has higher upper whiskers than the other two methods which indicates that more CI cycles with NAPFD values in that region occurred. In the higher NAPFD region uniform and CI cycle prioritized ER merely have a few outliers. On the GSDTSR task, three ER approaches have similar whiskers. Differences can be seen in terms of the quartiles and medians where uniform ER comes in first, buffer index prioritization second and CI cycle prioritization last.

Overall, our visual analysis underlined that only for the test case failure value function differences can be observed. However, this is of minor importance of this applied work as we are interested in one value function that performs best across all given data sets. Our previous experiments showed that that value function is not the test case failure value function; this can also be seen in Fig. 5 as both the time ranked and failure count value functions perform better than the test case failure value function. Corresponding statistical tests indicate the same[10]. Thus we pragmatically concentrate once more on the failure count and time-ranked approach.

Based on our visual evaluation we form the hypothesis that the choice of ER mechanism does not matter (for the better performing value functions) for a specific data set with respect to performance. Statistically formulated, we conjecture that for each data set the performance measurements collected any of the three ER mechanisms are from the same distribution. This can be examined using the Kruskal–Wallis test [13]. We decided against the more well-known ANOVA since the Kruskal–Wallis test is parameter-free and ANOVA's preconditions (normality, equality of variance, independence) cannot be ensured in our setting.

Table 4 shows the corresponding p values. We can see in 5 out of 6 cases high p values (between 0.96 and 0.9788). Only for one case (GSDTSR and time ranked) we observe a medium p value of 0.475280 which we attribute to the slightly higher quartiles of the two prioritized ER variants when compared to uniform ER (if we left out uniform ER then the p value would be 0.996). In order to properly interpret the p values we have to reconsider their meaning. For this test, the p value is the probability of observing similar or more extreme NAPFD data under the assumption that the three groups considered are from the same distribution. Since the p values are quite high, we can conclude that if our hypothesis is true then the observed NAPFD values are quite likely. This may be seen as very weak evidence in favor of our hypothesis which adds to the evidence we gathered from the visual analysis.

[10] We used one-sided Wilcoxon tests to compare each combination of one of the three ER methods with one of the failure count or time ranked value functions with each combination of the three ER methods with the test case failure value function (a total of $(3 \times 2) \times (3 \times 1) = 18$ comparisons) and for each checked the null hypothesis of whether the first performs worse than the second. Since all the p-values are less than 10^{-21}, we conclude that the failure count and time-ranked value functions yield significantly better results.

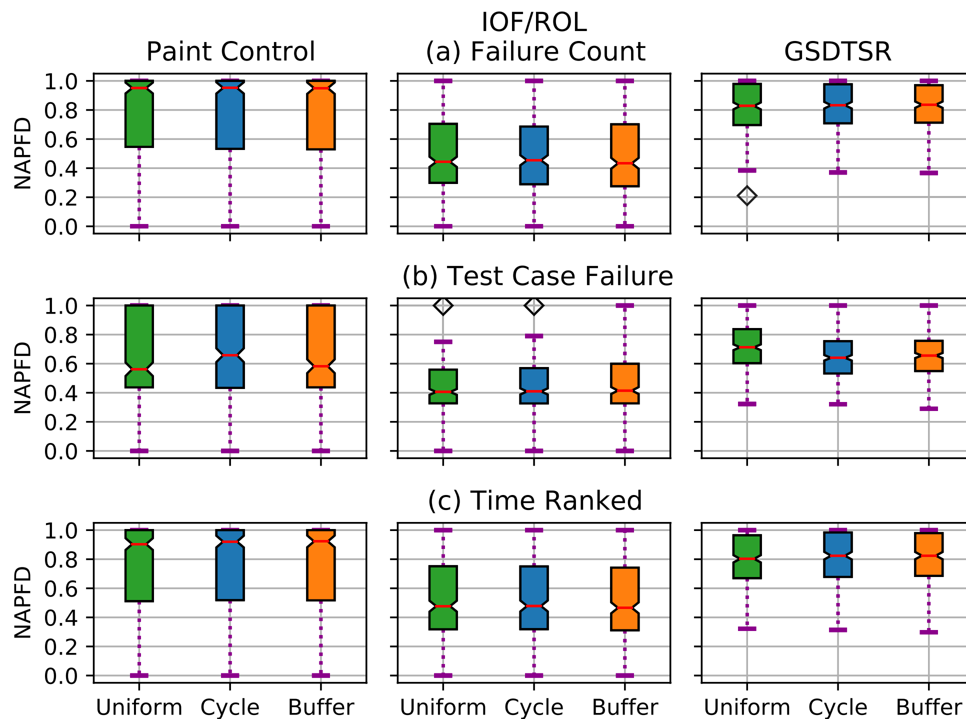


Fig. 5 Boxplots of the considered buffer mechanisms on the three data sets (columns) for each of the three value functions (rows). Note that Cycle stands for CI cycle prioritized and Buffer for buffer index prioritized

