Reinforcement Learning for Automatic Test Case Prioritization and Selection in Continuous Integration

Helge Spieker
Simula Research Laboratory
P.O. Box 134
1325 Lysaker, Norway
helge@simula.no

Arnaud Gotlieb
Simula Research Laboratory
P.O. Box 134
1325 Lysaker, Norway
arnaud@simula.no

Dusica Marijan
Simula Research Laboratory
P.O. Box 134
1325 Lysaker, Norway
dusica@simula.no

Morten Mossige
University of Stavanger
Stavanger, Norway
ABB Robotics
Bryne, Norway
morten.mossige@uis.no

ABSTRACT

Testing in Continuous Integration (CI) involves test case prioritization, selection, and execution at each cycle. Selecting the most promising test cases to detect bugs is hard if there are uncertainties on the impact of committed code changes or, if traceability links between code and tests are not available. This paper introduces Retecs, a new method for automatically learning test case selection and prioritization in CI with the goal to minimize the round-trip time between code commits and developer feedback on failed test cases. The Retecs method uses reinforcement learning to select and prioritize test cases according to their duration, previous last execution and failure history. In a constantly changing environment, where new test cases are created and obsolete test cases are deleted, the Retecs method learns to prioritize error-prone test cases higher under guidance of a reward function and by observing previous CI cycles. By applying Retecs on data extracted from three industrial case studies, we show for the first time that reinforcement learning enables fruitful automatic adaptive test case selection and prioritization in CI and regression testing.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**; **Software testing and debugging**;

KEYWORDS

Regression testing, Test case prioritization, Test case selection, Reinforcement Learning, Machine Learning, Continuous Integration

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ISSTA'17, July 2017, Santa Barbara, CA, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5076-1/17/07…$15.00
https://doi.org/10.1145/3092703.3092709

ACM Reference format:
Helge Spieker, Arnaud Gotlieb, Dusica Marijan, and Morten Mossige. 2017. Reinforcement Learning for Automatic Test Case Prioritization and Selection in Continuous Integration. In *Proceedings of 26th International Symposium on Software Testing and Analysis , Santa Barbara, CA, USA, July 2017 (ISSTA'17),* 11 pages.
https://doi.org/10.1145/3092703.3092709

1 INTRODUCTION

Context. Continuous Integration (CI) is a cost-effective software development practice commonly used in industry [10, 13] where developers frequently integrate their work. It involves several tasks, including version control, software configuration management, automatic build and regression testing of new software release candidates. Automatic regression testing is a crucial step which aims at detecting defects as early as possible in the process by selecting and executing available and relevant test cases. CI is seen as an essential method for improving software quality while keeping verification costs at a low level [24, 34].

Unlike usual testing methods, testing in CI requires tight control over the selection and prioritization of the most promising test cases. By most promising, we mean test cases that are prone to detect failures early in the process. Admittedly, selecting test cases which execute the most recent code changes is a good strategy in CI, such as, for example in coverage-based test case prioritization [9]. However, traceability links between code and test cases are not always available or easily accessible when test cases correspond to system tests. In system testing for example, test cases are designed for testing the overall system instead of simple units of code and instrumenting the system for code coverage monitoring is not easy. In that case, test case selection and prioritization has to be handled differently and using historical data about failures and successes of test cases has been proposed as an alternative [16]. Based on the hypothesis that test cases having failed in the past are more likely to fail in the future, *history-based test case prioritization* schedules these test cases first in new CI cycles [19]. Testing in CI also means

This work is supported by the Research Council of Norway (RCN) through the research-based innovation center Certus, under the SFI program.

to control the time required to execute a complete cycle. As the durations of test cases strongly vary, not all tests can be executed and *test case selection* is required.

Despite algorithms have been proposed recently [19, 23], we argue that these two aspects of CI testing, namely test case selection and history-based prioritization, can hardly be solved by using only non-adaptive methods. First, the time allocated to test case selection and prioritization in CI is limited as each step of the process is given a contract of time. So, time-effective methods shall be privileged over costly and complex prioritization algorithms. Second, history-based prioritization is not well adapted to changes in the execution environment. More precisely, it is frequent to see some test cases being removed from one cycle to another because they test an obsolete feature of the system. At the same time, new test cases are introduced to test new or changed features. Additionally, some test cases are more crucial in certain periods of time, because they test features on which customers focus the most, and then they loose their prevalence because the testing focus has changed. In brief, non-adaptive methods may not be able to spot changes in the importance of some test cases over others because they apply systematic prioritization algorithms.

Reinforcement Learning. In order to tame these problems, we propose a new lightweight test case selection and prioritization approach in CI based on reinforcement learning and neural networks. Reinforcement learning is well-tuned to design an adaptive method capable to learn from its experience of the execution environment. By adaptive, it is meant, that our method can progressively improve its efficiency from observations of the effects its actions have. By using a neural network which works on both the selected test cases and the order in which they are executed, the method tends to select and prioritize test cases which have been successfully used to detect faults in previous CI cycles, and to order them so that the most promising ones are executed first.

Unlike other prioritization algorithms, our method is able to adapt to situations where test cases are added to or deleted from a general repository. It can also adapt to situations where the testing priorities change because of different focus or execution platforms, indicated by changing failure indications. Finally, as the method is designed to run in a CI cycle, the time it requires is negligible, because it does not need to perform computationally intensive operations during prioritization. It does not mine in detail code-based repositories or change-logs history to compute a new test case schedule. Instead it facilitates knowledge about test cases which have been the most capable to detect failures in a small sequence of previous CI cycles. This knowledge to make decisions is updated only after tests are executed from feedback provided by a reward function, the only component in the method initially embedding domain knowledge.

The contributions of this paper are threefold:

(1) This paper shows that history-based test case prioritization and selection can be approached as a reinforcement learning problem. By modeling the problem with notions such as states, actions, agents, policy, and reward functions, we demonstrate, as a first contribution, that RL is suitable to automatically prioritize and select test cases;
(2) Implementing an online RL method, without any previous training phase, into a Continuous Integration process is shown to be effective to learn how to prioritize test cases. According to our knowledge, this is the first time that RL is applied to test case prioritization and compared with other simple deterministic and random approaches. Comparing two distinct representations (i.e., tableau and neural networks) and three distinct reward functions, our experimental results show that, without any prior knowledge and without any model of the environment, the RL approach is able to learn how to prioritize test cases better than other approaches. Remarkably, the number of cycles required to improve on other methods corresponds to less than 2-months of data, if there is only one CI cycle per day;
(3) Our experimental results have been computed on industrial data gathered over one year of Continuous Integration. By applying our RL method on this data, we actually show that the method is deployable in industrial settings. This is the third contribution of this paper.

Paper Outline. The rest of the paper is organized as follows: Section 2 provides notations and definitions. It also includes a formalization of the problem addressed in our work. Section 3 presents our Retecs approach for test case prioritization and selection based on reinforcement learning. It also introduces basic concepts such as artificial neural network, agent, policy and reward functions. Section 4 presents our experimental evaluation of the Retecs on industrial data sets, while Section 5 discusses related work. Finally, Section 6 summarizes and concludes the paper.

2 FORMAL DEFINITIONS

This section introduces necessary notations used in the rest of the paper and presents the addressed problem in a formal way.

2.1 Notations and Definitions

Let $\mathcal{T}_i$ be a set of test cases $\{t_1, t_2, \ldots, t_N\}$ at a CI cycle i. Note that this set can evolve from one cycle to another. Some of these test cases are selected and ordered for execution in a test schedule called $\mathcal{TS}_i$ ($\mathcal{TS}_i \subseteq \mathcal{T}_i$). For evaluation purposes, we define further $\mathcal{TS}_i^{total}$ as being the ordered sequence of all test cases ($\mathcal{TS}_i^{total} = \mathcal{T}_i$) as if all test cases are scheduled for execution regardless of any time limit. Note that $\mathcal{T}_i$ is an unordered set, while $\mathcal{TS}_i$ and $\mathcal{TS}_i^{total}$ are ordered sequences. Following up on this idea, we define a ranking function over the test cases: $rank : \mathcal{TS}_i \rightarrow \mathbb{N}$ where $rank(t)$ is the position of t within $\mathcal{TS}_i$.

