UiO : **University of Oslo**

Helge Spieker

Software Testing in Continuous Integration with Machine Learning and Constraint Optimization

Thesis submitted for the degree of Philosophiae Doctor

Department of Informatics
The Faculty of Mathematics and Natural Sciences

Simula Research Laboratory

2020

© Helge Spieker, 2020

Series of dissertations submitted to the
The Faculty of Mathematics and Natural Sciences, University of Oslo
No. 2282

ISSN 1501-7710

All rights reserved. No part of this publication may be
reproduced or transmitted, in any form or by any means, without permission.

Cover: Hanne Baadsgaard Utigard.
Print production: Reprosentralen, University of Oslo.

Abstract

Software testing within a continuous integration process is a crucial task for modern software development. It has the goal to evaluate a software's functionality and be confident about its quality after recent changes and before the integration of new features or deployment. Further challenges are introduced when testing software for cyber-physical systems that integrate both software and dedicated hardware components, e.g. industrial robots or embedded devices.

This thesis investigates the usage of machine learning and constraint optimization to achieve the desired efficiency and to create an intelligent testing process. Specifically, we contribute new methodology for the test suite optimization process of test case prioritization, test case scheduling, and test case selection and assignment. All of these steps are relevant to decide which test cases are most relevant in a continuous integration cycle and when to execute them on which test agent. Within the context of the thesis, a test agent most often refers to a cyber-physical system such as an industrial robot. Nevertheless, all methods are also applicable to the more general use case of testing generic software systems without the specialization on cyber-physical systems.

For test case prioritization, we contribute a method using reinforcement learning, a machine learning technique, where the prioritization method learns to continuously adapt to failure characteristics and error distribution of the system-under-test.

In test case scheduling, we propose a constraint optimization model to schedule a suite of test cases over multiple test agents with shared resources, e.g. measurement devices that can only be used by a single agent at a time.

Our contribution in test case selection and assignment addresses a problem when the number of tests outgrows the number of agents. At each resource-constrained testing iteration only a subset of tests can be selected and only be executed on single test agents, but over the course of multiple iterations it is desirable to have all test cases executed on all compatible test agents. We introduce a method that maintains rotational diversity, i.e. test cases are subsequently selected and assigned such that all possible combinations are made over time while also focusing on the most relevant test cases as given by their priority in each individual cycle.

Finally, not directly integrated in the test suite optimization process, but related to testing in continuous integration, we propose an extension to metamorphic testing. Metamorphic Testing is a software testing paradigm which aims at using necessary properties of a system-under-test, called metamorphic relations, to either check its expected outputs, or to generate new test cases. Metamorphic Testing has been successful to test programs for which a full oracle is not available or to test programs for which there are uncertainties on

expected outputs such as learning systems. Our contribution is to introduce Adaptive Metamorphic Testing, which utilizes contextual bandits, a machine learning technique, to select one of the multiple metamorphic relations available for a program. By receiving feedback about previous test results, Adaptive Metamorphic Testing learns which metamorphic relations are likely to transform a source test case, such that it has higher chance to discover faults.

In conclusion, this thesis focuses on the domain of software testing in continuous integration with a focus on testing of cyber-physical systems. We evaluate the usage of data-driven machine learning techniques, such as reinforcement learning and contextual bandits, and exact, logic-based constraint optimization techniques for dedicated tasks within the testing process.

Preface

This thesis is submitted in partial fulfillment of the requirements for the degree of *Philosophiae Doctor* at the University of Oslo. The research presented here is conducted at Simula Research Laboratory, in the context of the Certus Centre for Software Validation and Verification (Certus SFI), and the Department of Informatics, University of Oslo. The primary supervisor was Arnaud Gotlieb (Simula Research Laboratory). The secondary supervisors were Morten Mossige (University of Stavanger, ABB Robotics Norway) and Magne Jørgensen (Simula Metropolitan, University of Oslo).

The thesis is a collection of four papers (Papers A-D) with Paper C being an extended version of a previous conference publication. These papers have been published or are currently under submission in international conferences and journals and are subject to the academic peer-review process. The papers are printed in chronological order of their creation. All papers have been the result of collaborations with great researchers, namely Arnaud Gotlieb (Paper A-D), Morten Mossige (Paper A-C), Mats Carlsson (Paper B), Dusica Marijan (Paper A), and Hein Meling (Paper B).

Acknowledgements

Completing this thesis would not have been possible without the tremendous support of several people. First and foremost, I would like to express my gratitude to my supervisors. Arnaud, you in particular were an outstanding mentor to me and I appreciate the motivation, effort and sympathy you brought with you. Working together with you helped me develop myself as a researcher and having your scientific and practical advice available made a huge contribution towards the progress of my PhD project. Morten, thank you for grounding the work that we are doing to the industrial reality and for offering us challenges that actually matter. I have learned a lot from our discussions.

Furthermore, my thanks go to Simula Research Laboratory and the Certus Centre for providing me with the opportunity to pursue a PhD at an excellent workplace. Being located next to the Oslofjord and within a great academic community allowed for the right environment, albeit relaxing and calm or intellectually stimulating, whenever needed. My colleagues within the software engineering department at Simula were a great inspiration for me and always had an open ear and helpful advise in any situation.

Finally, my special gratitude goes to my family, and in particular to Neele, for their encouragement and loving support through the PhD. Whenever I need help, you are always there. Thank you!

Helge Spieker
Oslo, March 2020

List of Papers

Paper A

Spieker, Helge and Gotlieb, Arnaud and Marijan, Dusica and Mossige, Morten; 'Reinforcement Learning for Automatic Test Case Prioritization and Selection in Continuous Integration'. In: *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)* (2017), pp. 12–22. DOI: 10.1145/3092703.3092709. Preprint: arXiv:1811.04122v1.

Paper B

Mossige, Morten and Gotlieb, Arnaud and Spieker, Helge and Meling, Hein and Carlsson, Mats; 'Time-aware Test Case Execution Scheduling for Cyber-Physical Systems'. In: *Principles and Practice of Constraint Programming (CP)* (2017), Lecture Notes in Computer Science, Vol. 10416, pp. 387–404.
DOI: 10.1007/978-3-319-66158-2_25. Preprint: arXiv:1902.04627v1.

Paper C

Spieker, Helge and Gotlieb, Arnaud and Mossige, Morten; 'Rotational Diversity in Multi-Cycle Assignment Problems'. In: *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence* Vol. 33 (2019), pp. 7724–7731.
DOI: 10.1609/aaai.v33i01.33017724. Preprint: arXiv:1811.03496v1.

The thesis contains an extended version "Multi-Cycle Assignment Problems with Rotational Diversity" that has been submitted to: *Journal of Artificial Intelligence Research* (May 2019). Preprint: arXiv:1811.03496v2.

Paper D

Spieker, Helge and Gotlieb, Arnaud; 'Adaptive Metamorphic Testing'. Revision submitted to: *Journal of Systems and Software* (March 2020).
Preprint: arXiv:1910.00262v2.

The above papers are self-contained and therefore some information might be repeated among them. Some acronyms and terminology may be used differently across papers.

Papers not included in the thesis. Furthermore, I have been involved with the following papers, which are not included in the thesis, as they focus on topics that are outside the main context of the PhD project:

Paper E

Gotlieb, Arnaud and Marijan, Dusica and Spieker, Helge; 'Stratified Constructive Disjunction and Negation in Constraint Programming'. In: *2018 IEEE 30th International Conference on Tools with Artificial Intelligence (ICTAI)* (2018), pp. 106–113. DOI: 10.1109/ICTAI.2018.00026. Preprint: arXiv:1811.03906v1.

Extended version "ITE: A Lightweight Implementation of Stratified Reasoning for Constructive Logical Operators" submitted to *International Journal on Artificial Intelligence Tools* (April 2019). Preprint: arXiv:1811.03906v2.

Paper F

Spieker, Helge; 'Towards Sequence-to-Sequence Reinforcement Learning for Constraint Solving with Constraint-Based Local Search'. In: *Proceedings of the AAAI Conference on Artificial Intelligence (Student Abstracts)* Vol. 33 (2019), pp. 10037-10038. DOI: 10.1609/aaai.v33i01.330110037.

The published papers are reprinted with permission from Association of Computing Machinery and Springer Nature. All rights reserved.

Contents

Preface iii

Acknowledgements v

List of Papers vii

Contents ix

I Summary 1

1 Introduction 3
- 1.1 Motivation . . . 3
- 1.2 Research Objectives . . . 4
- 1.3 Contributions . . . 5
- 1.4 Structure of the Thesis . . . 7

2 Test Suite Optimization for Continuous Integration Testing 9
- 2.1 Software Testing . . . 9
- 2.2 Continuous Integration . . . 10
- 2.3 Machine Learning . . . 10
- 2.4 Constraint Programming . . . 12
- 2.5 Test Suite Optimization . . . 13

3 Summary of Results 17
- 3.1 Paper A . . . 17
- 3.2 Paper B . . . 18
- 3.3 Paper C . . . 18
- 3.4 Paper D . . . 20

4 Discussion 21
- 4.1 Future Work . . . 21
- 4.2 Conclusion . . . 22

References 23

II Papers **27**

A Reinforcement Learning for Automatic Test Case Prioritization and Selection in Continuous Integration **29**
1 Introduction . . . 30
2 Formal Definitions . . . 32
3 The RETECS Method . . . 33
4 Experimental Evaluation . . . 39
5 Related Work . . . 47
6 Conclusion . . . 48
References . . . 49

B Time-aware Test Case Execution Scheduling for Cyber-Physical Systems **55**
1 Introduction . . . 56
2 Existing Solutions and Related Work . . . 57
3 Problem Modeling . . . 58
4 The TC-Sched Method . . . 62
5 Implementation and Exploitation . . . 64
6 Experimental Evaluation . . . 65
7 Conclusion . . . 69
References . . . 70

C Multi-Cycle Assignment Problems with Rotational Diversity **75**
1 Introduction . . . 76
2 Related Work . . . 77
3 Problem Description . . . 79
4 Maintaining Rotational Diversity . . . 81
5 Experimental Evaluation . . . 87
6 Conclusion . . . 98
References . . . 99

D Adaptive Metamorphic Testing with Contextual Bandits **105**
1 Introduction . . . 106
2 Background . . . 107
3 Adaptive Metamorphic Testing . . . 111
4 Application Scenarios of Adaptive Metamorphic Testing . . . 115
5 Experimental Evaluation . . . 116
6 Experimental Results . . . 122
7 Conclusion . . . 130
References . . . 131

Part I

Summary

Chapter 1

Introduction

1.1 Motivation

Industrial control systems cover hereby a wide range of possible actual systems. One prominent example of control systems are robotic systems interacting with their environment, but also distributed communication systems, such as conference systems, are an example. The methods presented in this thesis do not focus on a specific type of control system, but seek to find a general approach towards them. However, often we discuss of cyber-physical systems (CPS) such as industrial robots as the main case study application in our method, due to the collaboration with ABB Robotics Norway, which have a central role in the requirements analysis and data collection parts of the study. A main distinction in testing of cyber-physical systems versus standard software is the dependency on specific physical hardware, e.g. the robotic platform or an embedded device like in Internet-of-Things scenarios. This physical hardware needs to be included and therefore considered as a key component in the testing process.

Exhaustive software testing is a time-intensive task, even when using automation. Recent years moved the best practice in software development towards shorter development and release cycles, where at all time a releasable software state is available. After each change in the software, a test suite is run to check for possibly introduced errors in the system and complete functionality. If errors are found, the developer is noticed and can react quickly, reducing the time between discovery and elimination of an error. This process is called *continuous integration*.

Early negative feedback, i.e. early detection of errors, is desirable, as well as being able to trust a positive feedback. However, due to large test suite sizes, execution all tests is not necessary within reasonable time frames or resource amounts. Therefore, three main techniques to handle this problem have emerged, namely *test case prioritization*, *test case selection*, and *test case scheduling*. The first, *test case prioritization*, rates test case by their importance for the upcoming test run. The second, *test case selection* or *test case minimization*, deals with identification of unnecessary or redundant test cases, which can be skipped. The third, *test case scheduling*, assigns test cases to test machines to be executed at a specified time. By creating a dedicated schedule for test execution it sets their order to maximize resource usage and possibly early failure detection. It is important to note here, that the assignment and scheduling of test cases not only concerns the selection of the test cases but to the same degree considers which test agent is executing the test cases, as different test agents might represent another subset of system functionality. These three techniques are interconnected and each can make use of the others' results.

1.2 Research Objectives

The main research objective investigated in this PhD thesis is how software testing of complex software systems, such as cyber-physical systems, in continuous integration can be done efficiently. We investigate the usage of machine learning and constraint optimization to achieve the desired efficiency and to create an adaptive testing process. To further structure the goals of the thesis, we introduce two other research objectives in more detail.

The first research objective is to consider the generic testing process, i.e. *test case selection, prioritization and scheduling*, under consideration of safety and reliability of the tested system. It considers how test management can learn from historical test data to detect more important test cases and prioritize them in the test schedule. Special interest on these methodologies stems from the continuous integration perspective, which aims to automate the whole testing process as much as possible. Through automation instead of manual efforts, regular software builds and testing both increase the reliability of the overall process and allow to repeat the build-test-deploy cycle more often than with a manual process and its additional overhead. Without human intervention, the continuous integration environment allows the machine learning mechanism to fully take control and optimize the schedule for an efficient development cycle. Automation results of following research questions would be integrated into the mechanism for production usage. The thesis project aims at handling each of the three testing mechanisms individually and in combination.

While the previous research objective addresses the testing process and the management of test cases without detailed insight into each individual test case, the second research objective focuses on a lower-level approach to software testing. We consider the case where a system is tested with *metamorphic testing*. Metamorphic testing is a software testing paradigm, in which a source test case is transformed into a new follow-up test case for which the exact expected test outcome is not known, but a relation between the source and follow-up test case is available. By execution of the follow-up test case it can be confirmed whether the system-under-test behaves according to the so-called metamorphic relation. If the relation is violated, a failure in the system has been identified.

Metamorphic Testing has been successfully applied to test programs for which a full test oracle is not available or to test programs for which there are uncertainties on expected outputs such as machine learning systems. Examples for applications of metamorphic testing are search engines, where it is generally expected that additional keywords reduce the number of search results or learning systems, where the exact reaction is not known, but similar inputs should reveal similar results, given an adequate definition of similarity. Here, the thesis project aims to improve the process of metamorphic testing in the context of continuous integration, i.e. repeatedly testing the same system, but preferably with different generated test cases.

1.3 Contributions

1.3.1 Research Contributions

We have introduced new methods for each of the components of the testing process. We propose a combination of data-driven machine learning, especially reinforcement learning, which extracts domain knowledge from historical data and experiences, with logic-driven constraint optimization, which is modeled based on existing domain knowledge. Machine learning, on the one hand, is useful where the exact rules are not known and can be extracted or refined over time. Constraint optimization, on the other hand, is exact and follows a strict set of rules to find a solution for a problem instance that has many requirements and constraints to be valid and optimal. By combining both techniques, we benefit from the characteristics of the different problems in the testing process. Test case prioritization has only few strict constraints on the result, i.e. any assignment of priorities is valid, but only few are good, but it is not possible to determine the quality without execution of the tests.

Selection and scheduling, however, follow strict constraints on the available resources, e.g. test machines, time, or external global resources that have to be shared by multiple tests, and possible assignments between test cases and test machines. While rules could be learned from historical assignments, there are downsides to this approach. Historical schedules might only cover parts of the total set of constraints. The human engineers, who created previous schedules, follow best practices or simple heuristics to finish the schedules. Validation of the generated rules and the corresponding schedules and assignments is still necessary to avoid problems during their execution. Our approach is therefore to apply exact optimization techniques where the constraints are explicitly modeled from expert knowledge. A highly-optimized constraint solver can then find a good or even optimal solution for every new problem instance by performing a defined search strategy.

Both approaches are generally independent of each other, but are still integrated and connected. If we consider the whole testing process, the output of test case prioritization influences the selection and scheduling, because we want to focus on the most important test cases and are willing to exclude test cases with low priority. In the opposite direction, the scheduling and selection influences for which test cases we have feedback to further train and improve the test case prioritization technique. Thereby, the whole process is connected and forms a feedback loop over all its components. We will now highlight the connection between each of the components and the papers in the thesis.

In Paper A, we use reinforcement learning [1], a machine learning technique, to derive project-specific test case prioritization over time by observing failures. Our method receives feedback from the execution of previously highly prioritized test cases and adjusts its strategy such that test cases similar to those that failed will be ranked higher in future iterations. We have evaluated two ways to represent the strategy, either by using tabular functions or neural networks, and proposed five reward functions to formulate feedback for the agent. Experimental

results on industry datasets show that an agent can, without any initial domain knowledge, learn which test cases are more likely to fail and then assign those a higher priority.

Paper B addresses the question of creating a test schedule, that follows constraints on the usage of external equipment, e.g. measurement devices, and minimizes the time taken to run all test cases. The proposed method uses constraint optimization to model the problem as a constraint-based scheduling problem where test cases tasks to execute on a set of machines, e.g. robots. Our contributed model uses virtual dummy machines for each of the shared resources and introduces a custom search heuristic to quickly derive near-optimal solutions for a new problem instance.

In Paper C, the set of requirements and constraints changed in contrast to Paper B to focus on a different problem variant. First, the focus is not to generate the shortest schedule, but to select the most relevant test cases to fill the available time for testing. Second, the time-limited test execution does not allow to run each test case on every different test agent, which would be desirable to maximize the confidence in the test results. To mitigate this issue, we design a multi-cycle rotation mechanism, called *rotational diversity*, that rotates the assignment of all available test cases to their compatible test agents over the course of subsequent cycles. Our method extends the inner assignment problem with an additional rotation mechanism that combines both the rotation goal and the goal to select the most relevant test cases and is compatible with a wide range of applications outside the software testing domain.

Towards the second research objective, we have introduced a machine learning-based approach for the generation of follow-up test cases in metamorphic testing [2, 3] in Paper D. Our new method, named *adaptive metamorphic testing*, uses contextual bandits [4], a variant of reinforcement learning, to decide which metamorphic relation to apply for a source test case. The contextual bandit receives context information about the source test case and decides for one of multiple metamorphic relations to generate a follow-up test case. From its execution, a feedback is formulated and given to the bandit to update its strategy. We have applied our method to test deep learning systems for computer vision. Adaptive metamorphic testing showed to find the same failure rate and distribution than exhaustive testing while requiring less test executions, which can be costly or at least time-consuming. At the same time, its error discovery is stronger than pure random testing, which is the common approach to metamorphic testing, because it can adapt to the strengths of the available metamorphic relations for certain test cases.

1.3.2 Industrial Adaptation

Besides the publication of research articles, the developed methods were implemented in software for industrial usage. This software package, called SWMOD which historically stands for software modularity project, covers test case prioritization, test case selection and scheduling in a single tool. It is currently integrated in the automated testing pipeline at ABB Robotics in

Norway and other international locations, where it handles the selection of test cases and their assignment to a set of robot agents. These agents are either virtual controllers that simulate the physical robot execution or the actual full physical robot in an isolated testing environment. Everyday SWMOD organizes the testing of different software subsystems, including the custom paint control system, the robots' safety systems, and the underlying robot operating systems [5].

SWMOD is implemented in SICStus Prolog [6] and clp(FD) [7] and based on the methods and research results presented in Paper B and Paper C. Since 2016, SWMOD is an on-going collaboration project between Simula Research Laboratory and ABB Robotics Norway and under on-going development and maintenance.

Recently, the work on test case prioritization and selection using reinforcement learning (see Paper A) was inspiration for a new test management tool at Netflix[1]. Following our approach, their software prioritizes the most relevant test cases of a test suite which are then selected and distributed onto the actual test devices; the result of the test execution is again used to improve the decision for future iterations [8].

1.4 Structure of the Thesis

This thesis is a collection of papers and the remainder of the thesis is structured as two parts.

Summary (Part 1) In Chapter 2, we introduce the relevant background and context the thesis. Specifically, we discuss the application and use cases of machine learning and artificial intelligence techniques in the software testing domain. An overview of the key results of this thesis in Chapter 3, followed by a concluding discussion in Chapter 4 closes the first part.

Papers (Part 2) This part holds four research papers, that form the main content of the thesis. All papers have either been published or are currently under submission in international, peer-reviewed conferences and journals. We present the main findings of the papers in the thesis summary of the first part, in Chapter 3.

[1] See https://www.simula.no/news/netflix-takes-inspiration-certus-sfi-research-paper

Chapter 2

Test Suite Optimization for Continuous Integration Testing

In this chapter, we discuss the background and general problem setting for the thesis project in context of the existing literature. This context is summarized as the process of test suite optimization, which we will define below, in Continuous Integration (CI) testing. We discuss the application of artificial intelligence (AI) techniques for the different steps within the process, which have varying requirements and problem characteristics. Both data-driven machine learning (ML) methods and exact logic-driven optimization techniques, such as Constraint Programming (CP), are effective techniques to address specific subproblems in test suite optimization.

In the context of CI testing with a focus on cyber-physical systems, we structure the test suite optimization process into three tasks, that have to be repeatedly solved in each CI cycle. These three tasks are a) test case prioritization, b) test case selection, and c) test case assignment. Each of these tasks will be discussed in detail, but first the necessary technical background on software testing, machine learning, and constraint programming is presented.

2.1 Software Testing

We consider the task of automated software testing as the detection of faults in an underlying system-under-test (SUT). The SUT is often a software program, but can also represent a cyber-physical system, which combines software with a specific physical hardware platform. Examples for cyber-physical systems are robotic systems, internet of things (IoT) appliances, or generic embedded systems with external sensors or actuators. When discussing software testing, we refer specifically to automated software testing, where the setup, execution and evaluation of a test can be performed automatically without human intervention. Opposite to automated testing, there is the area of manual testing where the SUT is exercised manually or with software-assistance by a human tester according to a test protocol.

According to the IEEE Standard Glossary of Software Engineering Terminology testing is "an activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or environment." [9, p. 74]. In this activity, we usually execute the system using *test cases*. A *test case* is defined as "A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement" [9, p. 74].

The collection of one or more test cases is named as a *test suite*. We choose a generic definition of a test suite as it can have several, more specific meanings. A test suite can be dedicated for a certain component, for a certain functionality of a system, focused on only test cases with short execution times, or generally contain all test cases for a system.

2.2 Continuous Integration

Continuous Integration (CI) is a cost-effective software engineering practice with wide adoption in the industry [10, 11, 12]. CI automates the repeated steps of software compilation, build, test, and deployment, involving all recent changes made to the software. The goal is to find faults and regressions in the software through frequent, i.e. almost continuous, integration of changes. One iteration of all steps in the CI process is named a *cycle*. CI cycles can be triggered upon every single change, using a small test suite of short tests, or at fixed intervals, for example, daily CI cycles that use the night for extensive testing.

By having limited time windows until a software change is integrated into the overall systems, it is easier to detect faults and their causes. This reduces the cost of debugging and bug fixing in later stages of the software development process [12].

However, the effective use of CI also creates additional challenges. CI requires the organization to make their compilation, build, test and deploy tasks automatically executable without human intervention. Generally, it is also desirable to control the duration of a CI cycle, such that the feedback to the developers about the compatibility of their changes is not delayed too much, or such that the time and resources available for the CI cycle are used as good as possible. For the context of this discussion, we focus exclusively on the testing step in CI and its resource requirements. We discuss the testing-related challenges in CI together with the test suite optimization process in Section 2.5.

2.3 Machine Learning

Machine learning (ML) algorithms are data-driven statistical method that derive a set of rules from data. How this data needs to be structured and what kind of rules are derived depends on the category and type of ML algorithm.

ML algorithms are broadly grouped into three categories: In *supervised learning*, the first category, a ML model is trained from a set of example inputs and outputs. During training the model learns to approximate the mapping between model inputs and the expected outputs. Examples for supervised ML models are classification models, e.g. to identify an object on an image, or regression models, for example, to predict the price of a house given a set of attributes.

The second category is *unsupervised learning*, where only model inputs are given, but without labeling of the expected output. These types of models are often used to identify similarities or differences in the data. Example methods

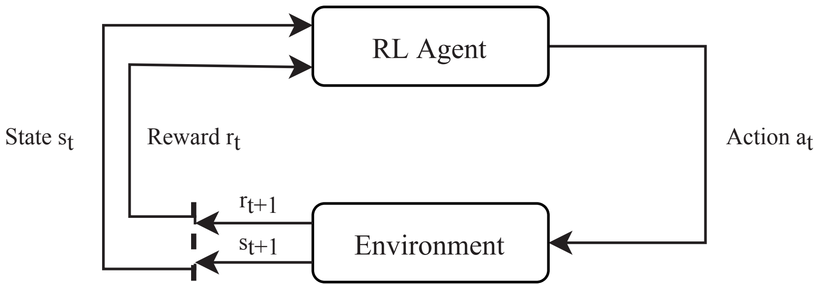


Figure 2.1: The agent-environment interaction in reinforcement learning (adapted from [1])

are clustering, where similar data points grouped into distinct clusters, or compressing data for dimensionality reduction, i.e. identifying the most relevant attributes to reduce the amount of data for further processing.

The third category is *reinforcement learning* where the model continuously learns to interact with an external environment through the observation of the outside state and issuing corresponding actions. Examples are game-playing agents that learn to play video [13] or board games [14, 15] through trial-and-error, learning robots [16, 17] or content placement agents that learn which news articles or advertisements to recommend for certain users [4, 18]. We will further introduce reinforcement learning in the following section as it is the most relevant category for this thesis.

2.3.1 Reinforcement Learning

In the reinforcement learning (RL) setting (see Figure 2.1) an agent observes the state s_t of the environment at time t and picks an action a_t according to its internal policy. For this action, it later receives a reward r_t from the environment [1]. After an action has been selected, it is executed within the environment and a new state s_{t+1} is received. The policy is represented by a ML model that is continuously trained via the tuple of state, action, received reward and follow-up state $\langle s_t, a_t, r_t, s_{t+1} \rangle$.

RL is thereby a sequential decision making algorithm where an action in one state influences which state is observed next. This is an important characteristic as the agents learns how its actions influence the environment and how they lead to states that can yield high rewards, e.g. winning a game or identifying faults in a software system.

The goal of the RL agent is to maximize the sum of total rewards over all subsequent episodes, that is, at each state the action for which the highest future reward is expected should be chosen. This is not necessarily the action with the highest immediate reward, but instead the trajectories of multiple future decisions has to be considered

A specific variant of RL are contextual bandits [4, 19] with the main distinction that an agents actions do not affect the environment such that it influences the next. This means each state is independent of the previous state regardless the chosen action, unlike in the previously explained RL setting where the earlier state and action influence future states. Scenarios with this setting include, for example, news placement on websites. Independent of which news article is chosen for a visitor, it does not have an influence on who visits the website next.

Otherwise, contextual bandits work similar to RL agents. They observe the state of the environment, which is here called context, and select an action, here called arm in correspondence to multi-armed bandits. Afterwards a reward is received, which is here often called either feedback or cost.

Due to the simpler setting of not being designed for sequential decision making, contextual bandits have been more in focus of researchers before and the underlying theory is more advanced than the theory of RL. For a dedicated introduction to RL and contextual bandits, we refer to the books by Sutton and Barto [1] respectively Lattimore and Szepesvari [19].

2.4 Constraint Programming

Constraint Programming (CP) [20] is a programming paradigm for solving combinatorial problems with additional constraints. Programs in this paradigm are called *constraint satisfaction problems* (CSPs).

CSPs are defined via a set of variables V, each with a domain of possible values D, and a set of constraints C that form relations between the variables. The goal is to find an assignment of values to all variables such that all constraints are satisfied, i.e. a *satisfiable* assignment. Finding this assignment is referred to as constraint reasoning or solving the constraint satisfaction problem [21]. Constraint reasoning is usually performed via a dedicated constraint solver, such as SICStus Prolog [6] and its clp(FD) library [7] or Gecode [22]. These solvers implement dedicated filtering techniques and search heuristics to efficiently remove infeasible domain values and derive a solution for the constraint system. Constraint satisfaction is sufficient when the problem only has a single solution or if there is no preference between one of the multiple solutions of the problem. In other cases, where the solutions are of different quality that can be quantified, constraint optimization can be applied. *Constraint optimization problems* (COPs) extend CSPs by introducing an additional objective function over the variables. In this scenario, not only a satisfying solution has to be found, but also the objective function has to be either minimized or maximized. Examples for COPs are scheduling problems, where the assignment from tasks to workers has to satisfy several constraints, e.g. each worker can only be used by one task at a time, and at the same time the total duration of the schedule has to be minimized.

Using CP for combinatorial problems consists of modeling the problem in terms of constraints, which requires the availability of domain knowledge either through a domain expert or from thorough analysis of the problem. The model

describes how a solution to the problem looks like, i.e. which constraints does it follow and how does it relate to the specific instance of the problem that is to be solved. From this model, the solver tries to find a solution using variant of branch-and-bound tree search. To further improve the search process, a problem-specific search heuristic can be defined to instrument the solver how to traverse the search tree to reach satisfiable and high-quality solutions earlier.

One particular strength of CP for solving combinatorial problems is the availability of global constraints. Global constraints are building blocks that can be used to solve common subproblems in combinatorial modeling and for which efficient filtering techniques are available. Using these global constraints rather than the naive modeling of the subproblems using a number of other, simpler constraints enables faster solving and reasoning by the solver and makes CP efficient. An example for a global constraint is `alldifferent`. This constraint restricts all involved variables, over which it is defined, to take a different value. Naively this would be modeled by adding inequality constraints between all variables, but the intention behind this construct would not be known to the solver and no optimization can take place.

However, CP is not only the only suitable approach for combinatorial problems. Alternatives include are satisfiability (SAT) and satisfiability modulo theories (SMT) solvers, as well as mixed-integer programming (MIP), which is commonly used in mathematical optimization and operations research. An important distinction has to be made regarding the possible values of a domain. Within the context of this thesis, we focus only on the *finite domain* that holds a discrete set of values, for example integers, as opposed to the real-valued domain of floating point values. For a further reading on CP and its variants, we refer to the book by Rossi, Beek, and Walsh [21].

2.5 Test Suite Optimization

The test suite optimization process aims to provide tools and methods for the efficient usage of the limited time available for testing in CI. Traditionally, for pure software systems, this includes selecting subsets of tests, their prioritization to establish an order, and their execution. This assumes that the execution environment is generic or decoupled from the software, and the test suite can be executed on any test environment. For testing of cyber-physical systems it is further relevant to include the test environment into the test suite optimization process as its resources need to be managed and considered, too. In 2012, Yoo and Harman published a major review on test minimization, selection and prioritization, and we refer the interested reader to this survey for an in-depth overview of the earlier literature [23].

2.5.1 Test Case Prioritization

Test case prioritization is the ordering of test cases for high effectiveness [24]. The effectiveness is expressed such that failing test cases should be executed

before passing test cases to identify faults in the SUT as soon as possible. For the sequential execution of the test cases, the test case prioritization method can directly sort the test cases. More formally, the test case prioritization method assigns a rank, e.g. a numerical priority, to each test case, but a downstream task decides on the actual order, i.e. sorting in the sequential case or more dedicated assignment and scheduling in more complex cases.

Test case prioritization has been extensively studied in the research community since the seminal works by Rothermel, Untch, Chu, and Harrold [24] and Elbaum, Malishevsky, and Rothermel [25] and the relevance of the problem is also addressed in industrial settings and use cases, for example, to name a few published case studies, at Cisco [26], Google [27, 28, 29], Salesforce.com [30], or Westermo Research and Development AB [31, 32].