Table 4 p values for different Kruskal-Wallis tests. H_0: "For data set d and value function V, the NAPFD performances of the three ER approaches considered are from the same distribution"

Data set	Failure count	Time ranked
Paint Control	0.978821	0.970148
IOF/ROL	0.978822	0.967298
GSDTSR	0.960042	0.475280

In summary, we conclude that for the two value functions that are of interest for practitioners (due to their performance) the three ER mechanisms we considered perform equivalently. If we combine this with our previous analysis (Sects. 7.1–7.2) then we can infer that while it *is* important that ER is used in order to achieve a higher performance, the exact mechanism seems to play a minor role.

Transfer Learning

Many companies such as Bosch or Microsoft do not solely develop one product. Hence a reuse of previously acquired testing knowledge is of interest. This motivated us to develop the population transformation described in Sect. 5. Here we examine this transformation's effects.

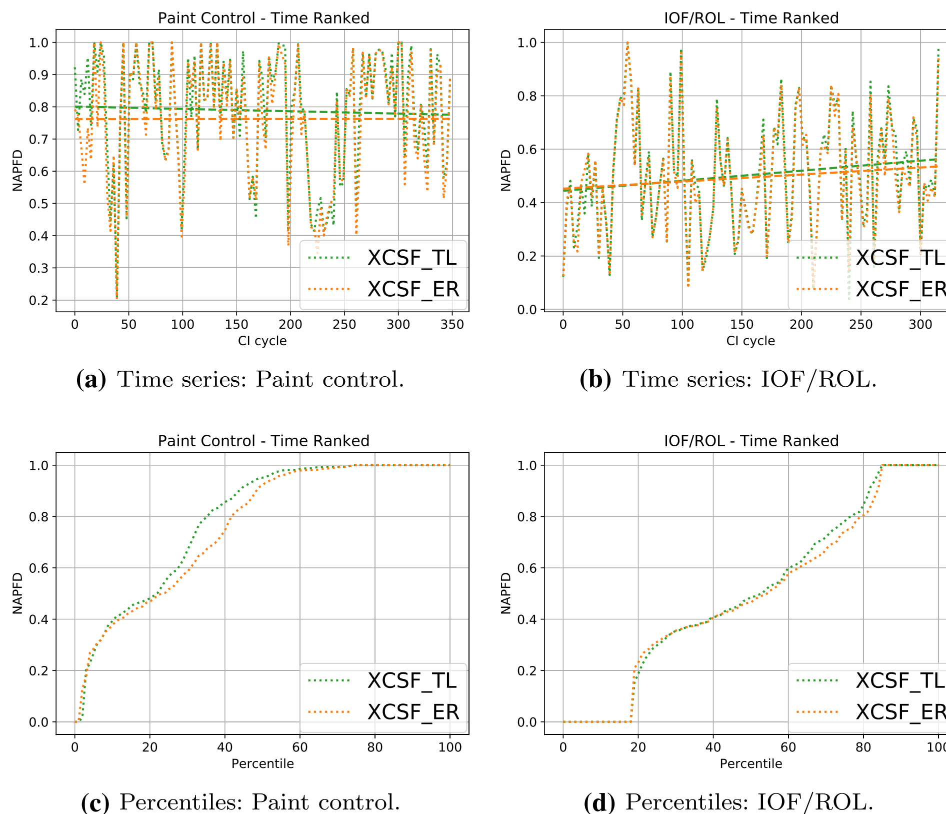
In one of our previous works we determined that the GSDTSR data set is suited best for pretraining the LCSs [25]. Certain data sets being better suited for pretraining than others is in line with other transfer learning approaches such as NN-based object recognition where a well-known approach is to use a model pretrained on the large ImageNet data set (note that, in our case, GSDTSR being suited best for pretraining also coincides with it being the largest data set). In the following, we evaluate the performance of an XCSF-ER that was pretrained on GSDTSR by comparing it to plain XCSF-ER.

We focus on the time ranked value as the previous experiments revealed that it is well-suited for XCSF-ER[11]. Figure 6a and b display the smoothed temporal behaviour (averaged over 30 runs) while 6c and d show the NAPFD distribution. Note that *XCSF-TL* denotes XCSF-ER with transfer learning.

Figure 6a and b reveal that positive effects of transfer learning can be seen across the entire time line and not just at the start. XCSF-TL seems to have smaller performance breakdowns, especially for the paint control data set. On the IOF/ROL data set, these positive effects are smaller.

Figure 6c and d underline this generally positive influence. For both distributions, the XCSF-TL graph lies above

[11] For the failure count value we can observe similar results; the corresponding plots can be found in Appendix A.



(a) Time series: Paint control. **(b)** Time series: IOF/ROL.