In $\mathcal{TS}_i$, each test case t has a verdict $t.verdict_i$ and a duration $t.duration_i$. Note that these values are only available after executing the test case and that they depend on the cycle in which the test case has been executed. For the sake of simplicity, the verdict is either 1 if the test case has passed, or 0 if it has failed or has not been executed in cycle i, i.e. it is not included in $\mathcal{TS}_i$. The subset of all failed test cases in $\mathcal{TS}_i$ is noted $\mathcal{TS}_i^{fail} = \{t \in \mathcal{TS}_i \text{ s.t. } t.verdict_i = 0\}$. The failure of an executed test case can be due to one or several actual faults in the system under test, and conversely a single fault can be responsible of multiple failed test cases. For the remainder of this paper, we will focus only on failed test cases (and not actual faults of the system) as the link between actual faults and executed test cases is not explicit in the available data of our context. Whereas $t.duration_i$ is the actual duration and only available after executing the test case, $t.duration$ is a simple over-approximation of previous durations and can be used for planning purposes.

Finally, we define $q_i(t)$ as a performance estimation of a test case in the given cycle i. By performance, we mean an estimate of its efficiency to detect failures. The performance Q_i of a test suite $\{t_1, \dots, t_n\}$ can be estimated with any cumulative function (e.g., sum, max, average, etc.) over $q_i(t_1), \dots q_i(t_n)$, e.g., $Q_i(\mathcal{TS}_i) = \frac{1}{|\mathcal{TS}_i|} \sum_{t \in \mathcal{TS}_i} q(t)$.

2.2 Problem Formulation

The goal of any test case prioritization algorithm is to find an optimal ordered sequence of test cases that reveal failures as early as possible in the regression testing process. Formally speaking, following and adapting the notations proposed by Rothermel et al. in [32]: *Test Case Prioritization Problem (TCP)*
Let $\mathcal{TS}_i$ be a test suite, and $\mathcal{PT}$ be the set of all possible permutations of $\mathcal{TS}_i$, let Q_i be the performance, then *TCP* aims at finding $\mathcal{TS}'_i$ a permutation of $\mathcal{TS}_i$, such that $Q_i(\mathcal{TS}'_i)$ is maximized. Said otherwise, TCP aims at finding $\mathcal{TS}'_i$ such that $\forall\ \mathcal{TS}_i \in \mathcal{PT}$: $Q_i(\mathcal{TS}'_i) \geq Q_i(\mathcal{TS}_i)$. Although it is fundamental, this problem formulation does not capture the notion of a time limit for executing the test suite. *Time-limited Test Case Prioritization* extends the TCP problem by limiting the available time for execution. As a consequence, not all the test cases may be executed when there is a time-contract. Note that other resources (than time) can constrain the test case selection process, too. However, the formulation given below can be adapted without any loss of generality.

Time-limited Test Case Prioritization Problem (TTCP)
Let M be the maximum time available for test suite execution, then *TTCP* aims at finding a test suite $\mathcal{TS}_i$, such that $Q_i(\mathcal{TS}_i)$ is maximized and the total duration of execution of $\mathcal{TS}_i$ is less than M. Said otherwise, TTCP aims at finding $\mathcal{TS}_i$ such that $\forall\ \mathcal{TS}'_i \in \mathcal{PT}$: $Q_i(\mathcal{TS}_i) \geq Q_i(\mathcal{TS}'_i) \wedge \sum_{t_k \in \mathcal{TS}'_i} t_k.duration \leq M \wedge \sum_{t_k \in \mathcal{TS}_i} t_k.duration \leq M$.

Still the problem formulation given above does not take into account the history of test suite execution. In case the links between code changes and test cases are not available as discussed in the introduction, history-based test case prioritization can be used. The final problem formulation given below corresponds to the problem addressed in this paper and for which a solution based on reinforcement learning is proposed. In a CI process, TTCP has to be solved in every cycle, but under the additional availability of historical information as a basis for test case prioritization. *Adaptive Test Case Selection Problem (ATCS)*
Let $\mathcal{TS}_1, \dots, \mathcal{TS}_{i-1}$ be a sequence of previously executed test suites, then the *Adaptive Test Case Selection Problem* aims at finding $\mathcal{TS}_i$, so $Q_i(\mathcal{TS}_i)$ is maximized and $\sum_{t \in \mathcal{TS}_i} t.duration \leq M$.

We see that ATCS is an optimization problem which gathers the idea of time-constrained test case prioritization, selection and performance evaluation, without requesting more information than previous test execution results in CI.

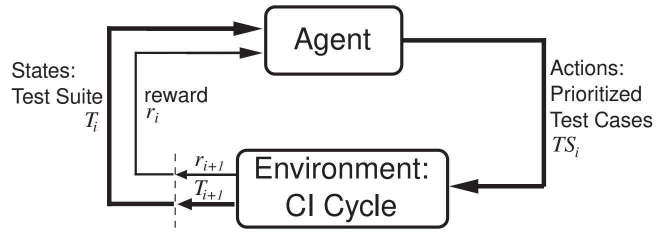

Figure 1: Interaction of Agent and Environment (adapted from [36, Fig 3.1])

3 THE RETECS METHOD

This section introduces our approach to the ATCS problem using reinforcement learning (RL), called *Reinforced Test Case Selection* (Retecs). It starts by describing how RL is applied to test case prioritization and selection (section 3.1), then discusses test case scheduling in one CI cycle (section 3.2). Finally, integration of the method within a CI process is presented (section 3.3).

3.1 Reinforcement Learning for Test Case Prioritization

In this section, we describe the main elements of reinforcement learning in the context of test case prioritization and selection. If necessary, a more in-depth introduction can be found in [36]. We apply RL as a *model-free* and *online learning* method for the ATCS problem. Each test case is prioritized individually and after all test cases have been prioritized, a schedule is created from the most important test cases, and afterwards executed and evaluated.

Model-free means the method has no initial concept of the environment's dynamics and how its actions affect it. This is appropriate for test case prioritization and selection, as there is no strict model behind the existence of failures within the software system and their detection.

Online learning describes a method constantly learning during its runtime. This is also appropriate for software testing, where indicators for failing test cases can change over time according to the focus of development or variations in the test suite. Therefore it is necessary to continuously adapt the prioritization method for test cases.

In RL, an agent interacts with its environment by perceiving its *state* and selecting an appropriate *action*, either from a learned *policy* or by random exploration of possible actions. As a result, the agent receives feedback in terms of *rewards*, which rate the performance of its previous action.

Figure 1 illustrates the links between RL and test case prioritization. A state represents a single test case's metadata, consisting of the test case's approximated duration, the time it was last executed and previous test execution results. As an action the test case's priority for the current CI cycle is returned. After all test cases in a test suite are prioritized, the prioritized test suite is scheduled, including a selection of the most important test cases, and submitted for execution. With the test execution results, i.e., the test verdicts, a reward is calculated and fed back to the agent. From this reward, the agent adapts its experience and policy for future actions. In case of positive rewards previous behavior is encouraged, i.e. reinforced, while in case of negative rewards it is discouraged.

Test verdicts of previous executions have shown to be useful to reveal future failures [16]. This raises the question how long the history of test verdicts should be for a reliable indication. In general, a long history provides more information and allows better knowledge of the failure distribution of the system under test, but it also requires processing more data which might have become irrelevant with previous upgrades of the system as the previously error-prone feature got more stable. To consider this, the agent has to learn how to time-weight previous test verdicts, which adds further complexity to the learning process. How the history length affects the performance of our method, is experimentally evaluated in Section 4.2.2.