2.5.2 Test Case Selection

Once a priority has been assigned, the test case selection step picks the most relevant test cases for actual execution. Selection is necessary when the resources for test execution are limited, e.g. only a fixed time period is allowed for test execution. For the case of sequential execution, the selection method can choose the most highly-prioritized test cases that fill the available resources. Test case selection is also referred to as test suite minimization [33] or reduction [34, 35, 36] as it reduces the number of test cases from the full initial test suite to the test suite that is actually executed. More complex approaches to test case selection can consider additional constraints and requirements on the resulting test suite like demanding certain coverage criteria to be fulfilled to cover the whole system even though the risk of failure in some systems is lower prioritized than in other areas, or by considering more resource constraints than just the available time, such as compatibility to available test agents or dependencies on external devices.

Test case selection is often combined with test case prioritization and sometimes seen as a combined technique. Nevertheless, both tasks are different and might entail a variation of requirements and constraints. Within the context of test suite optimization, which could also be seen as a single task from a higher level, these tasks are separated.

2.5.3 Test Case Scheduling

The third subtask, test case scheduling, is again linked with the selection of test cases. Scheduling describes the generation of a test execution plan that defines which test case is executed at which time on which test agent. It thereby differs from test case selection, which only picks a subset, by also making an assignment to an execution time and location. The scheduling model captures the constraints of the individual assignment between test cases and test agents as well as potential constraints on the execution orders of test cases. While test cases should generally be independent of each other, it can be necessary to group certain kinds of test cases to avoid costly setup times, or to avoid certain groupings of test cases because they rely on similar external resources. These

constraints need to be developed together with domain experts, e.g. the quality engineers developing the tests, and need to be formalized and stored as metadata for the test cases.

The task of scheduling is not specific to software testing, but has a long history as its own area of research [37, 38, 39, 40] and many application domains, e.g. machine scheduling, project scheduling, or sports league scheduling. For testing purposes, we can rely on many of the developed techniques and best practices, e.g. from constraint programming [41, 42, 43] and adapt them to the specific constraints of the domain [44].

Chapter 3

Summary of Results

In this chapter, we present a summary and the key contributions for each of the papers included in this thesis.

The first three papers address the test optimization process for repeated testing in Continuous Integration environments, consisting of test case prioritization, selection, and scheduling.. The fourth paper focuses on the application of machine learning to generate follow-up test cases for an existing test suite, which is complementary to the previous methods, but not specifically focused on the optimization of the testing process.

3.1 Paper A

'*Reinforcement Learning for Automatic Test Case Prioritization and Selection in Continuous Integration*'. *Spieker, Helge and Gotlieb, Arnaud and Marijan, Dusica and Mossige, Morten. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA) (2017), pp. 12–22. DOI: 10.1145/3092703.3092709.*

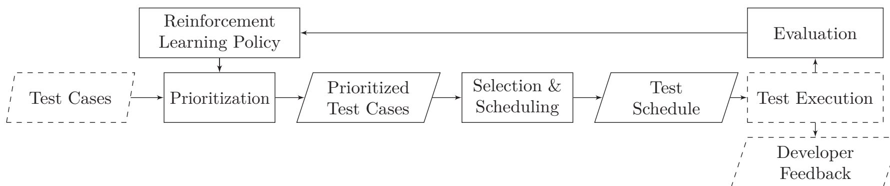


Figure 3.1: Testing in CI process: Retecs uses test execution results for learning test case prioritization (solid boxes: Included in Retecs, dashed boxes: Interfaces to the CI environment)

In this paper, we explore the usage of reinforcement learning (RL) for prioritizing test cases in a Continuous Integration (CI) environment (see Figure 3.1). Testing in a CI environment is characterized by its repeated and frequent execution, which accumulates historical data and insights about the error distribution and its characteristics in a software project. By using RL, our method, named Retecs (Reinforced Test Case Selection), captures this error distribution from historical test executions and learns to prioritize likely-to-fail test cases higher.

Still, the design of a RL method requires several decisions, including the selection of a memory representation, the design of an action space, and the selection of a reward function. Our study evaluates different choices for these components and compares them empirically on three datasets: one dataset of

test executions from Google, and two datasets from our industrial partner ABB Robotics Norway. The experiments show, that RETECS is capable of learning project-specific test case prioritization strategies and improves its performance over time. The implementation of RETECS and our datasets are available at: https://bitbucket.org/helges/retecs and https://bitbucket.org/HelgeS/atcs-data

3.2 Paper B

'Time-aware Test Case Execution Scheduling for Cyber-Physical Systems'. Mossige, Morten and Gotlieb, Arnaud and Spieker, Helge and Meling, Hein and Carlsson, Mats. In: Principles and Practice of Constraint Programming (CP) (2017), Lecture Notes in Computer Science, Vol. 10416, pp. 387–404. DOI: 10.1007/978-3-319-66158-2_25

When testing large-scale systems with hundreds of test cases in continuous integration, it is crucial to minimize the round-trip time, that is, the time from when a source code change is committed until the test results are reported back to the developer. To this end, scheduling as many test case executions as possible in the minimum amount of time is essential to increasing the effectiveness of continuous integration.

In contrast to Paper C, this paper is concerned with the minimization of a test schedule, i.e. how to assign tests such that the required time is minimized, whereas the previous work aims to maximize the value of the assignment under resource constraints. Both scenarios have practical applications in software testing, depending on the actual testing setup and which constraints apply on the environment.

More specifically, this paper introduces TC-Sched, a time-aware method for test execution scheduling on multiple machines with constraints on accessible resources, such as measurement devices or network equipment. The method uses as input a test suite, a set of machines, and a set of shared resources and produces an execution schedule. The schedule guarantees that each test will be executed once and minimizes the round-trip time.

TC-Sched has undergone extensive experimental evaluation using generated test suites derived from existing industrial test suites augmented with randomly-selected values. Our results provide evidence that TC-Sched conducts effective test execution scheduling and is suitable for deployment in continuous integration. In particular, we show that automatic optimal scheduling of 500 test cases over 100 machines is reachable in less than 4 minutes for 96.6 % of the instances.

3.3 Paper C

'Rotational Diversity in Multi-Cycle Assignment Problems'. Spieker, Helge and Gotlieb, Arnaud and Mossige, Morten. In: Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence, Vol. 33 (2019), pp. 7724–7731. DOI: 10.1609/aaai.v33i01.33017724

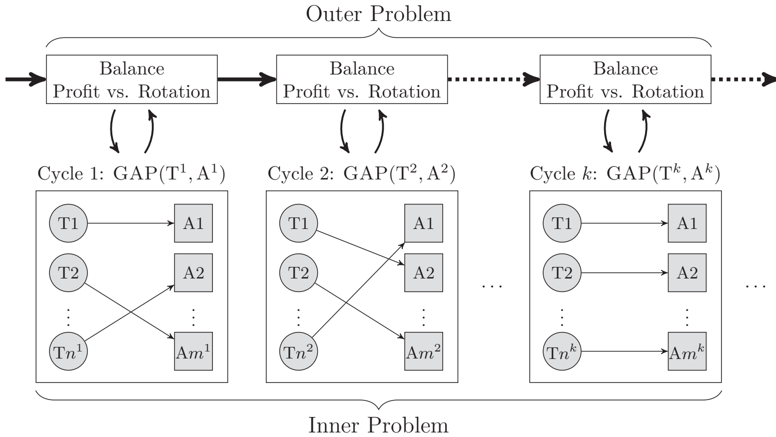


Figure 3.2: Multi-Cycle Assignment Problem: At each cycle an independent assignment problem is solved while maintaining rotational diversity over multiple cycles. Our contribution introduces metrics and strategies to achieve frequent rotation while maintaining high assignment quality in each cycle.

Testing of cyber-physical systems imposes additional challenges over testing of stand-alone software. Ideally, all test cases should be run on all physical test agents in every testing iteration, but this is often not possible in practice. For example, the availability of test cases and test agents can not be planned or assured in advance, but changes unforeseeable. Another challenge, that we address in this paper, is the trade-off between having limited resources for testing available and the desire to maximize the coverage of test cases to test agents. Therefore, the goal is to select a subset of the test cases that maximize an objective function, in this case the sum of test case priorities, while following resource constraints such as the available time for test execution.

We have developed a method to achieve rotational diversity in multi-cycle, i.e. iterative, testing processes. The problem of assigning test cases to test agents is formulated as a general assignment problem, one of the most general and well-studied, but nevertheless challenging problems in computer science. Our method assigns the most highly prioritized test cases, as defined by an external process (e.g. from Paper A), in one iteration and manages to rotate the assignment from test cases to test agents over the course of multiple cycles. To balance rotation and maximization of profits, i.e. assigned test case priorities, the problem is split into an outer problem that considers long-term decisions and an inner problem that optimizes the assignment in each individual iteration (see Figure 3.2) Using this technique, we achieve to maintain a frequent rotation while only reducing the quality, as measured by the priority of test cases, by a small margin.

More technically, we introduce a metric to quantify the *affinity* between a test case and an agent and propose multiple strategies to consider these affinities

in the optimization process. Besides the application for test case selection and assignment, we propose our solution for rotational diversity in a general setting for assignment problems and show results for the multi-cycle multiple knapsack problem, a variant of the general assignment problem with additional constraints.

The version of the paper that is printed in the thesis is an extension of the original conference paper. In the extended version, we further investigate methods to balance priority optimization and rotation and include an additional application, the multi-cycle multiple subset sum problem, for the experimental evaluation. One newly introduced approach is to limit the possible assignments of test cases to agents artificially and thereby enforce assignments that are overdue. In our experiments, we observe that this limitation increases the ability to maintain rotational diversity, but comes with a trade-off in priority optimization. Our implementation and data generators are available at: https://github.com/HelgeS/mcap_rotational_diversity

3.4 Paper D

'Adaptive Metamorphic Testing'. Spieker, Helge and Gotlieb, Arnaud. Revision submitted to: Journal of Systems and Software (2019). Preprint: arXiv:1910.00262

Metamorphic testing is a testing paradigm, in which a source test case is transformed into a new follow-up test case for which the exact expected outcome is unknown, but a relation between the source and follow-up test case is available. By execution of the follow-up test case it can be confirmed whether the system-under-test behaves according to the so-called metamorphic relation. If the relation is violated, a failure in the system has been identified. Metamorphic testing thereby addresses the oracle problem in software testing, where it is impossible or difficult to know the exact system output for a test case. Examples for successful applications of metamorphic testing are search engines, where it is generally expected that additional keywords reduce the number of search results or learning systems, where the exact reaction is unknown, but similar inputs should reveal similar results, given an adequate definition of similarity.

Our new method, named *adaptive metamorphic testing*, uses contextual bandits, a variant of reinforcement learning, to decide which metamorphic relation to apply for a source test case. The contextual bandit receives context information about the source test case and decides for one of multiple metamorphic relations to generate a follow-up test case. From its execution, a feedback is formulated and given to the bandit to update its strategy.

We have applied our method to test deep learning systems for computer vision. Adaptive metamorphic testing showed to find the same failure rate and distribution than exhaustive testing while requiring less test executions, which can be costly or at least time-consuming. At the same time, it's error discovery is stronger than pure random testing, because it can adapt to the strengths of the available metamorphic relations for certain test cases. Our implementation and datasets are available at: https://github.com/HelgeS/tetraband

Chapter 4

Discussion

This thesis investigates automatic management and optimization of software testing in Continuous Integration (CI) environments with a focus on cyber-physical systems. Methods for test case prioritization, test case scheduling, test case selection and assignment, and generation of new test cases are proposed, which utilize machine learning and constraint optimization.

Machine learning, in our context reinforcement learning and contextual bandits, requires the availability of historical data and an external feedback signal are available for learning. It has shown to be beneficial in scenarios where an exact heuristic or algorithm can not be explicitly modeled without in-depth domain knowledge and analysis, i.e. test case prioritization and the generation of follow-up test case in metamorphic testing.

In settings with strict constraints, where a set of limited resources has to utilized for testing, we applied constraint optimization, an exact combinatorial optimization technique. Our applications are test case scheduling, i.e. assigning test cases to a set of test agents while minimizing the total execution time required, and test case selection and assignment, i.e. selecting the subset of test cases that best make use of a given time period. Both applications have strict constraints on satisfying and acceptable solutions, while requiring to be repeatedly solved with different inputs due to variations in test cases and test agents from cycle to cycle.

The thesis bridges multiple subfields in computer science and makes contribution both to the application domain, software testing, as well as the technical domain of constraint optimization respectively constraint programming, and artificial intelligence in general. This is reflected in the chosen publication venues of the included papers and the nature of the technical contributions made, which are, e.g. in the case of rotational diversity (see Paper C), also wider applicable than only the software testing area.

4.1 Future Work

As research is rarely complete, there are also directions for future research in this case. While the work on test case prioritization using reinforcement learning (see Paper A) shows promising results, there are further steps that can be taken. Our method uses a lightweight set of historical test case metadata for prioritization. Further work should also consider the actual changes made and the affected components of the system-under-test. At the same time, this also increases the complexity of the method in each CI cycle - for change analysis - and requires source code access, while our current approach is easier to integrate in any existing CI environment.

For our work on test case scheduling (see Paper B) as well as test case selection and assignment (see Paper C), a technical challenge for future work is to introduce some of the experiences made with learning directly into the optimization model. So far, in each CI cycle, the optimization model starts from scratch using the test case and test agent data to create a solution. Still, in a CI environment the actual tasks to be solved in each cycle are often similar to previous cycles. Future work on re-using the previous results should allow for potential speedups in future cycles. Another challenge

Regarding the usage of machine learning in metamorphic testing (see Paper D), future work should extend beyond the selection among a set of metamorphic relations to generate follow-up test cases to the discovery or to the approximating of actual metamorphic relations. Currently, our setup still requires the set of metamorphic relations to be available for application of adaptive metamorphic testing, but, given a sufficient observation of the system behavior, machine learning could be used to identify and learn new metamorphic relations. These metamorphic relations can then be used for the normal metamorphic testing process or be inspected by a domain expert whether these metamorphic relations should hold for a system or if a faulty relation is discovered.

4.2 Conclusion

In conclusion, we have presented methods for the automation of the test suite optimization process in Continuous Integration testing, that individually, yet integrated with each other can make decisions for a testing schedule under consideration of historical test information and current resource constraints. Our methods focus on testing of cyber-physical systems and use machine learning and constraint optimization. The results of the thesis present an effective approach for the whole test suite optimization pipeline and can be applied in a variety of project environments, either as a whole, or through the modular nature of test suite optimization, in individual aspects. Especially for environments with physical agents or dedicated requirements on the execution environments for the test cases, test case selection and assignment as well as scheduling enable the automatic creation of test scenarios.

References

[1] Sutton, R. S. and Barto, A. G. *Reinforcement Learning: An Introduction*. 2nd. MIT Press, 2018.

[2] Chen, T., Cheung, S., and Yiu, S. *Metamorphic Testing: A New Approach for Generating Next Test Cases*. Technical Report HKUST-CS98-01. Hong Kong: Department of Computer Science, Hong Kong University of Science and Technology, 1998.

[3] Segura, S., Fraser, G., Sanchez, A. B., and Ruiz-Cortes, A. "A Survey on Metamorphic Testing". In: *IEEE Transactions on Software Engineering* vol. 42, no. 9 (2016), pp. 805–824.

[4] Langford, J. and Zhang, T. "The Epoch-Greedy Algorithm for Multi-Armed Bandits with Side Information". In: *Advances in Neural Information Processing Systems 20 (NIPS 2007)*. 2007, pp. 817–824.

[5] Mossige, M. *A Brief History of CI-Based Testing at ABB Robotics, Bryne*. 14th Certus User Partner Workshop (UPW), Larvik, Norway, Sept. 2019.

[6] Carlsson, M. et al. *SICStus Prolog User's Manual, Release 4.5.1*. RISE SICS AB, 2019.

[7] Carlsson, M., Ottosson, G., and Carlson, B. "An Open-Ended Finite Domain Constraint Solver". In: *Proc. of the 9th Int. Symp. on Prog. Languages, Implementations, Logics, and Programs (PLILP '97)*. 1997, pp. 191–206.

[8] Netflix Technology Blog. *Lerner — Using RL Agents for Test Case Scheduling*. en. https://netflixtechblog.com/lerner-using-rl-agents-for-test-case-scheduling-3e0686211198. May 2019.

[9] Standards Coordinating Committee. "IEEE Standard Glossary of Software Engineering Terminology". In: *IEEE Std 610.12-1990* (Dec. 1990), pp. 1–84.

[10] Booch, G. *Object Oriented Design: With Applications*. en. Benjamin/Cummings Pub., 1991.

[11] Fowler, M. and Foemmel, M. *Continuous Integration*. 2006.

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

[13] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. a, Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. "Human-Level Control through Deep Reinforcement Learning". In: *Nature* vol. 518, no. 7540 (2015), pp. 529–533.

[14] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. "Mastering the Game of Go with Deep Neural Networks and Tree Search". In: *Nature* vol. 529, no. 7587 (2016), pp. 484–489.

[15] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., and Hassabis, D. "A General Reinforcement Learning Algorithm That Masters Chess, Shogi, and Go through Self-Play". en. In: *Science* vol. 362, no. 6419 (2018), pp. 1140–1144.

[16] Kober, J., Bagnell, J. A., and Peters, J. "Reinforcement Learning in Robotics: A Survey". en. In: *The International Journal of Robotics Research* vol. 32, no. 11 (Sept. 2013), pp. 1238–1274.

[17] Kormushev, P., Calinon, S., and Caldwell, D. "Reinforcement Learning in Robotics: Applications and Real-World Challenges". en. In: *Robotics* vol. 2, no. 3 (July 2013), pp. 122–148.

[18] Tang, L., Rosales, R., Singh, A., and Agarwal, D. "Automatic Ad Format Selection via Contextual Bandits". In: *Proceedings of the 22nd ACM International Conference on Conference on Information & Knowledge Management*. 2013, pp. 1587–1594.

[19] Lattimore, T. and Szepesvari, C. *Bandit Algorithms*. en. Vol. Revision: 8b22b8b6131c37e388d5e3b2eecf0b4ff5d7db92. https://banditalgs.com/, 2019.

[20] Van Hentenryck, P. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.

[21] Rossi, F., Beek, P. V., and Walsh, T. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. New York, USA: Elsevier Science Inc., 2006.

[22] Schulte, C., Tack, G., and Lagerkvist, M. Z. *Modeling and Programming with Gecode*. 2018.

[23] Yoo, S. and Harman, M. "Regression Testing Minimization, Selection and Prioritization: A Survey". In: *Software Testing, Verification and Reliability* vol. 22, no. 2 (2012), pp. 67–120.

[24] Rothermel, G., Untch, R. H., Chu, C., and Harrold, M. J. "Prioritizing Test Cases For Regression Testing". In: *IEEE Transactions on Software Engineering* vol. 27, no. 10 (2001), pp. 929–948.

[25] Elbaum, S., Malishevsky, A., and Rothermel, G. "Test Case Prioritization: A Family of Empirical Studies". In: *IEEE Transactions on Software Engineering* vol. 28, no. 2 (2002), pp. 159–182.

[26] Marijan, D., Gotlieb, A., and Sen, S. "Test Case Prioritization for Continuous Regression Testing: An Industrial Case Study". In: *Proc. of Int. Conf. on Soft. Maintenance (ICSM'13), Industry Track*. 2013, pp. 540–543.

[27] Elbaum, S., Rothermel, G., and Penix, J. "Techniques for Improving Regression Testing in Continuous Integration Development Environments". In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*. ACM, 2014, pp. 235–245.

[28] Memon, A., Zebao Gao, Bao Nguyen, Dhanda, S., Nickell, E., Siemborski, R., and Micco, J. "Taming Google-Scale Continuous Testing". In: *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, May 2017, pp. 233–242.

[29] Leong, C., Singh, A., Papadakis, M., Traon, Y. L., and Micco, J. "Assessing Transition-Based Test Selection Algorithms at Google". In: *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*. ICSE-SEIP '10. Montreal, Quebec, Canada: IEEE Press, 2019, pp. 101–110.

[30] Busjaeger, B. and Xie, T. "Learning for Test Prioritization: An Industrial Case Study". In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 975–980.

[31] Strandberg, P. E., Sundmark, D., Afzal, W., Ostrand, T. J., and Weyuker, E. J. "Experience Report: Automated System Level Regression Test Prioritization Using Multiple Factors". In: *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2016, pp. 12–23.

[32] Strandberg, P. E., Afzal, W., Ostrand, T. J., Weyuker, E. J., and Sundmark, D. "Automated System-Level Regression Test Prioritization in a Nutshell". In: *IEEE Software* vol. 34, no. 4 (2017), pp. 30–37.

[33] Jabbarvand, R., Sadeghi, A., Bagheri, H., and Malek, S. "Energy-Aware Test-Suite Minimization for Android Apps". In: *Proceedings of the 25th International Symposium on Software Testing and Analysis - ISSTA 2016* (2016), pp. 425–436.

[34] Wang, R., Lu, Y., and Qu, B. "Empirical Study of the Effects of Different Profiles on Regression Test Case Reduction". In: *IET Software* vol. 9, no. 2 (2015), pp. 29–38.

[35] Felbinger, H. and Wotawa, F. "Test-Suite Reduction Does Not Necessarily Require Executing The Program Under Test". In: *QRS-C* (2016).

[36] Shi, A., Gyori, A., Mahmood, S., Zhao, P., and Marinov, D. "Evaluating Test-Suite Reduction in Real Software Evolution". In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis - ISSTA 2018*, no. 1 (2018), pp. 84–94.

[37] Blazewicz, J., Lenstra, J. K., and Kan, A. R. "Scheduling Subject to Resource Constraints: Classification and Complexity". In: *Discrete Applied Mathematics* vol. 5, no. 1 (1983), pp. 11–24.

[38] Morihara, I., Ibaraki, T., and Hasegawa, T. “Bin Packing and Multiprocessor Scheduling Problems with Side Constraint on Job Types”. In: *Discrete Applied Mathematics* vol. 6 (1983), pp. 173–191.

[39] Brucker, P. and Schlie, R. “Job-Shop Scheduling with Multi-Purpose Machines”. In: *Computing* vol. 45, no. 4 (1990), pp. 369–375.

[40] Hartmann, S. and Briskorn, D. “A Survey of Variants and Extensions of the Resource-Constrained Project Scheduling Problem”. In: *European Journal of Operational Research* vol. 207, no. 1 (2010), pp. 1–14.

[41] Baptiste, P., Le Pape, C., and Nuijten, W. *Constraint-Based Scheduling: Applying Constraint Programming to Scheduling Problems.* 1st ed. International Series in Operations Research & Management Science 39. Springer US, 2001.

[42] Beldiceanu, N. and Carlsson, M. “A New Multi-Resource Cumulatives Constraint With Negative Heights”. In: *Principles and Practice of Constraint Prog. (CP’02).* 2002, pp. 63–79.

[43] Beck, J. C., Feng, T. K., and Watson, J. P. “Combining Constraint Programming and Local Search for Job-Shop Scheduling”. In: *INFORMS Journal on Computing* vol. 23, no. 1 (2011), pp. 1–14.

[44] Hebrard, E., Huguet, M. J., Veysseire, D., Sauvan, L. B., and Cabon, B. “Constraint Programming for Planning Test Campaigns of Communications Satellites”. In: *Constraints* vol. 22, no. 1 (2017), pp. 73–89.

Part II

Papers

Paper A

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

Helge Spieker, Arnaud Gotlieb, Dusica Marijan, Morten Mossige

Published in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017)*, 2017, ACM, New York, NY, USA, pp. 12–22. DOI: 10.1145/3092703.3092709. Preprint: arXiv:1811.04122v1.

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.

©ACM Association for Computing Machinery, 2017. The layout has been revised.

1 Introduction

Context. Continuous Integration (CI) is a cost-effective software development practice commonly used in industry [1, 2] 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 [3, 4].

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 [5]. 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 [6]. 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 [7]. Testing in CI also means 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 [7, 8], 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 $\{t_1, \ldots, t_n\}$ can be estimated with any cumulative function (e.g., sum, max, average, etc.) over $q_i(t_1), \ldots q_i(t_n)$, 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 [9]: *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 $\mathcal{TS}_1, \ldots, \mathcal{TS}_{i-1}$ 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.

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 (subsection 3.1), then discusses test case scheduling in one CI cycle (subsection 3.2). Finally, integration of the method within a CI process is presented (subsection 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

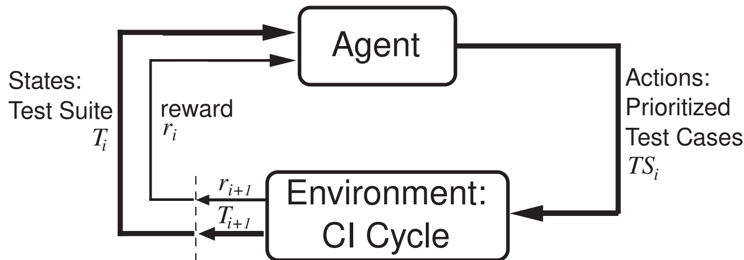


Figure A.1: Interaction of Agent and Environment (adapted from [10, Fig 3.1])

introduction can be found in [10]. 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 A.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 [6]. 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 [11]. 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 subsection 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*

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

In the first reward function (A.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*

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

The second reward function (A.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 (subsection 3.2) this automatically leads to an earlier execution.

Definition 3.3. *Time-ranked Reward*

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

The third reward function (A.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 (A.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* [10]. 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 [12]. 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.

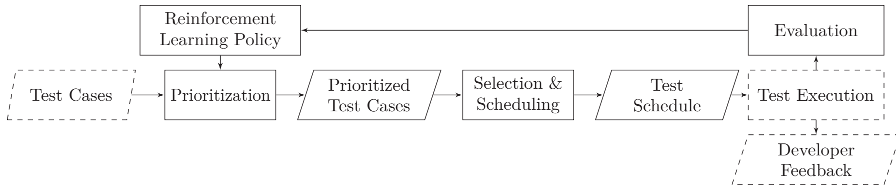


Figure A.2: Testing in CI process: RETECS uses test execution results for learning test case prioritization (solid boxes: Included in RETECS, dashed boxes: Interfaces to the CI environment)

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 re-visited during training phase to achieve repeated learning impulses, which is called *experience replay* [13]. 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 [14].

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 A.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 (subsection 4.1) is given before the experimental setup is introduced (subsection 4.2). In subsection 4.3 we present and discuss the experimental results. A discussion of possible threats (subsection 4.4) and extensions (subsection 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. 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 [15] 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) [16] 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 [17] using scikit-learn's implementation of artificial neural networks [18].

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) [19] 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.

[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 A.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 A.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

Table A.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 subsubsection 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 A.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 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.

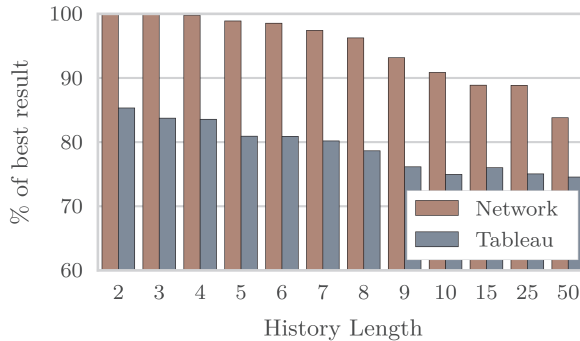


Figure A.3: Relative performance of different history lengths. A longer history can reduce the performance due to more complex information. (Data set: ABB Paint Control)

The effect of different history lengths is evaluated experimentally on the Paint Control data set. As Figure A.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 A.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.

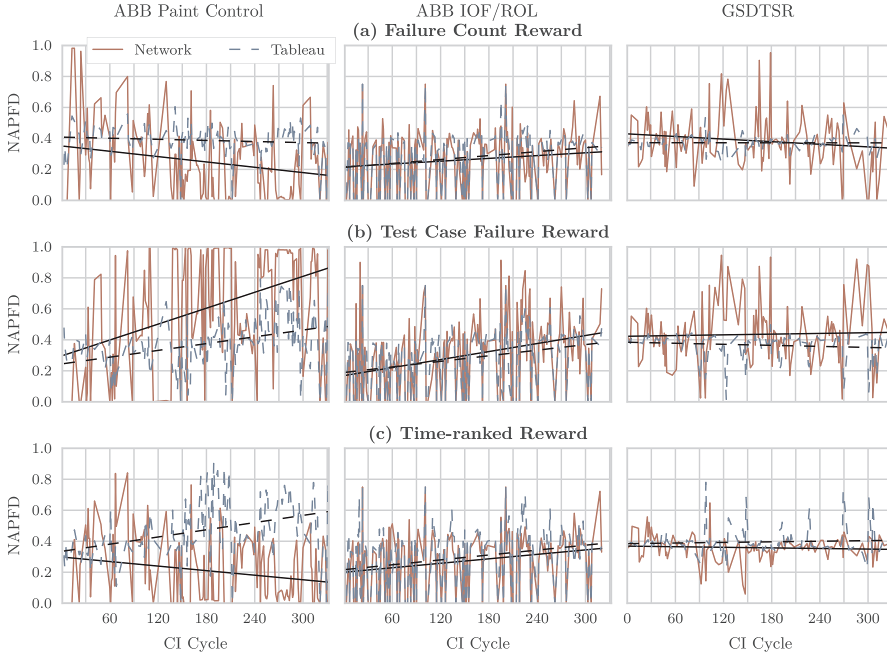


Figure A.4: Comparison of reward functions and memory representations: A Network-based agent with Test Case Failure reward delivers best performance on all three data sets (Black lines indicate trend over time)

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.

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 A.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 A.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

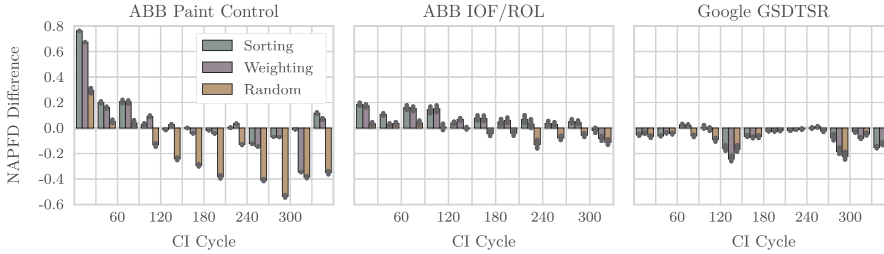


Figure A.5: Performance difference between network-based agent and comparison methods: After an initial exploration phase RETECS adapts to competitive performance. Each group of bars compares 30 CI cycles.

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 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 A.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

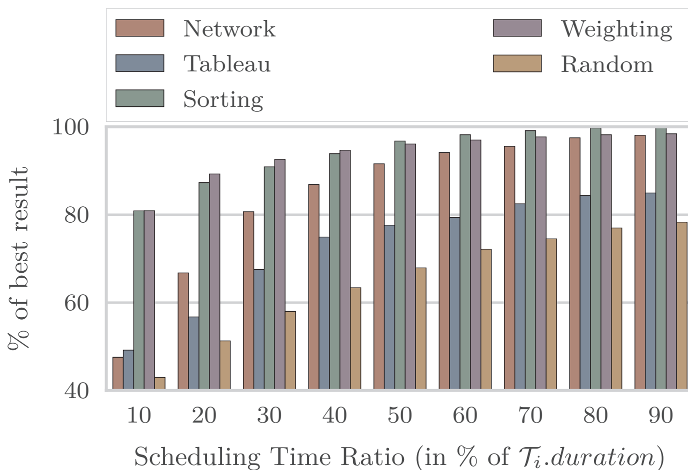


Figure A.6: Relative performance under different time limits. Shorter scheduling times reduce the information for rewards and delay learning. The performance differences for Network and Tableau also arise from the initial exploration phase, as shown in Figure A.5 (Data set: ABB Paint Control).