(c) Percentiles: Paint control. **(d)** Percentiles: IOF/ROL.

Fig. 6 Overview of the effects of transfer learning on XCSF-ER. **a** and **b** show time series and trend lines of the averaged (30 runs) and smoothed NAPFDs. **c** and **d** show NAPFD distributions in the form of a percentile plot as described in Sect. 7.2. Note that XCSF-TL denotes XCSF-ER with transfer learning. The temporal plots are based on [25]

the XCSF-ER graph indicating an overall higher NAPFD level. Aside from that, we can observe a shorter 0-plateau for the paint control data set which confirms the decreased amount of performance breakdowns in that case. On the paint control data set this leads to less performance breakdowns as the percentage of CI cycles with a NAPFD of zero becomes smaller.

Both the temporal and distribution plots give rise to the hypothesis that transfer learning is beneficial in terms of NAPFD. We evaluated that hypothesis with corresponding Wilcoxon signed rank tests[12]. The corresponding p-values were below 0.05. Thus we infer that transfer learning indeed delivers a performance boost.

We once more want to measure how big this performance increase is. Therefore we reuse our KPI, but exchange the baseline. We do not normalize with the integral of the NN but with XCSF-ER's integral. For paint control, we could increase the output by about 3.4 % and for IOF/ROL by about 2 %.

[12] We examined null hypotheses of the form: Our transfer learning approach leads to worse results than the raw XCSF-ER on data set x with value function y.

Random Selection

The previous analysis compared different LCSs and an NN with each other. These have all in common that they are a version of artificial intelligence (AI). In order to widen our view we look outside and compare our approaches against a standard approach which is used in many sources for benchmarking: a pure random selection [2, 14, 27]. Here, the method assigns a priority drawn uniformly at random from [0, 1). We focus on the time ranked value, the observations for the failure count value are similar (those results can be found in Appendix B).

We begin with a visual analysis by employing a series of boxplots (Fig. 7). We can see throughout all plots that the medians of the LCSs are always above the ones of the random selection. For XCSF-ER we can see in two cases that even the lower quartiles are higher. The LCSs show a wider bandwidth of values, but have fewer outliers. However, the mass of the distribution is on higher NAPFD values (Table 5).

The previous observation leads to the speculation that LCSs perform better than a pure random selection. In order to back this statistically we investigate the null hypotheses

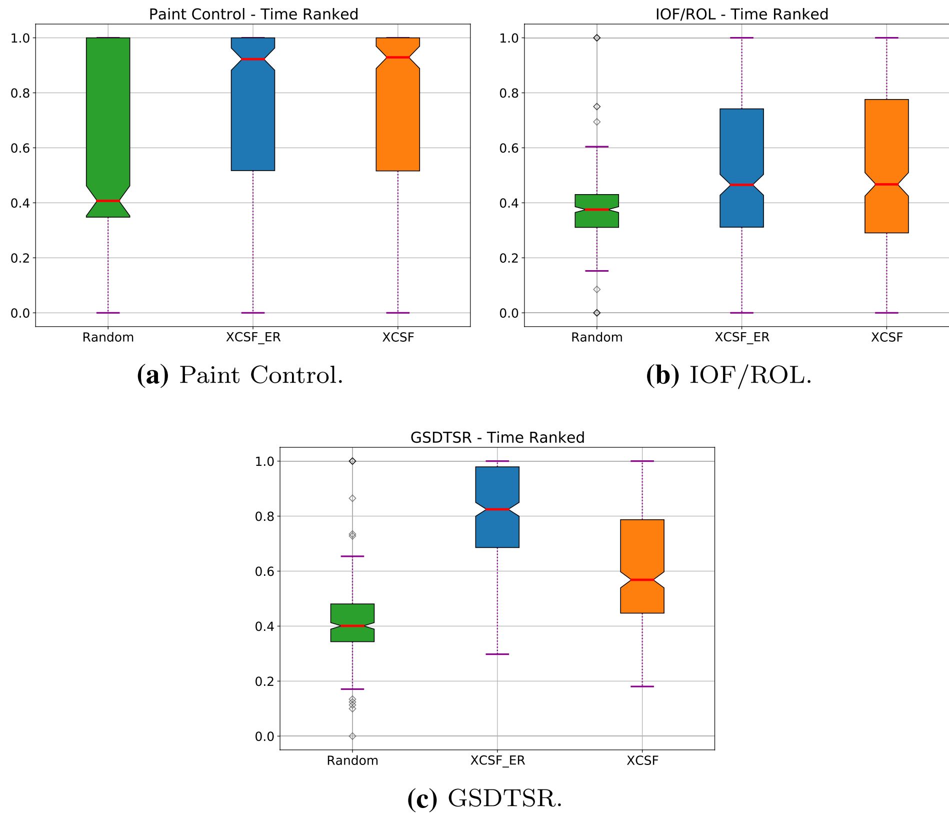


Fig. 7 Boxplots displaying the performance of the used LCSs and random selection

Table 5 *p* values for comparing XCSF and XCSF-ER to pure random selection. For the ML methods we consider the time ranked value function