Of further importance for RL applications are the agent's policy, i.e. the way it decides on actions, the memory representation, i.e. how it stores its experience and policy, and the reward function to provide feedback for adaptation and policy improvement.

In the following, we will discuss these components and their relevance for Retecs.

3.1.1 Reward Functions. Within the ATCS problem, a good test schedule is defined by the goals of test case selection and prioritization. It contains those test cases which lead to detection of failures and executes them early to minimize feedback time. The reward function should reflect these goals and thereby domain knowledge to steer the agent's behavior [20]. Referring to the definition of ATCS, the reward function implements Q_i and evaluates the performance of a test schedule.

Ideally, feedback should be based on common metrics used in test case prioritization and selection, e.g. NAPFD (presented in section 4.1). However, these metrics require knowledge about the total number of faults in the system under test or full information on test case verdicts, even for non-executed test cases. In a CI setting, test case verdicts exist only for executed test cases and information about missed failures is not available. It is impossible to teach the RL agent about test cases which should have been included, but only to reinforce actions having shown positive effects. Therefore, in Retecs, rewards are either zero or positive, because we cannot automatically detect negative behavior.

In order to teach the agent about both the goal of a task and the way to approach this goal the reward, two types of reward functions can be distinguished. Either a single reward value is given for the whole test schedule, or, more specifically, one reward value per individual test case. The former rewards the decisions on all test cases as a group, but the agent does not receive feedback how helpful each particular test case was to detect failures. The latter resolves this issue by providing more specific feedback, but risks to neglect the prioritization strategy of different priorities for different test cases for the complete schedule as a whole.

Throughout the presentation and evaluation of this paper, we will consider three reward functions.

Definition 3.1. Failure Count Reward

$$reward_i^{fail}(t) = |\mathcal{TS}_i^{fail}| \qquad (\forall\, t \in \mathcal{T}_i) \tag{1}$$

In the first reward function (1) all test cases, both scheduled and unscheduled, receive the number of failed test cases in the schedule as a reward. It is a basic, but intuitive reward function directly rewarding the RL agent on the goal of maximizing the number of failed test cases. The reward function acknowledges the prioritized test suite in total, including positive feedback on low priorities for test cases regarded as unimportant. This risks encouraging low priorities for test cases which would have failed if executed, and could encourage undesired behavior, but at the same time it strengthens the influence all priorities in the test suite have.

Definition 3.2. Test Case Failure Reward

$$reward_i^{tcfail}(t) = \begin{cases} 1 - t.verdict_i & \text{if } t \in \mathcal{TS}_i \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

The second reward function (2) returns the test case's verdict as each test case's individual reward. Scheduling failing test cases is intended and therefore reinforced. If a test case passed, no specific reward is given as including it neither improved nor reduced the schedule's quality according to available information. Still, the order of test cases is not explicitly included in the reward. It is implicitly included by encouraging the agent to focus on failing test cases and prioritizing them higher. For the proposed scheduling method (section 3.2) this automatically leads to an earlier execution.

Definition 3.3. Time-ranked Reward

$$reward_i^{time}(t) = |\mathcal{TS}_i^{fail}| - t.verdict_i \times \sum_{\substack{t_k \in \mathcal{TS}_i^{fail} \wedge \\ rank(t) < rank(t_k)}} 1 \tag{3}$$

The third reward function (3) explicitly includes the order of test cases and rewards each test case based on its rank in the test schedule and whether it failed. As a good schedule executes failing test cases early, every passed test case reduces the schedule's quality if it precedes a failing test case. Each test cases is rewarded by the total number of failed test cases, for failed test cases it is the same as reward function (1). For passed test cases, the reward is further decreased by the number of failed test cases ranked after the passed test case to penalize scheduling passing test cases early.

3.1.2 Action Selection: Prioritizing Test Cases. Action selection describes how the RL agent processes a test case and decides on a priority for it by using the policy. The policy is a function from the set of states, i.e., test cases in our context, to the set of actions, i.e., how important each test case is for the current schedule, and describes how the agent interacts with its execution environment. The policy function is an approximation of the optimal policy. In the beginning it is a loose approximation, but over time and by gathering experience it adapts towards an optimal policy.

The agent selects those actions from the policy which were most rewarding before. It relies on its learned experience on good actions for the current state. Because the agent initially has no concept of its actions' effects, it explores the environment by choosing random actions and observing received rewards on these actions. How often random actions are selected instead of consulting the policy, is controlled by the exploration rate, a parameter which usually decreases over time. In the beginning of the process, a high exploration rate encourages experimenting, whereas at a later time exploration is reduced and the agent more strongly relies on its learned policy. Still, exploration is not disabled, because the agent interacts in a dynamic environment, where the effects of certain actions change and where it is necessary to continuously adapt the policy. Action selection and the effect of exploration are also influenced by non-stationary rewards, meaning that the same action for the same test case does not always yield the same reward. Test cases which are likely to fail, based on previous experiences, do not fail when the software is bug-free, although their failure would be expected. The existence of non-stationary rewards has motivated our selection of an online-learning approach, which enables continuous adaptation and should tolerate their occurence.

3.1.3 Memory Representation. As noted above, the policy is an approximated function from a state (a test case) to an action (a priority). There exist a wide variety of function approximators in literature, but for our context we focus on two main approximators.

The first function approximator is the *tableau representation* [36]. It consists of two tables to track seen states and selected actions. In one table it is counted how often each distinct action was chosen per state. The other table stores the average received reward for these actions. The policy is then to choose that action with highest expected reward for the current state, which can be directly read from the table. When receiving rewards, cells for each rewarded combination of states and actions are updated by increasing the counter and calculating the running average of received rewards.

As an exploration method to select random actions, ϵ-greedy exploration is used. With probability $(1 - \epsilon)$ the most promising action according to the policy is selected, otherwise a random action is selected for exploration.

Albeit a straightforward representation, the tableau also restricts the agent. States and actions have to be discrete sets of limited size as each state/action pair is stored separately. Furthermore, with many possible states and actions, the policy approximation takes longer to converge towards an optimal policy as more experiences are necessary for the training. However, for the presented problem and its number of possible states a tableau is still applicable and considered for evaluation.

Overcoming the limitations of the tableau, artificial neural networks (ANN) are commonly used function approximators [37]. ANNs can approximate functions with continuous states and actions and are easier to scale to larger state spaces. The downside of using ANNs are more complex configuration and higher training efforts than for the tableau. In the context of Retecs, an ANN receives a state as input to the network and outputs a single continuous action, which directly resembles the test case's priority.

Exploration is different when using ANNs, too. Because a continuous action is used, ϵ-greedy exploration is not possible. Instead, exploration is achieved by adding a random value drawn from a Gaussian distribution to the policy's suggested action. The variance of the distribution is given by the exploration rate and a higher rate allows for higher deviations from the policy's actions. The lower the exploration rate is, the closer the action is to the learned policy.

Whereas the agent with tableau representation processes each experience and reward once, an ANN-based agent can be trained differently. Previously encountered experiences are stored and revisited during training phase to achieve repeated learning impulses, which is called *experience replay* [18]. When rewards are received, each experience, consisting of a test case, action and reward, is stored in a separate replay memory with limited capacity. If the replay memory capacity is reached, oldest experiences get replaced first. During training, a batch of experiences is randomly sampled from this memory and used for training the ANN via backpropagation with stochastic gradient descent [44].

3.2 Scheduling

Test cases are scheduled under consideration of their priority, their duration and a time limit. The scheduling method is a modular aspect within Retecs and can be selected depending on the environment, e.g. considering execution constraints or scheduling onto multiple test agents. As an only requirement it has to maximize the total priority within the schedule. For example, in an environment with only a single test agent and no further constraints, test cases can be selected by descending priority (ties broken randomly) until the time limit is reached.