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 [20]. 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 [21], e.g. deterministic methods with desirable behavior, or by using historical data before it is introduced in the CI process [22]. 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 [23].

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 [24] 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 [5], 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 [25, 26], including swarm optimization [27] and ant colony optimization [28]. Walcott et al. use genetic algorithms for time-aware regression test suite prioritization for frequent code rebuilding [29]. Similarly, Zhang et al. propose time-aware prioritization using ILP [30]. Strandberg et al. [31] 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 [6, 7, 8, 32], code dependencies [33], or information retrieval [34, 35]. 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 [36], 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. [37] 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 [38], combining machine learning and program slicing for regression test case prioritization [39], learning agent-based test case prioritization [40], or clustering approaches [41]. 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, Fern, Pinto, Bauer, Alipour, Erwig, and Lopez [42]. Furthermore, Reichstaller, Eberhardinger, Knapp, Reif, and Gehlen [43] 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 [44]. 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.

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

References

[1] Fowler, M. and Foemmel, M. *Continuous Integration.* 2006.

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

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

[4] Stolberg, S. "Enabling Agile Testing through Continuous Integration". In: *Agile Conference, 2009. AGILE'09.* 2009, pp. 369–374.

[5] Di Nardo, D., Alshahwan, N., Briand, L., and Labiche, Y. "Coverage-Based Regression Test Case Selection, Minimization and Prioritization: A Case Study on an Industrial System". In: *Software Testing, Verification and Reliability* vol. 25, no. 4 (2015), pp. 371–396.

[6] Kim, J.-M. and Porter, A. "A History-Based Test Prioritization Technique for Regression Testing in Resource Constrained Environments". In: *Proceedings of the 24th International Conference on Software Engineering.* 2002, pp. 119–129.

[7] Marijan, D., Gotlieb, A., and Sen, S. "Test Case Prioritization for Continuous Regression Testing: An Industrial Case Study". In: *Proc. of Int. Conf. on Soft. Maintenance (ICSM'13), Industry Track.* 2013, pp. 540–543.

[8] Noor, T. B. and Hemmati, H. “A Similarity-Based Approach for Test Case Prioritization Using Historical Failure Data”. In: *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*. Nov. 2015, pp. 58–68.

[9] Rothermel, G., Untch, R. H., Chu, C., and Harrold, M. J. “Prioritizing Test Cases For Regression Testing”. In: *IEEE Transactions on Software Engineering* vol. 27, no. 10 (2001), pp. 929–948.

[10] Sutton, R. S. and Barto, A. G. *Reinforcement Learning: An Introduction*. 1st. MIT press Cambridge, 1998.

[11] Matarić, M. J. “Reward Functions for Accelerated Learning”. In: *Machine Learning: Proceedings of the Eleventh International Conference*. 1994, pp. 181–189.

[12] Van Hasselt, H. and Wiering, M. A. “Reinforcement Learning in Continuous Action Spaces”. In: *Proceedings of the 2007 IEEE Symposium on Approximate Dynamic Programming and Reinforcement Learning, ADPRL 2007* (2007), pp. 272–279.

[13] Lin, L.-J. J. “Self-Improving Reactive Agents Based on Reinforcement Learning, Planning and Teaching”. In: *Machine Learning* vol. 8, no. 3 (1992), pp. 293–321.

[14] Zhang, T. “Solving Large Scale Linear Prediction Problems Using Stochastic Gradient Descent Algorithms”. In: *Proceedings of the Twenty-First International Conference on Machine Learning*. 2004, pp. 116–116.

[15] Rothermel, G., Untch, R. H., and Harrold, M. J. *Test Case Prioritization*. Tech. rep. 1999, pp. 1–32.

[16] Qu, X., Cohen, M. B., and Woolf, K. M. “Combinatorial Interaction Regression Testing: A Study of Test Case Generation and Prioritization”. In: *IEEE International Conference on Software Maintenance, 2007 (ICSM)*. IEEE, 2007, pp. 255–264.

[17] Van Rossum, Guido and Drake Jr, F. L. *Python Reference Manual*. Tech. rep. Amsterdam, The Netherlands, The Netherlands: CWI (Centre for Mathematics and Computer Science), 1995.

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

[19] Elbaum, S., Mclaughlin, A., and Penix, J. “The Google Dataset of Testing Results”. In: (2014).

[20] Elbaum, S., Rothermel, G., and Penix, J. “Techniques for Improving Regression Testing in Continuous Integration Development Environments”. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*. ACM, 2014, pp. 235–245.

[21] Abbeel, P. and Ng, A. Y. “Apprenticeship Learning via Inverse Reinforcement Learning”. In: *Proceedings of the 21st International Conference on Machine Learning (ICML)*. 2004, pp. 1–8.

[22] Riedmiller, M. “Neural Fitted Q Iteration - First Experiences with a Data Efficient Neural Reinforcement Learning Method”. In: *European Conference on Machine Learning*. Vol. 3720 LNAI. Springer, 2005, pp. 317–328.

[23] Qu, B., Nie, C., and Xu, B. “Test Case Prioritization for Multiple Processing Queues”. In: *2008 International Symposium on Information Science and Engineering (ISISE)*. Vol. 2. IEEE, 2008, pp. 646–649.

[24] Mirarab, S., Akhlaghi, S., and Tahvildari, L. “Size-Constrained Regression Test Case Selection Using Multicriteria Optimization”. In: *IEEE Transactions on Software Engineering* vol. 38, no. 4 (2012), pp. 936–956.

[25] Yu, L., Xu, L., and Tsai, W.-T. “Time-Constrained Test Selection for Regression Testing”. In: *International Conference on Advanced Data Mining and Applications*. Springer Berlin Heidelberg, 2010, pp. 221–232.

[26] de Souza, L. S., Prudêncio, R. B. C., de A. Barros, F., and da S. Aranha, E. H. “Search Based Constrained Test Case Selection Using Execution Effort”. In: *Expert Systems with Applications* vol. 40, no. 12 (2013), pp. 4887–4896.

[27] de Souza, L. S., de Miranda, P. B., Prudencio, R. B., and Barros, F. d. A. “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, Nov. 2011, pp. 245–252.

[28] Noguchi, T., Washizaki, H., Fukazawa, Y., Sato, A., and Ota, K. “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)*. Apr. 2015, pp. 1–2.

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

[30] Zhang, L., Hou, S.-S., Guo, C., Xie, T., and Mei, H. “Time-Aware Test-Case Prioritization Using Integer Linear Programming”. In: *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2009, pp. 213–224.

[31] Strandberg, P. E., Sundmark, D., Afzal, W., Ostrand, T. J., and Weyuker, E. J. “Experience Report: Automated System Level Regression Test Prioritization Using Multiple Factors”. In: *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2016, pp. 12–23.

[32] Park, H., Ryu, H., and Baik, J. “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*. IEEE, July 2008, pp. 39–46.

[33] Gligoric, M., Eloussi, L., and Marinov, D. “Ekstazi: Lightweight Test Selection”. In: *Proceedings of the 37th International Conference on Software Engineering*. Vol. 2. May 2015, pp. 713–716.

[34] Kwon, J.-H. H., Ko, I.-Y. Y., Rothermel, G., and Staats, M. “Test Case Prioritization Based on Information Retrieval Concepts”. In: *Proceedings - Asia-Pacific Software Engineering Conference, APSEC* vol. 1 (2014), pp. 19–26.

[35] Saha, R. K., Zhang, L., Khurshid, S., and Perry, D. E. “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. May 2015, pp. 268–279.

[36] Busjaeger, B. and Xie, T. “Learning for Test Prioritization: An Industrial Case Study”. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 975–980.

[37] Chen, S., Chen, Z., Zhao, Z., Xu, B., and Feng, Y. “Using Semi-Supervised Clustering to Improve Regression Test Selection Techniques”. In: *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. IEEE, Mar. 2011, pp. 1–10.

[38] Bowring, J. F., Rehg, J. M., and Harrold, M. J. “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, 2004, pp. 195–205.

[39] Wang, F., Yang, S.-C., and Yang, Y.-L. “Regression Testing Based on Neural Networks and Program Slicing Techniques”. In: *Practical Applications of Intelligent Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 409–418.

[40] Abele, S. and Göhner, P. “Improving Proceeding Test Case Prioritization with Learning Software Agents”. In: *Proceedings of the 6th International Conference on Agents and Artificial Intelligence - Volume 2 (ICAART)*. 2014, pp. 293–298.

[41] Chaurasia, G., Agarwal, S., and Gautam, S. S. “Clustering Based Novel Test Case Prioritization Technique”. In: *2015 IEEE Students Conference on Engineering and Systems (SCES)*. IEEE, Nov. 2015, pp. 1–5.

[42] Groce, A., Fern, A., Pinto, J., Bauer, T., Alipour, A., Erwig, M., and Lopez, C. “Lightweight Automated Testing with Adaptation-Based Programming”. In: *Proceedings - International Symposium on Software Reliability Engineering, ISSRE*. 2012, pp. 161–170.

[43] Reichstaller, A. A., Eberhardinger, B., Knapp, A., Reif, W., and Gehlen, M. "Risk-Based Interoperability Testing Using Reinforcement Learning". In: *28th IFIP WG 6.1 International Conference, ICTSS 2016, Graz, Austria, October 17-19, 2016, Proceedings*. Ed. by Petrenko, A., Simão, A., and Maldonado, J. C. Vol. 6435. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2016, pp. 52–69.

[44] Veanes, M., Roy, P., and Campbell, C. "Online Testing with Reinforcement Learning". In: *Formal Approaches to Software Testing and Runtime Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 240–253.

Authors' addresses

Helge Spieker Simula Research Laboratory, Martin Linges vei 25, 1364 Fornebu, Norway, helge@simula.no

Arnaud Gotlieb Simula Research Laboratory, Martin Linges vei 25, 1364 Fornebu, Norway, arnaud@simula.no

Dusica Marijan Simula Research Laboratory, Martin Linges vei 25, 1364 Fornebu, Norway, dusica@simula.no

Morten Mossige University of Stavanger, Postboks 8600, 4036 Stavanger, Norway, ABB Robotics, Nordlysvegen 7, 4340 Bryne, Norway morten.mossige@uis.no

Paper B

Time-aware Test Case Execution Scheduling for Cyber-Physical Systems

Morten Mossige, Arnaud Gotlieb, Helge Spieker, Hein Meling, Mats Carlsson

Published in *Proceedings of the 23rd International Conference on Principles and Practice of Constraint Programming. CP 2017*, Lecture Notes in Computer Science, volume 10416, Springer, Cham, DOI: 10.1007/978-3-319-66158-2_25.
Preprint: arXiv:1902.04627v1.

II

Abstract

Testing cyber-physical systems involves the execution of test cases on target-machines equipped with the latest release of a software control system. When testing industrial robots, it is common that the target machines need to share some common resources, e.g., costly hardware devices, and so there is a need to schedule test case execution on the target machines, accounting for these shared resources. With a large number of such tests executed on a regular basis, this scheduling becomes difficult to manage manually. In fact, with manual test execution planning and scheduling, some robots may remain unoccupied for long periods of time and some test cases may not be executed.

This paper introduces TC-Sched, a time-aware method for automated test case execution scheduling. TC-Sched uses Constraint Programming to schedule tests to run on multiple machines constrained by the tests' access to shared resources, such as measurement or networking devices. The CP model is written in SICStus Prolog and uses the Cumulatives global constraint. Given a set of test cases, a set of machines, and a set of shared resources, TC-Sched produces an execution schedule where each test is executed once with minimal time between when a source code change is committed and the test results are reported to the developer. Experiments reveal that TC-Sched can schedule 500 test cases over 100 machines in less than 4 minutes for 99.5% of the instances. In addition, TC-Sched largely outperforms simpler methods based on a greedy algorithm and is suitable for deployment on industrial robot testing.

Reprinted by permission from Springer Nature 2017. The layout has been revised.

1 Introduction

Continuous integration (CI) aims to uncover defects in early stages of software development by frequently building, integrating, and testing software systems. When applied to the development of cyber-physical systems (CPS)[1], the process may include running integration test cases involving real hardware components on different machines or machines equipped with specific devices. In the last decade, CI has been recognized as an effective process to improve software quality at reasonable costs [1, 2, 3, 4].

Different from traditional testing methods, running a test case in CI requires tight control over the *round-trip time*, that is, the time from when a source code change is committed until the success or failure of the build and test processes is reported back to the developer [5]. Admittedly, the easiest way to minimize the round-trip time is simply to execute as many tests as possible in the shortest amount of time. But the achievable parallelism is limited by the availability of scarce global resources, such as a costly measurement instrument or network device, and the compatible machines per test case, targeting different machine architecture and operating systems. These global resources are required in addition to the machine executing the test case and thereby require parallel adjustments of the schedule for multiple machines.

Thus, computing an optimal test schedule with minimal round-trip time is a challenging optimization problem. Since different test cases have different execution times and may use different global resources that are locked during execution, finding an optimal schedule manually is mostly impossible. Nevertheless, manual scheduling still is state-of-the-practice in many industrial applications, besides simple heuristics. In general, successful approaches to scheduling use techniques from Constraint Programming (CP) and Operations Research (OR), additionally metaheuristics are able to provide good solutions to certain scheduling problems. We discuss these approaches further in Section 2.

Informally, the optimal test scheduling problem (OTS) is to find an execution order and assignment of all test cases to machines. Each test case has to be executed once and no global resource can be used by two test cases at the same time. The objective is to minimize the overall test scheduling and test execution time. The assignment is constrained by the compatibility between test cases and machines, that is, each test case can only be executed on a subset of machines.

This paper introduces TC-Sched, a time-aware method to solve OTS. Using the Cumulatives [6, 7] global constraint, we propose a cost-effective constraint optimization search technique. This method allows us to 1) automatically filter invalid test execution schedules, and 2) find among possible valid schedules, those that minimize the global test execution time (i.e., makespan). To the best of our knowledge, this is the first time the problem of optimal scheduling test suite execution is formalized and a fully automated solution is developed using constraint optimization techniques. TC-Sched has been developed and deployed together with ABB Robotics, Norway.

[1] CPS can simply be seen as communicating embedded software systems.

An extensive experimental evaluation is conducted over test suites from industrial software systems, namely an integrated control system for industrial robots and a product line of video-conferencing systems. The primary goal in this paper is to demonstrate the scalability of the proposed approach for CI processes involving hundreds of test cases and tens of machines, which corresponds to a realistic development environment. Furthermore, we demonstrate the cost-effectiveness of integrating our approach within an actual CI process.

2 Existing Solutions and Related Work

Automated solutions to address the OTS problem are not yet common practice. In industrial settings, test engineers manually design the scheduling of test case execution by allocating executions to certain machines at a given time or following a given order. In practice, they manage the constraints as an aggregate and try to find the best compromise in terms of the time needed to execute the test cases. Keeping this process manual in CI is paradoxical, since every activity should, in principle, be automated.

Regression testing [8], i.e. the repeated testing of systems after changes were made, in CI covers a broad area of research works, including automatic test case generation [9], test suite prioritization and test suite reduction [4]. There, the idea of controlling the time taken by optimization processes in test suite prioritization is not new [10]. In test suite prioritization, [11] proposed to use time-aware genetic algorithms to optimize the order in which to execute the test cases. Zhang *et al.* further refined this approach in [12] by using integer linear programming. On-demand test suite reduction [13] also exploits integer linear programming for preserving the fault-detection capability of a test suite while performing test suite reduction. Cost-aware methods are also available for selecting minimal subsets of test cases covering a number of requirements [14, 15]. All these approaches participate in a general effort to better control the time allocated to the optimization algorithms when they are used in CI processes. Note however that test suite execution scheduling is different to prioritization or reduction as it deals with the notion of scheduling in time the execution of all test cases, without paying attention to any prioritization or reduction.

Scheduling problems have been studied in other contexts for decades and an extensive body of research exists on resource-constrained approaches. The scheduling domain is divided into distinct areas such as process execution scheduling in operating systems and scheduling of workforces in a construction project. The scheduling problem of this paper belongs to a scheduling category named resource-constrained project scheduling problem (RCPSP; see [16, 17, 18] for an extensive overview). RCPSP is concerned with finding schedules for resource-consuming tasks with precedence constraints in a fixed time horizon, such that the makespan is minimized [18]. From the angle of RCPSP, global resources can be expressed as *renewable resources* which are available with exactly one unit per timestep and can therefore only be consumed by a single job per timestep.

RCPSP has been addressed by both exact methods [19, 20, 21, 22], as well as heuristic methods [23, 24]. Due to the vast amount of literature, we will focus on CP/OR-methods most closely related to the work of this paper. The clear trend in both CP and OR is to solve such problems with hybrid approaches, like, for instance, the work by Schutt et al. [25] or Beck et al. [26]. Furthermore, *disjunctive scheduling problems*, a subfamily of RCPSP addressing unary resources (in our terms global resources), have been effectively solved, e.g. by lazy clause generation [27].

RCPSP is considered to be a generalization of *machine scheduling problems* where *job shop scheduling* (JSS) is one of the best known [28]. JSS is the special case of RCPSP where each operation uses exactly one resource, and FJSS (*flexible job shop scheduling*) further extends JSS such that each operation can be processed on any machine from a given set. The FJSS is known to be NP-hard [29].

While OTS is closely related to FJSS, and efficient approaches to FJSS are known [22, 30], there are some differences. First, in OTS, execution times are machine-independent. Second, each job in OTS consists of only one operation, while in FJSS one job can contain several operations, where there are precedences between the operations. Finally, some operations additionally require exclusive access to a global resource, preventing overlap with other operations.

3 Problem Modeling

This section contains a formal definition of the OTS problem for test suite execution on multiple machines with resource constraints. Based on this definition, we propose a constraint optimization model using Cumulatives global constraint.

3.1 Optimal Test Case Execution Scheduling

Optimal test case scheduling[2] (OTS) is an optimization problem $(\mathcal{T}, \mathcal{G}, \mathcal{M}, d, g, f)$, where $\mathcal{T}$ is a set of n test cases along with a function $d : \mathcal{T} \longrightarrow \mathbb{N}$ giving each test case a duration d_i; a set of global resources $\mathcal{G}$ along with a function $g : \mathcal{T} \longrightarrow 2^{\mathcal{G}}$ that describes which resources are used by each test case; and a set of machines $\mathcal{M}$ and a function $f : \mathcal{T} \longrightarrow 2^{\mathcal{M}}$ that assigns to each test case a subset of machines on which the test case can be executed. The function d is usually obtained by measuring the execution time of each test case in previous test campaigns and by over-approximating each duration to account for small variations between the different execution machines. OTS is the optimization problem of finding an execution ordering and assignment of all test cases to machines, such that each test case is executed once, no global resource is used by two test cases at the same time, and the overall test execution time, T_t, is minimized. We define T_t as the time needed to compute the schedule (T_s) plus the time needed to execute the schedule (C^*), $T_t = T_s + C^*$. Machine assignment and test case execution

[2] OTS was part of the Industrial Modelling Competition at CP 2015.

ordering can be described either by a time-discretized table containing a line per machine or a starting time for each test case and its assignment to a given machine.

The problem addressed in this paper aims to execute each test case once while minimizing the total duration of the execution of the test cases. That is, to find an assignment $a : \mathcal{T} \longrightarrow \mathcal{M}$ and an execution order for each machine to run its test cases.

In its basic version, the OTS problem includes the following constraints:

Disjunctive scheduling: Two test cases cannot be executed at the same time on a single machine.

Non-preemptive scheduling: The execution of a test case cannot be temporarily interrupted to execute another test case on the same machine.

Non-shared resources: When a test case uses a global resource, no other test case needing this resource can be executed at the same time.

Machine-independent execution time: The execution time of a test case is assumed to be independent of the executing machine. This is reasonable for test cases in which the time is dominated by external physical factors such as a robot's motion, the opening of a valve, or sending an Ethernet frame. Such test cases typically have execution times that are uncorrelated with machine performance. In any case, a sufficient over-approximation will satisfy the assumption.

There are cases where OTS can be trivially solved, e.g. with only one machine executing all test cases in sequence. Indeed, the global execution time remains unchanged, whatever the execution order. Similarly, when there are no global resources and when test cases can be executed on any available machine, then simply allocating the longest test cases first to the available execution machine easily calculates a best-effort solution.

Example Considering the test suite in Table B.1, we present a small example. Let $\mathcal{T}$ be the test cases $\{1, \ldots, 10\}$, $\mathcal{G}$ be the global resources $\{1, 2\}$, and $\mathcal{M}$ be the machines $\{1, 2, 3\}$. The machines on which each test case in $\mathcal{T}$ can run is given in Table B.1. This table can be extracted by analyzing the test scripts or querying the test management. By sharing the same resource 1, test cases $2, 3, 4$ cannot be executed at the same time, even if their execution is scheduled on different machines. Since test case 7 can only be executed on machine 1, test case 8 on machine 2, test case 9 on machine 3, and test case 10 on machines 1 or 3, we have to solve a complex scheduling problem. One possible *optimal* schedule is given in Figure B.1, where the time needed to execute the test campaign is $C^* = 11$. For this small problem the solving time, T_s, can be assumed to be very short, so the total execution time will be $T_t \approx C^*$.

3.2 The CUMULATIVES Global Constraint

The CUMULATIVES global constraint [7] is a powerful tool for modeling cumulative scheduling of multiple operations on multiple machines, where each operation can be set up to consume a given amount of a resources, and each machine can be set up to provide a given amount of resources.

Table B.1: Test suite for example.

Test	Duration	Executable on	Use of global resource
1	2	$1, 2, 3$	-
2	4	$1, 2, 3$	1
3	3	$1, 2, 3$	1
4	4	$1, 2, 3$	1
5	3	$1, 2, 3$	-
6	2	$1, 2, 3$	-
7	1	1	-
8	2	2	-
9	3	3	-
10	5	$1, 3$	2

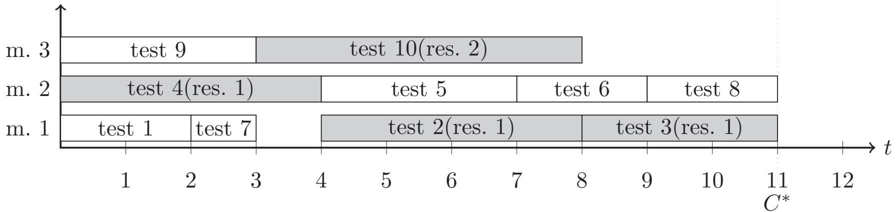


Figure B.1: An optimal solution to the scheduling problem given in Table B.1. Test cases in light gray require exclusive access to a global resource.

Cumulatives$([O_1, \ldots, O_n], [c_1, \ldots, c_p])$[3] constrains n operations on p machines such that the total resource consumption on each machine j does not exceed the given threshold c_j at any time [31]. An operation O_i is typically represented by a tuple $(S_i, d_i, E_i, r_i, M_i)$[4] where S_i (resp. E_i) is a variable that denotes the starting (resp. ending) instant of the operation, d_i is a constant representing the total duration of the operation, r_i is a constant representing the amount of resource used by the operation. S_i, E_i and M_i are bounded integer variables. S_i and E_i have the domains $est_i \ldots let_i$, where est_i denotes the operation's earliest starting time and let_i denotes its latest ending time and $let_i \geq est_i + d_i$. M_i is bounded by the number of machines available, that is $1, \ldots, p$. By reducing the domain of M_i it is possible to force a specific operation to be assigned to only a subset of the available machines, or even to one specific machine. It is worth noting that this formalization implicitly uses discrete time instants. Indeed, since est_i and let_i are integers, a function associating each time instant to the current executed operations can automatically be constructed. Formally, if h represents an instant in time, we have:

$$r_i^h = \begin{cases} r_i & \text{if } S_i \leq h < S_i + d_i \\ 0 & \text{otherwise} \end{cases}$$

[3]In [7] an additional third argument to Cumulatives, $Op \in \{\leq, \geq\}$ is defined. We omit it throughout our work and always set $Op = \leq$.

[4]Throughout the paper, lower-case characters are used to represent constants and upper-case characters are used to represent variables.

Cumulatives holds if and only if, for every operation O_i, $S_i + d_i = E_i$, and, for all machines k and instants h, $\sum_{i|M_i=k} r_i^h \leq c_k$. In fact, Cumulatives captures a disjunctive relation between different scenarios and applies deductive reasoning to the possible values in the domains of its variables. This constraint provides a cost-effective process for pruning the search space of some impossible schedules.

3.3 Modeling Test Case Execution Scheduling

This section shows how the Cumulatives constraint can be used to model a schedule. In this small example, we disregard the use of global resources, and the constraints that some operations can only be executed on a subset of the available machines, since that will be covered in Section 3.4. By the schedule in Figure B.1, we have ten operations $\mathcal{O} = \{O_1, \ldots, O_{10}\}$ and three available machines. By encoding the data from Table B.1, we get $O_1 = (S_1, 2, E_1, 1, M_1)$, $O_2 = (S_2, 4, E_2, 1, M_2) \ldots$, $O_{10} = (S_{10}, 5, E_{10}, 1, M_{10})$, $c_1 = 1$, $c_2 = 1$, $c_3 = 1$. Note that each operation has a resource consumption of one and all three machines have a resource capacity of one. This implies that one machine can only execute one operation at a time. Here, a resource refers to an execution machine and not to a global resource.

3.4 Introducing Global Resources

As mentioned above, global resources corresponding to physical equipment such as valves, air sensors, measurement instruments, or network devices, have limited and exclusive access. To avoid concurrent access from two test cases, additional constraints are introduced. Note that global resources must not be confused with the resource consumption or resource bounds of operations and machines.

The Cumulatives constraint does not support native modelling of these global resources without additional, user-defined constraints. However, there are ways to model exclusive access to such global resources by means of further constraints. The naive approach to prevent two operations from overlapping is to consider constraints over the start and stop time of the operations. For instance, if O_1 and O_2 both require exclusive access to a global resource, then the constraint $E_1 \leq S_2 \vee E_2 \leq S_1$ can be added. A less naive approach is to use a Disjunctive($\mathcal{O}^k$) constraint per global resource k, where $\mathcal{O}^k$ is the set of tasks that require that global resource, and Disjunctive prevents any pair of tasks from overlapping.

Referring to the example in Figure B.1, there are ten operations to be scheduled on three machines, and two global resources, 1 and 2. The basic scheduling constraint is set up as explained in Section 3.3. Yet another way to model the global resources is to treat each resource as a new quasi-machine $1'$ corresponding to $c_{1'} = 1$ and $2'$ corresponding to $c_{2'} = 1$. For each operation requiring a global resource, we create a "mirrored" operation of the corresponding quasi-machine: $\mathcal{O}'_1 = \{O'_2, O'_3, O'_4\}$ and $\mathcal{O}'_2 = \{O'_{10}\}$. Finally, we can express the schedule with a single constraint: Cumulatives($\mathcal{O} \cup \mathcal{O}'_1 \cup \mathcal{O}'_2, [c_1, c_2, c_3, c_{1'}, c_{2'}]$). For each operation in $\mathcal{O}'_1$ and $\mathcal{O}'_2$ we also reuse the same domain variables for

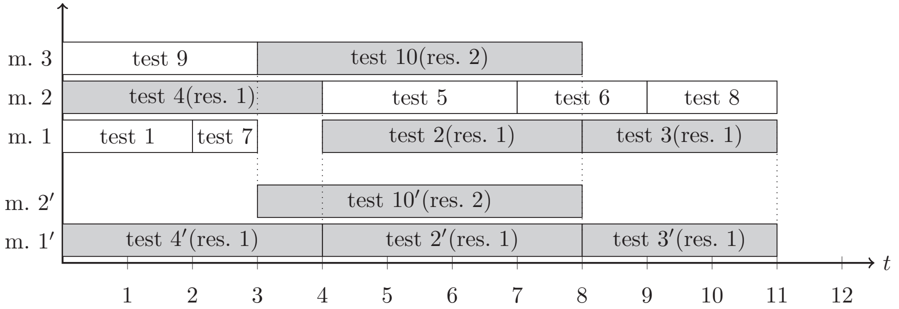


Figure B.2: Modeling global resources by creating quasi-machines and Cumulatives

start-time, duration and end-time. The operation O_4 will be forced to have the same start-/end-time as O_4', while they are scheduled on two different machines 2 and $1'$.

4 The TC-Sched Method

This section describes our method, TC-Sched, to solve the OTS problem. It is a *time-constrained cumulative scheduling technique*, as 1) it allows to keep fine-grained control over the time allocated to the constraint solving process (i.e., *time-constrained*), 2) it encodes exclusive resource use with constraints (i.e., *constraint-based*), and 3) it solves the problem by using the Cumulatives constraint. The TC-Sched method is composed of three elements, namely, the constraint model described in Section 4.1, the search procedure described in Section 4.2, and the time-constrained minimization process described in Section 4.3.

4.1 Constraint Model

We encode the OTS problem with one Cumulatives($\mathcal{O}, \mathcal{C}$) constraint, one Disjunctive($\mathcal{O}^k$) constraint per global resource k, using the second scheme from Section 3.4, and a search procedure able to find an optimal schedule among many feasible schedules. Each test case i is encoded as an operation $(S_i, d_i, E_i, 1, M_i)$ as explained in Section 3.2. $\mathcal{O}$ is simply the array of all such operations and $\mathcal{C}$ is an array of 1s of length equal to the number of machines. Suppose that there are three execution machines numbered $1, 2$, and 3; then, to say that test i can be executed on any machine, we just add the domain constraint $M_i \in \{1, 2, 3\}$, whereas to say that test i can only be executed on machine 1, we replace M_i by 1. Finally, to complete the model, we introduce the variable *MakeSpan* representing the completion time of the entire schedule and seek to minimize it. *MakeSpan* is lower bounded by the ending time of each

individual test case. The generic model is captured by:

$$\begin{array}{l}\textsc{Cumulatives}(\mathcal{O},\mathcal{C}) \wedge \\ \forall \text{global resource } k : \textsc{Disjunctive}(\mathcal{O}^k) \wedge \\ \forall 1 \le i \le n : M_i \in f(i) \wedge \\ \forall 1 \le i \le n : E_i \le \mathit{MakeSpan} \wedge \\ \textsc{Label}(\textsc{Minimize}(\mathit{MakeSpan}), [S_1, M_1, \ldots, S_n, M_n])\end{array} \tag{B.1}$$

Note that the ending times depend functionally on the starting times. Thus, a solution to the OTS problem can be obtained by searching among the starting times and the assignment of test cases to execution machines.

4.2 Search Procedure

Our search procedure is called *test case duration splitting*, and is a branch-and-bound search that seeks to minimize the *Makespan*. The procedure makes two passes over the set of test cases. A key idea is to allocate the most demanding test cases first. To this end, the test cases are initially sorted by decreasing r_i where r_i is the number of global resources used by test case i, breaking ties by choosing the test case with the longest duration d_i.

In Phase 1, two actions are performed on each test case. First, in order to avoid a large branching factor in the choice of start time and to effectively fix the relative order among the tasks on the same machine or resource, we split the domain of the start variable, forcing an obligatory part of the corresponding task, as described in [32, Section 3.6]. Next, in order to balance the load on the machines, we choose machines in round-robin fashion. These two choices are of course backtrackable, to ensure completeness of the search procedure.