Data set	XCSF-ER	XCSF
Paint control	2.80e–23	5.60e–22
IOF/ROL	2e–4	9.04e–05
GSDTSR	1.98e–71	7.36e–15

Note that all are significant

that the random selection performs equally well or better than XCSF on each specific data set (analogously for XCSF-ER). To do so, we use Wilcoxon signed rank tests and report the p-values in Table 6. It is worth mentioning that this statistical test has no preconditions that must be checked. The p-values are all close to zero and we can hence reject the null hypothesis and infer that the LCSs indeed perform better.

An Evolutionary Computation Application

In our previous experiments we underlined the performance of our approach. This, however, does not yet answer one vital question: How do we deliver the scientific value created to the industry? One step in that direction has been creating a system architecture and design specification; we refer the reader to the corresponding publication [26]. However, software architecture exceeds mere design documents. The architectural properties of the solution created are often regarded as more important [23]. These are also known as quality attributes [6].

In order to determine the desirable architectural properties we first want to outline for whom we deployed our solution. BSH Hausgeräte GmbH (BSH) develops and produces various kinds of embedded systems (home appliances). Therefore the developers need a wide range of knowledge (electronics, mechanics, C/C++ software development etc.). However, based on our personal experience, this usually does not include ML or EC. Thus the solution must be easy to operate even without such knowledge, making usability our primary quality property.

Usability is linked to other architectural properties such as accessibility and installability. Our solution is written in Python 3 and hosted on a company-internal Python package management system which is accessible from every BSH computer. It can be installed via Python's package manager pip using a one-line command.

The software itself is a simple command line tool. This matches the workflow of CI servers such as Jenkins

or Bamboo well as these define CI pipelines in the form of scripts (similar to .sh or .bat files). We coined our tool *Q_auto* after the engineer *Q* from the James Bond movies (as he equips the LCSs with the necessary equipment to do their job). Q_auto just takes three arguments: The name of the pipeline, the CI cycle index and the available time budget, the first two of which are usually provided as environment variables. Since BSH employs a testing process which predefines the location of test cases, Q_auto can parse them from there automatically. Overall, in order to use our software in practice, an operator simply has to integrate the following commands into the corresponding pipeline script:

- `pip install test_abstraction_layer`: Installs the Python package.
- `Q_auto pipeline_name CI_cycle time_budget`: Runs the prioritization, selects the test cases, executes them, retrieves the results and performs the learning updates.

Thus it boils down to two commands which require no knowledge of ML or EC whatsoever.The documentation is hosted on internal servers that the engineers are used to work with; this allows access to the newest documentation if changes come up. Overall, we conclude that our usability goal is satisfied. As a side-effect, we also enabled good accessability and installability.

It is worth outlining that many more architectural properties exist, for example, security. Since our application is not directly connected to the internet but instead hidden behind BSH's corporate security measures, this vital goal can be seen as having already been achieved by BSH's IT department. Other quality attributes such as maintainablity or adaptability depend more on the used design; for an overview of these, we recommend our publication that focuses on that topic [26].

Performance-related quality attributes such as fault-tolerance or robustness are more related to our algorithmic approach and can already be seen in our experimental evaluation. However, we still want to give an overview of how our system performs for our industrial partner. We started a pilot phase with three different projects across differing test levels. A dishwasher system test (DC ST), a dishwasher user interface test (DC UI) and a control and power module test (CPM) of an oven.

As mentioned before, in practice, we cannot evaluate the system's performance using NAPFD as it requires the knowledge of the outcomes of all possible test cases. Instead we use a metric that we coin the failure velocity at CI cycle i which computes the quotient of failed tests and execution time of the chosen test suite:

$$\frac{|\mathcal{TS}_i^f|}{\sum_{T \in \mathcal{TS}_i} d_{\text{actual}}(T)} \tag{18}$$

where $d_{\text{actual}}(T)$ denotes the measured runtime of T. This is analogous to velocity; however, instead of measuring how far we went per time, we measure how many failing test cases the system found in the given time. Thus the metric takes both objectives into account (failure revealing capabilities, limited time budget).