3.3 Integration within a CI Process

In a typical CI process (as shown in Figure 2), a set of test cases is first prioritized and based on the prioritization a subset of test cases is selected and scheduled onto the testing system(s) for execution.

The Retecs method fits into this scheme by providing the *Prioritization* and *Selection & Scheduling* steps. It extends the CI process by requiring an additional feedback channel to receive test results after each cycle, which is the same or part of the information also provided as developer feedback.

4 EXPERIMENTAL EVALUATION

In this section we present an experimental evaluation of the Retecs method. During the first part, an overview of evaluation metrics (section 4.1) is given before the experimental setup is introduced (section 4.2). In section 4.3 we present and discuss the experimental results. A discussion of possible threats (section 4.4) and extensions (section 4.5) to our work close the evaluation.

Within the evaluation of the Retecs method we investigate if it can be successfully applied towards the ATCS problem. Initially, before evaluating the method on our research questions, we explore how different parameter choices affect the performance of our method.

- **RQ1** Is the Retecs method effective to prioritize and select test cases? We evaluate combinations of memory representations and reward functions on three industrial data sets.
- **RQ2** Can the lightweight and model-free Retecs method prioritize test cases comparable to deterministic, domain-specific methods? We compare Retecs against three comparison methods, one random prioritization strategy and to basic deterministic methods.

4.1 Evaluation Metric

In order to compare the performance of different methods, evaluation metrics are required as a common performance indicator.

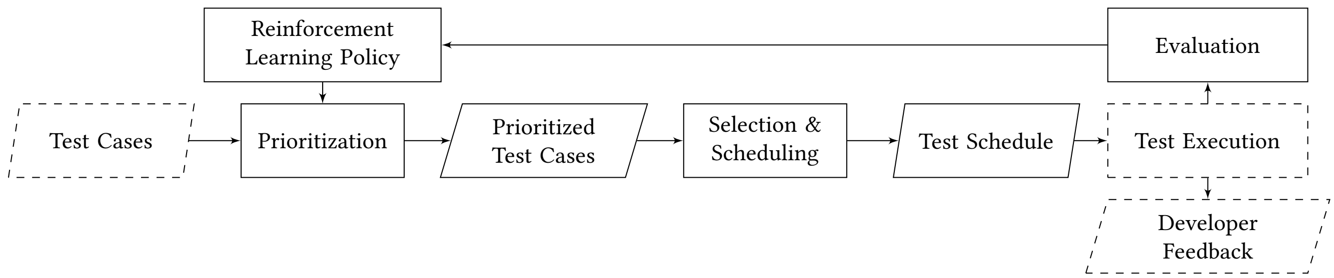

Figure 2: Testing in CI process: RETECS uses test execution results for learning test case prioritization (solid boxes: Included in RETECS, dashed boxes: Interfaces to the CI environment)

Following, we introduce Normalized Average Percentage of Faults Detected as the applied evaluation metric.

Definition 4.1. Normalized APFD

$$NAPFD(\mathcal{TS}_i) = p - \frac{\sum\limits_{t \in \mathcal{TS}_i^{fail}} rank(t)}{|\mathcal{TS}_i^{fail}| \times |\mathcal{TS}_i|} + \frac{p}{2 \times |\mathcal{TS}_i|}$$

$$\text{with } p = \frac{|\mathcal{TS}_i^{fail}|}{|\mathcal{TS}_i^{total,fail}|}$$

Average Percentage of Faults Detected (APFD) was introduced in [31] to measure the effectiveness of test case prioritization techniques. It measures the quality via the ranks of failure-detecting test cases in the test execution order. As it assumes all detectable faults get detected, APFD is designed for test case prioritization tasks without selecting a subset of test cases. Normalized APFD (NAPFD) [28] is an extension of APFD to include the ratio between detected and detectable failures within the test suite, and is thereby suited for test case selection tasks when not all test cases are executed and failures can be undetected. If all faults are detected ($p = 1$), NAPFD is equal to the original APFD formulation.

4.2 Experimental Setup

Two RL agents are evaluated in the experiments. First uses a tableau representation of discrete states and a fixed number of actions, named *Tableau-based* agent. And a second, *Network-based* agent uses an artificial neural network as memory representation for continuous states and a continuous action. The reward function of each agent is not fixed, but varied throughout the experiments.

Test cases are scheduled on a single test agent in descending order of priority until the time limit is reached.

To evaluate the efficiency of the Retecs method, we compare it to three basic test case prioritization methods. First is random test case prioritization as a baseline method, referred to as *Random*. The other two methods are deterministic. As a second method, named *Sorting*, test cases are sorted by their recent verdicts with recently failed test cases having higher priority. For the third comparison method, labeled as *Weighting*, the priority is calculated by a sum of the test case's features as they are used as an input to the RL agent. Weighting considers the same information as Retecs and corresponds to a weighted sum with equal weights and is thereby a naive version of Retecs without adaptation. Although the three comparison methods are basic approaches to test case prioritization, they utilize the same information as provided to our method, and are likely to be encountered in industrial environments.

Due to the online learning properties and the dependence on previous test suite results, evaluation is done by comparing the NAPFD metrics for all subsequent CI cycles of a data set over time. To account for the influence of randomness within the experimental evaluation, all experiments are repeated 30 times and reported results show the mean, if not stated otherwise.

Retecs[1] is implemented in Python [38] using scikit-learn's implementation of artificial neural networks [26].

4.2.1 Industrial Data Sets. To determine real-world applicability, industrial data sets from ABB Robotics Norway[2], *Paint Control* and *IOF/ROL*, for testing complex industrial robots, and *Google Shared Dataset of Test Suite Results* (GSDTSR) [11] are used.[3] These data sets consist of historical information about test executions and their verdicts and each contain data for over 300 CI cycles.

Table 1 gives an overview of the data sets' structure. Both ABB data sets are split into daily intervals, whereas GSDTSR is split into hourly intervals as it originally provides log data of 16 days, which is too short for our evaluation. Still, the average test suite size per CI cycle in GSDTSR exceeds that in the ABB data sets while having fewer failed test executions. For applying Retecs constant durations between each CI cycle are not required.

For the CI cycle's time limit, which is not present in the data sets, a fixed percentage of 50% of the required time is used. A relative time limit allows better comparison of results between data sets and keeps the difficulty at each CI cycle on a comparable level. How this percentage affects the results is evaluated in section 4.3.3.

4.2.2 Parameter Selection. A couple of parameters allow adjusting the method towards specific environments. For the experimental evaluation the same set of parameters is used in all experiments, if not stated otherwise. These parameters are based on values from literature and experimental exploration.

Table 2 gives an overview of the chosen parameters. The number of actions for the Tableau-based agent is set to 25. Preliminary tests showed a larger number of actions did not substantially increase

[1] Implementation available at https://bitbucket.org/helges/retecs
[2] Website: http://new.abb.com/products/robotics
[3] Data Sets available at https://bitbucket.org/helges/atcs-data

Table 1: Industrial Data Sets Overview: All columns show the total amount of data in the data set

Data Set	Test Cases	CI Cycles	Verdicts	Failed
Paint Control	114	312	25,594	19.36%
IOF/ROL	2,086	320	30,319	28.43%
GSDTSR	5,555	336	1,260,617	0.25%

Table 2: Parameter Overview

RL Agent	Parameter	Value
All	CI cycle's time limit M	$50\% \times \mathcal{T}_i.duration$
	History Length	4
Tableau	Number of Actions	25
	Exploration Rate ϵ	0.2
Network	Hidden Nodes	12
	Replay Memory	10000
	Replay Batch Size	1000

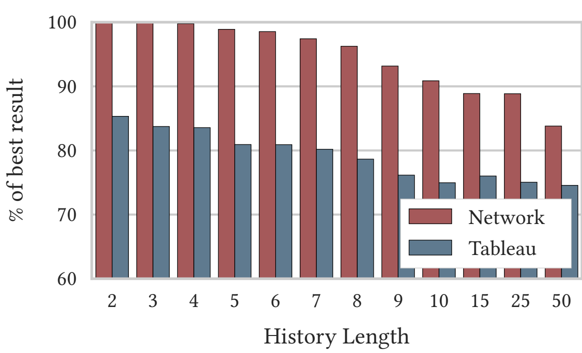

Figure 3: Relative performance of different history lengths. A longer history can reduce the performance due to more complex information. (Data set: ABB Paint Control)

the performance. Similar tests were conducted for the ANN's size, including variations on the number of layers and hidden nodes, but a network larger than a single layer with 12 nodes did not significantly improve performance.