Note that at the end of Phase 1, the constraint system effectively forms a directed acyclic graph where every node is a task and every arc is a precedence constraint induced by the relative order. It is well known that such constraint systems can be solved without search by topologically sorting the start variables and assigning each of them to its minimal value. This is Phase 2 of the search.

In this procedure, the load-balancing component has shown to be particularly effective in a CI context and makes the first solution found a good compromise between solving and execution time of the schedule, which is one of the key factors in CI. Our preliminary experiments concluded, that the presented strategy provided the best compromise between cost and solution quality. Furthermore, we tried a more precise but costlier load-balancing scheme, but it did not significantly improve the quality. We also tried to sort the tests by decreasing $d_i \cdot (r_i + 1)$, which did not significantly improve the quality, either.

4.3 Time-constrained Minimization

The third necessary ingredient of the TC-Sched method is to perform branch-and-bound search under a time contract. That is, to settle on the schedule with the shortest *MakeSpan* found when the time contract ends. When the number of test cases grows to be several hundred, finding a globally optimal schedule

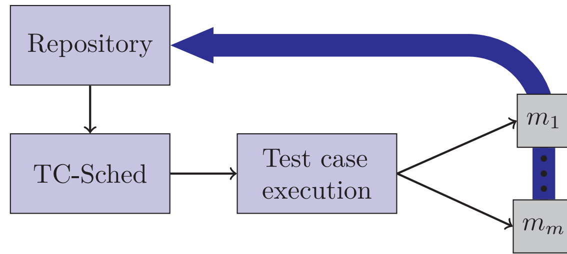


Figure B.3: Integration of TC-Sched into a CI process. The test case schedule solved by TC-Sched is transmitted for execution to the machines in the machine pool, $\mathcal{M}$. The results including actual test case durations are then feed back into the repository.

may become an intractable problem[5], but in practical applications it is often sufficient to find a "best-effort" solution. This leads to the important question to select the most appropriate contract of time for the minimization process, as the time used to optimize the schedule is not available to actually execute the schedule. We address this question in the experimental evaluation.

5 Implementation and Exploitation

This section details our implementation of the TC-Sched method and its insertion into CI. We implemented the TC-Sched method in SICStus Prolog [34]. The Cumulatives constraint is available as part of the `clpfd` library [31]. The `clpfd` library also provides an implementation of the time-constrained branch-and-bound with the option to express individual search strategy (see Section 4.2). Using `clpfd`, a generic constraint model for the TC-Sched method is designed, which takes an OTS problem as input and returns an (quasi-)optimal schedule.

Since TC-Sched is designed to run as part of a CI process, we describe how it can be integrated within the CI environment. Because CI environments change and test cases and agents are constantly added or removed, TC-Sched has to be provided with a list of test cases and available machines at runtime. Furthermore, an estimation of the test case durations on the available agents has to be provided. This can either be gathered from historical execution data and then (over-)estimated to account for differences in execution machines, or, for some kinds to test suites, they are fixed and can be precisely given [35], e.g. for robotic applications where the duration is determined by the movement of the robot.

A test campaign in a CI cycle is typically initiated upon a successful build of the software being tested. As a first step, all machines available for test execution are identified and updated with the newly built software. Then, TC-Sched takes as input the test cases of the test campaign and the previous test case execution times from the storage repository. After TC-Sched calculated an

[5]The general cumulative scheduling problem is known to be NP-hard [33].

optimal schedule, that schedule is handed over to a dedicated dispatch server which is responsible for distributing the test cases to the physical machines and the actual execution. Finally, after the test execution finished, the overall result of the test campaign is reported back to the users and the storage repository is updated with the latest test case execution times. Of course, minimizing the *round-trip time* leads to earlier notifications of the developers in case the software system fails and helps to improve the development cycle in CI.

6 Experimental Evaluation

This section presents our findings from the experimental evaluation of TC-Sched. To this end, we address the following three research questions:
RQ1: How does the first solution provided by TC-Sched compare with simpler scheduling methods in terms of schedule execution time? This research question states the crucial question of whether using complex constraint optimization is useful despite simpler approaches being available at almost no cost to implement.
RQ2: For TC-Sched, will an increased investment in the solving time in TC-Sched reduce the overall time of a CI cycle? This question is about finding the most appropriate trade-off between the solving time and the execution time of the test campaign in the proposed approach.
RQ3: In addition to random OTS problem instances, can TC-Sched efficiently and effectively handle industrial case studies? These cases can lead to structured problems which exhibit very different properties than random instances.

All experiments were performed on a 2.7 GHz Intel Core i7 processor with 16 GB RAM, running SICStus Prolog 4.3.5 on a Linux operating system.

6.1 Experimental Artifacts

To answer RQ1, we implemented two scheduling methods, referred to as the *random* method and the *greedy* method.

The *random* method works as follows: It first picks a test case at random and then picks a machine at random such that no resource constraint is violated. Finally, the test case is assigned the lowest possible starting time on the selected machine. The *greedy* method is more advanced. At first, it assigns test cases by decreasing resource demands. Afterwards, test cases without any resource demands are assigned to the remaining machines. For each assignment, the machine that can provide the earliest starting time is selected. Note that none of the two methods can backtrack to improve upon the initial solution.

The reason we have chosen to compare with these two methods is threefold: 1) As explained in Section 2, we are not aware of any previously published work related to test case execution scheduling, which means that there is no baseline to compare against; 2) From cooperation with our industrial partners, we know that this is, in the best case, the industrial state of the art (i.e., non-optimal schedules computed manually); 3) We manually checked the results on simple schedules and found them to be satisfactory, so they are a suitable comparison.

Table B.2: Randomly generated test suites.

	# of tests	20	30	40	50	100	500
# machines	100	-	-	-	-	-	TS11
	50	-	-	-	-	TS8	TS12
	20	-	TS2	TS4	TS6	TS9	TS13
	10	TS1	TS3	TS5	TS7	TS10	TS14

To answer our research questions, we have considered randomly generated benchmarks and industrial case studies. Although there are benchmark test suites for both JSS and FJSS, e.g., [36] or [29], they cannot be used as a comparison baseline. Furthermore, as our method approaches testing applications, a thorough evaluation on data from the target domain is justifiable.

We generated a benchmark library containing 840 OTS instances[6]. The library is structured by data collected from three different real-world test suites, provided by our industrial partners: a test suite for video conferencing systems (VCS) [38], a test suite for integrated painting systems (IPS) [35], and a test suite for a mobile application called *TV-everywhere*.

VCS is a test suite for testing commercial video conferencing systems, developed by CISCO Systems, Norway. It contains 132 test cases and 74 machines. The duration of test cases varies from 13 seconds to 4 hours, where the vast majority has a duration between 100 s and 800 s. The IPS test suite aims at testing a distributed paint control system for complex industrial robots, developed at ABB Robotics, Norway. It contains 33 test cases, with duration ranging from 1 s to 780 s, and 16 distinct machines. There are two global resources for this test suite, an airflow meter and a simulator for an optical encoder. *TV-everywhere* is a mobile application that allows users to watch TV on tablets, smart phones, and laptops. Its test suite only contains manual test cases, but, in our benchmark, it serves as a useful example of a test suite with a large number of constraints limiting the number of possible machines for each test case.

Based on data from the three industrial test suites, we composed 14 groups of test suites, denoted TS1-TS14, with randomized assignments of test cases to machines and exclusive usages of global resources. Let $|T|$ be the number of test cases, and $|M|$ be the number of machines, and $|R| = \{3, 5, 10\}$ be the number of resources. Table B.2 gives an overview of the groups of test suites. For test suite TSx, we write TSxR3, TSxR5, or TSxR10 to indicate the number of resources.

For each of the $14 \cdot 3$ variants, we generated 20 random test suites. The duration of each test case was chosen randomly between 1 s and 800 s, and each test case had a 30 % chance of using a global resource. The number of resources was chosen randomly between 1 and $|R|$. A total of 80 % of the tests were considered to be executable on all machines, while the remaining 20 % were executable on a smaller subset of machines. For these tests, the number

[6] All generated instances are available in CSPLib, a library of test problems for constraint solvers [37]

Figure B.4: The differences in schedule execution times produced by the different methods for test suites TS1–TS14, with *greedy* as the baseline of 100%. The blue is the difference between the first solution C_f^* and *greedy* and the red shows the difference between the final solution C_l^* and *greedy*.

of machines on which each test case could be executed was selected randomly between 1 % and 40 % of the number of available machines. This means that a test case was executable either on all machines (part of the 80 % group) or only on at most 40 % of the machines. In total, we generated $14 \cdot 3 \cdot 20 = 840$ different test suites.

6.2 RQ1: How does TC-Sched compare with simpler scheduling?

To compare our TC-Sched method with the *greedy* and *random* methods, we recorded the first solution, C_f^*, found by TC-Sched. We also recorded the last solution, C_l^*. This is either a proved optimal solution, or the best solution found after 5 minutes of solving time. For each of the 840 test suites, we computed the differences between the *random* and *greedy*, C_f^* and *greedy*, and C_l^* and *greedy*, where *greedy* is the baseline of 100 %. The results show that *random* is 30 %-60 % worse than *greedy*, which means that *random* can clearly be discarded from further analysis. Our findings are summarized in Figure B.4, showing the difference between TC-Sched and *greedy*. For all test suites but the hardest subset of TS1 and some instances of TS2, C_f^* is better than *greedy*. We also observe that for larger test suites, i.e., TS11-TS14, there is only a marginal difference between C_f^* and C_l^*. Hence, running the solver for a longer time has only little benefit.

Furthermore, to evaluate the effectiveness of the test case duration splitting search strategy, we compared it to standard strategies available in SICStus Prolog's `clpfd` with the same constraint model on the test suites TS1 and TS14. The search first enumerates on the machine assignments increasingly, i.e. without load-balancing, and afterwards assigns end times via domain splitting by bisecting the domain, starting from the earliest end times. As variable selection strategies, we tested both the default setting, selecting the leftmost variable, and a first-fail strategy, selecting the variable with the smallest domain. Additionally, we tried sorting the variables by decreasing resource usage.

All variants of the standard searches performed substantially worse than test case duration splitting, with first-fail search on sorted variables being the best. After finding an initial solution, further improvements are rare and the makespan of the final solution is in average 4 times larger compared to using test case duration splitting with the same time contract of 5 minutes.

6.3 RQ2: Will longer solving time reduce the total execution time?

RQ2 aims at finding an appropriate trade-off between the time spent in solving the constraint model, T_s, and the time spent in executing the schedule, C^*. As mentioned in Section 1, the round-trip time is critical in CI and has to be kept low. It is therefore crucial to determine the most appropriate timeout for the constraint optimizer. The ultimate goal being to generate a schedule which is quasi-optimal w.r.t. total execution time, $T_t = T_s + C^*$.

As mentioned above, TC-Sched can be given a time-contract for finding a quasi-optimal solution when minimizing the execution time of the schedule. More precisely, with this time-constrained process four outcomes are possible.

No solution with proof: TC-Sched proves that the OTS problem has no solution due to unsatisfiable constraints.

No solution without proof: TC-Sched was not able to find a solution within the given time. Thus, there could be a solution, but it has not been found.

Quasi-optimal solution: At the end of the time-contract, a solution is returned, but TC-Sched was interrupted while trying to prove its optimality. Such a best-effort solution is usually sufficient in the examined industrial settings.

Optimal solution: Before the end of the time-contract, TC-Sched returns an optimal solution along with its proof. This is obviously the most desired result.

Each solution i generated by TC-Sched can be represented by a tuple $(C_i^*, T_{s,i})$ where C_i^* is the makespan of solution i and $T_{s,i}$ is the time the solver spent finding solution i. The goal of RQ2 is to find the value of $T_{s,i}$ that minimizes $(C_i^* + T_{s,i}), \forall\, i$ and use this value as the time-contract.

To answer RQ2, we executed TC-Sched on all 840 test suites, with a time-contract of 5 minutes. During this process, we recorded all intermediate search results to calculate the optimal value of T_s for each test suite.

Figure B.5 shows the distribution in solving time for the first solution found by TC-Sched, the last solution and also how the optimal value of T_s is distributed. For the group of 600 test suites containing up to 100 test cases (TS1-TS10), the results show that a solution that minimizes the total execution time, noted T_t, is found in $T_s < 5$ s for 96.8 % of the test suites. If we extend the search time to $T_s < 10$ s, the number grows to 98 % of the test suites. For this group, the worst case optimal solving time was $T_s = 122.3$ s. We see that a solution is always found in less than 0.1 s. For the group of 240 test suites containing 500 test cases (TS11-TS14), the results show that a solution that minimizes T_t is found in $T_s < 120$ s for 97.5 % of the test suites. A solution minimizing T_t is found in less than 240 s for all test suites, except one instance with $T_t = 264$ s.

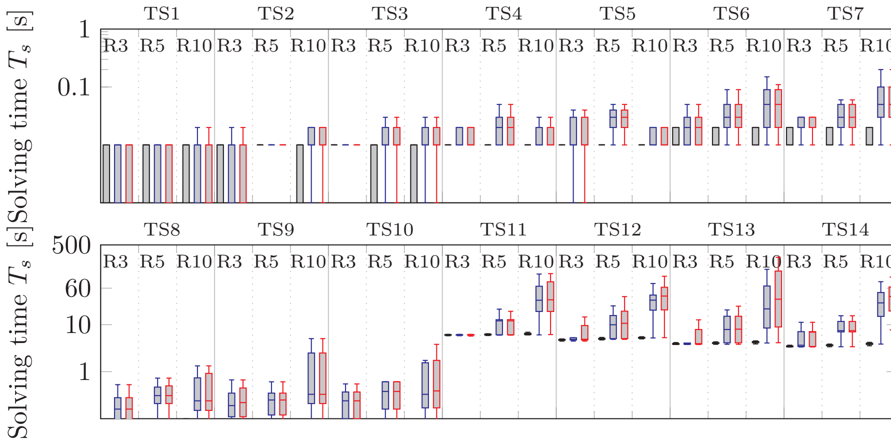


Figure B.5: The black boxes show the distribution in solving time, T_s, for the first solution found by TC-Sched. The blue boxes show the distribution in T_s where the total execution time, T_t, is optimal. Finally, the red boxes show the distribution in T_s for the last solution found by TC-Sched, which can be the optimal value or the last value found before timeout. The timeout was set to 5 min.

An increased investment in the solving part does not seem to necessarily pay off if one considers the total execution time. The reported experiments give hints to evaluate and select the optimal test contract for the solving part.

6.4 RQ3: Can TC-Sched efficiently solve industrial OTS problems?

To answer RQ3, we consider two of the three industrial case studies, namely, IPS and VCS. These case studies are composed of automated test scripts, which makes the application of the TC-Sched method especially pertinent.

In both case studies, the guaranteed optimal solution is already found as the first solution in less than 200 ms. This avoids the necessity to compromise between C^* and T_s for these industrial applications.

When applying TC-Sched to the IPS test suite, we find the optimal solution, $C^* = 780$ s, at $T_s = 10$ ms. For the VCS test suite, the optimal solution, $C^* = 14637$ s is found at $T_s = 160$ ms.

In summary, TC-Sched can easily be applied to both VCS and IPS, and in both cases, the best result is achieved when C^* is minimized and T_s is neglected.

7 Conclusion

This paper introduced TC-Sched, a time-aware method for solving the optimal test suite scheduling (OTS) problem, where test cases can be executed on multiple execution machines with non-shareable global resources. TC-Sched

exploits the CUMULATIVES global constraint and a time-aware minimization process, and a dedicated search strategy, called *test case duration splitting*. To our knowledge, the OTS problem is rigorously formalized for the first time and a method is proposed to solve it in CI applications. An experimental evaluation performed over 840 generated test suites revealed that TC-Sched outperforms simple scheduling methods w.r.t. total execution time. More specifically, we showed that automatic optimal scheduling of 500 test cases over 100 machines is reachable in less than 4 minutes for 99.5% instances of the problem. By considering trade-offs between the solving time and the total execution time, the evaluation allowed us to find the best compromise to allocate time-contracts to the solving process. Finally, by using TC-Sched with two industrial test suites, we demonstrated that finding the guaranteed optimal test execution time is possible and that TC-Sched can effectively solve the OTS problem in practice.

Further work includes consideration of test case priorities, non-unitary shareable global resources, as well as explicit symmetry breaking in the model. Additional evaluation and comparison against heuristic methods, such as evolutionary algorithms, or Mixed-Integer Linear Programming could extend the presented work and support the integration of TC-Sched in practical CI processes.

References

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

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

[3] Stolberg, S. "Enabling Agile Testing through Continuous Integration". In: *Agile Conference, 2009. AGILE'09.* 2009, pp. 369–374.

[4] Elbaum, S., Rothermel, G., and Penix, J. "Techniques for Improving Regression Testing in Continuous Integration Development Environments". In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*. ACM, 2014, pp. 235–245.

[5] Fowler, M. and Foemmel, M. *Continuous Integration*. 2006.

[6] Aggoun, A. and Beldiceanu, N. "Extending {CHIP} in Order to Solve Complex Scheduling and Placement Problems". In: *Mathematical and Computer Modelling* vol. 17, no. 7 (1993), pp. 57–73.

[7] Beldiceanu, N. and Carlsson, M. "A New Multi-Resource Cumulatives Constraint With Negative Heights". In: *Principles and Practice of Constraint Prog. (CP'02)*. 2002, pp. 63–79.

[8] Orso, A., Shi, N., and Harrold, M. J. "Scaling Regression Testing to Large Software Systems". In: *Proc. of the Symp. on Foundations of Software Eng. (FSE'04)*. ACM Press, 2004, pp. 241–251.

[9] de Campos, J., Arcuri, A., Fraser, G., and de Abreu, R. "Continuous Test Generation: Enhancing Continuous Integration with Automated Test Generation". In: *Proc. of Int. Conf. on Automated Soft. Eng. (ASE'14)*. 2014, pp. 55–66.

[10] Do, H., Mirarab, S., Tahvildari, L., and Rothermel, G. "The Effects of Time Constraints on Test Case Prioritization: A Series of Controlled Experiments". In: *IEEE Trans. on Soft. Eng.* vol. 36, no. 5 (2010), pp. 593–617.

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

[12] Zhang, L., Hou, S.-S., Guo, C., Xie, T., and Mei, H. "Time-Aware Test-Case Prioritization Using Integer Linear Programming". In: *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2009, pp. 213–224.

[13] Hao, D., Zhang, L., Wu, X., Mei, H., and Rothermel, G. "On-Demand Test Suite Reduction". In: *Proc. of the Int. Conf. on Soft. Eng.* 2012, pp. 738–748.

[14] Lin, T.-Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., Dollár, P., and Zitnick, C. L. "Microsoft COCO: Common Objects in Context". In: *European Conference on Computer Vision*. Vol. 8693. LNCS. 2014, pp. 740–755.

[15] Gotlieb, A. and Marijan, D. "FLOWER: Optimal Test Suite Reduction as a Network Maximum Flow". In: *Proc. of Int. Symp. on Soft. Testing and Analysis (ISSTA'14)*. 2014, pp. 171–180.

[16] Brucker, P., Drexl, A., Möhring, R., Neumann, K., and Pesch, E. "Resource-Constrained Project Scheduling: Notation, Classification, Models, and Methods". In: *European Journal of Operational Research* vol. 112, no. 1 (1999), pp. 3–41.

[17] Brucker, P. and Knust, S. *Complex Scheduling (GOR-Publications)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[18] Hartmann, S. and Briskorn, D. "A Survey of Variants and Extensions of the Resource-Constrained Project Scheduling Problem". In: *European Journal of Operational Research* vol. 207, no. 1 (2010), pp. 1–14.

[19] Szeredi, R. and Schutt, A. "Modelling and Solving Multi-Mode Resource-Constrained Project Scheduling". In: *Principles and Practice of Constraint Programming. CP 2016*. Vol. 9892. 2016, pp. 877–878.

[20] Kreter, S., Schutt, A., and Stuckey, P. J. "Modeling and Solving Project Scheduling with Calendars". In: *International Conference on Principles and Practice of Constraint Programming*. 2015, pp. 262–278.

[21] Schutt, A., Chu, G., Stuckey, P. J., and Wallace, M. G. "Maximising the Net Present Value for Resource-Constrained Project Scheduling". In: *CPAIOR 2012*. Vol. 7514. LNCS. 2012, pp. 362–378.

[22] Schutt, A., Feydy, T., and Stuckey, P. J. "Scheduling Optional Tasks with Explanation". In: *International Conference on Principles and Practice of Constraint Programming*. 2013, pp. 628–644.

[23] Hartmann, S. and Kolisch, R. "Experimental Evaluation of State-of-the-Art Heuristics for the Resource-Constrained Project Scheduling Problem". In: *European Journal of Operational Research* vol. 127, no. 2 (2000), pp. 394–407.

[24] Kolisch, R. and Hartmann, S. "Experimental Investigation of Heuristics for Resource-Constrained Project Scheduling: An Update". In: *European Journal of Operational Research* vol. 174, no. 1 (2006), pp. 23–37.

[25] Schutt, A., Feydy, T., Stuckey, P. J., and Wallace, M. G. "Why Cumulative Decomposition Is Not As Bad As It Sounds". In: *Principles and Practice of Constraint Programming (CP'09)*. Springer, 2009, pp. 746–761.

[26] Beck, J. C., Feng, T. K., and Watson, J. P. "Combining Constraint Programming and Local Search for Job-Shop Scheduling". In: *INFORMS Journal on Computing* vol. 23, no. 1 (2011), pp. 1–14.

[27] Siala, M., Artigues, C., and Hebrard, E. "Two Clause Learning Approaches for Disjunctive Scheduling". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* vol. 9255 (2015), pp. 393–402.

[28] Herroelen, W., De Reyck, B., and Demeulemeester, E. "Resource-Constrained Project Scheduling: A Survey of Recent Developments". In: *Computers & Operations Research* vol. 25, no. 4 (1998), pp. 279–302.

[29] Behnke, D. and Geiger, M. J. *Test Instances for the Flexible Job Shop Scheduling Problem with Work Centers*. Tech. rep. Hamburg, Germany, 2012.

[30] Brandimarte, P. "Routing and Scheduling in a Flexible Job Shop by Tabu Search". In: *Annals of Operations research* vol. 41, no. 3 (1993), pp. 157–183.

[31] Carlsson, M., Ottosson, G., and Carlson, B. "An Open-Ended Finite Domain Constraint Solver". In: *Proc. of the 9th Int. Symp. on Prog. Languages, Implementations, Logics, and Programs (PLILP '97)*. 1997, pp. 191–206.

[32] Simonis, H. and O'Sullivan, B. "Search Strategies for Rectangle Packing". In: *Proc. of Principles and Practice of Constraint Prog. (CP'08)*. 2008, pp. 52–66.

[33] Baptiste, P., Le Pape, C., and Nuijten, W. *Constraint-Based Scheduling: Applying Constraint Programming to Scheduling Problems*. 1st ed. International Series in Operations Research & Management Science 39. Springer US, 2001.

[34] Carlsson, M. et al. *{SICStus} {Prolog} User's Manual, Release 4*. Tech. rep. 2007.

[35] Mossige, M., Gotlieb, A., and Meling, H. "Using CP in Automatic Test Generation for ABB Robotics' Paint Control System". In: *Principles and Practice of Constraint Programming*. Ed. by O'Sullivan, B. Vol. 8656. LNCS. Cham: Springer International Publishing, 2014, pp. 25–41.

[36] Taillard, E. "Benchmarks for Basic Scheduling Problems". In: *European Journal of Operational Research* vol. 64, no. 2 (1993), pp. 278–285.

[37] Mossige, M. "CSPLib Problem 073: Test Scheduling Problem". In: (). Ed. by Jefferson, C., Miguel, I., Hnich, B., Walsh, T., and Gent, I. P.

[38] Marijan, D., Gotlieb, A., and Sen, S. "Test Case Prioritization for Continuous Regression Testing: An Industrial Case Study". In: *Proc. of Int. Conf. on Soft. Maintenance (ICSM'13), Industry Track*. 2013, pp. 540–543.

Authors' addresses

Morten Mossige University of Stavanger, Postboks 8600, 4036 Stavanger, Norway,
ABB Robotics, Nordlysvegen 7, 4340 Bryne, Norway morten.mossige@uis.no

Arnaud Gotlieb Simula Research Laboratory, Martin Linges vei 25, 1364 Fornebu, Norway, arnaud@simula.no

Helge Spieker Simula Research Laboratory, Martin Linges vei 25, 1364 Fornebu, Norway, helge@simula.no

Hein Meling University of Stavanger, Postboks 8600, 4036 Stavanger, Norway, hein.meling@uis.no

Mats Carlsson RISE Research Institutes of Sweden AB, ICT SICS, Box 1263, SE-164 29 Kista, Sweden, mats.carlsson@ri.se

Paper C

Multi-Cycle Assignment Problems with Rotational Diversity

Helge Spieker, Arnaud Gotlieb, Morten Mossige

Initial conference paper published as "Rotational Diversity in Multi-Cycle Assignment Problems" in: *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence* Vol. 33 (2019), pp. 7724–7731. DOI: 10.1609/aaai.v33i01.33017724. Preprint: arXiv:1811.03496v1.
The thesis contains an extended version that has been submitted to: *Journal of Artificial Intelligence Research* (May 2019). Preprint: arXiv:1811.03496v2.

Abstract

Multi-cycle assignment problems address scenarios where a series of general assignment problems has to be solved sequentially. Subsequent cycles can differ from previous ones due to changing availability or creation of tasks and agents, which makes an upfront static schedule infeasible and introduces uncertainty in the task-agent assignment process. We consider the setting where, besides profit maximization, it is also desired to maintain diverse assignments for tasks and agents, such that all tasks have been assigned to all agents over subsequent cycles. This problem of multi-cycle assignment with rotational diversity is approached in two sub-problems: The outer problem which augments the original profit maximization objective with additional information about the state of rotational diversity while the inner problem solves the adjusted general assignment problem in a single execution of the model. We discuss strategies to augment the profit values and evaluate them experimentally. The method's efficacy is shown in three case studies: multi-cycle variants of the multiple knapsack and the multiple subset sum problems, and a real-world case study on the test case selection and assignment problem from the software engineering domain.

1 Introduction

General assignment problems are well-studied in artificial intelligence and can be solved efficiently. Their goal is to assign a set of weighted tasks to a set of agents, such that capacity constraints are satisfied and a profit function is maximized. These problems are relevant in a broad context, of which many consider some form of rotation. In aircraft rotation [1] or machine scheduling [2], rotation mechanisms allow to keep maintenance schedules or optimize usage patterns of machinery. In nurse rostering [3, 4] and workforce scheduling [5, 6], rotation is relevant to avoid boredom, fatigue and prolonged high workloads or to cover a constrained shift system. It is not always possible to address such scheduling requirements upfront, albeit for personnel availability due to vacations and sickness leaves, changing demand patterns, or short-term planning horizons for other reasons. Conclusively, it can be necessary to include rotation mechanisms in scenarios of iterative and recurring planning due to problem constraints and requirements.

This paper addresses multi-cycle assignment problems, where there is uncertainty regarding the availability of tasks and agents, under the additional goal to rotate assignments from tasks to agents over successive cycles. Tasks and agents can be unavailable for one or several cycles without previous notice or information about their next availability. We refer to the subsequent diverse assignments as *rotational diversity*. A full example of the problem and our solution is given in Section 4.1.

It should be noted, that this work defines *rotational diversity* in a temporal manner. That is, the solution to multiple subsequent instances of a problem has to differ in the assignments made. This is different from the notion of solution diversity, where it is desirable to find multiple distinctly diverse solutions to one instance of a problem [7, 8, 9, 10].

We develop an method, that combines profits and affinities, a metric to describe the state of rotation, into a single optimization criterion. Solving this model incrementally, that is, at each cycle, allows to control rotational diversity. A central component for this control is the strategy, that defines how profits and affinities are combined. Processwise, the method is split into two sub-problems: The outer problem which augments the original profit maximization objective with additional information about the state of rotational diversity, while the inner problem solves the adjusted general assignment problem in a single execution of the solver for the assignment problem.

Part of the technical contribution is the presentation of five strategies for this combination of values. All of these strategies can be further combined with a *Limited Assignment* extension, that restricts the possible assignments between tasks and agents with the goal to more rigorously enforce the solver to produce diverse solutions. Using Limited Assignment increases the ability to maintain rotational diversity, but with a trade-off in profit.

As part of the experimental evaluation, three case studies are considered. The first case study is a multi-cycle extension of the multiple knapsack problem, and in the second, the multiple subset sum problem is considered in a multi-cycle

environment (MCMSSP). The third case study is a real-world case study of test case selection and assignment problem (TCSA), originating from the software engineering domain. Our results show that in all case studies rotational diversity can be effectively maintained by the introduced method, while sacrificing only a small percentage of the original goal of profit maximization, e.g. less than 4 % in TCSA.

In previous work [11], we introduced the problem of rotational diversity in multi-cycle assignment problems and presented initial strategies to address the problem for the first time. This paper builds upon the existing results with additional insights and explanations, as well as an extended experimental evaluation. We furthermore introduce the Limited Assignment, which enforces diversity through manipulating the compatibility between tasks and agents. This extended strategy can be combined with any of the previous approaches and shows to be effective to further improve the rotational diversity in our experiments.

The remainder of this paper is structured as follows: Section 2 gives an overview on the related work in the area of general assignment problems, assignments under uncertainty and alternative approaches to our method, then Section 3 introduces and formalizes the problem of multi-cycle assignment with rotational diversity. Our method to approach rotational diversity is presented in Section 4 along with the six evaluated strategies. In Section 5, we perform an experimental evaluation on two case studies before concluding the paper with a final discussion in Section 6.

2 Related Work

The general multi-cycle assignment problem is a variant of the *General Assignment Problem* (GAP) [12, 13, 14]. A set of tasks, each associated with a profit and a weight, has to be assigned to a set of agents with limited capacity. The goal is to maximize (or minimize) the summed profits of the assigned tasks, while the weights do not exceed the agent capacities. Not all tasks are mandatory to be assigned. Profits and weights can vary between agents. The classical assignment problem formulates a cost minimization objective, although maximization, which we use throughout this work, is also commonly found in problem variants.

In this paper, we formulate rotational diversity in terms of the broad class of general assignment problems, as our contribution is steered towards the general rotation mechanism. The closest problem variant is the group of knapsack problems. One or multiple agents have to be filled in to maximize the value of the selected tasks [15]. A multi-cycle knapsack variant is presented in [16], although only the unassigned items from previous cycles are available in subsequent cycles.

Assignment rotation is found in job rotation scheduling [6]. Here, a common goal is to find schedules and work assignments for humans to avoid fatigue, boredom [17] or accidents [18], or to evenly distribute shifts to personnel [19, 20]. This is often solved by a fixed schedule, where the assignment between workers and

their tasks frequently changes. While there is existing work on repairing schedules in case of disruptions or stochastic elements [21], for example in the application of university timetabling [22] or repair scheduling [23], these approaches require a defined planning horizon with one, possibly large, optimization problem in the beginning and a number of follow-up problems in case of disruptions. A specific application is further scheduling and assignment under the awareness of uncertainties within the given data [24], especially for varying task durations or weights, which can be addressed by robust local search [25] or stochastic optimization models [26]. In our approach, we solve subsequent assignment problem without a fixed planning horizon. That is, we do not fix one assignment over multiple cycles, but have to repeatedly create individual assignments at each cycle due to changing availability of agents and tasks. In relation to the terminology introduced in [21], our presented method is a progressive technique, where each part of the overall schedule is created sequentially at each cycle. However, their framework and terminology addresses the ability to repair schedules under uncertainty and does not include the desire to actively introduce diversity in the assignments between each time-step or cycle.