Figure 8 displays the failure velocity for the three aforementioned projects. We decided to use a logarithmic scale since our metric is not normalized and might have high values (see DC UI); this slightly dampens the visibility of the system learning. Also, two of the graphs end early as—at the time of this writing—no more recent results were available for them. Over the up to 40 CI cycles, we can see some variation in failure velocity but, even if performance breaks down, the system is capable to recover just as we could see in the robustness experiments in Chapter 7. Even though, in the middle of the timeline, the testers decided to reduce the available testing time (by roughly 50 %), the system is still capable to achieve high values in terms of our metric. Further, on DC ST, the BSH testers experimented with the time budget by making it smaller and even set it to only one minute for several CI cycles (which lead to the break-down in CI cycles 8 and 9). After increasing it again, the system recovered. Overall, the plot shows that the system is capable of detecting failing test cases and that it already shows some robustness. We keep our analysis at this level concluding

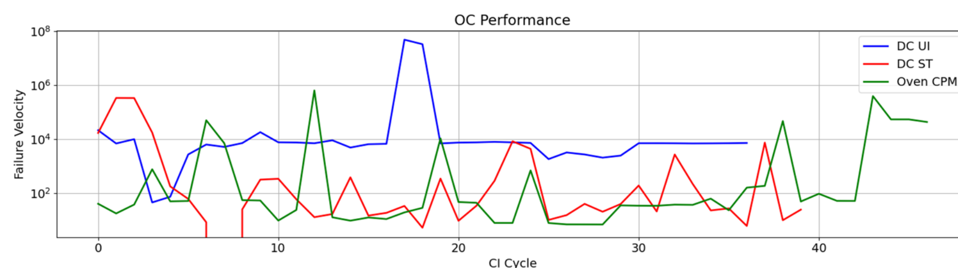


Fig. 8 Failure velocity for the three BSH pilot projects

that there is definitely a need for a long-term evaluation after the system ran several months.

Future Work

We implemented the methodology presented as an easy to use command line tool for one of our industrial partners. The system is already up and running but we identified an open point when it comes to the practical evaluation of the performance. We cannot use NAPFD as we do not have the knowledge about the outcome of all test cases. While we already came up with an alternative metric, failure velocity, it has a high total variation in terms of its magnitude. Also, in the long run, we desire a more human-interpretable metric.

Our experiments showed that the workflow used by a software project has an impact on every ML approach considered. Especially the frequency of the CI runs seems to be a key point; a high sampling rate in terms of software builds and test cases seems to be beneficial at first glance. This hypothesis would be in line with the well-known Shannon theorem (we could reconstruct the “software signal”). We intend to examine this hypothesis using data from the industrial solution that we discussed in Sect. 8. However, in order to be able to do so, more data needs to be collected first.

One of the key implications of this study is that LCSs are a good choice for this task. There is a variety of other mechanisms available that might further push their performance (see, e. g., [20]and [19]). We plan to evaluate several of them in order to see whether this can be observed here as well. In particular, interpolation-based methods caught our interest: Stein et al. [34] empirically showed in a series of articles on toy problems that this mathematical technique may be a desirable addition to LCSs. Nonetheless it is still to be shown if this effect is also the case on real-world problems such as this testing use case.

Furthermore, while our experiments underlined that transfer learning is useful for this task, a more fine-grained transformation may be required to really increase performance.

Our field tests at BSH showed that the testers sometimes may change the time budget available and the system is nonetheless able to cope with that. A further idea to boost the system’s performance would be to adapt our ER mechanism to react to a changed time budget by replaying (or storing) more experiences.

Additionally we are going to evaluate another selection mechanism as the current one of Spieker et al. does not consider the test execution time for test cases of equal rank.

Aside from our ML goals we also pursue more traditional engineering tasks: We work closely with BSH’s test engineers to continuously improve our solution from a technical standpoint.

Conclusion

Our work focused on a task in software verification which arises in the modern development practice of continuous integration (CI): Since only a limited time is available for executing test cases, a set of crucial test cases has to be compiled.

A recent trend is to rely on machine learning (ML) techniques and CI-related data to compute such a set of test cases. One state-of-the-art method is based on an artificial neural network (NN). We presented an alternative approach that uses a learning classifier system (LCS), more specifically, XCSF, and compared it with the NN-based solution. In our experiments we could see that, in most cases, the LCS is superior. We could further see that the LCS never falls behind on any data set considered.

The problem at hand shows a lot of variation when employing the typical, purely temporal view on the gathered experimental data. We thus extended our analysis by a distributional perspective. This removed the variation of the signal and provided further insights, revealing that our method is superior for the majority of the data points gathered and indicating a higher robustness. It exceeded the state-of-the-art performance by up to 2.7 times and performed optimally for large fractions of the considered software projects’ lifetimes.

Finally, we reported on our experiences with deploying our solution as a full stack software for an industrial partner. The product is an easy-to-use program which requires no knowledge about ML. It is already being used to automatedly select test cases and increase failure detection in three pilot software projects with varying time budgets.

A Transfer Learning for Failure Count

Figure 9 displays the transfer learning experiments if the failure count value instead of the time ranked value is employed.