The effect of different history lengths is evaluated experimentally on the Paint Control data set. As Figure 3 shows, does a longer history not necessarily correspond to better performance. From an application perspective we interpret the most recent results to also be the most relevant results. Many historical failures indicate a relevant test case better than many passes, but individual consideration of each of these results on their own is unlikely to lead to better conclusions of future verdicts. From a technical perspective, this is supported by the fact, that a longer history increases the state space of possible test case representations. A larger state space is in both memory representations related to a higher complexity and requires generally more data to adapt, because the agent has to learn to handle earlier execution results differently than more recent ones, for example by weighting or aggregating them.

4.3 Results

4.3.1 RQ1: Learning Process & Effectiveness. Figure 4 shows the performance of Tableau- and Network-based agents with different reward functions on three industrial data sets. Each column shows results for one data set, each row for a particular reward function.

It is visible that the combination of memory representation and reward function strongly influences the performance. In some cases it does not support the learning process and the performance stays at the initial level or even declines. Some combinations enable the agent to learn which test cases to prioritize higher or lower and to create meaningful test schedules.

Performance on all data sets is best for the Network-based agent with the Test Case Failure reward function. It benefits from the specific feedback for each test case and learns which test cases are likely to fail. Because the Network-based agent prioritizes test cases with continuous actions, it adapts more easily than the Tableau-based agent, where only specific actions are rewarded and rewards for one action do not influence close other actions.

In all results a similar pattern should be visible. Initially, the agent has no concept of the environment and cannot identify failing test cases, leading to a poor performance. After a few cycles it received enough feedback by the reward function to make better choices and successively improves. However, this is not true for all combinations of memory representation and reward function. One example is the combination of Network-based agent and Test Case Failure reward. On Paint Control, the performance at early CI cycles is superior to the Tableau-based agent, but it steadily declines due to misleading feedback from the reward function.

One general observation are performance fluctuations over time. These fluctuations are correlated to noise in the industrial data sets, where failures in the system occur for different reasons and are hard to predict. For example, in the Paint Control data set between 200 and 250 cycles a performance drop is visible. For these cycles a larger number of test cases were repeatedly added to the test suite manually. A large part of these test cases failed, which put additional difficulty on the task. However, as the test suite was manually adjusted, from a practical perspective it is arguable whether a fully automated prioritization technique is feasible during these cycles.

In GSDTSR only few failed test cases occur in comparison to the high number of successful executions. This makes it harder for the learning agent to discover a feasible prioritization strategy. Nevertheless, as the results show, it is possible for the Network-based agent to create effective schedules in a high number of CI cycles, albeit with occasional performance drops.

Regarding RQ1, we conclude that it is possible to apply Retecs on the ATCS problem. In particular, the combination of memory representation and reward function strongly influences the performance of the agent. We found both Network-based agent and Test Case Failure Reward, as well as Tableau-based agent with Time-ranked Reward, to be suitable combinations, with the former delivering an overall better performance. The Failure Count Reward function does not support the learning processes of the two agents. Providing only a single reward value without further distinction is not helping the agents towards an effective prioritization strategy. It is better to reward each test case's priority individually according to its contribution to the previous schedule.

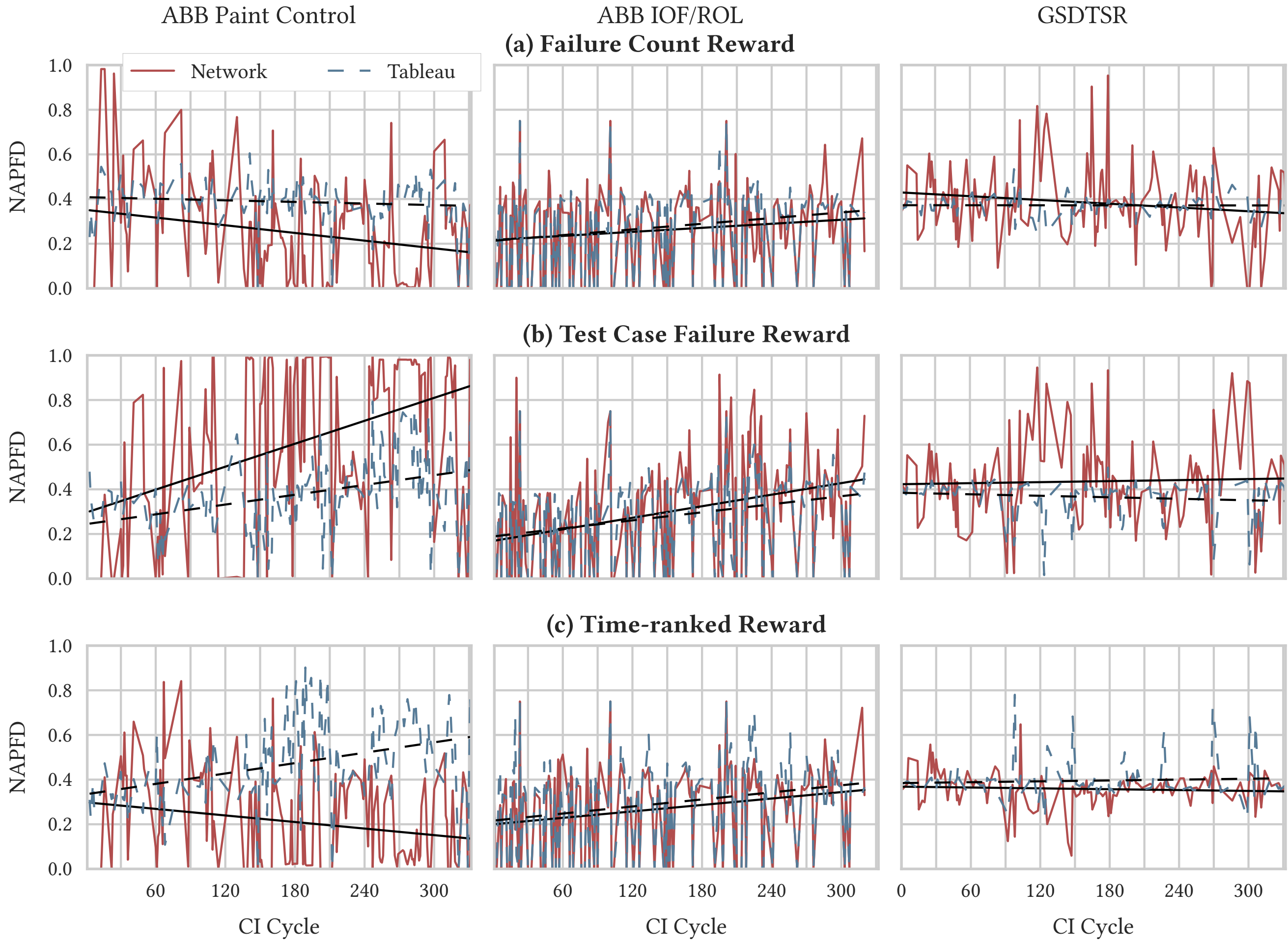

Figure 4: Comparison of reward functions and memory representations: A Network-based agent with Test Case Failure reward delivers best performance on all three data sets (Black lines indicate trend over time)

4.3.2 RQ2: Comparison to Other Methods. Where the experiments on RQ1 focus on the performances of different component combinations, is the focus of RQ2 towards comparing the best-performing Network-based RL agent (with Test Case Failure reward) with other test case prioritization methods. Figure 5 shows the results of the comparison against the three methods on each of the three data sets. A comparison is made for every 30 CI cycles on the difference of the average NAPFD values of each cycle. Positive differences show better performance by the comparison method, a negative difference shows better performance by Retecs.