Opposite to diverse rotations is the concept of *persistence* in robust optimization [27, 28]. Persistence [27, 29] considers finding stable assignments during optimization, such that improvements in the solution objective only cause small changes in the variable assignment of the solution. By maximizing the affinity between tasks and agents, we can adjust the presented method to support persistent instead of diverse assignments in subsequent iterations of a problem using similar techniques. We note that the concept of affinity, which we introduce in Section 3.1, can be transferred to multi-cycle problems with persistence.

Fair allocations, which maximize a social welfare function, are considered in game theory research. Mechanisms for the resource distribution include *combinatorial auctions* and *exchanges* [30, 31]. Both have shown to result in a balanced and fair distribution of resources, although it is complex to determine which resources to offer in an auction or exchange and who is the resulting winner [32]. Recent works further discuss aspects of repeated matching between tasks and agents, under consideration of dynamic preferences and fairness [33], or repeated matching of previously unmatched tasks [34]. Because combinatorial auctions and exchanges can be decentralized, these techniques are commonly used for resource allocation in multi-agent systems [35, 36].

In this work, we do not directly solve the GAP, but instrument a general solver to maintain a fair distribution of tasks to agents. An alternative is a system where agents exchange tasks among them to achieve rotation. However, preliminary experiments showed this approach to be inferior to the one presented. The evaluated exchange model first focused on profits only, and afterwards aimed for a fair rotation by allowing one-task exchanges between agents. It showed that a high-quality GAP solution limits the number of choices for one-task exchanges and only minimal improvements in rotational diversity occur.

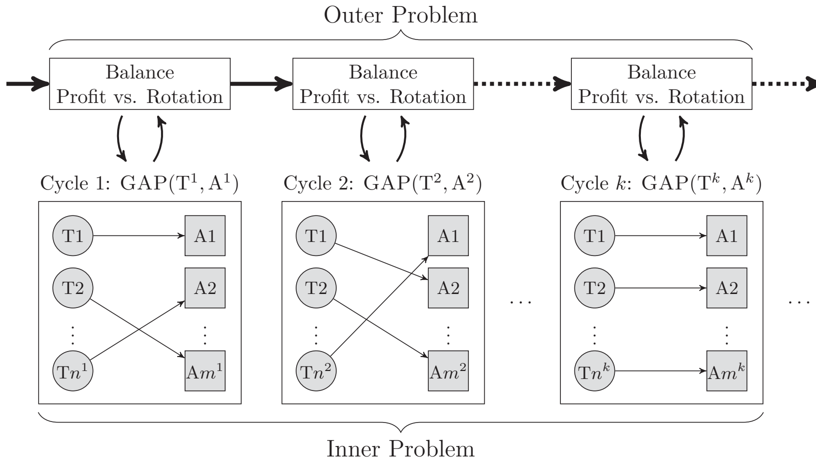


Figure C.1: Multi-Cycle Assignment Problem: At each cycle an independent GAP has to be solved, which includes an optimization objective, e.g. maximizing the sum of profits of all assigned tasks. The sets of available tasks and agents can vary between cycle due to different availability.

3 Problem Description

We first introduce the multi-cycle assignment problem as a combination of two sub-problems, which we further define afterwards. Then, we discuss the characteristics of the class of general assignment problems at the core of our approach. Finally, we formulate and discuss the requirements for maintaining rotational diversity over multiple cycles.

In a multi-cycle assignment problem, every cycle is a distinct planning unit, because, due to the availability of tasks and agents, planning ahead is not possible. Therefore, rotational diversity has to be considered at every cycle. This separates the overall problem into two partial sub-problems (as visualized in Figure C.1): First, the *inner problem* is to solve an independent GAP in each cycle k. The GAP selects a subset of the available tasks while maximizing the sum of their values.

Second, the *outer problem* aims to maintain a diverse assignment between tasks to agents, meaning that the tasks are frequently assigned to all compatible agents over subsequent cycles. As a mechanism for this balance, we utilize the *affinity* between a single task and each agent, and the *affinity pressure* as a metric to evaluate the whole set of tasks and agents. The balancing mechanism between profit optimization and rotation of tasks, is called a *strategy*.

The inner problem, as well as both the affinity and the affinity pressure will be further defined and introduced in the following section. As part of our method, we introduce six strategies for achieving rotational diversity in Section 4.2.

3.1 Multi-Cycle General Assignment Problem

The general assignment problem, $\mathrm{GAP}(\mathcal{T}^k, \mathcal{A}^k)$, receives as inputs the tasks and agents available at cycle k. The set of agents, $\mathcal{A}^k$, consists of m integers i, each with a fixed *capacity*, b_i, and the set of tasks, $\mathcal{T}^k$, consists of n integers j. Both sets are given at each cycle and can unpredictably change from cycle k to $k+1$.

The relation between a task and an agent has three fixed attributes: both the *profit* p_{ij} and the *weight* w_{ij} are externally fixed and describe the benefit respectively the resource demand of task j when assigned to agent i. Each task further has a set of *compatible agents*, $\mathcal{C}_j^k$, that it can be assigned to.

The *affinity* a_{ij} is not fixed, but changes between cycles. The affinity numerically describes the preferred assignments from tasks to agents, with higher values giving a higher preference for a task to be assigned to that agent. It is not given as a problem input parameter, unlike the profit, weight and compatibility, but it is determined as part of method to maintain rotational diversity.

Additionally, we refer to *values* in the context of the optimization objective of the GAP. Here, the value v_{ij} is a combination of profits and affinities, a way to balance profit- and rotation-oriented assignments. For a standard assignment problem without affinities, the values equal the profits.

The affinity between a task and an agent, a_{ij}, is the number of cycles since the last assignment of task j to agent i. The affinity quantifies the preference of a task to be assigned to certain agents during the next cycles. The affinity pressure is the maximum of all affinities in the set of tasks. Both the affinity and the affinity pressure will be further discussed after a definition of the inner assignment problem.

Definition 3.1. Multi-Cycle General Assignment Problem

$$\text{Maximize} \quad \sum_{i \in \mathcal{A}^k} \sum_{j \in \mathcal{T}^k} x_{ij} v_{ij} \tag{C.1}$$

$$\text{subject to} \quad \sum_{j \in \mathcal{T}^k} x_{ij} w_{ij} \leq b_i, \qquad \forall\, i \in \mathcal{A}^k \tag{C.2}$$

$$\sum_{i \in \mathcal{A}^k} x_{ij} \leq 1, \qquad \forall\, j \in \mathcal{T}^k \tag{C.3}$$

with

k : Index of the current cycle

$\mathcal{A}^k$: A set of integers i labeling m agents

$\mathcal{T}^k$: A set of integers j labeling n tasks

b_i : Capacity of agent i

$$v_{ij} : \text{Value of task } j \text{ when assigned to agent } i \tag{C.4}$$

w_{ij} : Weight of task j on agent i

$$x_{ij} : \begin{cases} 1 & \text{Task } j \text{ is assigned to agent } i \land i \in \mathcal{C}_j^k \\ 0 & \text{otherwise} \end{cases} \tag{C.5}$$

The problem's objective is to maximize the total sum values of the assigned tasks (Equation C.1). Each agent can hold multiple tasks up to its resource limit (Equation C.2) and each task is assigned to at most one agent (Equation C.3). The assignment of tasks to agents is constrained by compatibility constraints (Equation C.5), such that each task can only be placed on a subset of agents.

We state a very general GAP formulation, although our proposed approach is able to handle different GAP variants. The most important and required properties of the formulation are: a) the possibility to have different values per task and agent (Equation C.4), and b) the value maximization objective (Equation C.1).

GAP is NP-hard as it reduces to the NP-hard one-dimensional knapsack optimization problem [37].

3.2 Rotational Diversity

To maintain rotational diversity, it is necessary to control the affinities between tasks and agents. As an indicator, the affinity pressure must not grow too high, which can be avoided by a diverse rotation between tasks and agents.

As part of solving the outer problem, it is necessary to balance profit maximization and reducing affinities by rotating the assignments from tasks to agents. Additional complexity stems from the fact, that at each cycle different sets of agents and tasks are available and the assignment can only take the current cycle into account.

The optimization in the outer problem could be solved by an exhaustive search of possible combinations between profits and affinities, such that an optimal solution can be found. In practice, this is infeasible, as it requires to solve the computationally expensive inner GAP problem multiple times before deciding for the final solution.

4 Maintaining Rotational Diversity

The central idea for maintaining rotational diversity is the manipulation of the values contributing to the objective of the inner assignment problem (see Figure C.2). This adjustment steers the optimization process towards an assignment which is balancing profit maximization and making diverse assignments. The adjustment is made according to a strategy and the state of the available resources, that is tasks and agents available in the current cycle, and their affinities.

Before introducing different adjustment strategies, we describe the mechanism to calculate the affinities and the affinity pressure, and the relevance of their values.

4.1 Assignment Diversity

To achieve rotation of tasks over agents in subsequent cycles, the cycle-specific assignment problem needs an incentive to assign a task to a different agent than

$$\overset{\text{Profits}}{\begin{bmatrix} p_{1,1} & \cdots & p_{1,n} \\ \vdots & \ddots & \vdots \\ p_{m,1} & \cdots & p_{m,n} \end{bmatrix}} \bigcirc \overset{\text{Affinities}}{\begin{bmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{m,1} & \cdots & a_{m,n} \end{bmatrix}} = \overset{\text{Values}}{\begin{bmatrix} v_{1,1} & \cdots & v_{1,n} \\ \vdots & \ddots & \vdots \\ v_{m,1} & \cdots & v_{m,n} \end{bmatrix}}$$

Figure C.2: In the outer problem, profits p and affinities a are combined by a strategy $\bigcirc$ into single values v. These values are used to optimize the GAP in the inner problem.

in previous assignments.

This incentive is described by the notion of affinities between tasks and agents, describing how important an assignment of a task to an agent is to achieve high rotational diversity. A low affinity value corresponds to a recent assignment from the task to the agent, whereas a high affinity indicates the necessity to make this assignment again soon.

The affinities are determined by *Affinity Counting*.

Definition 4.1. Affinity Counting

$$a_{ij}^k = \begin{cases} 0 & \text{if } i \notin \mathcal{C}_j^k & \text{(C.6a)} \\ 1 & \text{if } k = 1 \lor x_{ij}^{k-1} = 1 & \text{(C.6b)} \\ a_{ij}^{k-1} + 1 & \text{if } i \in \mathcal{A}^k \land j \in \mathcal{T}^k & \text{(C.6c)} \\ a_{ij}^{k-1} & \text{otherwise} & \text{(C.6d)} \end{cases}$$

Affinity Counting counts the number of cycles since the last assignment from task j to agent i, starting from 1 at the first cycle or the last assignment (C.6b). If a task and agent are incompatible, the affinity is always 0 (C.6a). At cycle k, the affinity increases for non-selected, but possible assignments in the previous cycle $k-1$ (C.6c)(C.6d).

Naturally, the affinity values increase over time as each task can only be assigned to one of the compatible agents in each cycle. This growth is anticipated and acceptable to a certain degree, while at the same time, growing affinities show the need to make the corresponding assignment soon.

To monitor the overall state of rotational diversity, we define the *Affinity Pressure* metric.

Definition 4.2. Affinity Pressure (AP)
The Affinity Pressure is defined per cycle k and task j:

$$\text{AP}_j^k = \frac{\sum_{i \in \mathcal{A}^k} a_{ij}^k}{|\mathcal{C}_j^k|} - \frac{|\mathcal{C}_j^k| + 1}{2}$$

It is the scaled difference between the actual and ideal affinities, as described below. For the AP calculation, only the task and agents available in that cycle

are considered. Hence, tasks and agents can be added or completely removed without affecting the AP values of the remaining tasks.

In an ideal rotation setting, the affinities of a task j form the set $\{\, i \mid 1 \leq i \leq |\mathcal{C}_j^k| \,\}$, with its sum being the triangular number $\frac{1}{2} \cdot |\mathcal{C}_j^k| \cdot (|\mathcal{C}_j^k| + 1)$. As the task is (ideally) assigned in every cycle, the last assignment has affinity 1, the previous assignment has affinity 2, and so on. With $|\mathcal{C}_j^k|$ compatible agents, the longest unassigned task then has affinity $|\mathcal{C}_j^k|$.

However, in a practical rotation setting, this perfect rotation is hindered by non-availability and limited capacities of the agents. To evaluate the state of rotational diversity, it is, therefore, crucial to consider how long a task has not been assigned to each agent, but also, from an agent's perspective, the time it has not executed certain tasks.

The AP metric is derived from the difference between the sum of current affinities and the ideal values. For comparability and normalization, it is scaled by the number of possible agents: $\frac{1}{|\mathcal{C}_j^k|} \cdot \left[\sum_{i \in \mathcal{A}^k} a_{ij}^k - \frac{1}{2} \cdot |\mathcal{C}_j^k| \cdot (|\mathcal{C}_j^k| + 1)\right]$

In this formula, the minuend describes the current affinities relative to the number of possible agents, the subtrahend the ideal case with fully regular rotation. A positive excess indicates missed assignments to achieve ideal rotation. Note that the bottom value of 0 is an ideal value, which in practice is usually not achievable, due to selection and limited availability of tasks and agents, and the necessary selection in the GAP assignment problem. During the first $|\mathcal{C}_j^k|$ cycles, the AP for a task is negative, as initially all affinities equal 1. After $|\mathcal{C}_j^k|$ cycles, the AP is always ≥ 0.

Example. *Figure C.3 presents an example of affinities and their development over four cycles. In the initial cycle* 1*, all affinities equal 1 (or 0 for incompatible assignments) and there is no preferred assignment among all possible assignments.*

Over the next cycles, tasks T1 and T2 rotate over all compatible agents, resulting in the AP *value 0 for T1 and T3. Task T3 does not rotate, but is assigned to agent C in two subsequent cycles, which increases the affinity for the assignment to agent B and raises the* AP *to 0.5, an indicator for the imbalance of T3. Note that, in cycle* 3*, T3 is unavailable, but this does not affect its affinities in cycle* 4*.*

Theorem 4.3. *For any set of tasks $\mathcal{T}^k$ and agents $\mathcal{A}^k$ with constant availability, if a task j is always assigned to one of the agents for which it has the highest affinity, a perfect rotation is achieved and the Affinity Pressure is 0.*

Proof. With N possible agents, it takes N cycles to assign a task once to every agent. The affinity is set to 1 after the assignment was made and is increased by 1 at every cycle. After each assignment was made once, the affinity to the first assigned agent is N again, the affinity of the second assigned agent is $N - 1$, and the affinity of the last assigned agent is 1. The sum of affinities is $\sum_{i \in \mathcal{A}^k} a_{ij}^k = \sum_{i=1}^{N} i = \frac{1}{2} N(N+1)$. Using Definition 4.2, and because the number of available agents is constantly $|\mathcal{C}_j^k| = N$, it follows AP $= 0$. $\square$

	A	B	C	AP
T1	1	1	0	-0.5
T2	1	1	1	-1.0
T3	0	1	1	-0.5

(a) Cycle 1

A	B	C	AP
1	**2**	0	0
2	1	**2**	-0.3
0	**2**	1	0

(b) Cycle 2

A	B	C	AP	
T1	**2**	1	0	0
T2	1	2	**3**	0
T3	~~0~~	~~**3**~~	~~1~~	~~0.5~~

(c) Cycle 3

A	B	C	AP
1	**2**	0	0
2	**3**	1	0
0	**3**	1	0.5

(d) Cycle 4

Figure C.3: Affinities and Affinity Pressure of three tasks T1, T2, T3 and agents A, B, C over four cycles (Bold: Ideal; Highlighted: Assignment in cycle k; Strikethrough: Task unavailable)

Algorithm 1 Solving a single cycle of the multi-cycle assignment problem under consideration of rotational diversity.

1: **function** ExecuteCycle($\mathcal{T}^k, \mathcal{A}^k$)
2: $AP^k \leftarrow$ Calculate Affinity Pressure AP($\mathcal{T}^k$, $\mathcal{A}^k$)
3: $\mathcal{T}^k_{Values} \leftarrow$ Strategy($\mathcal{T}^k$, $\mathcal{A}^k$)
4: Assignment $\leftarrow$ Solve GAP($\mathcal{T}^k_{Values}$, $\mathcal{A}^k$)
5: $\mathcal{T}^k$, $\mathcal{A}^k \leftarrow$ UpdateAffinity($\mathcal{T}^k$, $\mathcal{A}^k$, Assignment)
6: **return** $\mathcal{T}^k$, $\mathcal{A}^k$, Assignment
7: **end function**

4.2 Strategies

A central strategy balances profit maximization and diverse assignments, by controlling the combination of profits and affinities into values. This combination then steers the focus of the single-objective GAP solver.

The general optimization scheme for a single cycle is shown in Algorithm 1. First, the state of the system, given by available tasks and agents, and the affinity pressure are gathered. Second, the task values are derived, and the cycle's GAP is solved. Finally, based on the actual assignments, the affinities of the available tasks are updated. This procedure adds little overhead to a process where no rotation is considered, as the main computational effort remains in the central GAP.

The selected strategy remains constant for the whole process, i.e. every cycle uses the same strategy. It is nevertheless possible for the strategy to be adaptive and adjust its behaviour according to current state of tasks, agents, and affinities. At the beginning of every cycle, the strategy calculates the profit values, based on profits, affinities, and (if required) other information about the current state.

These values are then taken as parameters in the current cycle's GAP instance.

In the following, we present six strategies to control rotational diversity:

Strategy 1: Objective Switch (OS/γ)
The Objective Switch strategy maintains rotational diversity by monitoring the affinity pressure, and, if it reaches a threshold γ, switches from profit to affinity values:

$$v_{ij} \triangleq \begin{cases} p_{ij} & \text{if } \gamma > \max_{j \in \mathcal{T}^k} \mathrm{AP}_j^k \\ a_{ij} & \text{otherwise} \end{cases}, \quad \forall\, i \in \mathcal{A}^k,\ j \in \mathcal{T}^k$$

The threshold γ is a fixed, user-defined configuration parameter, and selected according to the desired trade-off between maximized profits and high rotational diversity.

The objective switch strategy exchanges the focus of the optimization procedure to specifically address a single optimization goal. It has the intuition, that it is most effective to focus on the rotation goal as soon as the need, quantified by the affinity pressure and γ, arises.

Strategy 2: Product Combination (PC)
In the Product Combination strategy, profit and affinities are multiplied to form the task values:

$$v_{ij} \triangleq p_{ij}^{\alpha} \cdot a_{ij}^{\beta}, \quad \forall\, i \in \mathcal{A}^k,\ j \in \mathcal{T}^k$$

The exponents α and β allow configuration of the strategy to emphasize one aspect or to account for different scales of profits and affinities in specific applications. In our experiments, we found a standard configuration of $\alpha = \beta = 1$ to be intuitive and well-performing. Therefore, this strategy does not require additional configuration, but allows for adjustments if necessary.

In the PC strategy, there is not active reaction on the overall state of rotational diversity, as in the OS/γ strategy, but higher affinities values implicitly influence the profits and put an emphasis on tasks with missing rotation.

Strategy 3: Weighted Partial Profits (WPP)
The WPP strategy calculates task values with a weighted sum:

$$v_{ij} \triangleq \lambda_j^k \cdot \frac{p_{ij}}{\max\limits_{i \in \mathcal{A}^k} \max\limits_{j \in \mathcal{T}^k} p_{ij}} + (1 - \lambda_j^k) \cdot \frac{a_{ij}}{\max\limits_{i \in \mathcal{A}^k} \max\limits_{j \in \mathcal{T}^k} a_{ij}}, \quad \forall\, i \in \mathcal{A}^k,\ j \in \mathcal{T}^k$$

The task- and cycle-specific weight parameter λ_j^k balances the influence of each objective on the final value v_{ij}. λ_j^k is self-adaptive and depends on the ratio between ideal and actual affinities, similar to the affinity pressure. When the rotational diversity is high, the influence of the profits is high, too, otherwise the affinities have higher influence:

$$\lambda_j^k = \frac{\frac{1}{2} \cdot |\mathcal{C}_j^k| \cdot (|\mathcal{C}_j^k| + 1)}{\sum_{i \in \mathcal{A}^k} a_{ij}}, \quad \forall\, j \in \mathcal{T}^k$$

To account for different value ranges, both profits and affinities are scaled to $[0, 1]$ by their respective maxima.

Strategy 4: Fixed Objective: Profit (FOP)
Each task value equals the static profit value:

$$v_{ij} \triangleq p_{ij}\,, \quad \forall\, i \in \mathcal{A}^k,\; j \in \mathcal{T}^k$$

Strategy 5: Fixed Objective: Affinity (FOA)
Each task value equals the affinity value:

$$v_{ij} \triangleq a_{ij}\,, \quad \forall\, i \in \mathcal{A}^k,\; j \in \mathcal{T}^k$$

FOP and FOA represent special cases of the PC strategy, with $\beta = 0$ respectively $\alpha = 0$. These strategies are the two most extreme approaches, because each of them ignores the other goal, albeit profits or affinities. They serve as comparison baselines to evaluate the trade-offs by the other strategies.

In contrast to the discussed strategies, which manipulate the task values, we consider an additional approach to maintain rotational diversity. This approach does not only manipulate the task values, but also restricts the possible assignments between tasks and agents.

4.3 Limited Assignment

The *Limited Assignment* approach explicitly constrains a task i to be assigned to compatible and available agents with a high affinity value. This is achieved by artificially limiting the possible assignments through temporarily manipulating the compatibility between tasks and agents. Limited Assignment does not further manipulate the profit values, but only works on the level of possible assignments, i.e. the task value equals the static profit value: $v_{ij} \triangleq p_{ij}$ Therefore, this approach can be combined with any of the strategies as an additional control mechanism.

Specifically, the compatibility between a task and an agent is removed if the affinity a_{ij} is below a threshold value. The threshold value th can either be a static, user-defined parameter, or dynamically adapted. As a heuristic for the threshold value, we propose using the mean affinity between a task and all available and compatible agents, rounded to the next smallest integer:

$$Threshold_j = \left\lfloor \frac{1}{|\mathcal{C}_j^k|} \sum_{i\,\in\,\mathcal{C}_j^k} a_{ij} \right\rfloor, \quad \forall\, j \in \mathcal{T}^k \tag{C.7}$$

As this approach reduces the search space of possible solutions for the instance, it can lead to the removal of optimal solutions from the solution space. However, in many settings it is neither necessary nor possible to find the optimal solution, and finding a near-optimal solution in a reduced solution space is sufficient.

To further understand the Limited Assignment approach, we revisit the example for affinity calculation (Figure C.3) from Section 4.1.

	A	B	C	Th
T1	1	1	0	1
T2	1	1	1	1
T3	0	1	1	1

(a) Cycle 1

A	B	C	Th
1	**2**	0	1
2	X	**2**	2
0	**2**	1	1

(b) Cycle 2

A	B	C	Th	
T1	**2**	1	0	1
T2	X	2	**3**	2
T3	~~0~~	~~**3**~~	~~1~~	~~2~~

(c) Cycle 3

A	B	C	Th
1	**2**	0	1
2	**3**	X	2
0	**3**	X	2

(d) Cycle 4

Figure C.4: Effect of Limited Assignment on three tasks T1, T2, T3 and agents A, B, C over four cycles (Th: Threshold value; X: Compatibility temporarily removed through Limited Assignment; Bold: Ideal next assignment; Highlighted: Assignment in cycle k; Strikethrough: Task unavailable)

Example. *Figure C.4 presents an example of affinities and their development over four cycles. In the initial cycle* 1*, all affinities equal 1 (or – for incompatible assignments) and there is no preferred assignment among all possible assignments.*

Over the next cycles, tasks T1 and T2 rotate over all compatible agents, resulting in the AP *value 0 for T1 and T3. Task T3 does not rotate, but is assigned to agent C in two subsequent cycles, which increases the affinity for the assignment to agent B and raises the* AP *to 0.5, an indicator for the imbalance of T3. Note that, in cycle* 3*, T3 is unavailable, but this does not affect its affinities in cycle* 4*.*

5 Experimental Evaluation

We consider three problems for evaluation: a) a multi-cycle variant (MCMKP) of the known multiple knapsack problem (MKP) to evaluate trade-offs between the strategies; b) a multi-cycle variant (MCMSSP) of the multiple subset sum problem (MSSP) to evaluate the behaviour with a smaller number of agents and low task availability; c) test case selection and assignment (TCSA) as a real-world case study from the software testing domain to evaluate the practical interest of our approach.

5.1 Implementation and Setup

Our strategies and the experimental setup are implemented in Python. The assignment problem is modeled with MiniZinc 2.0 [38], following the presented GAP formulation, and is solved with IBM CPLEX 12.8.0. Our implementation

and all test data is available online at https://github.com/HelgeS/mcap_rotational_diversity.

We note that there are further domain-specific heuristics and exact algorithms to solve knapsack problems, e.g. [39], but as the GAP model and its solver are a black-box to our strategies, their optimization is not in the scope of our work, and we employ a generic model formulation and solver. To ensure the solution quality with a reasonable time-contract for the solver, we compared it on a set of sample instances with mulknap[1], an exact MKP solver [40]. With a 60 second timeout, CPLEX achieves on average 99.5 % of the optimal solution calculated by mulknap.

All strategies are run on each scenario with a 60 second timeout for the GAP solver. The thresholds γ for the Objective Switch strategy are 10, 20, 30, and 40, except for the MCMSSP problem, where we use smaller γ of 1, 2, 4, and 10.

We evaluate the full rotation of tasks over agents, both looking at all tasks, and at each individual task. One full rotation over all tasks is achieved, when each task was assigned once to all compatible agents. The rotation over one task describes how often a task is assigned to its compatible agents on average. These numbers can be different. If few tasks are not rotated, those forestall full rotations, but allow other tasks to be frequently rotated.

Furthermore, we compare the achieved profit of the assignments with the profit of the FOP strategy, which does not consider rotation and only maximizes profit. As the other experimental parameters are the same and also the same assignment model is used, FOP simulates the baseline setting without rotation-awareness.

We have considered an additional baseline, where the full multi-cycle assignment problem is optimized as one single optimization model. This differs from our method, as each task's and agent's availability is known already in the beginning. However, due to the exceeding model size, solving the extended GAP model is computationally expensive and did not yield a comparable solution within 24 CPU hours, which is substantially more than the total computational cost of successively optimizing individual cycles. Therefore, we do not further consider this baseline.

5.2 Multi-Cycle Multiple Knapsack Problem

MKP is a variant of the 0-1 knapsack problem, and thereby of GAP, with multiple agents, i.e. knapsacks [13, 41].

We extend MKP to a multi-cycle variant (MCMKP) with limited availability of tasks and agents. In every cycle, the same MKP instance has to be solved under consideration of the assignments made in previous cycles and changing availability of tasks and agents.

[1] http://www.diku.dk/~pisinger/codes.html

5.2.1 Setup

To generate problem instances, we employ the procedure by Pisinger (1999) [40], as described in Fukunaga (2011) [42], and extend it to the notion of compatibility and availability. An instance is generated by first creating random tasks with weights from a uniform distribution ($w_j \sim \mathcal{U}[10, 1000]$). The profits of the tasks are either uncorrelated, i.e. profits are drawn from the same uniform distribution, or weakly correlated, i.e. the profits are calculated by $p_j = w_j + \mathcal{U}[-99, +99], \forall j \in \mathcal{T}^k$.

After generating the tasks, the agents $a_1, a_2, a_i, \ldots, a_{m-1}$ are generated and set to 40–60 % of the tasks' weight. An exception is the last agent a_m, whose capacity is set such that the total capacity of all agents equals half of the tasks' demand. The instance sizes are 30/75, 15/45, and 12/48 agents, respectively tasks. For this generation scheme, a ratio $|\mathcal{T}^k|/|\mathcal{A}^k|$ slightly larger than 2 leads to hard instances, while instances of higher ratios become easier to solve [42]. The number of cycles is three times the number of tasks, to allow multiple assignments between tasks and agents, even if an agent has only capacity for one task.

A notion of compatibility is implicit in the generation procedure. Tasks that do not fit into an agent's capacity are automatically incompatible. However, this skews the number of compatible tasks to those agents with high capacities, and puts more emphasis onto their assignments.

From all combinations of the four parameters, we generate 24 instances with in total 4032 assignment problems for evaluation. Every instance is run with each strategy alone and in combination with Limited Assignment.

5.2.2 Pure Strategies

We first discuss the rotational diversity results grouped by agent and task availability, that is in four different groups, as this is the main differentiating attribute of the MCMKP scenarios. The results for the pure strategies without limited assignment are shown in the upper half of Table C.1 and the results for the combination with limited assignment in the lower half. The profit values of all MCMKP results have been scaled in relation to the best achieved profit, which corresponds to the profit-only optimization without limited assignment. That means, also the results for the strategies with limited assignment in Table C.1 are scaled in relation to the results without limited assignment. All strategies rank between the extreme baselines, FOA and FOP, that only focus on one aspect of the problem formulation, either rotation or profit optimization.

In the MCMKP, specifically the OS/γ strategies are effective to achieve either good rotation or good profit optimization while still having better results in the other optimization goal in comparison to the baselines. However, as the γ value is a fixed parameter of these strategies, the strategies cannot effectively balance the two goals for a better result. With a low $\gamma = 10$, OS/10 shows similar performance than FOA, but with a small improvement on profit optimization in cycles where the overall affinity pressure is low. With the higher $\gamma = 40$, OS/40

		Available Agents Available Tasks	75% 75%	75% 100%	100% 75%	100% 100%	Average
(a) Pure Strategies	Rotational Diversity	OS/10	**2.0 (4.3)**	**2.0 (4.6)**	**3.0 (5.3)**	**3.3 (5.9)**	**2.0 (5.0)**
		OS/20	1.6 (3.8)	2.0 (4.4)	1.6 (3.9)	3.0 (5.2)	2.0 (4.3)
		OS/30	1.0 (3.1)	1.3 (3.8)	1.3 (3.3)	2.3 (4.4)	1.0 (3.7)
		OS/40	0.6 (2.7)	1.3 (3.3)	0.6 (3.0)	1.6 (3.8)	1.0 (3.2)
		PC	0.0 (4.2)	0.0 (4.4)	0.0 (5.4)	0.0 (5.9)	0.0 (5.0)
		WPP	1.3 (4.0)	1.6 (4.2)	1.6 (4.9)	1.6 (5.3)	1.0 (4.6)
		FOA	2.3 (4.5)	2.0 (4.8)	3.0 (5.7)	3.3 (6.1)	2.0 (5.3)
		FOP	0.0 (1.6)	0.0 (1.5)	0.0 (1.9)	0.0 (2.0)	0.0 (1.7)
	Profit (% of FOP)	OS/10	87.8	83.7	88.5	84.9	86.2
		OS/20	90.0	84.6	92.0	86.8	88.4
		OS/30	92.9	87.1	94.2	89.7	91.0
		OS/40	**94.9**	89.2	**95.7**	**92.1**	**93.0**
		PC	92.3	**90.5**	90.5	90.5	91.0
		WPP	88.6	83.1	93.3	88.1	88.3
		FOA	87.0	82.8	86.8	84.1	85.1
		FOP	100.0	100.0	100.0	100.0	100.0
(b) Limited Assignment	Rotational Diversity	OS/10	2.0 (4.4)	2.0 (4.7)	2.6 (5.3)	3.3 (6.1)	2.0 (5.1)
		OS/20	1.6 (4.1)	2.0 (4.6)	1.6 (4.7)	3.0 (5.7)	2.0 (4.8)
		OS/30	1.3 (3.7)	1.6 (4.3)	1.3 (4.5)	2.3 (5.2)	1.0 (4.4)
		OS/40	1.0 (3.4)	1.3 (4.0)	0.6 (4.2)	1.6 (4.6)	1.0 (4.1)
		PC	0.0 (4.2)	0.0 (4.5)	0.0 (5.4)	0.0 (5.9)	0.0 (5.0)
		WPP	**2.3 (4.4)**	**2.3 (4.6)**	**3.3 (5.8)**	**4.0 (6.0)**	**3.0 (5.2)**
		FOA	2.0 (4.5)	2.0 (4.8)	3.0 (5.8)	3.3 (6.2)	2.0 (5.3)
		FOP	0.0 (2.9)	0.0 (3.0)	0.0 (3.5)	0.0 (3.6)	0.0 (3.3)
	Profit (% of FOP)	OS/10	87.6	83.5	88.2	84.8	86.0
		OS/20	89.1	84.3	90.8	86.4	87.6
		OS/30	91.8	86.3	91.6	88.7	89.6
		OS/40	**93.0**	88.3	**92.3**	**90.4**	**91.0**
		PC	92.0	**90.2**	90.4	**90.4**	90.8
		WPP	85.5	81.2	86.3	83.2	84.1
		FOA	87.0	82.8	86.7	84.1	85.1
		FOP	95.9	95.6	91.3	94.5	94.3

Table C.1: Results for MCMKP. The best results for each (a) pure strategies and (b) Limited Assignment are marked in bold without considering the FOA/FOP baselines.

has the highest profit among the strategies, except FOP, and still better rotation results than FOP and, in terms of full rotations, also than PC.