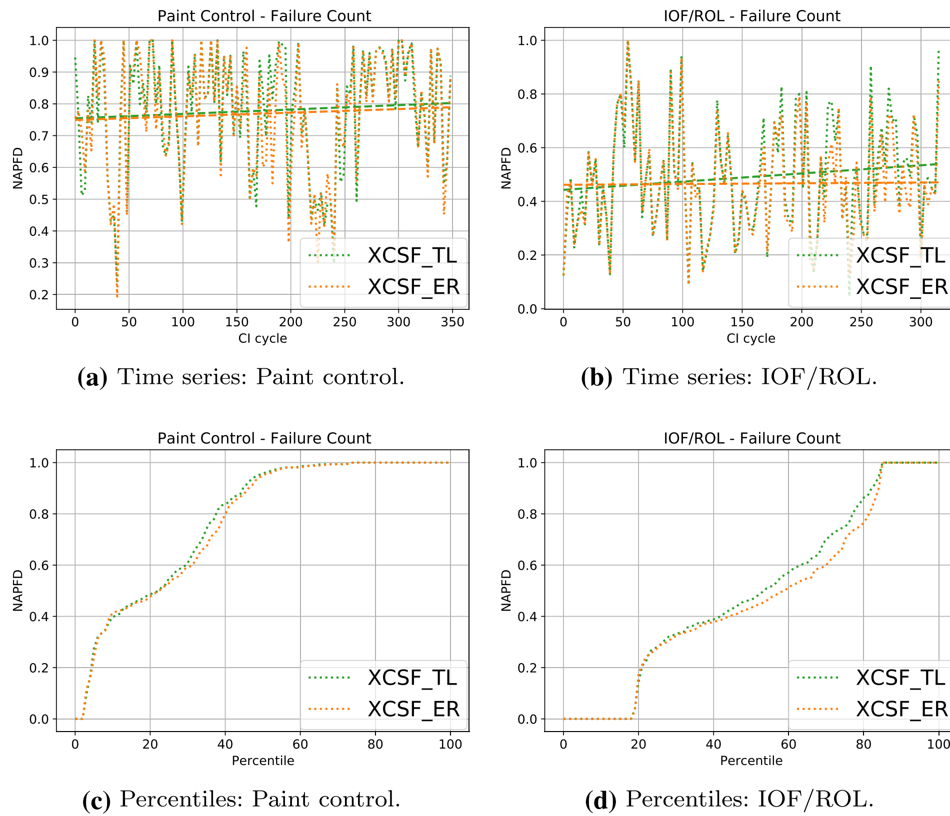
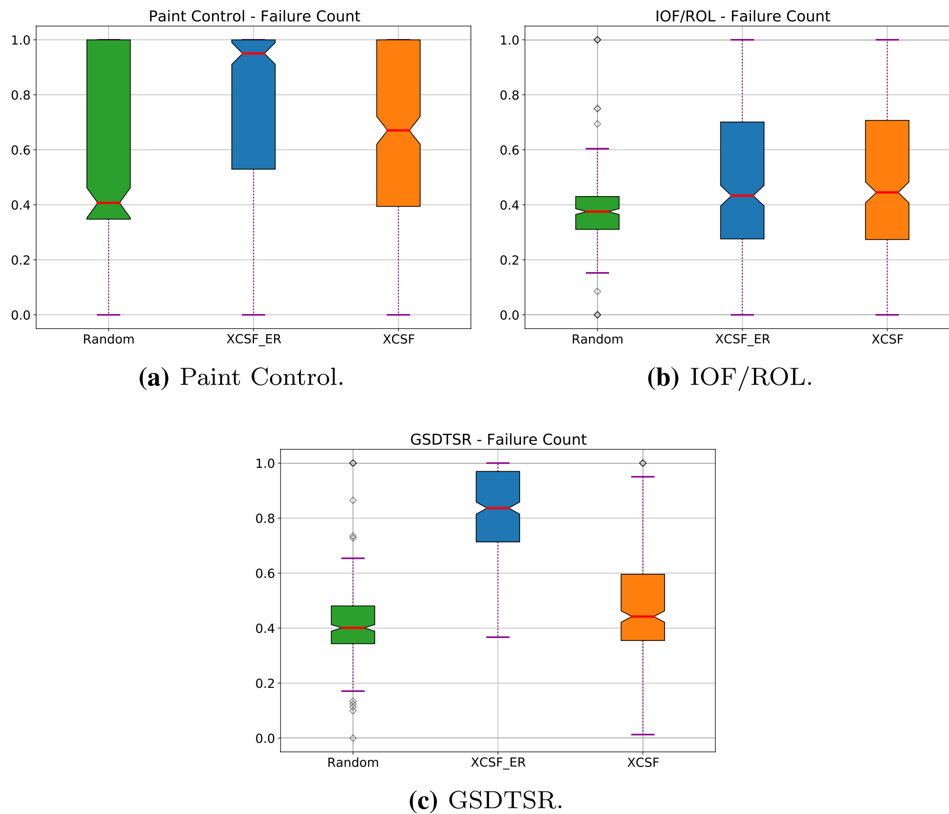


Fig. 9 Overview of the effects of transfer learning on XCSF-ER (both temporal and in terms of the distribution). Note that XCSF-TL denotes XCSF-ER with transfer learning and the temporal plots are based on [25]. Here we employ the failure count value

B Random Testing Comparison using Failure Count

See Fig. 10 and Table 6.