During early CI cycles, the deterministic comparison methods show mostly better performance. This corresponds to the initial exploration phase, where Retecs adapts to its environment. After approximately 60 CI cycles, for Paint Control, it is able to prioritize with similar or better performance than the comparison methods. Similar results are visible on the other two data sets, with a longer adaptation phase but less performance differences on IOF/ROL and an early comparable performance on GSDTSR.

For IOF/ROL, where the previous evaluation (see Figure 4) showed lower performance compared to Paint Control, also the comparison methods are not able to correctly prioritize failing test cases higher, as the small performance gap indicates.

For GSDTSR, Retecs is performing overall comparable with an NAPFD difference up to 0.2. Due to the few failures within the data set, the exploration phase does not impact the performance in the early cycles as strongly as for the other two data sets. Also, it appears as if the indicators for failing test cases are not as correlated to the previous test execution results as they were in the other data sets, which is visible from the comparatively low performance of the deterministic methods.

In summary, the results for RQ2 show, that Retecs can, starting from a model-free memory without initial knowledge about test case prioritization, in around 60 cycles, which corresponds to two month for daily intervals, learn to effectively prioritize test cases. Its performance compares to that of basic deterministic test case prioritization methods. For CI, this means that Retecs is a promising method for test case prioritization which adapts to environment specific indication of system failures.

4.3.3 Internal Evaluation: Schedule Time Influence. In the experimental setup, the time limit for each CI cycle's reduced test

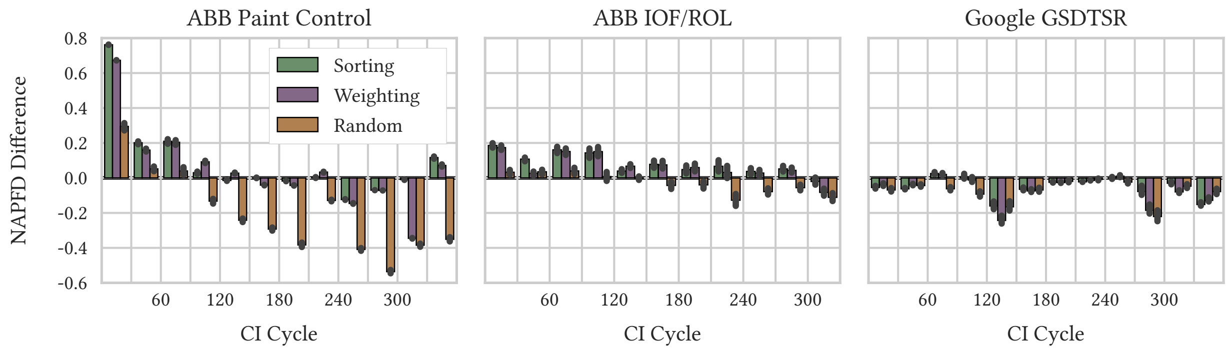

Figure 5: Performance difference between network-based agent and comparison methods: After an initial exploration phase RETECS adapts to competitive performance. Each group of bars compares 30 CI cycles.

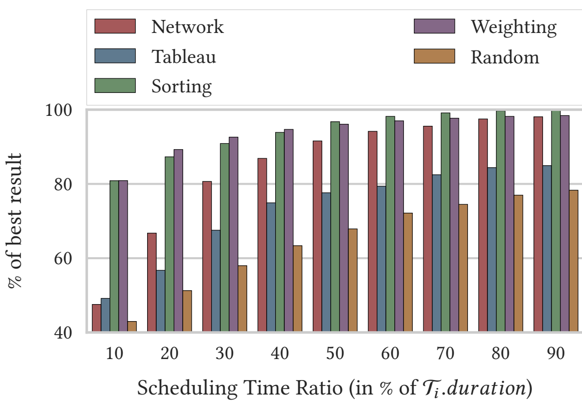

Figure 6: Relative performance under different time limits. Shorter scheduling times reduce the information for rewards and delay learning. The performance differences for Network and Tableau also arise from the initial exploration phase, as shown in Figure 5 (Data set: ABB Paint Control).

schedule is set to 50% of the execution time of the overall test suite $\mathcal{T}_i$. To see how this choice influences the results and how it affects the learning process, an additional experiment is conducted with varying scheduling time ratios.

Figure 6 shows the results on the Paint Control data set. The NAPFD result is averaged over all CI cycles, which explains the overall better performance by the comparison methods due to an initial learning period. As it is expected, performance decreases with lower time limits for all methods. However, for RL agents a decreased scheduling time directly decreases available information for learning as fewer test cases can be executed and fewer actions can meaningfully be rewarded, resulting in a slower learning process.

Nevertheless, the decrease in performance is not directly proportional to the decrease in scheduling time, a sign that Retecs learns at some point how to prioritize test cases even though the amount of data in previous cycles was limited.

4.4 Threats to Validity

Internal. The first threat to internal validity is the influence of random decisions on the results. To mitigate the threat, we repeated our experiments 30 times and report averaged results.

Another threat is related to the existence of faults within our implementation. We approached this threat by applying established components, such as scikit-learn, within our software where appropriate. Furthermore, our implementation is available online for inspection and reproduction of experiments.

Finally, many machine learning algorithms are sensible to their parameters and a feasible parameter set for one problem environment might not work for as well for different one. During our experiments, the initially selected parameters were not changed for different problems to allow better comparison. In a real-world setting, those parameters can be adjusted to tune the approach for the specific environment.

External. Our evaluation is based on data from three industrial data sets, which is a limitation regarding the wide variety of CI environments and failure distributions. One of these data sets is publicly available, but according to our knowledge it has only been used in one publication and a different setting [12]. From what we have analyzed, there are no further public data sets available which include the required data, especially test verdicts over time. This threat has to be addressed by additional experiments in different settings once further data is accessible. To improve the data availability, we publish the other two data sets used in our experiments.

Construct. A threats to construct validity is the assumption, that each failed test cases indicates a different failure in the system under test. This is not always true. One test case can fail due to multiple failures in the system and one failure can lead to multiple failing test cases. Based on the abstraction level of our method, this information is not easily available. Nevertheless, our approach tries to find all failing test cases and thereby indirectly also all detectable failures. To address the threat, we propose to include failure causes as input features in future work.

Further regarding the input features, our proposed method uses only few test case metadata to prioritize test cases and to reason about their importance for the test schedule. In practical environments, more information about test cases or the system under test is available and should be utilized.

We compared our method to baseline approaches, but we have not considered additional techniques. Although further methods exist in literature, they do not report results on comparable data sets or would need adjustment for our CI setting.

4.5 Extensions

The presented results give perspectives to extensions from two angles. First perspective is on the technical RL approach. Through a pre-training phase the agent can internalize test case prioritization knowledge before actually prioritizing test cases and thereby improve the initial performance. This can be approached by imitation of other methods [1], e.g. deterministic methods with desirable behavior, or by using historical data before it is introduced in the CI process [30]. The second perspective focuses on the domain-specific approach of test case prioritization and selection. Here, only few metadata of a test case and its history is facilitated. The number of features of a test case should be extended to allow better reasoning of expected failures, e.g. links between source code changes and relevant test cases. By including failure causes, scheduling of redundant test cases can be avoided and the effectiveness improved.