The PC strategy multiplies profits and affinities into one value and achieves competitive results for the MCMKP scenario in terms of profit maximization and average rotations, but it does not yield a schedule with full rotations of task. This is an indication of one or a few tasks that are blocked from achieving full rotations, e.g. because they have low profit values or limited availability either in themselves or the compatible agents, and are therefore not sufficiently scheduled. As noted before, the MCMKP generation procedure shifts the compatibility of

tasks to the last agent, to which most tasks are compatible. Still, its capacity is limited and only a few tasks, especially those with a height weight, can be assigned per cycle. MCKMP thereby creates a bottleneck for diverse assignments, which PC does not efficiently overcome. Strategies which focus more directly on achieving rotational diversity have an advantage in that case. However, we see this result specifically in the MCMKP scenario, but not for the other case study, which we will discuss below.

WPP shows a similar performance as OS/20 while following a more flexible value combination strategy than switching the objective. This results in very similar average results over all scenarios, but larger differences, for example in the case of 100 % availability for both agents and tasks. Here, OS/20 achieves 3 full rotations, but WPP only 1.6, even though the average rotations are similar with 5.2 respectively 5.3. We can attribute the potential miss of full rotations in WPP to the same causes as PC as both strategies follow a similar approach of combining the profit and the affinities into a single value, whereas OS/γ uses either the one or the other.

Furthermore, we analyze the influence of varying availability on the achievable rotations. As the setting is such that a selection of tasks has to occur (the resource demand is higher than the resource supply), the availability of a large number of agents has a stronger influence than a high task availability. However, for making diverse assignments, a high task availability is beneficial. This can be seen when comparing the results with 75 %/100 % and 100 %/75 % agent respectively task availability. The more profit-oriented variants of OS/γ achieve better rotation in the former than in the latter case, as OS/γ switches focus, and potentially the optimization objective of a cycle does not match the availability of the tasks. Then, one task might only be present at profit maximization, but not for rotation optimization.

5.2.3 Limited Assignment

When considering the strategies in combination with Limited Assignment, where the compatibility between tasks and agents is limited to those pairings with high affinity values, we observe two main effects in the MCMKP case study. First, reducing the search space for assignments is effective for increasing rotational diversity. For all strategies, an increase in average rotations can be observed and for all strategies, except PC, also an increase in full rotations. Even FOA, which already in its pure version optimizes rotations can benefit from the limitations on possible assignments, albeit only to a small degree. This is an indication that the pure version might make assignments with little benefit, either to optimize the cycle's objective by choosing two low-affinity tasks with small weights over a high-affinity task with higher weight or due to limited time to solve the inner problem. The biggest performance difference in rotational diversity is observed for WPP, which maintains the highest level of rotational diversity, and actually surpasses even the FOA baseline in terms of full rotations, but not average rotations.

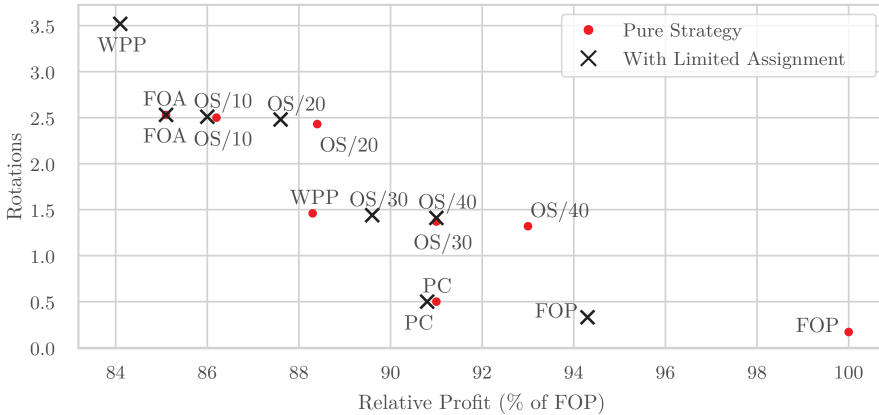


Figure C.5: MCMKP: Distribution of the average results for each strategy without (marked with red circle) and with (marked with black X) limited assignment. Rotations are calculated as Full Rotations + Average Rotations/10.

Second, while increasing the maintainability of rotational diversity through limited the search space, solutions with a high profit value are potentially removed and the total profit is expected to be reduced when using limited assignment. This effect is reflected in the results, but only to a small degree for most strategies. The largest difference occurs in the scenario with 100 % available agents and 75 % available tasks. Here, the profit of WPP is reduced by 7 and FOP by 8.7 percentage points.

5.2.4 Distribution of Results

In Figure C.5, the distribution of the average profit and rotational diversity over all problem instances is visualized. Each point corresponds to the average result of the strategy, including those with the additional limited assignment strategy. The full and average rotations shown in Table C.1 are aggregated into a single value as Full Rotations + Avg. Rotations/10. It shows the trade-off between profit maximization and achieving high rotational diversity, as seen by the extreme points FOP and WPP with limited assignment.

The figure visually confirms the previous discussion on the spread of the strategies' performance with WPP on the one end, achieving high rotational diversity, but comparatively low profit-optimization, and FOP on the other end. In between these extreme points, the other strategies are spread. In terms of solution dominance, OS/20 and OS/40 without Limited Assignment show a balanced performance that surpasses the other strategies in either one of the objectives, while not being worse in the other.

However, in general, the role of the trade-off between rotations and profit has to evaluated in the context of the specific use case. While it can be acceptable to observe a moderate to high decrease in profit in some applications and therefore

maintain rotational diversity in a balanced manner, in other use cases profit optimization is the main driver and rotations only of secondary importance. Depending on how these two goals are valued, or if both are equally important, a more informed decision on the applicable strategy can be made.

5.3 Multi-Cycle Multiple Subset Sum

The second experiment evaluates the proposed method on an extension of another assignment problem variant, the multiple subset sum problem (MSSP) [13, 43]. In MSSP tasks are assigned to a number of identical bins with limited capacity, such that the total sum of weights of packed items is maximized. For our GAP formulation MSSP is implemented as an assignment problem with a number of agents with equal capacity and tasks with same weight and profit. The multi-cycle variant of MSSP (MCMSSP) requires, as in the previous experiments, to repeatedly solve variations of a MSSP instance under the additional goal to produce diverse assignments over subsequent cycles.

One practical example of this problem is described in [43]. In a production planning setting where a given number of fixed size raw material is available at each day, e.g. steel slabs, it is necessary to distribute the products, which have to be produced, onto the slabs such that only little material is wasted. The aspect of rotational diversity can be introduced if the raw material is of fixed size, but with different attributes, e.g. color, and all products should be more or less equally often produced in each material type. If the product demand is further irregular, it is difficult to plan ahead and the product respectively task availability can each between production cycles.

5.3.1 Setup

For problem instances, we base our generation procedure again on [42]. In this scenario task weights are uniformly drawn from the low precision distribution $[10, 100]$ and a task's profit equals its weight, $p_j = w_j$ with the same weight for all agents. All agents have the same capacity, $b_i = c, \forall i \in \mathcal{A}^k$, where c is chosen to be 50 % of the weight of all tasks, $c = 0.5 \sum_{j \in \mathcal{T}^k} w_j$. Our focus in this experiment is on the influence of task availability on the rotation mechanism. Therefore, we do not limit the agents' availability or the compatibility between tasks and agents. All agents are available in every cycle and every task can be assigned to every agent. However, the total capacity of the agents is limited, such that a selection of tasks has to be made, and the availability of tasks is limited. We generate instances of 100 cycles with 20 tasks and either 1 or 5 agents and a task availability of either 50 or 75 % per cycle. Due to the smaller number of agents in this experiment, we reduce the values for γ in the OS/γ strategy to 1, 2, 4 and 10.

		Number of Agents Num. of Tasks/Available	1 20 / 50 %	1 20 / 75 %	5 20 / 50 %	5 20 / 75 %
(a) Pure Strategies	Rotational Diversity	OS/1	**32 (44.9)**	29 (55.5)	5 (7.6)	6 (9.5)
		OS/2	28 (44.0)	36 (51.9)	4 (7.0)	5 (9.4)
		OS/4	27 (43.7)	31 (49.0)	3 (5.7)	5 (9.1)
		OS/10	27 (43.7)	28 (47.8)	1 (4.6)	2 (7.6)
		PC	30 (44.1)	**41 (49.8)**	**5 (7.6)**	5 (9.0)
		WPP	30 (43.7)	38 (48.4)	5 (7.4)	**6 (9.0)**
		FOA	29 (45.4)	28 (55.5)	4 (7.8)	5 (9.5)
		FOP	27 (43.7)	27 (47.8)	1 (4.4)	0 (6.6)
	Profit (% of FOP)	OS/1	97.8	95.2	96.0	93.0
		OS/2	99.5	97.1	96.5	93.2
		OS/4	**100.0**	99.1	98.2	93.9
		OS/10	**100.0**	**100.0**	**99.8**	**97.7**
		PC	99.9	99.9	97.6	95.0
		WPP	99.9	99.9	99.3	94.8
		FOA	96.4	95.0	95.5	92.9
		FOP	100.0	100.0	100.0	100.0
(b) Limited Assignment	Rotational Diversity	OS/1	**32 (44.9)**	29 (55.5)	4 (7.5)	5 (9.4)
		OS/2	28 (44.0)	36 (51.9)	4 (7.2)	5 (9.5)
		OS/4	27 (43.7)	31 (49.0)	4 (7.0)	5 (9.3)
		OS/10	27 (43.7)	28 (47.8)	2 (6.7)	3 (8.8)
		PC	30 (44.1)	**41 (49.8)**	**6 (7.7)**	5 (9.1)
		WPP	30 (43.7)	38 (48.4)	5 (7.6)	**6 (9.1)**
		FOA	29 (45.4)	28 (55.5)	5 (7.7)	5 (9.5)
		FOP	27 (43.7)	27 (47.8)	2 (6.7)	0 (8.6)
	Profit (% of FOP)	OS/1	97.8	95.2	95.3	92.7
		OS/2	99.5	97.1	96.7	92.9
		OS/4	**100.0**	99.1	98.8	93.8
		OS/10	**100.0**	**100.0**	**99.5**	**97.9**
		PC	99.9	99.9	97.3	94.5
		WPP	99.9	99.9	96.0	92.9
		FOA	96.4	95.0	95.0	92.6
		FOP	100.0	100.0	99.5	99.8

Table C.2: Results for multi-cycle MSSP. The best results for each (a) pure strategies and (b) Limited Assignment are marked in bold without considering the FOA/FOP baselines.

5.3.2 Single Agent Scenarios

The results for the MCMSSP experiments are shown in Table C.2. In the scenario with only a single agent, the Limited Assignment approach has no effect, because no compatibility between tasks and the single agent can be removed and it is not part of this approach to completely remove tasks from the optimization problem. Therefore, the results for the scenarios with a single agent are identical to the previously discussed Pure Strategies. Due to the single agent scenario, the number of full rotations is high and expresses how often each tasks has been assigned, as there is no actual rotation over different but just being assigned or

not being assigned. Still, there is a difference visible between the strategies in the diversity of the assignments and task selection. For example, in the scenario with one agent and a task availability of 75 %, the PC strategy with 41 full rotations achieves 40 % more rotations compared to OS/10 with 28 full rotations, while maintaining a similar level of profit (99.9 % vs. 100 %).

In general, in the single agent scenarios, rotational diversity can be maintained with only 0.1 % reduction in profits by using the PC or WPP strategy. Selecting the tasks to be assigned carefully is still important, as it is not possible to assign all available tasks per cycle. The utilization of the agent and the percentage of assigned tasks shows no major differences between the strategies. In the scenario with 50 % available tasks, on average 90 % of these tasks are assigned, with an agent utilization of 90 %, including for the profit strategy. With 75 % available tasks, the utilization increases to 99 % and 65–67 % of the available tasks are assigned on average.

5.3.3 Multiple Agent Scenarios

When comparing the results for five agents and 20 tasks, the difference is in the task availability within each cycle, which is either 50 or 75 %. Between the results of the pure strategies and Limited Assignment, we see a similar effect as before, such that Limited Assignment is effective to increase full rotations for most strategies with a profit trade-off.

The difference between scenarios with five agents but different task availability, i.e. 50 % respectively 75 %, shows as a higher task availability increases the possible assignments and thereby the profit gap between rotation-oriented strategies and the profit-only baseline FOP. A low task availability restricts the possible assignments of all strategies and thereby leads to a smaller gap in profits. Still, employing a rotational diversity strategy allows to control this trade-off.

5.4 Test Case Selection and Assignment

As a third and final case study, we employ the real-world application of Test Case Selection and Assignment (TCSA) for cyber-physical systems [44], such as industrial robots. TCSA usually occurs in Continuous Integration (CI) processes, where new releases of the robot control software are regularly integrated and released [45]. Typically, CI involves assigning test cases to test agents several times a day. Comprehensive test suites exist, but available time and hardware for their execution are limited. Then it is necessary to distribute a selection of the most relevant test cases over the available agents. The test case relevance is given by an upstream test case prioritization process [46, 47]. This priority can be different at each cycle, due to discovered failures or changes in the system-under-test. The assignment of tests to agents is constrained by the available time and compatibility between test and agent. In the GAP terminology, the test case priority resembles the profit, the test's duration the task weight, and an agent's available time its capacity.

Additionally, the availability of agents is influenced by maintenance, technical faults, or short-term usage in other projects, and the set of test cases changes due to the ongoing development. If TCSA were to be solved by a static assignment, this changing availability would create a need for frequent updates and schedule repairs. Therefore, a static schedule is not practically applicable without additional effort. Instead, to capture the dynamic setting, an individual selection and assignment has to be made at each cycle. Enforcing diverse assignments increases the coverage of tasks and agents, and thereby the confidence into the system-under-test.

5.4.1 Setup

We evaluate the strategies on TCSA, based on actual test data from our industrial partner. A set of test cases is to be divided among a number of test agents. The test case selection has to select those tests with the highest priorities, which is assigned externally, and to ensure a rotation of tests between agents, such that a test is frequently executed on all compatible agents. All test agents have the same capacity, that is the time available for a test cycle, which is 10 hours. Due to unique hardware specifications and different functionality, a test case is compatible with approximately 60 % of the test agents. The runtime of a test case varies from 1 to 21 minutes, but is identical for each test agent. In practice, test agents are not exclusively available for testing or might be defect and test cases are temporarily removed from the test suite. Therefore, an average of 40 % of the agents and 10 % of the test cases are unavailable for 3–7 cycles. In total, we consider four scenarios, 20 agents with 750, 1500, and 3000 test cases, and 30 agents with 3000 test cases.

5.4.2 Pure Strategies

For the pure strategies, without additional Limited Assignment, the results are shown in the upper half of Table C.3. In the smallest scenario, a full rotation of all tests over all possible agents is achieved 14–15 times over 365 cycles, i.e. every 24–26 days. Here, each task is compatible to circa 12 agents (60 %), and 60 % of the agents are unavailable for multiple cycles. For the larger scenarios with the same number of agents, the number of full rotations reduces approximately linear, but not the number of average rotations per task. This shows, that some tests are not evenly rotated and hinder the completion of full rotations. With a larger number of agents, the average number of rotations per tasks drops, as there are more compatible agents and more cycles are necessary for a full rotation.

The profits earned from the assignments (see lower half of Table C.3) are close to the FOP baseline for all strategies in the smallest scenario, but decrease with a higher number of tests, except for PC, which is able to balance profit maximization and rotation better than the other strategies and even outperforms FOA for complete rotations. For PC, the profit trade-off is always less than 10 %, and on average less than 4 % in comparison to the profit-oriented FOP.

		Number of Agents Number of Tasks	20 750	20 1500	20 3000	30 3000	Average
(a) Pure Strategies	Rotational Diversity	OS/10	13 (22.0)	6 (15.5)	**3 (9.3)**	**3 (8.4)**	6 (13.8)
		OS/20	8 (18.5)	6 (15.2)	3 (9.2)	3 (8.3)	5 (12.8)
		OS/30	6 (17.0)	5 (14.2)	3 (9.0)	3 (8.1)	4 (12.1)
		OS/40	6 (16.4)	4 (13.1)	3 (8.9)	3 (7.9)	4 (11.6)
		PC	**15 (24.0)**	**7 (14.3)**	3 (8.2)	3 (7.5)	**7 (13.5)**
		WPP	14 (24.0)	7 (14.1)	3 (7.3)	3 (7.0)	6 (13.1)
		FOA	15 (24.3)	6 (15.6)	3 (9.5)	3 (8.5)	6 (14.5)
		FOP	5 (15.9)	0 (10.8)	0 (7.0)	0 (4.6)	1 (9.6)
	Profit (% of FOP)	OS/10	96.4	79.5	67.8	74.9	79.6
		OS/20	98.1	80.0	68.3	75.3	80.5
		OS/30	99.0	84.7	69.0	75.9	82.2
		OS/40	99.4	90.8	69.6	76.5	84.1
		PC	**99.7**	**97.7**	**91.1**	**96.6**	**96.3**
		WPP	98.1	77.3	54.7	66.6	74.2
		FOA	96.1	79.0	67.4	74.4	79.2
		FOP	100.0	100.0	100.0	100.0	100.0
(b) Limited Assignment	Rotational Diversity	OS/10	14 (22.5)	6 (15.5)	3 (9.4)	3 (8.4)	6 (13.9)
		OS/20	10 (20.4)	6 (15.4)	3 (9.3)	3 (8.3)	5 (13.4)
		OS/30	11 (20.2)	5 (14.8)	3 (9.2)	3 (8.2)	5 (13.1)
		OS/40	11 (20.3)	4 (14.2)	3 (9.1)	3 (8.1)	5 (12.9)
		PC	14 (23.9)	7 (14.4)	3 (8.2)	3 (7.5)	6 (13.5)
		WPP	**16 (24.1)**	**8 (14.1)**	**4 (7.5)**	**4 (7.2)**	**8 (13.2)**
		FOA	15 (24.3)	6 (15.6)	3 (9.5)	3 (8.5)	6 (14.5)
		FOP	9 (20.2)	2 (14.0)	0 (9.3)	0 (7.0)	2 (12.6)
	Profit (% of FOP)	OS/10	96.7	79.6	67.8	74.9	79.8
		OS/20	99.8	80.1	68.4	75.4	80.9
		OS/30	**100.0**	87.2	68.9	76.0	83.0
		OS/40	**100.0**	94.9	69.6	76.5	85.3
		PC	99.6	**97.7**	**91.1**	**96.6**	**96.3**
		WPP	95.7	72.7	54.8	65.4	72.2
		FOA	96.1	79.1	67.3	74.4	79.2
		FOP	100.0	100.0	100.0	100.0	100.0

Table C.3: Results for TCSA. The best results for each (a) pure strategies and (b) Limited Assignment are marked in bold without considering the FOA/FOP baselines.

WPP, who showed compelling results in the previous experiment, achieves similar rotational diversity to PC, but is less effective in profit optimization.

5.4.3 Limited Assignment

The results for the strategies in combination with Limited Assignment are shown in the lower half of Table C.3. As we observed in the experiments on MCMKP, adding Limited Assignment and thereby reducing the space of possible assignments between tasks and agents, increases the rotational diversity. This is

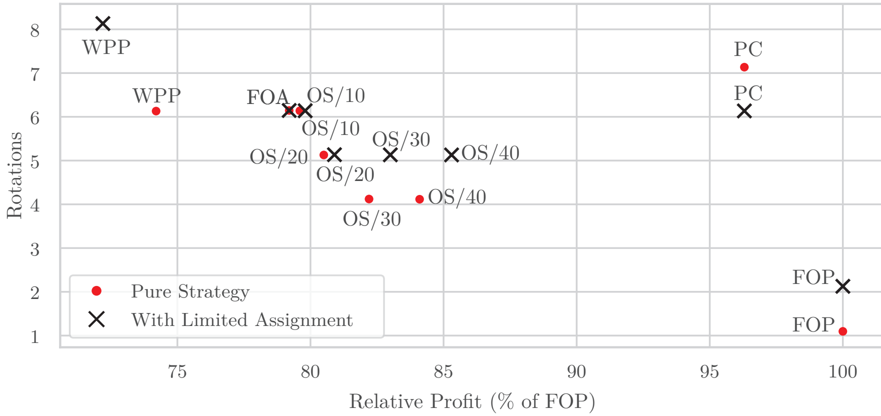


Figure C.6: TCSA: Distribution of the average results for each strategy without (marked with red ∘) and with (marked with black x) limited assignment. Rotations are calculated as Full Rotations + Average Rotations/100.

also true for TCSA. For most strategies, the number of full and average rotations is increased, with the exception of PC in the smallest scenario, where the number of rotations is decreased by a small amount. Interestingly, we do not observe a substantial reduction in assigned profits, but all strategies can maintain similar levels of profits with and without Limited Assignment.

5.4.4 Distribution of Results

As for the MCMKP case study, we visualize the average results of each strategy in terms of rotations and relative profit in Figure C.6. For TCSA, the rotations are calculated as Full Rotations + Average Rotations/100 to account for the high number of average rotations in these scenarios.

The extreme points being WPP and FOP, as in the MCMKP case study, the different OS/γ are located closely to each other. The strong performance of product combination (PC) is also visible in the figure, with pure PC even outperforming the combination of PC and Limited Assignment due to the higher number of full rotations.

6 Conclusion

Rotational diversity is the frequent assignment of a task to all its compatible agents over subsequent cycles. We present a two-part model for its optimization in multi-cycle assignment problems with variable availability of tasks and agents: 1) an inner assignment problem, to optimize the assignment from tasks to agents, and 2) an outer problem, to adjust the task values for the maximization objective of the inner problem.

Five strategies, each having a different approach and trade-offs, and approaches, one using only the strategies to introduce rotational diversity, the other also controlling the compatibility between tasks agents, are evaluated on three case studies. Achieving rotational diversity is possible with a profit trade-off of only 4 % in the test case selection and assignment case study. Both the product combination of profits and affinities, and the objective switch strategy, that focuses on either profit maximization or diverse assignments, efficiently achieve rotational diversity.

For applications of this method, we encourage the reader to start from the product combination strategy. It is straightforward to implement and does not require initial configuration, but it can still be adjusted if necessary.

The combination of profits and affinities into a single task value is efficient for balancing profits and rotation. This is especially the case in settings where an extended multi-objective optimization model is not an alternative. Splitting the problem and its responsibilities allows to use problem-specific, single-objective solvers for the inner problem, or to use problems with additional requirements, e.g. precedence constraints or task-dependencies.

In future work, we aim to apply our approach to other settings, which are derived from the general assignment problem. These settings can include scheduling problems with precedence constraints and task-dependencies. Again, rotation of tasks should be achieved without adding substantial computational costs for rotational diversity.

The affinity metric is also not restricted to assignment problems with rotational diversity, but can be transferred to other domains. One related concept is the idea of persistence, although in an opposite sense to rotational diversity. There, a solution should remain stable, even under uncertainties.

Acknowledgements. This work is supported by the Research Council of Norway (RCN) through the research-based innovation center Certus, under the SFI program. The experiments were performed on the Abel Cluster, owned by the University of Oslo and Uninett/Sigma2, and operated by the Department for Research Computing at USIT, the University of Oslo IT-department.

References

[1] Clarke, L., Johnson, E., Nemhauser, G., and Zhu, Z. “The Aircraft Rotation Problem”. In: *Transportation Research Part A: Policy and Practice* vol. 30, no. 1 (1996), p. 51.

[2] Ma, Y., Chu, C., and Zuo, C. “A Survey of Scheduling with Deterministic Machine Availability Constraints”. In: *Computers and Industrial Engineering* vol. 58, no. 2 (2010), pp. 199–211.

[3] Chiaramonte, M. V. “Competitive Nurse Rostering and Rerostering”. PhD thesis. Arizona State University, 2008.

[4] Azizi, N., Zolfaghari, S., and Liang, M. “Modeling Job Rotation in Manufacturing Systems: The Study of Employee’s Boredom and Skill Variations”. In: *International Journal of Production Economics* vol. 123, no. 1 (2010), pp. 69–85.

[5] Ernst, A. T., Jiang, H., Krishnamoorthy, M., and Sier, D. “Staff Scheduling and Rostering: A Review of Applications, Methods and Models”. In: *European Journal of Operational Research* vol. 153, no. 1 (2004), pp. 3–27.

[6] Musliu, N., Schutt, A., and Stuckey, P. J. “Solver Independent Rotating Workforce Scheduling”. In: *Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR).* Vol. 10848. LNCS. 2018, pp. 429–445.

[7] Glover, F., Løkketangen, A., and Woodruff, D. “Scatter Search to Generate Diverse MIP Solutions”. In: *OR Computing Tools for Modeling, Optimization and Simulation: Interfaces in Computer Science and Operations Research* vol. 12 (2000), pp. 299–317.

[8] Hebrard, E., Hnich, B., O’Sullivan, B., and Walsh, T. “Finding Diverse and Similar Solutions in Constraint Programming”. In: *Proceedings of the Twentieth National Conference on Artificial Intelligence.* 2005, pp. 372–377.

[9] Trapp, A. C. and Konrad, R. A. “Finding Diverse Optima and Near-Optima to Binary Integer Programs”. In: *IIE Transactions* vol. 47, no. 11 (2015), pp. 1300–1312.

[10] Petit, T. and Trapp, A. C. “Finding Diverse Solutions of High Quality to Constraint Optimization Problems”. In: *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI).* 2015, pp. 260–266.

[11] Spieker, H., Gotlieb, A., and Mossige, M. “Rotational Diversity in Multi-Cycle Assignment Problems”. In: *Thirty-Third AAAI Conference on Artificial Intelligence.* 2019, pp. 7724–7731.

[12] Ross, G. T. and Soland, R. M. “A Branch and Bound Algorithm for the Generalized Assignment Problem”. en. In: *Mathematical Programming* vol. 8, no. 1 (Dec. 1975), pp. 91–103.

[13] Martello, S. and Toth, P. *Knapsack Problems: Algorithms and Computer Implementations.* en. Wiley-Interscience Series in Discrete Mathematics and Optimization. Chichester ; New York: J. Wiley & Sons, 1990.

[14] Pentico, D. W. “Assignment Problems: A Golden Anniversary Survey”. In: *European Journal of Operational Research* vol. 176, no. 2 (2007), pp. 774–793.

[15] Martello, S. and Toth, P. “Algorithms for Knapsack Problems”. In: *North-Holland Mathematics Studies* vol. 132, no. C (1987), pp. 213–257. arXiv: 1011.1669v3.

[16] Faaland, B. H. “Technical Note–The Multiperiod Knapsack Problem”. In: *Operations Research* vol. 29, no. 3 (1981), pp. 612–616.

[17] Bhadury, J. and Radovilsky, Z. “Job Rotation Using the Multi-Period Assignment Model”. In: *International Journal of Production Research* vol. 44, no. 20 (2006), pp. 4431–4444.

[18] Carnahan, B. J., Redfern, M. S., and Norman, B. “Designing Safe Job Rotation Schedules Using Optimization and Heuristic Search.” In: *Ergonomics* vol. 43, no. 4 (2000), pp. 543–560.

[19] Bard, J. F. and Purnomo, H. W. “Preference Scheduling for Nurses Using Column Generation”. In: *European Journal of Operational Research* vol. 164, no. 2 (2005), pp. 510–534.

[20] Ayough, A., Zandieh, M., and Farsijani, H. “GA and ICA Approaches to Job Rotation Scheduling Problem: Considering Employee’s Boredom”. In: *International Journal of Advanced Manufacturing Technology* vol. 60, no. 5-8 (2012), pp. 651–666.

[21] Bidot, J., Vidal, T., Laborie, P., and Beck, J. C. “A Theoretic and Practical Framework for Scheduling in a Stochastic Environment”. en. In: *Journal of Scheduling* vol. 12, no. 3 (June 2009), pp. 315–344.

[22] Lindahl, M., Stidsen, T., and Sørensen, M. “Quality Recovering of University Timetables”. In: *European Journal of Operational Research* vol. 276, no. 2 (July 2019), pp. 422–435.

[23] Bajestani, M. A. and Beck, J. C. “Scheduling a Dynamic Aircraft Repair Shop with Limited Repair Resources”. en. In: *Journal of Artificial Intelligence Research* vol. 47 (May 2013), pp. 35–70.

[24] Herroelen, W. and Leus, R. “Project Scheduling under Uncertainty: Survey and Research Potentials”. In: *European Journal of Operational Research*. Project Management and Scheduling vol. 165, no. 2 (Sept. 2005), pp. 289–306.

[25] Fu, N., Lau, H. C., Varakantham, P., and Xiao, F. “Robust Local Search for Solving RCPSP/Max with Durational Uncertainty”. en. In: *Journal of Artificial Intelligence Research* vol. 43 (Jan. 2012), pp. 43–86.

[26] Song, W., Kang, D., Zhang, J., Cao, Z., and Xi, H. “A Sampling Approach for Proactive Project Scheduling under Generalized Time-Dependent Workability Uncertainty”. en. In: *Journal of Artificial Intelligence Research* vol. 64 (Feb. 2019), pp. 385–427.

[27] Bertsimas, D., Natarajan, K., and Teo, C.-P. “Persistence in Discrete Optimization under Data Uncertainty”. In: *Mathematical Programming* vol. 108, no. 2-3 (2006), pp. 251–274.

[28] Morrison, T. “A New Paradigm for Robust Combinatorial Optimization: Using Persistence As a Theory of Evidence”. In: *PhD thesis* (2010).