(a) Paint Control. **(b)** IOF/ROL.

(c) GSDTSR.

Fig. 10 Boxplots displaying the performance of the used LCSs and random selection

Table 6 p values for XCSF and XCSF-ER with a pure random selection. For the methods we consider the failure count value

Data set	Metric	XCSF-ER	XCSF
Paint control	p value	2.80e–23	5.60e–22
IOF/ROL	p value	0.0002	9.04e–05
GSDTSR	p value	1.98e–71	7.36e–15

Note all are significant

Author Contributions Not applicable.

Funding Not applicable.

Availability of Data and Material The used data sets may be found here: https://bitbucket.org/HelgeS/atcs-data/src/master/.

Declarations

Conflict of interest Not applicable.

Code Availability The source code for the ML approaches etc. can be retrieved from here: https://github.com/LagLukas/transfer_learning.

Consent to Participate Not applicable (no medical study).

Consent to Publish Not applicable (no medical study).

References

1. Anand S, Burke EK, Chen TY, Clark J, Cohen MB, Grieskamp W, Harman M, Harrold MJ, McMinn P, Bertolino A, Li JJ, Zhu H. An orchestrated survey of methodologies for automated software test case generation. J Syst Softw. 2013;86(8):1978–2001.
2. Arrieta A, Wang S, Arruabarrena A, Markiegi U, Sagardui G, Etxeberria L. Multi-objective black-box test case selection for cost-effectively testing simulation models. In: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO '18, 2018. New York: Association for Computing Machinery, p. 1411–8.
3. Butz MV, Wilson SW. An algorithmic description of XCS. In: Lanzi PL, Stolzmann W, Wilson SW, editors. Advances in learning classifier systems. Berlin: Springer; 2001. p. 253–72.
4. Dijkstra EW. Chapter I: notes on structured programming. GBR: Academic Press Ltd.; 1972. p. 1–82.
5. Fedus W, Ramachandran P, Agarwal R, Bengio Y, Larochelle H, Rowland M, Dabney W. Revisiting fundamentals of experience replay. CoRR. http://arxiv.org/abs/2007.06700, 2020.
6. International Organization for Standardization. ISO/IEC 25010. https://iso25000.com/index.php/en/iso-25000-standards/iso-25010, 2014. Accessed 15 Jun 2021.
7. Fowler M. Continuous integration. https://www.martinfowler.com/articles/continuousIntegration.html, 2006. Accessed 21 Feb 2021.
8. Fraser G, Wotawa F. Redundancy based test-suite reduction. In: Dwyer MB, Lopes A, editors. Fundamental approaches to software engineering. Berlin: Springer; 2007. p. 291–305.
9. Hsu H-Y, Orso A. Mints: a general framework and tool for supporting test-suite minimization. In: 2009 IEEE 31st International Conference on Software Engineering, 2009. p. 419–429.