Furthermore, this work used a linear scheduling model, but in industrial environments more complex environments are encountered, e.g. multiple systems for test executions or additional constraints on test execution besides time limits. Another extension of this work is therefore to integrate different scheduling methods under consideration of prioritization information and integration into the learning process [27].

5 RELATED WORK

Test case prioritization and selection for regression testing: Previous work focuses on optimizing regression testing based on mainly three aspects: cost, coverage, and fault detection, or their combinations. In [21] authors propose an approach for test case selection and prioritization using the combination of Integer Linear Programming (ILP) and greedy methods by optimizing multiple criteria. Another study investigates coverage-based regression testing [9], using four common prioritization techniques: a test selection technique, a test suite minimization technique and a hybrid approach that combines selection and minimization. Similar approaches have been proposed using search-based algorithms [7, 42], including swarm optimization [8] and ant colony optimization [22]. Walcott et al. use genetic algorithms for time-aware regression test suite prioritization for frequent code rebuilding [40]. Similarly, Zhang et al. propose time-aware prioritization using ILP [43]. Strandberg et al. [35] apply a novel prioritization method with multiple factors in a real-world embedded software and show the improvement over industry practice. Other regression test selection techniques have been proposed based on historical test data [16, 19, 23, 25], code dependencies [14], or information retrieval [17, 33]. Despite various approaches to test optimization for regression testing, the challenge of applying most of them in practice lies in their complexity and the computational overhead typically required to collect and analyze different test parameters needed for prioritization, such as age, test coverage, etc. By contrast, our approach based on RL is a lightweight method, which only uses historical results and its experience from previous CI cycles. Furthermore, Retecs is adaptive and suited for dynamic environments with frequent changes in code and testing, and evolving test suites.
Machine learning for software testing: Machine learning algorithms receive increasing attention in the context of software testing. The work closest to ours is [4], where Busjaeger and Xie use machine learning and multiple heuristic techniques to prioritize test cases in an industrial setting. By combining various data sources and learning to rank in an agnostic way, this work makes a strong step into the definition of a general framework to automatically learn to rank test cases. Our approach, only based on RL and ANN, takes another direction by providing a lightweight learning method using one source of data, namely test case failure history. Chen et al. [6] uses semi-supervised clustering for regression test selection. The downside of such an approach may be higher computational complexity. Other approaches include active learning for test classification [3], combining machine learning and program slicing for regression test case prioritization [41], learning agent-based test case prioritization [2], or clustering approaches [5]. RL has been previously used in combination with adaptation-based programming (ABP) for automated testing of software APIs, where the combination of RL and ABP successively selects calls to the API with the goal to increase test coverage, by Groce et al. [15]. Furthermore, Reichstaller et al. [29] apply RL to generate test cases for risk-based interoperability testing. Based on a model of the system under test, RL agents are trained to interact in an error-provoking way, i.e. they are encouraged to exploit possible interactions between components. Veanes et al. use RL for online formal testing of communication systems [39]. Based on the idea to see testing as a two-player game, RL is used to strengthen the tester's behavior when system and test cases are modeled as Input-Output Labeled Transition Systems. While this approach is appealing, Retecs applies RL for a completely different purpose, namely test case prioritization and selection. Our approach aims at CI environments, which are characterized by strict time and effort constraints.

6 CONCLUSION

We presented Retecs, a novel lightweight method for test case prioritization and selection in Continuous Integration, combining reinforcement learning methods and historical test information. Retecs is adaptive and learns important indicators for failing test cases during its runtime by observing test cases, test results, and its own actions and their effects.

Evaluation results show fast learning and adaptation of Retecs in three industrial case studies. An effective prioritization strategy is discovered with a performance comparable to basic deterministic prioritization methods after an initial learning phase of approximately 60 CI cycles without previous training on test case prioritization. Necessary domain knowledge is only reflected in a reward function to evaluate previous schedules. The method is model-free, language-agnostic and requires no source code or program access. It only requires test metadata, namely historical results, durations and last execution times. However, we expect additional metadata to enhance the method's performance.

In our evaluation we compare different variants of RL agents for the ATCS problem. Agents based on artificial neural networks have shown to be best performing, especially when trained with test case-individual reward functions. While we applied only small networks in this work, with extended available data amounts, an extension towards larger networks and deep learning techniques can be a promising path for future research.

REFERENCES

[1] Pieter Abbeel and Andrew Y Ng. 2004. Apprenticeship learning via inverse reinforcement learning. *Proceedings of the 21st International Conference on Machine Learning (ICML)* (2004), 1–8. https://doi.org/10.1145/1015330.1015430 arXiv:1206.5264

[2] Sebastian Abele and Peter Göhner. 2014. Improving Proceeding Test Case Prioritization with Learning Software Agents. In *Proceedings of the 6th International Conference on Agents and Artificial Intelligence - Volume 2 (ICAART).* 293–298.

[3] James F Bowring, James M Rehg, and Mary Jean Harrold. 2004. Active Learning for Automatic Classification of Software Behavior. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '04).* ACM, New York, NY, USA, 195–205. https://doi.org/10.1145/1007512.1007539

[4] Benjamin Busjaeger and Tao Xie. 2016. Learning for Test Prioritization: An Industrial Case Study. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering.* ACM, New York, NY, USA, 975–980. https://doi.org/10.1145/2950290.2983954

[5] G Chaurasia, S Agarwal, and S S Gautam. 2015. Clustering based novel test case prioritization technique. In *2015 IEEE Students Conference on Engineering and Systems (SCES).* IEEE, 1–5. https://doi.org/10.1109/SCES.2015.7506447

[6] S Chen, Z Chen, Z Zhao, B Xu, and Y Feng. 2011. Using semi-supervised clustering to improve regression test selection techniques. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation.* IEEE, 1–10. https://doi.org/10.1109/ICST.2011.38

[7] Luciano S de Souza, Pericles BC de Miranda, Ricardo BC Prudencio, and Flavia de A Barros. 2011. A Multi-objective Particle Swarm Optimization for Test Case Selection Based on Functional Requirements Coverage and Execution Effort. In *2011 IEEE 23rd International Conference on Tools with Artificial Intelligence.* IEEE, 245–252. https://doi.org/10.1109/ICTAI.2011.45

[8] Luciano S de Souza, Ricardo B C Prudêncio, Flavia de A. Barros, and Eduardo H da S. Aranha. 2013. Search based constrained test case selection using execution effort. *Expert Systems with Applications* 40, 12 (2013), 4887–4896. https://doi.org/10.1016/j.eswa.2013.02.018

[9] Daniel Di Nardo, Nadia Alshahwan, Lionel Briand, and Yvan Labiche. 2015. Coverage-based regression test case selection, minimization and prioritization: a case study on an industrial system. *Software Testing, Verification and Reliability* 25, 4 (2015), 371–396. https://doi.org/10.1002/stvr.1572

[10] P M Duvall, S Matyas, and A Glover. 2007. *Continuous Integration: Improving Software Quality and Reducing Risk.* Pearson Education.