[29] Brown, G. G., Dell, R. F., and Wood, R. K. “Optimization and Persistence”. In: *INFORMS Journal on Applied Analytics* vol. 27, no. 5 (1997).

[30] De Vries, S. and Vohra, R. V. “Combinatorial Auctions: A Survey”. In: *INFORMS Journal on Computing* vol. 15, no. 3 (2003), pp. 284–309.

[31] Endriss, U., Maudet, N., Sadri, F., and Toni, F. “Negotiating Socially Optimal Allocations of Resources”. In: *Journal of Artificial Intelligence Research* vol. 25 (2006), pp. 315–348. arXiv: 1109.6340v1.

[32] Sandholm, T. and Suri, S. “BOB: Improved Winner Determination in Combinatorial Auctions and Generalizations”. In: *Artificial Intelligence* vol. 145, no. 1-2 (2003), pp. 33–58.

[33] Hosseini, H., Larson, K., and Cohen, R. “Matching with Dynamic Ordinal Preferences”. In: *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*. 2015, pp. 936–943.

[34] Anshelevich, E., Chhabra, M., Das, S., and Gerrior, M. “On the Social Welfare of Mechanisms for Repeated Batch Matching”. In: *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence*. 2013, pp. 60–66.

[35] Liu, Y. and Mohamed, Y. “Multi-Agent Resource Allocation (MARA) for Modeling Construction Processes”. In: *Winter Simulation Conference (WSC)*. 2008, pp. 2361–2369.

[36] Nongaillard, A., Mathieu, P., and Jaumard, B. “A Multi-Agent Resource Negotiation for the Utilitarian Welfare”. In: *Engineering Societies in the Agents World IX (ESAW)*. Vol. 5485. LNCS. 2008, pp. 208–226.

[37] Karp, R. M. “Reducibility among Combinatorial Problems”. In: *Complexity of Computer Computations*. Springer, 1972, pp. 85–103.

[38] Nethercote, N., Stuckey, P. J., Becket, R., Brand, S., Duck, G. J., and Tack, G. “MiniZinc: Towards a Standard CP Modelling Language”. en. In: *Principles and Practice of Constraint Programming – CP 2007*. Ed. by Bessière, C. Vol. 4741. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 529–543.

[39] Fukunaga, A. S. and Korf, R. E. “Bin Completion Algorithms for Multicontainer Packing, Knapsack, and Covering Problems”. en. In: *Journal of Artificial Intelligence Research* vol. 28 (Mar. 2007), pp. 393–429.

[40] Pisinger, D. “An Exact Algorithm for Large Multiple Knapsack Problems”. In: *European Journal of Operational Research* vol. 114, no. 3 (1999), pp. 528–541.

[41] Pisinger, D. “Algorithms for Knapsack Problems”. PhD thesis. University of Copenhagen, 1995. arXiv: 1011.1669v3.

[42] Fukunaga, A. S. “A Branch-and-Bound Algorithm for Hard Multiple Knapsack Problems”. In: *Annals of Operations Research* vol. 184, no. 1 (2011), pp. 97–119.

[43] Caprara, A., Kellerer, H., and Pferschy, U. “The Multiple Subset Sum Problem”. en. In: *SIAM Journal on Optimization* vol. 11, no. 2 (Jan. 2000), pp. 308–319.

[44] Yoo, S. and Harman, M. "Regression Testing Minimization, Selection and Prioritization: A Survey". In: *Software Testing, Verification and Reliability* vol. 22, no. 2 (2012), pp. 67–120.

[45] Mossige, M., Gotlieb, A., Spieker, H., Meling, H., and Carlsson, M. "Time-Aware Test Case Execution Scheduling for Cyber-Physical Systems". In: *Proceedings of the 23rd International Conference on Principles and Practice of Constraint Programming*. Vol. 10416. LNCS. 2017, pp. 387–404.

[46] Rothermel, G., Untch, R. H., Chu, C., and Harrold, M. J. "Prioritizing Test Cases For Regression Testing". In: *IEEE Transactions on Software Engineering* vol. 27, no. 10 (2001), pp. 929–948.

[47] Spieker, H., Gotlieb, A., Marijan, D., and Mossige, M. "Reinforcement Learning for Automatic Test Case Prioritization and Selection in Continuous Integration". In: *Proceedings of 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017)*. 2017, pp. 12–22.

Authors' addresses

Helge Spieker Simula Research Laboratory, Martin Linges vei 25, 1364 Fornebu, Norway, helge@simula.no

Arnaud Gotlieb Simula Research Laboratory, Martin Linges vei 25, 1364 Fornebu, Norway, arnaud@simula.no

Morten Mossige University of Stavanger, Postboks 8600, 4036 Stavanger, Norway, ABB Robotics, Nordlysvegen 7, 4340 Bryne, Norway morten.mossige@uis.no

Paper D

Adaptive Metamorphic Testing with Contextual Bandits

Helge Spieker, Arnaud Gotlieb

Revision submitted to *Journal of Systems and Software*, March 2019. Preprint: arXiv:1910.00262v2.

Abstract

Metamorphic Testing is a software testing paradigm which aims at using necessary properties of a system-under-test, called metamorphic relations, to either check its expected outputs, or to generate new test cases. Metamorphic Testing has been successful to test programs for which a full oracle is not available or to test programs for which there are uncertainties on expected outputs such as learning systems. In this article, we propose *Adaptive Metamorphic Testing* as a generalization of a simple yet powerful reinforcement learning technique, namely contextual bandits, to select one of the multiple metamorphic relations available for a program. By using contextual bandits, Adaptive Metamorphic Testing learns which metamorphic relations are likely to transform a source test case, such that it has higher chance to discover faults. We present experimental results over two major case studies in machine learning, namely image classification and object detection, and identify weaknesses and robustness boundaries. Adaptive Metamorphic Testing efficiently identifies weaknesses of the tested systems in context of the source test case.

IV

1 Introduction

Metamorphic Testing (MT) is a software testing paradigm that aims at using necessary properties of a software under test to either check its expected outputs or to generate new test cases [1, 2]. More precisely, MT tackles the so-called *oracle problem* which occurs whenever predicting the expected outputs of a system is just too difficult or even impossible. Typical examples include machine learning models used for classification tasks, for which only stochastic behaviors can be specified [3]. Indeed, these models are often initially trained with existing datasets and then exploited to classify new data samples. However, the expected class of any new data sample is unknown and thus, these samples cannot be used for testing the trained models. Fortunately, transformations over the data samples which do not change their (unknown) class, are usually available. By applying these transformations, called Metamorphic Relations (MRs) in MT, it becomes possible to effectively test trained machine learning models [4, 5, 6].

MT has been very successful to address testing issues in various application domains, e.g., driverless cars [7], search engines [8], or bioinformatics [9] just to name a few (see Section 2.3 for more references). However, generally speaking, applying MT in practice requires to address two issues: the *MR identification* and the *MR selection* problems [2]. The former occurs when trying to identify MRs for a specific system, i.e., to formalize input transformation properties which lead to a known transformation of the outputs. Finding such relations may be difficult when there is no obvious symmetries in the input data, or obvious system invariant, or else when the functional behavior of the system is unknown. The second occurs when several MRs have been identified, but determining which ones are best suited to discover faults in the system under test is hard. It is important to select appropriate MRs for testing the system to avoid redundancies in test cases and thus to avoid slack in the test execution process. This problem is especially critical when testing is part of a continuous integration process where there is usually a limit on the time allocated for testing in an integration cycle.

This paper addresses exclusively the latter problem, i.e., MRs selection, by formulating the effective selection of MRs as a reinforcement learning problem, based on *contextual bandits*. Our method, called *Adaptive Metamorphic Testing* (AMT), defines a *test transformation bandit* which sequentially selects a MR that is expected to provide the highest payoff, i.e., that is most likely to reveal faults. Which MRs are likely to reveal faults is learned from successive exploration trials. The bandit explores the different available MRs and evaluates the fault landscape of the system under test, thereby providing valuable information to the tester.

Learning the selection of MRs can be useful when testing under resource-constraints, for example in cases where the system under test changes are frequently integrated and tested, but also for infrequent testing when the number of MRs is large or their checking is costly. We also discuss a second application which is to identify robust boundaries of the MR parameters. Robust boundaries describe related scenarios but focus on a single MR that can be controlled via

parameters. The interest is to find parameters that produce fault-revealing test cases with minimal changes only.

In this paper, we evaluate Adaptive Metamorphic Testing on two case study applications for image analysis, namely, *image classification* and *object detection*. As implementations of these case studies, we test freely available and pre-trained deep learning systems that can be used as black-box components in other software systems. For each system, we explore both the general fault-revealing capabilities of metamorphic relations and the discovery of robustness boundaries.

The main contributions of this paper are three-fold. First, we introduce Adaptive Metamorphic Testing as a general adaptive selection method for metamorphic relations and test transformations. The method is based on reinforcement learning with contextual bandits and learns to identify those relations which are likely to reveal faults in the system under test. This method is useful in a context where a source test case can be modified by several different metamorphic relations and the system is to be repeatedly tested. To the best of our knowledge, it is the first time that reinforcement learning is applied to select MRs and embedded into a general methodology for MT.

Second, we provide an implementation of Adaptive Metamorphic Testing in a tool called Tetraband, dedicated to testing machine learning models for image analysis. Our tool facilitates metamorphic relations based on image augmentation functions and provides dedicated environments for adaptive metamorphic testing that can be integrated with other implementations.

Third, we explore the benefits of Tetraband on two case studies coming from image analysis, namely image classification and object recognition. Both of these case studies are relevant subsystems in a wide number of applications, such as autonomous cars, robot navigation, or industrial automation [10, 11]. For these applications, high-quality standards are essential and rigorous testing is a requirement. Our experiments show that Tetraband is highly beneficial to optimize the testing process towards fault-revealing MRs.

The remainder of the paper is structured as follows. We review the background on metamorphic testing and contextual bandits and related work in the area in Section 2. Section 3 introduces Adaptive Metamorphic Testing and its components. We discuss general application scenarios in Section 4 before introducing the experimental setup, consisting of our implementation of AMT, Tetraband, and the two case studies in Section 5. The results are presented in Section 6 and Section 7 finally concludes the paper.

2 Background

2.1 Metamorphic Testing

Metamorphic Testing (MT) aims at using necessary properties of a software under test to either check its expected outputs or to generate new test cases [1, 2]. Central to MT is the concept of *Metamorphic Relations* (MRs) which are high-level observable properties that must hold over inputs and outputs of the system under test.

In the following, we formalize the definition of a metamorphic relation, the transformation from source test cases to follow-up test cases, and metamorphic testing. Our definitions follow the formalization by Chen et al.[2], except for the transformation from a source to a follow-up test case. We interpret the transformation function to apply to the whole test case, which includes the test input, and formalize it accordingly in this more general, but compatible, way:

Definition 2.1 (Metamorphic Relation (MR))**.** Let f be a target function or algorithm. A metamorphic relation (MR) is a necessary property over a sequence of multiple inputs $\langle x_1, x_2, \ldots, x_n \rangle$ $(n \geq 2)$ and their corresponding outputs $\langle f(x_1), f(x_2), \ldots, f(x_n) \rangle$. It can be expressed as a relation $\mathcal{R} \subseteq X^n \times Y^n$, with $\subseteq$ being the subset relation, and X^n and Y^n being the Cartesian products of n input and output spaces.

Definition 2.2 (Transformation from Source Test Cases to Follow-up Test Cases)**.** Consider a MR $\mathcal{R}(x_1, x_2, \ldots, x_n, f(x_1), f(x_2), f(x_3))$. The sequence of inputs and their corresponding outputs defines the set of source test cases. For each source test case $\mathcal{S}(x) \in \mathcal{R}$, a follow-up test case $\mathcal{F}$ is derived by applying a possibly non-deterministic transformation function T to the input of the source test case x: $\mathcal{F}(x) = f(T(x))$. The transformation function T is constructed such that the follow-up test case $\mathcal{F}$ fulfills the necessary property of $\mathcal{R}$.

Definition 2.3 (Metamorphic Testing (MT))**.** Let P be an implementation of a target algorithm f. For an MR $\mathcal{R}$, suppose that we have $\mathcal{R}(x_1, x_2, \ldots, x_n, f(x_1), f(x_2), f(x_3))$. Metamorphic testing (MT) based on this MR for P involves the following steps:

1. Define $\mathcal{R}'$ by replacing f by P in $\mathcal{R}$.

2. Given a sequence of source test cases $\langle x_1, x_2, \ldots, x_k \rangle$, execute them to obtain their respective outputs $\langle P(x_1), P(x_2), \ldots, P(x_k) \rangle$. Construct and execute a sequence of follow-up test cases $\langle x_{k+1}, x_{k+2}, \ldots, x_n \rangle$ according to $\mathcal{R}'$ and obtain their respective outputs $\langle P(x_{k+1}), P(x_{k+2}), \ldots, P(x_n) \rangle$.

3. Examine the results with reference to $\mathcal{R}'$. If $\mathcal{R}'$ is not satisfied, then this MR has revealed that P is faulty.

For the remainder of this paper, we refer in general to a metamorphic relation R as the tuple of $\langle \mathcal{R}, T \rangle$, the combination of necessary properties over the outputs for a specific MR and the transformation function T to generate follow-up test cases from source test cases. Therefore, a source test case $\mathcal{S}$ produces output $P(x)$ for the system P with test input x. Using a metamorphic relation R, the follow-up test case $\mathcal{F}$ with test input $T(x)$, where T is a transformation over the input x, can be generated. Due to the metamorphic relation over R (including T), $\mathcal{S}$, and $\mathcal{F}$, the result $P(T(x))$ can be verified. Note that MRs are only partial properties, which means that only test cases that violate them indicate the presence of faults in the system under test. Showing that the system satisfies a MR on any input or a test suite does not guarantee the absence of faults but

increases our confidence in the system correctness. However, this issue concerns any software testing method, not only MT.

The transformation function T of a metamorphic relation R does not have to be a deterministic function, but is usually parameterized and can result in several different follow-up test cases from one source test case, depending on a parameter ϕ which specifies the exact transformation to be applied. In many cases and the simplest implementation of metamorphic testing, ϕ is chosen from a random distribution and T produces a random follow-up test case. We use R_ϕ to denote a metamorphic relation configured by ϕ, which makes it deterministic, and R for the general metamorphic relation, which might be non-stochastic if no additional configuration is possible.

2.2 Contextual Bandits

The selection of a test transformation to apply on a source test case is formalized as a multi-armed bandit problem with context information, also known as a contextual bandit [12, 13]. A contextual bandit acts in discrete iterations, where each iteration corresponds to the generation and execution of one follow-up test case. A bandit $\mathcal{B}$ has k arms, where every arm corresponds to the selection of one possible MR to generate the follow-up test case. In every iteration i, the bandit receives the context vector c_i which in our case describes the source test case.

The bandit then acts according to one or multiple policies π which formalize the action selection, i.e. the decision making strategy. These policies are trained from external feedback about the success of previously made decisions. A policy is often realized by function approximation techniques, for example, multiple linear regression or neural networks. Additionally, an exploration strategy is used to try previously unexplored actions instead of following the policy only. The bandit chooses an arm ($a_i = \pi(c_i)$) and receives a reward, also called payoff, r_i which is the external feedback for this decision. As only one arm can be selected, there is also only feedback for the effect of this single arm a_i. Afterwards, the bandit updates its policy from the observation (c_i, a_i, r_i). Updating the bandit works by adjusting a set of weights, such that the new set of weights better fits the previously made experiences and minimizes regret for historical decisions. The actual implementation of the weight update is dependent on the specific contextual bandits algorithm [14] and the learner used to approximate the policy, e.g. whether it is a form of linear regression or a non-linear neural network.

Our goal for the contextual bandit is to maximize the total payoff, i.e., the cumulative undiscounted reward over all iterations $\sum_{i=1}^{T} r_i$. To achieve this goal and to identify highly rewarded actions, contextual bandit algorithms are designed to also minimize the regret. The regret of a bandit is the gap between the expected reward over a number of iterations when following one policy and the cumulative reward the agent actually receives over the same number of iterations [15]. As a smaller regret means to choose actions more closely to the highest possible payoff, minimizing the regret implies the maximization of

the total payoff, but by maintaining the concept of regret also badly rewarded actions contribute to the improvement of the policy.

A challenge in the design of contextual bandits is to find a balance between *exploration*, i.e., evaluating the effect of rarely used actions, and *exploitation*, i.e., repeating those actions that showed to be effective before. This process is called the *exploration-exploitation trade-off*. To this extent, several exploration techniques have been developed as part of bandit algorithms. We introduce here two techniques that are useful in Adaptive Metamorphic Testing. *Epsilon-greedy* [16] decides to explore a random action with probability ϵ and with probability $1 - \epsilon$ the current learned policy is used to select an action. The parameter $\epsilon \in [0, 1)$ is chosen by the user. A more advanced exploration strategy is *online cover* [14]. Instead of training a single policy, m different polices are trained to produce diverse behaviors. The exploration algorithm then chooses from those actions which have not been learned to perform bad, i.e., having high regret, in the current context. Again, m is a parameter to be chosen by the user.

Contextual bandits are related to Reinforcement Learning (RL) [16]. The main distinction between bandits and RL agents is that bandits perceive each iteration as independent of the previous one [15], i.e., the selected action does not affect the next context that is observed. In our scenario, the next test case is independent of the test transformation chosen for the previous test case. RL agents, however, are designed to operate over multiple subsequent iterations, where a chosen action influences the context in the next iteration. General RL agents could be applied by reducing the length of each scenario to one step, but our early experiments found contextual bandits to be more efficient.

Bandit algorithms have been successfully applied in a variety of domains, such as news article recommendation [17], advertisement selection [18, 19], statistical software testing [20], constraint optimization [21, 22], or real-time strategy games [23]. In this work, we apply contextual bandits to the selection and configuration of metamorphic relations in software testing.

2.3 Related Work

Metamorphic Testing (MT) has been applied to a variety of domains and applications, see [2, 24] for an in-depth overview. Successful application domains include testing of driverless cars [7], search engines [8], machine translation systems [25], performance testing [26, 27], constraint solvers [28] or bioinformatics [9]. Previous works already focused on its automation, for example by exploring algorithms to specifically identify fault-revealing inputs [29] or performing an empirical study on selecting good Metamorphic Relations (MRs) [30]. Other works predict the applicability of a MR for a system [31, 32]. Based on source code traces, a classification model predicts which MRs of a given set can be applied.

Due to the emergent success and usage of machine learning in different application areas, the verification and validation of these systems has received increasing attention. Several works approach testing machine learning systems

based on software testing techniques, such as differential, multi-implementation [33] or mutation testing [34].

Because testing machine learning systems, due to their stochastic nature, is affected by the oracle problem [3], there has been work to especially apply MT for this purpose. Murphy et al.identify a set of general MRs, that hold for a variety of machine learning algorithms [4], and are shown to be effective [35]. Chan et al.further use MT to identify violations of MRs from passed test cases of classification models [36]. It has also been shown that MT can be used for deep learning-based applications, e.g., to test the classification of biological cells [5]. Dwarakanath et al.identify implementation faults in image classifiers [6]. They introduce MRs that affect the training and test data used during model training and demonstrate how these MRs can be applied to find implementation errors in training procedures and model architecture. Yang et al.propose to test unsupervised clustering methods [37] and Mekala et al.explore the application of MRs to detect adversarial examples for deep learning models [38]. However, Saha and Kanewala [39] recently evaluated the effectiveness of MRs for testing supervised classifiers based on mutations of the system under test. They found that the detection rates for the used MRs of previous studies are limited when generating a large set of mutants.

Previous work also considered the adaptive control of software testing through feedback while testing [40, 41]. Similar to our method, these works exploit the behavior of the system during the test execution and adjust the testing strategy when the understanding of the software changes. In these works, the adjustment of the testing strategy focuses on test case prioritization and selection, whereas our method focuses on the generation of follow-up test cases using metamorphic testing and under consideration of many repeated testing cycles.

3 Adaptive Metamorphic Testing

3.1 Overview

In this section, we introduce Adaptive Metamorphic Testing (AMT) with contextual bandits. Our method is based on a test transformation bandit that learns to select follow-up test cases from a set of applicable metamorphic test cases. Algorithm 2 shows an overview of the main steps of AMT. At the core of AMT, a contextual bandit receives a description of the source test case. Based on this context vector, the bandit selects an action, which resembles a MR, and the configuration of this transformation. Both are applied to generate a follow-up test case. After generating the follow-up test case, the system under test is executed and the test result evaluated according to the MR acceptance criterion. The method can be directly deployed without any pre-training step. However, during the first iterations, MR selection is partly random to gather initial experiences about the different MRs effectiveness and their potential payoffs, when applied to the available source test cases. After several iterations have been performed, the bandit learns to focus on MRs which are most likely to reveal faults. Nevertheless, the bandit continues to explore among the MRs,

Algorithm 2 Adaptive Metamorphic Testing with Contextual Bandits

Input: M: set of MRs; SUT: system under test P; TS: set of test cases; Iter: Number of iterations
Output: $\mathcal{B}$: trained bandit

1: $i \leftarrow 0$
2: $\mathcal{B} \leftarrow$ Load *existing* or initialize *new* Bandit
3: **while** $i <$ Iter **do**
4: **Select** $\mathcal{S} \in TS$ ▷ Draw source test case from test suite
5: $c \leftarrow \mathcal{B}$.ExtractContextFeatures($\mathcal{S}$) ▷ Generate feature vector for source test case $\mathcal{S}$
6: $R_\phi \leftarrow \mathcal{B}$.SelectBanditArm($c, M$) ▷ Select one MR R_ϕ using the bandit
7: $v \leftarrow$ Apply($SUT, R_\phi(\mathcal{F})$) ▷ Execute SUT with transformed test $\mathcal{F}$, get a verdict v
8: $\mathcal{B} \leftarrow \mathcal{B}$.UpdateBandit($R_\phi, c, v$) ▷ Train the bandit with the feedback
9: $i \leftarrow i + 1$
10: **end while**
11: **return** $\mathcal{B}$ ▷ Return updated bandit for future test cycle

i.e., it sometimes chooses MRs which do not promise the highest payoff. This is important to adjust to changes in the system under test as well as to gather additional information about the effect of MRs in different contexts.

Definition 3.1 (Adaptive Metamorphic Testing)**.** Let P be an implementation of a target algorithm f with TS being its test suite; let M be a set of metamorphic relations applicable on TS and let $\mathcal{B}$ be a contextual bandit. Adaptive Metamorphic Testing (AMT) is an iterative variant of metamorphic testing and involves the following steps at each iteration:

1. A test case $\mathcal{S}$ is (randomly) selected from the test suite TS and executed to obtain its output $P(\mathcal{S})$.

2. The bandit $\mathcal{B}$ selects a MR R based on the context features of $\mathcal{S}$.

3. Construct one or more follow-up test cases $\langle \mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_k \rangle$ according to R and obtain their respective outputs $\langle P(\mathcal{F}_1), P(\mathcal{F}_2), \ldots, P(\mathcal{F}_k) \rangle$.

4. Examine the results with reference to R. If R is not satisfied, then this MR has revealed that P is faulty.

5. Report the results of the execution back to $\mathcal{B}$ for adaptation of the learning algorithm.

The arm selection of the bandit, i.e., the selection of a metamorphic relation R with its parameters is handled by a hierarchy of contextual bandits using the context features c at each iteration of the algorithm.

On the highest level, the main contextual bandit $\mathcal{B}$ selects one MR R from the set of supported MRs. Afterwards another action-specific contextual bandit

$\mathcal{B}_R$ is queried, using the same context information as the main bandit, for the configuration parameter ϕ: $\mathcal{B}_R(c_i) \rightarrow \phi$. If the metamorphic relation does not require additional configuration, the second step is skipped. The MR R_ϕ can then be used to generate a follow-up test case for the current iteration. After test execution, both bandits $\mathcal{B}$ and $\mathcal{B}_R$ are trained from the received feedback. Tetraband consists of one main contextual bandit plus one additional contextual bandit for each configurable MR.

Adaptive Metamorphic Testing is independent of the application domain or the implementation or else the specific MRs that can be applied. It only takes as inputs a set of MRs, the system under test, a set of test cases, and a user-defined parameter corresponding to the maximum number of iterations to run. As output, the method returns a trained bandit $\mathcal{B}$ which has learned to select the MRs which have the greatest chances to detect faults in the system under test.

3.2 Components

Adaptive Metamorphic Testing requires only a few system-specific components. In the following, we discuss each of these components.

3.2.1 Extract Context Features

In order to select an appropriate MR which is likely to reveal a fault, for a source test case, it is mandatory to feed the contextual bandit with relevant context information about the source test case. This context is captured with a feature vector, which is a real-valued vector of fixed size n. The function receives the source test case as input and returns the feature vector: $ExtractContextFeatures : \mathcal{S} \rightarrow \mathbb{R}^n$.

By using representative features, the contextual bandit can learn a mapping from the source test cases, which are described through the features, to the MRs. For that, it has to include test characteristics that can be affected by the metamorphic relations. Therefore, the features need to capture the necessary details distinctive about the individual test case, especially those that relate to the effect of the MR.

How the feature vector is formed, is a domain-specific problem and requires some degree of domain knowledge. For example, when testing scientific software for matrix calculations [30], the features should describe the characteristics of the original input matrix in order to select which transformation is applied. In our experiments, which are based on testing computer vision problems, we rely on common feature modeling used in machine learning. For instance, using a pre-trained neural network to extract image features is common in computer vision. A similar approach could be used for text processing, where textual features, e.g., used vocabulary, text sentiment, or sentence structure, can be derived using pre-trained networks.

3.2.2 Metamorphic Relations

The most relevant component to acquire for applying Adaptive Metamorphic Testing is the set of MRs, which is highly domain-dependent. The automatic and systematic identification of MRs for a system is an ongoing research topic [2], but in many cases, MRs can be extracted by using domain knowledge about the system or reviewing the existing literature from the Software Testing community. In the special case of testing machine learning systems, a starting point to uncover MRs is to exploit data augmentation methods that are commonly used. As shown in our experiments, these augmentations can serve as a basis for MRs and help to identify weaknesses in ML systems.

3.2.3 Select Bandit Arm

The selection of an appropriate MR is mostly handled by the internal contextual bandit algorithm and does not have to be individually implemented for a new system under test. The main bandit selects the MR and, if necessary, the action-specific bandit for this MR selects the parameter to configure the MR. Nevertheless, the configuration of the contextual bandit influences the performance of the system and should be adjusted, depending on the number of available MRs and the robustness of the SUT. This allows us to focus on exploiting MR that reveal faults in the system or to broadly explore the effects of many different MRs.

The most important configuration parameter to adjust is the exploration rate, i.e., how often does the bandit choose a different action than the most promising one. Using a high exploration rate allows us to examine different combinations of test cases and systems, which is relevant to detect new faults in the system and extend the coverage of different tests. Conversely, a lower exploration rate exploits combinations of tests and MRs that have often fail previously. Traditionally, Metamorphic Testing often creates follow-up test cases at random, which corresponds to a maximal exploration rate here.

Exploitation is relevant when repeatedly testing the system, e.g., in continuous integration settings, or when trying to understand the weaknesses of the system for a certain group of MRs. Still, it is not only desired to exploit known weaknesses, but broad coverage of the system behavior is desired for higher test confidence. If the behavior of the system changes, because it is becoming more robust to previously effective MRs, the bandit learns this and can adjust its selection for future iterations. Conclusively, compared to other applications of contextual bandits, where especially the exploitation of known good actions is in focus, exploration is more prominent in Adaptive Metamorphic Testing to broadly test the system behavior.

3.2.4 Transform, Execute and Evaluate

The selected MR and its configuration transform the source test case into a follow-up test case. That test case is then executed and the test verdict is

evaluated according to the MR. These core steps of Adaptive Metamorphic Testing are similar to those required in traditional MT.

3.2.5 Update Bandit

After the follow-up test case has been executed and the results have been evaluated, the bandit's policy is updated. This requires information about the initial context feature vector, the chosen MR and its configuration, and the test verdict. The *update routine* updates the expected reward for this MR. The exact update routine is specific to the contextual bandit algorithm and its configuration and we refer to the corresponding literature for its description [12, 15].

Nevertheless, choosing the appropriate reward for a failed test case has to be done while adjusting the bandit for a new system to test. In most scenarios, where the goal is to find the most fault-revealing MRs, as described below in Section 4.1, the reward is the same for every failing test case. However, if the bandit has the goal to identify certain properties of the SUT, it can be necessary to propose a different reward structure that depends on the selected MR. This second scenario is further described in Section 5.3.

4 Application Scenarios of Adaptive Metamorphic Testing

Contextual bandits are powerful to explore the effects of the MRs in different contexts, but can also exploit the gathered experiences to subsequently focus on those relations that are most likely to reveal faults. From these properties, we identify two application scenarios of Adaptive Metamorphic Testing that we discuss further and evaluate as part of the case studies.

4.1 Fault-Revealing MR Selection

The first application of Adaptive Metamorphic Testing is the selection of metamorphic relations which are prone to reveal faults in the system under test. In cases where the MR has parameters, an additional contextual bandit is responsible to select these parameters, as described before. This application, which we refer to as *fault-revealing MR selection*, steers MT towards greater effectiveness when there are many MRs available and not sufficient resources to apply them all. In this application, all MRs are considered distinct from each other. Accordingly, the achievable reward received for revealing a fault is identical for all MRs.

4.2 Robustness Boundaries

The second application uses the exploration/exploitation trade-off of contextual bandits to identify *robustness boundaries* of the software under test. With robust boundaries, we focus on MRs whose effect can be adjusted by user-defined parameters, especially those with continuous or a range of discrete values that control the distance between source and follow-up test cases. As an example

taken from the case studies, while testing an image analysis system, one possible transformation is to rotate the image, where the degree of rotation is a user-defined parameter. If the system is susceptible to treat wrongly rotated images, it is likely that large rotations, e.g. by 90 degrees, are more likely to cause mistakes than smaller rotations. By identifying the robust boundaries, information can be inferred about the trade-off between acceptable transformations and exceedingly strong manipulations. This result yields both a robustness characteristic and a starting point for a more curated set of requirements on the system under test.

5 Experimental Evaluation

5.1 Case Studies

We consider two case studies to evaluate our tool Tetraband. Both case studies come from the field of digital image processing, where deep learning methods commonly represent the state-of-the-art approaches [11]: *image classification* and *object detection*. For each of the case studies, we consider both previously introduced application scenarios (see Section 4) and identify fault-revealing MRs as well as robustness boundaries against configurable image transformations. We have formulated the following three research questions as a guideline for our experiments:

RQ1 Does Adaptive Metamorphic Testing, implemented as Tetraband, learn to select MRs whose follow-up test cases reveal faults?

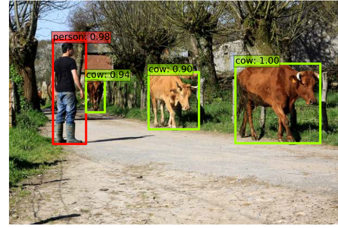
RQ2 Is AMT effective to approximate the distribution of faults in the system under test?

RQ3 Is AMT computation- and data-efficient compared to random sampling of MRs and exhaustive search of all follow-up test cases?