10. Huang R, Sun W, Xu Y, Chen H, Towey D, Xia X. A survey on adaptive random testing. IEEE Trans Softw Eng. 2021;47(10):2052–83.
11. Hunter JD. Matplotlib: a 2D graphics environment. Comput Sci Eng. 2007;9(3):90–5.
12. Kirdey S, Cureton K, Rick S, Ramanathan S, Mrinal S. Lerner—using RL agents for test case scheduling. https://netflixtechblog.com/lerner-using-rl-agents-for-test-case-scheduling-3e0686211198, 2019. Accessed 21 Feb 2021
13. Kruskal WH, Wallis WA. Use of ranks in one-criterion variance analysis. J Am Stat Assoc. 1952;47(260):583–621.
14. Lachmann R, Felderer M, Nieke M, Schulze S, Seidl C, Schaefer I. Multi-objective black-box test case selection for system testing. In: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO ’17. 2017. New York: Association for Computing Machinery, p. 1311–8.
15. Lin L-J. Reinforcement Learning for Robots Using Neural Networks. PhD thesis, Pittsburgh, PA, USA, 1992. UMI Order No. GAX93-22750.
16. Lukasczyk S, Kroiß F, Fraser G. Automated unit test generation for python. CoRR, abs/2007.14049, 2020.
17. Müller-Schloer C, Tomforde S. Organic computing—technical systems for survival in the real world. In: Autonomic Systems, 2017.
18. Papadakis M, Kintis M, Zhang J, Jia Y, Traon TL, Harman M. Chapter six—mutation testing advances: an analysis and survey. volume 112 of Advances in Computers, p. 275–378. Elsevier, 2019.
19. Pätzel D, Heider M, Wagner ARM. An overview of LCS research from 2020 to 2021. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO ’21. New York: Association for Computing Machinery, 2021, pp. 1648–56.
20. Pätzel D, Stein A, Nakata M. An overview of lcs research from iwlcs 2019–2020. In: Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion, GECCO ’20. New York: Association for Computing Machinery, 2020, pp. 1782–8.
21. Prothmann H, Tomforde S, Branke J, Hähner J, Müller-Schloer C, Schmeck H. Organic traffic control; 2011.
22. Qu X, Cohen MB, Woolf KM. Combinatorial interaction regression testing: a study of test case generation and prioritization. In: 2007 IEEE International Conference on Software Maintenance. 2007, p. 255–64.
23. Richards M, Ford N. Fundamentals of software architecture: an engineering approach. London: O’Reilly Media Incorporated; 2019.
24. Rosenbauer L, Stein A, Hähner J. An artificial immune system for adaptive test selection. In: 2020 IEEE Symposium Series on Computational Intelligence (SSCI), 2020; p. 2940–7.
25. Rosenbauer L, Pätzel D, Stein A, Hähner J. Transfer learning for automated test case prioritization using xcsf. In: EvoApplications: 24th International Conference on the Applications of Evolutionary Computation as part of evostar 2021, April 2021, Seville, Spain, 2021.
26. Rosenbauer L, Pätzel D, Stein A, Hähner J. An organic computing system for automated testing. In: Bauer L, Pionteck T, editors. Architecture of computing systems—ARCS 2021. Cham: Springer International Publishing; 2021.
27. Rosenbauer L, Stein A, Hähner J. An artificial immune system for black box test case selection. In: EvoCop: 21st European Conference on Evolutionary Computation in Combinatorial Optimisation as part of evostar 2021, April 2021, Seville, Spain, 2021.
28. Rosenbauer L, Stein A, Maier R, Pätzel D, Hähner J. Xcs as a reinforcement learning approach to automatic test case prioritization. In: Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion, GECCO’20, New York: Association for Computing Machinery, 2020, p. 1798–806.
29. Rosenbauer L, Stein A, Pätzel D, Hähner J. Xcsf for automatic test case prioritization. In: Merelo JJ, Garibaldi J, Wagner C, Bäck T, Madani K, Warwick K (eds) Proceedings of the 12th International Joint Conference on Computational Intelligence (ECTA), November 2–4, 2020, 2020.
30. Rosenbauer L, Stein A, Pätzel D, Hähner J. Xcsf with experience replay for automatic test case prioritization. In: Abbass H, Coello Coello CA, Singh HK (eds) 2020 IEEE Symposium Series on Computational Intelligence (SSCI), virtual event, Canberra, Australia, 1–4 December 2020, 2020.
31. Smart JF. Jenkins: the Definitive Guide. Beijing: O’Reilly; 2011.
32. Spieker H, Gotlieb A, Marijan D, Mossige M. Reinforcement learning for automatic test case prioritization and selection in continuous integration. CoRR. abs/1811.04122, 2018.
33. Stein A, Maier R, Rosenbauer L, Hähner J. Xcs classifier system with experience replay. In: Proceedings of the 2020 Genetic and Evolutionary Computation Conference, GECCO’20. New York: Association for Computing Machinery, 2020, p. 404–13.
34. Stein A, Menssen S, Hähner J. What about interpolation? a radial basis function approach to classifier prediction modeling in xcsf. In: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO ’18. New York: Association for Computing Machinery, 2018, p. 537–44.
35. Stein A, Rudolph S, Tomforde S, Hähner J. Self-learning smart cameras—harnessing the generalisation capability of XCS. In: Proceedings of the 9th International Joint Conference on Computational Intelligence, Funchal, Portugal, 2017.
36. Ståhl D, Bosch J. Modeling continuous integration practice differences in industry software development. J Syst Softw. 2014;87:48–59.
37. Urbanowicz RJ, Browne WN. Introduction to Learning Classifier Systems. Springer Publishing Company, Incorporated, 1st edn., 2017.
38. Wilson S. Classifiers that approximate functions. Nat Comput. 2002;1:1–2.
39. Wilson SW. Classifier fitness based on accuracy. Evol Comput. 1995;3(2):149–75.
40. Yoo S, Harman M. Regression testing minimization, selection and prioritization: a survey. Softw Test Verif Reliab. 2012;22(2):67–120.
41. Yu Y, Jones JA, Harrold MJ. An empirical study of the effects of test-suite reduction on fault localization. In: Proceedings of the 30th International Conference on Software Engineering, ICSE ’08. New York: Association for Computing Machinery, 2008, p. 201–10.

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.