[11] Sebastian Elbaum, Andrew Mclaughlin, and John Penix. 2014. The Google Dataset of Testing Results. (2014). https://code.google.com/p/google-shared-dataset-of-test-suite-results/

[12] Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering.* ACM, 235–245. https://doi.org/10.1145/2635868.2635910

[13] Martin Fowler and M Foemmel. 2006. Continuous integration. (2006). http://martinfowler.com/articles/continuousIntegration.html

[14] M Gligoric, L Eloussi, and D Marinov. 2015. Ekstazi: Lightweight Test Selection. In *Proceedings of the 37th International Conference on Software Engineering*, Vol. 2. 713–716. https://doi.org/10.1109/ICSE.2015.230

[15] A. Groce, A. Fern, J. Pinto, T. Bauer, A. Alipour, M. Erwig, and C. Lopez. 2012. Lightweight Automated Testing with Adaptation-Based Programming. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering.* 161–170. https://doi.org/10.1109/ISSRE.2012.1

[16] Jung-Min Kim Jung-Min Kim and A. Porter. 2002. A history-based test prioritization technique for regression testing in resource constrained environments. In *Proceedings of the 24th international conference on software engineering.* 119–129. https://doi.org/10.1109/ICSE.2002.1007961

[17] Jung-Hyun Kwon, In-Young Ko, Gregg Rothermel, and Matt Staats. 2014. Test case prioritization based on information retrieval concepts. *2014 21st Asia-Pacific Software Engineering Conference (APSEC)* 1 (2014), 19–26. https://doi.org/10.1109/APSEC.2014.12

[18] Long-Ji Lin. 1992. Self-Improving Reactive Agents Based on Reinforcement Learning, Planning and Teaching. *Machine Learning* 8, 3-4 (1992), 293–321. https://doi.org/10.1023/A:1022628806385

[19] Dusica Marijan, Arnaud Gotlieb, and Sagar Sen. 2013. Test case prioritization for continuous regression testing: An industrial case study. In *2013 29th IEEE International Conference on Software Maintenance (ICSM).* 540–543. https://doi.org/10.1109/ICSM.2013.91

[20] Maja J Matarić. 1994. Reward functions for accelerated learning. In *Machine Learning: Proceedings of the Eleventh international conference.* 181–189. https://doi.org/10.1.1.42.4313

[21] Siavash Mirarab, Soroush Akhlaghi Esfahani, and Ladan Tahvildari. 2012. Size-Constrained Regression Test Case Selection Using Multicriteria Optimization. *IEEE Transactions on Software Engineering* 38, 4 (jul 2012), 936–956. https://doi.org/10.1109/TSE.2011.56

[22] T Noguchi, H Washizaki, Y Fukazawa, A Sato, and K Ota. 2015. History-Based Test Case Prioritization for Black Box Testing Using Ant Colony Optimization. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST).* 1–2. https://doi.org/10.1109/ICST.2015.7102622

[23] Tanzeem Bin Noor and Hadi Hemmati. 2015. A similarity-based approach for test case prioritization using historical failure data. *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)* (2015), 58—-68.

[24] A Orso and G Rothermel. 2014. Software Testing: a Research Travelogue (2000–2014). In *Proceedings of the on Future of Software Engineering.* ACM, Hyderabad, India, 117–132.

[25] H Park, H Ryu, and J Baik. 2008. Historical Value-Based Approach for Cost-Cognizant Test Case Prioritization to Improve the Effectiveness of Regression Testing. In *2008 Second International Conference on Secure System Integration and Reliability Improvement.* 39–46. https://doi.org/10.1109/SSIRI.2008.52

[26] F Pedregosa, G Varoquaux, A Gramfort, V Michel, B Thirion, O Grisel, M Blondel, P Prettenhofer, R Weiss, V Dubourg, J Vanderplas, A Passos, D Cournapeau, M Brucher, M Perrot, and E Duchesnay. 2011. Scikit-learn: Machine Learning in {P}ython. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[27] Bo Qu, Changhai Nie, and Baowen Xu. 2008. Test case prioritization for multiple processing queues. *2008 International Symposium on Information Science and Engineering (ISISE)* 2 (2008), 646–649. https://doi.org/10.1109/ISISE.2008.106

[28] Xiao Qu, Myra B. Cohen, and Katherine M. Woolf. 2007. Combinatorial interaction regression testing: A study of test case generation and prioritization. In *IEEE International Conference on Software Maintenance, 2007 (ICSM).* IEEE, 255–264.

[29] Andre André Reichstaller, Benedikt Eberhardinger, Alexander Knapp, Wolfgang Reif, and Marcel Gehlen. 2010. Risk-Based Interoperability Testing Using Reinforcement Learning. In *28th IFIP WG 6.1 International Conference, ICTSS 2016, Graz, Austria, October 17-19, 2016, Proceedings*, Franz Wotawa, Mihai Nica, and Natalia Kushik (Eds.), Vol. 6435. Springer International Publishing, Cham, 52–69. https://doi.org/10.1007/978-3-642-16573-3

[30] Martin Riedmiller. 2005. Neural fitted Q iteration - First experiences with a data efficient neural Reinforcement Learning method. In *European Conference on Machine Learning.* Springer, 317–328. https://doi.org/10.1007/11564096_32

[31] Gregg Rothermel, Roland H Untch, Chengyun Chu, and Mary Jean Harrold. 1999. Test case prioritization: An empirical study. In *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on.* IEEE, 179–188.

[32] Gregg Rothermel, Roland H Untch, Chengyun Chu, Mary Jean Harrold, and Ieee Computer Society. 2001. Prioritizing Test Cases For Regression Testing. *IEEE Transactions on Software Engineering* 27, 10 (2001), 929–948. https://doi.org/10.1145/347324.348910

[33] Ripon K Saha, L Zhang, S Khurshid, and D E Perry. 2015. An Information Retrieval Approach for Regression Test Prioritization Based on Program Changes. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, Vol. 1. 268–279. https://doi.org/10.1109/ICSE.2015.47

[34] S Stolberg. 2009. Enabling agile testing through continuous integration. In *Agile Conference, 2009. AGILE'09.* IEEE, 369–374.

[35] Per Erik Strandberg, Daniel Sundmark, Wasif Afzal, Thomas Ostrand, and Elaine Weyuker. 2016. Experience Report: Automated System Level Regression Test Prioritization Using Multiple Factors. In *Software Reliability Engineering (ISSRE), 2016 IEEE 27th International Symposium on.* IEEE, 12—-23.

[36] Richard S. Sutton and Andrew G. Barto. 1998. *Reinforcement Learning: An Introduction* (1st ed.). MIT press Cambridge. https://doi.org/10.1109/TNN.1998.712192

[37] Hado Van Hasselt and Marco A Wiering. 2007. Reinforcement learning in continuous action spaces. *Proceedings of the 2007 IEEE Symposium on Approximate Dynamic Programming and Reinforcement Learning, ADPRL 2007* (2007), 272–279. https://doi.org/10.1109/ADPRL.2007.368199

[38] Fred L Van Rossum, Guido and Drake Jr. 1995. *Python Reference Manual.* Technical Report. Amsterdam, The Netherlands, The Netherlands.

[39] Margus Veanes, Pritam Roy, and Colin Campbell. 2006. *Online Testing with Reinforcement Learning.* Springer Berlin Heidelberg, Berlin, Heidelberg, 240–253. https://doi.org/10.1007/11940197_16

[40] K R Walcott, M L Soffa, G M Kapfhammer, and R S Roos. 2006. Time-Aware Test Suite Prioritization. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis (ISSTA).* ACM, Portland, Maine, USA, 1–12.

[41] Farn Wang, Shun-Ching Yang, and Ya-Lan Yang. 2011. *Regression Testing Based on Neural Networks and Program Slicing Techniques.* Springer Berlin Heidelberg, Berlin, Heidelberg, 409–418. https://doi.org/10.1007/978-3-642-25658-5_50

[42] Lian Yu, Lei Xu, and Wei-Tek Tsai. 2010. *Time-Constrained Test Selection for Regression Testing.* Springer Berlin Heidelberg, Berlin, Heidelberg, 221–232. https://doi.org/10.1007/978-3-642-17313-4_23

[43] Lu Zhang, Shan-Shan Hou, Chao Guo, Tao Xie, and Hong Mei. 2009. Time-aware test-case prioritization using integer linear programming. *Proceedings of the eighteenth International Symposium on Software Testing and Analysis (ISSTA)* (2009), 213–224. https://doi.org/10.1145/1572272.1572297

[44] Tong Zhang. 2004. Solving large scale linear prediction problems using stochastic gradient descent algorithms. In *Proceedings of the twenty-first international conference on Machine learning.* ACM, 116.