Previous work for testing image processing applications has considered random and metamorphic testing [42, 43], but focused on the evaluation of handcrafted image processing applications, whereas we focus in our experiments especially on machine learning-based computer vision systems. The existing studies focus on testing the basic functionality of the system by generating random images and transforming them using a set of transformations, different from our approach where we base on an existing dataset of images from the domain that the ML model has been trained on. We furthermore especially consider the selection of good MRs. Xu et al.recently presented another use case for metamorphic relations in image classification applications beyond testing [44]. Their work uses metamorphic relations, based on separation and occlusion, to augment the training data and fine-tune the model.

In the following, we present the two case studies, image classification and object detection, with their setup and the considered MRs. We further discuss the configuration of Tetraband, and finally, present the experimental results and our findings.

(a) Image Classification Example (from ImageNet [45]). The goal is to assign a single class label to the image, e.g. *toucan* here.



(b) Object Detection Example (from [46]). The goal is to identify and categorize objects by drawing a bounding box and assigning a class label.

Figure D.1: Image classification and object detection examples.

5.2 Image Processing Applications

We describe two case studies where testing image processing systems is necessary (see Figure D.1). In the first case study, an image classification system is tested. The second case study focuses on an object detection application.

5.2.1 Image Classification

An image classification task, also image recognition, has the goal to identify the object shown in an image, e.g., assign the image to one of a fixed number of classes. Since 2012, the state-of-the-art method for image classification, among other image analysis tasks, are deep neural networks, such as residual neural networks (ResNets) [47] or SqueezeNet [48].

In our case study, we test a SqueezeNet model that has been initially trained on the ImageNet dataset [45] and then fine-tuned for the 10 classes of the CIFAR-10 dataset [49]. For testing the model, we use the CIFAR-10 test set, consisting of $10,000$ labeled images.

Following the metamorphic relation between source and follow-up test cases, we consider a test as failed, if the transformed image leads to a different classification result than the original image. The correctness of the original class prediction does not influence the test result, because the bandit does not know the initial model performance. Instead, the bandit aims to select transformations that affect the outcome in a fault-revealing manner compared to the original output. Testing the difference in outputs between source and follow-up test cases removes the dependency from having to use labeled data, i.e. data where the ground-truth class is known and allows the integration of other data sources. Nevertheless, for testing the system, we monitor also the accuracy of the system for correctly classifying the images, but we do not use this information as feedback for the test transformation bandit, although that would also be a viable setup.

5.2.2 Object Detection

Object detection is a generalization of the image classification task in the sense that there can be multiple objects on a single image. Besides assigning classes to these objects, it is also necessary to provide bounding boxes around the location of each object. This means that the output of an object detection model consists of a class label and four coordinates for the bounding box for each detected object. Object detection systems employ deep neural networks of a similar, but extended, architecture compared to image classification systems.

The system to test in this case study is a pre-trained object detection model, based on the open-source TensorFlow Object Detection API[1] [50]. In particular, we test an implementation of a single shot multibox detector (SSD) [46] with a feature pyramid network (FPN) [51], based on a ResNet-50 network [47]. We refer the reader to the given references for an in-depth overview of the models. We see the system to test here as a black box. However, briefly said, the model detects objects in images with a single neural network by assigning one of multiple predefined box sizes, their size adjustment and classification scores at the same time. By reducing the complexity to a single neural network, it is a fast model for real-time object detection that achieves state-of-the-art performance. The used model was trained on Microsoft COCO dataset [52] and is available within the Object Detection API[2].

For testing, we use $5,000$ images from the validation set of the *MS COCO challenge 2017* as source test cases. We apply the same input transformations on the images as in the image classification case study and as described in Section 5.3. Because each image annotation consists of an additional bounding box per object in the image, the metamorphic relations are extended to transformation also on the bounding boxes. For example, rotating the image rotates the bounding box of the object to match the rotated object, and flipping the image from left to right also flips the positions of the bounding boxes in the image (see the next Section 5.3 for an introduction of the applied transformations).

Furthermore, we consider the different evaluation metrics for object detection tasks. In image classification, the result is easily verified by comparing the estimated class with the ground truth class. In object detection, it is necessary to evaluate the overlapping regions between the estimated bounding boxes and the ground truth, which is called the intersection-over-union (IoU), in addition to the class label of each box:

$$IoU(A, B) = \frac{A \cap B}{A \cup B}$$

where A are the proposed pixels from the object detection model and B is the ground-truth from the dataset. If the IoU value exceeds a certain threshold, the object is counted as correctly detected. We follow the evaluation guidelines from

[1]TensorFlow Object Detection API: github.com/tensorflow/models/tree/master/research/object_detection

[2]The exact name in the object detection model zoo is *ssd_resnet50_v1_fpn_shared_box_predictor_640x640_coco14_sync_2018_07_03*

the MS COCO challenge and compare the results for the mean average precision (mAP) which is calculated over all objects in an image and the average of different IoU thresholds $0.5, 0.55, 0.6, \ldots, 0.95$, for both the original and the transformed image. If the mAP of the transformed image is below the original mAP minus a performance reduction of 0.05 which is close to 10% of the average model performance, we interpret the test case to be violating the MR and therefore as failed.

5.3 Metamorphic Relations and Rewards

In the case studies of this paper, we use tests where the input is an image and the output is a classification of this image to recognize certain objects. The considered MRs are all related to image transformations, e.g. mirroring or rotating, under the property that the results of image classification and object detection do not change, i.e., the MR defines equality of the outputs, while the input is transformed.

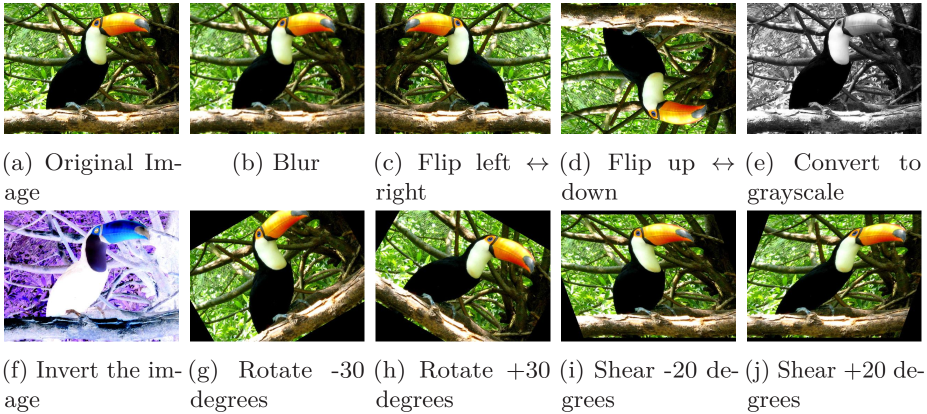
In most cases, these transformations must only not modify the class of these images. However, in object detection, some transformations of images that impact object location markers entail similar transformation over the outputs. Here, the MR defines a relation between the outputs that is similar to the transformation of the input.

We select seven common image transformations among all possible transformations as MRs, some of which have been used in previous work on metamorphic testing for image analysis methods [38, 42]. Among these MRs, two are configurable by an additional parameter ϕ. These transformations are the following:

1. Blur the image by the average value of neighbor pixels (Blur)
2. Flip the image from the left to the right (Flip L/R)
3. Flip the image upside down (Flip U/D)
4. Convert a colored image to grayscale (Grayscale)
5. Invert the colors of the image (Invert)
6. Rotate the image by x degrees (Rotation)
7. Shear the image by x degrees (Shear)

The effects of these transformations are shown in Figure D.2. The MRs Rotation and Shear expect a parameter to define the transformation effect. For Rotation, we consider 36 distinct values in steps of 5 degrees in the degree range $[-90; 90]$, excluding rotations of 0 degrees. For shear, we include 18 values in the range $[-45; 45]$ in steps of 5 degrees, again excluding 0 degrees.

The reward for the main MRs is set up such that a failed test case is rewarded with 1 and a passed test case with 0, independent of which MR was selected. For the action-specific bandits, which select the MR parameters for Rotation and Shear, the reward structure is designed to encourage the selection of the



(a) Original Image (b) Blur (c) Flip left ↔ right (d) Flip up ↔ down (e) Convert to grayscale

(f) Invert the image (g) Rotate -30 degrees (h) Rotate +30 degrees (i) Shear -20 degrees (j) Shear +20 degrees

Figure D.2: Metamorphic relations, i.e. image transformations, and their effects. Image taken from the ImageNet dataset [45].

smallest failing parameters. Therefore, the smallest parameter of -5 respectively 5 degrees receives a reward of 10000, if revealing a fault. For each additional step, the reward is divided by two. Thereby, choosing a smaller parameter value has always higher payoff than a larger rotation, if successful.

5.4 Implementation

We implemented Adaptive Metamorphic Testing in a tool called Tetraband. Our implementation package is available at http://github.com/helges/tetraband. The software is implemented in Python 2.7 and it is structured into two main components, as shown in Figure D.3.

One component provides the SUTs used in our case studies, i.e. the image classification and object detection systems. These SUTs are encapsulated via the OpenAI Gym[3] interface [53]. Having separate, standardized environments allows easier reproduction of the experiments and their usage in other work. Their functionality includes the feature extraction for each SUT, as well as the application of the available MRs. The metamorphic relations for image manipulation are realized with the imgaug library (version 0.2.6) for image augmentation[4]. Further details for the setup of the SUTs are given in the description of the case studies in Section 5.2.

The second component is Tetraband itself, our implementation of AMT. It is mostly an adaptation of a contextual bandit as the main actor, using the machine learning library Vowpal Wabbit 8.6.1[5]. Additionally, for comparison, we include a random agent that uniformly picks an arbitrary MR and configuration.

[3]OpenAI Gym: https://gym.openai.com/
[4]imgaug: https://imgaug.readthedocs.io/
[5]Vowpal Wabbit: github.com/VowpalWabbit/vowpal_wabbit

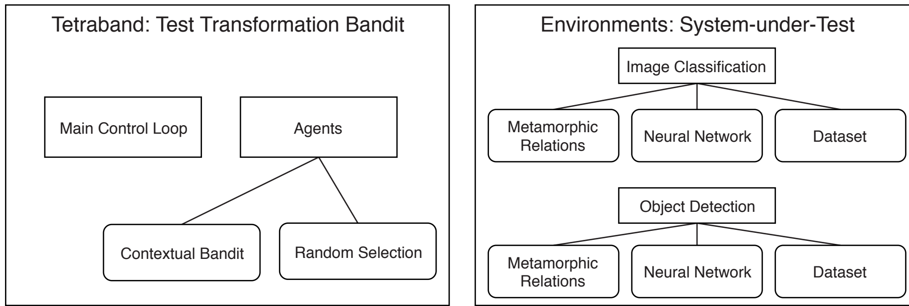


Figure D.3: Overview of Tetraband, our implementation of Adaptive Metamorphic Testing. It is a light-weight implementation that allows extension and adaptation to other environments and settings.

The contextual bandits use the doubly robust policy evaluation algorithm for learning and action selection [54]. Exploration is performed through a combination of epsilon-greedy exploration, where it chooses a random action in 10 % of the iterations, and online cover exploration [14] with three policies. The policy itself is approximated by a feed-forward artificial neural network with a single hidden layer with 16 neurons. We choose a moderately high exploration rate, because we not only want the bandits to converge on few single actions that repeatedly provide high payoff, but we also want to learn about the effectiveness of other actions. An important aspect of contextual bandits is that the exploration is not reduced or disabled after training, but stays active although self-adjusted to a lesser extent than in the initial iterations. This helps to re-use the bandit to test the SUT repeatedly, e.g. in Continuous Integration, because it can adapt to changing behaviors in the SUT.

5.5 Experimental Setup

At each iteration, the source test case is formed by an input image and its annotations from the data set. The context feature vector is extracted through an additional neural network [55] which processes the image and returns a feature vector of 512 floating-point numbers. The network is a pre-trained ResNet-18 network [47] from the PyTorch Torchvision model zoo[6]. The final layer, that usually outputs the identified image class, has been removed and the output of the previous layer is used as the feature vector. We have also experimented with perceptual image hashing for feature extraction but did not find the hash value expressive enough.

[6]PyTorch: https://pytorch.org / https://github.com/pytorch/vision

5.6 Fault-Revealing Metamorphic Relations

The first experiment focuses on identifying general weaknesses in the system, i.e. identifying which metamorphic relations are fault-revealing for the system under test.

The bandit can freely choose from the seven transformations described in Section 5.3 and their reward structure. A follow-up test case fails, i.e. it violates the MR, if the transformed image is classified differently than the source image: $SUT(\mathcal{S}) \neq SUT(\mathcal{F})$. We define the MR to produce equal outputs for the source and follow-up test cases, but we could instead also consider the annotated ground-truth labels for comparison. Using the difference in outputs between source and follow-up test cases reduces the dependency on this labeling, and is thereby applicable to unlabeled datasets.

5.7 Robustness Boundaries

The second experiment takes two specific transformations and learns the robustness boundaries of the SUT for these transformations. As the robustness boundary, we describe the parameterization of the transformation which changes the source test case as little as possible but is most likely to reveal a fault.

For evaluation, we use the two parameterized MRs that are already used in the other experiments, i.e. Rotation and Shear transformation. For both MRs the same parameter space as described in Section 5.3 is kept. The main difference in this experiment is that the focus lies on a single MR and its parameterization. This allows us to specifically examine the weaknesses of the SUT towards one transformation and learn about its robustness.

6 Experimental Results

For each case study, we have performed two experiments: identifying fault-revealing MRs and robustness boundaries. Each run consists of one pass over the full training set, i.e. 10,000 iterations for image classification and 5,000 iterations for object detection. For each of the experiments, we show the mean result of 10 runs with different random seeds. Our findings underline the effectiveness of Tetraband for controlling the metamorphic testing process in software testing, especially for testing machine learning systems.

6.1 Effectiveness of Metamorphic Relations on Image Classification

Before the evaluation of Tetraband, we first analyze the effectiveness of the selected MRs for our experiments on the image classification dataset, i.e. CIFAR-10. We aim to understand whether there are different effects of different MRs and how they affect certain classes of images. To this end, all MRs were applied to all images of the dataset, which we refer to in the other experiments as the *baseline* reference and the changes in the predicted classes are observed.

	Airplane	Automobile	Bird	Cat	Deer	Dog	Frog	Horse	Ship	Truck	Avg.
Blur	10.60	11.40	13.10	9.81	7.30	13.50	17.70	9.00	6.00	6.20	10.46
Flip L/R	2.90	1.00	4.10	6.71	2.20	6.80	1.30	2.40	0.90	2.40	3.07
Flip U/D	14.90	74.60	37.80	33.13	59.10	53.90	29.30	92.40	72.20	43.30	51.06
Grayscale	4.70	5.40	28.10	7.91	18.10	26.00	14.30	6.70	4.80	5.30	12.13
Invert	16.50	29.40	29.50	33.13	41.40	70.30	41.80	38.30	27.30	35.70	36.33
Rotation	25.49	37.09	35.43	17.70	69.00	46.10	20.63	60.44	42.44	50.01	40.43
Shear	11.22	4.99	26.69	35.79	45.45	51.97	15.63	40.24	19.78	55.24	30.70
Avg.	12.33	23.41	24.96	20.60	34.65	38.37	20.10	35.64	24.77	28.31	26.31

Table D.1: CIFAR-10 dataset: Effects of MRs by the true class of the image. Each cell value shows the percentage of images in the class, which are wrongly classified after applying the MR. Every class contains 1000 images. Rotation and Shear are parameterized by 30 degrees. The values are determined using the baseline method, i.e. exhaustive search over all MRs and all images.

The baseline results reported for image classification correspond thereby to the average violation rate over all classes as shown in the rightmost column of the table.

Table D.1 shows the percentage of images in a class that is affected by a MR, such that they are wrongly classified afterward. CIFAR-10 consists of ten classes of images, printed as column names. The effectiveness of the image transformations varies between both MRs and image classes. The MR Flip U/D is an example of a particularly effective transformation that affects over 50% of the images in the test set. However, when noting the different classes in the dataset, all images have a clear vertical orientation and the flipped image of the objects is unlikely to occur in the dataset, for example for automobiles where 74.6% of the images are misclassified. Images of frogs or airplanes, where the perspectives of the images vary more naturally are less affected, but still to a moderately high degree compared to other MRs. This is different for the MR Flip L/R where the horizontal orientation is reversed, which corresponds to a variation that is already included in the training data and therefore is the least effective MR. Other MRs identify stronger differences between classes. Converting the image to grayscale has little effect on airplanes or ships, which commonly have few distinct features related to color, but large effects for birds or dogs, where colors are more distinctive characteristics.

Conclusively, from the experiment using the image classification dataset, we see different effects per MR and image, which underlines the motivation to learn which MRs are effective for a particular image.

6.2 Fault-Revealing Metamorphic Relations

6.2.1 Case Study 1: Image Classification

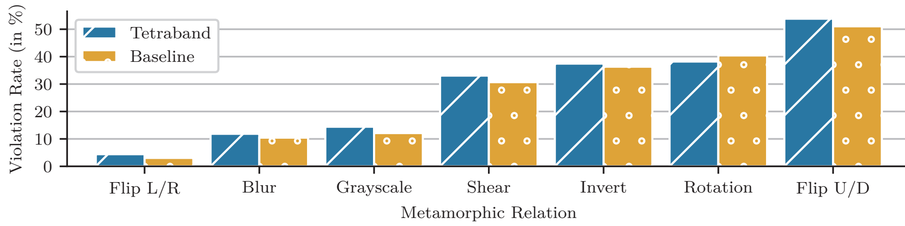
In our first case study, which is an image classification ML model, the goal of the bandit is to select a MR which leads to a different classification of the transformed image compared to the original image of the source test case. While we looked at the true classes of the source images in the previous initial experiment, we

only consider the change of the predicted class from source to a follow-up test case in this experiment. This approach focuses on the consistency of outputs for different variants of the same input image, i.e., the follow-up test cases. The actual true label is not as relevant in this case, as it is unlikely to find one MR which transforms the image consistently in a way that the correct label is predicted. Additionally, the accuracy, i.e., the correctness of the outputs is usually already tested during or after the training of the model. Focusing on the difference between outputs for source and follow-up test cases furthermore allows to extend the set of source test cases from unlabelled datasets, making it easier to enlarge the system's test suite.

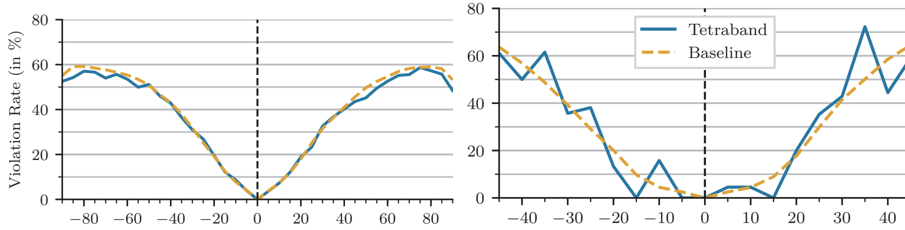
For the seven main transformations, Figure D.4a shows the distribution of the violation rate for each follow-up test case generated by a selected MR. We compare this violation rate of MRs selected by Tetraband to the true violation rate for the set of source test cases, which corresponds to the rightmost column (Avg.) in Table D.1. These ground-truth results form our baseline and are determined by applying all available MRs to all source test cases in an exhaustive search. While this exhaustive search covers all possible follow-up test cases, it is time- and resource-intense compared to adaptive metamorphic testing and not ideally suited for repeated testing. We have therefore considered the random selection of a MR, such as it is common in traditional metamorphic testing, as the comparison method for computational cost and resource consumption. Conclusively, to discuss the quality of the selected MRs with Tetraband, we use exhaustive search as the baseline. The evaluation of computational cost, including a comparison to the common random selection, is discussed separately in Section 6.2.3.

The *violation rate* estimates how often the image classifier changes its prediction after the MR was applied to the source image. This can be different from the *failure rate* in cases where the prediction on the original image was wrong, but the transformation made the model predict the correct output. However, as we do not expect the test data to be labeled, we do not require and consider this information for our experiments and instead focus on the robustness and consistency of the model for the predictions. Therefore, we mostly report the violation rate. Ideally, a perfectly-trained image classification model should not show any change and the violation rates should be zero independently of the selected MR.

Generally speaking, the violation rate distribution shows a different impact for different MRs. The least fault-revealing MR is the one that mirrors images over the middle vertical axis, i.e., flip left and right. This transformation is most likely to be found in the training datasets for the image classifier. Often the image is either symmetric by itself, like a human face or body, or included together with another image of the same object but taken from a different angle, showing a symmetric profile. Other MRs are more effective to reveal faults in the image classifier, which is related to the disturbance impact they have on the original image. They often represent transformations that could be expected in real-world applications and should be covered by a robust image classification system. As seen from the exhaustive search baseline experiment, the most



(a) Violation rate of MRs selected by Tetraband and baseline results for exhaustive search



(b) Configuration of Rotation transformation (in degrees)

(c) Configuration of Shear Transformation (in degrees)

Figure D.4: Fault-Revealing MRs for image classification: Violation rate and configuration of parameterized MR transformations. Tetraband approximates the true error distribution and select fault-revealing MRs and their parameters.

fault-revealing MR identified by Tetraband showed to be flipping the image upside-down, i.e., mirroring on the middle of the horizontal axis. This is within expectations, as it results in an image, which is unlikely to be represented in the distribution of training images, e.g., an image of a car is likely to be shown with its wheels on the ground. Nevertheless, high violation rates are also observed for less invasive MRs, such as rotating the image or inverting it, and these MRs are either likely to be encountered in practical applications or preserve many of the distinctive features. Having this statistic not only allows us to identify the weaknesses of the classifier (model testing), but it can also be used as a basis to configure image augmentation techniques to train a new version of the image classification model (model training). With image augmentation, the training set of images is extended by including modified versions of the original image, through small perturbations or affinity scaling, while preserving the original label. By knowing the MRs that fail the old classifier, the necessary transformations to include in future image augmentation are known and can help to improve the performance of new models. However, not all MRs are necessarily suitable image augmentations at training time, as they might produce images that are not within the distribution of inputs, i.e. images, for which the model is trained.

The performance of Tetraband closely approximates the violation distribution for the baseline results and exceeds them for all MRs, except Rotation where the violation rate is close to the baseline violation rate. The reason for the lower

violation rate is related to the necessary exploration to select an appropriate parameter to rotate the source image. Due to the large number of parameters from -90 to 90 the bandit exploration had to take ineffective actions to learn, which leads to an initially lower violation rate.

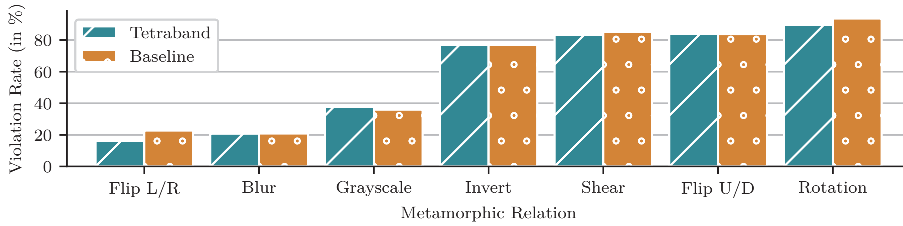
However, we also observe that longer runtime and more iterations over the dataset further increase the averaged violation rate as the bandit algorithm can focus more on exploitation than exploration. This can be explained as initial iterations do not have knowledge about the MRs effectiveness and the selection is more random, which includes selecting ineffective MRs. While these actions are valuable for exploration, they do not contribute to the violation rate. Later, when the effect of MRs has been sufficiently explored, the focus changes on also exploiting the MRs and selecting fault-revealing MRs. These actions then contribute to increasing the violation rate. Accordingly, if the system under test does not rapidly change, more iterations will increase the amount of exploitation with high violation rates and the impact of the initial exploration on the violation rate decreases.

The violation distribution for the different parameters of the MRs Rotation and Shear are shown in Figure D.4b and Figure D.4c. The bandit effectively picks the appropriate degree of rotation to closely resemble the true error distribution. The only exceptions are the largest degrees of rotation, where the selection does not completely approximate the true distribution. However, due to our reward structure, the agent is encouraged to focus on minimal rotations in the images that lead to some misclassification. Especially in the range between -45 and 45 degrees the parameter selection is appropriate, which indicates the successful convergence towards the most revealing parameter for image rotation.

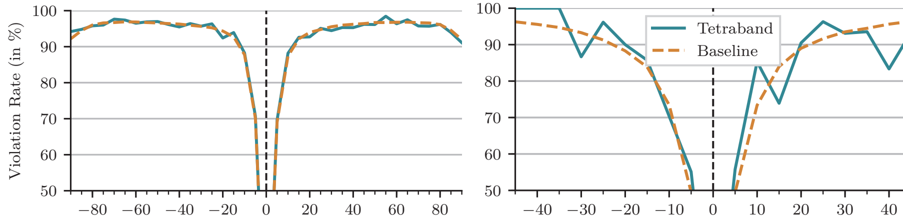
In the Shear transformation, Tetraband was less effective to smoothly approximate the true error distribution but broadly follows its shape. Here, we observe that due to the lower general violation rate of the MR, there are fewer chances for successful exploration of the parameter space than for the Rotation MR. Accordingly, the approximation of the parameter distribution for Shear is not as close as for Rotation, but still follows the general distribution.

6.2.2 Case Study 2: Object Detection

The second case study application is the object detection neural network. Similar to the presentation of the first case study, Figure D.5 shows the results of the object detection case study. As a first main difference, the results show a higher violation rate for object detection, with over 70% violation rate for four of the seven MRs. The ranking of MRs is similar but applying the Shear MR with parameter selection is more efficient here than in the image classification case study, where the Invert MR showed a higher violation rate, and Rotation is the most effective MR. The MRs Flip L/R, Blur and Grayscale have the lowest violation rate, i.e., the model is most robust to these changes. However, while the total violation rate is higher, the case study itself is more difficult as the dataset on which we apply Tetraband is smaller and only consists of 5000 images, which means the overall process takes half of the iterations of the image classification



(a) Violation rate of MRs selected by Tetraband and baseline results for exhaustive search



(b) Configuration of Rotation transformation (in degrees)

(c) Configuration of Shear transformation (in degrees)

Figure D.5: Fault-Revealing MRs for object detection: Violation rate and configuration of parameterized MR transformations. Tetraband approximates the true error distribution and select fault-revealing MRs and their parameters.

case study. This difference explains the approximation difference for some of the main MRs in comparison to the baseline violation rates.

For the two MRs Rotation and Shear, the approximation quality for the additional parameter is similar to the image classification results. For Rotation, the distribution is close to the true distribution of the exhaustive search baseline results, related to the high overall violation rate for this MR and its corresponding higher selection and thereby better exploration opportunities. The Shear parameters match the true distribution less closely in this case study, which is also related to the higher general effectiveness of this MR for the object detection dataset.

In general, the results of the object detection case study confirm the results of the image classification case study while at the same time respecting the higher difficulty of fewer iterations, due to which the violation rate of the main MRs is close to the true violation rate, but does not exceed it after the given number of iterations.

6.2.3 Computational Cost and Random Selection

From the previous experiments, we have evaluated the effectiveness of Tetraband for selecting fault-revealing MRs in image classification and object detection systems. However, we did not consider the computational cost of introducing machine learning in the MT process or compared Tetraband to the commonly used

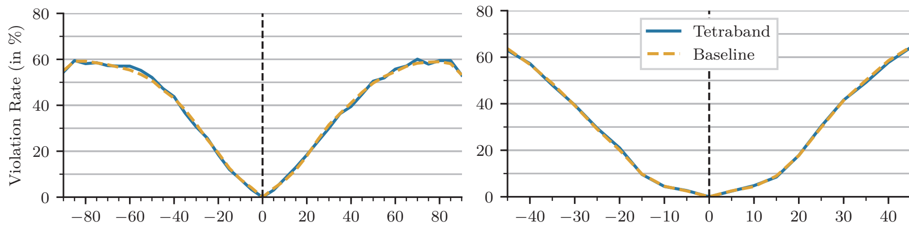
Case Study	Runtime (in s)		Accuracy (in %)		
	Tetraband	Random	Unmodified	Tetraband	Random
Image Classification	0.1	0.06	96.6	54.0	72.9
Object Detection	0.63	0.56	54.3	23.1	36.3

Table D.2: Computational cost and accuracy of Tetraband and random selection for selecting fault-revealing MRs. The average runtime per image shows that the overhead introduced by the ML model is relatively small, especially for object detection with a more costly test execution. Tetraband selects MRs more effectively and the accuracy is smaller than with random selection.

approach of randomly selecting MRs and their parameters. In this experiment, to answer RQ3, we analyze common characteristics for both case studies, which consider the computational cost of introduced contextual bandits and the additional learning step in the MR process. We further briefly discuss another comparison method that more closely resembles the state-of-practice in MT, which is to randomly select MRs and their parameters. A summary of the results is given in Table D.2.

Running Tetraband is computationally cheaper and more sample-efficient than the exhaustive search that we consider as a baseline. While the exhaustive search considers all MRs with all different parameters for Rotation and Shear, in total 59 different transformations, Tetraband selects one MR and one parameter per iteration. In addition to the application of the MR and the execution of the SUT, Tetraband has a small computational overhead for selecting the MR and its parameter and learning from the received feedback. The average duration per iteration in the image classification case study is 0.1s using Tetraband and 0.06s when using a random MR and not learning from feedback. While this overhead increases the execution time, it is still faster and more efficient than running all possible transformations, as we will see below. For object detection, the average duration is 0.63s with learning and 0.56s without, here the main computation lies in the object detection neural network. The overhead for training the contextual bandit could be further reduced by moving the learning step outside the main processing loop; however, we argue that in a practical application the overhead is negligible due to the lower number of executions and the availability of highly optimized contextual bandit implementations.

We also considered a random selection of MRs and their parameters. At each iteration, a random MR is sampled uniformly instead of using the bandit selection. This selection is less efficient than Tetraband. Using Tetraband, the accuracy of the image classifier is reduced from 96.6% for the unmodified images to 54.0% for the images modified by the selected MRs. With random selection, the accuracy for the modified images remains at 72.9%, which is a substantial reduction, but not as high as Tetraband. In object detection, the precision is reduced from 54.3% to 23.1% with Tetraband and to 36.3% with random selection. Due to the lower general effectiveness of random selection, we do not



(a) Configuration of Rotation transformation (in degrees)

(b) Configuration of Shear transformation (in degrees)

Figure D.6: Image classification: Average violation rate per degree step. The approximated violation rate closely approximates the true violation distribution of the ground truth baseline, i.e. exhaustive search.

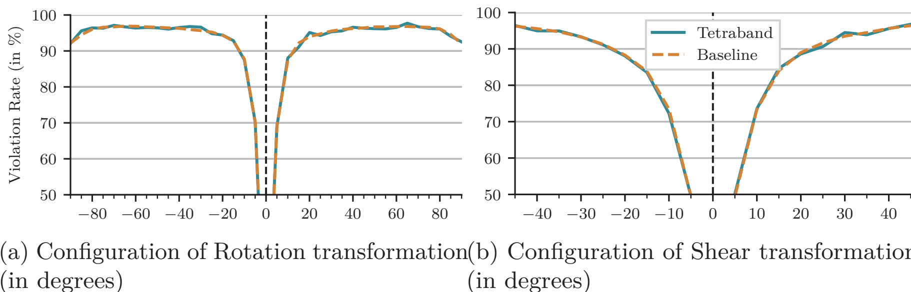
further discuss its results in detail.

6.3 Robustness Boundaries

As a second experiment and application of Tetraband to learning MR selection, we aim to find the robustness boundaries of the case study systems against different degrees of image rotations and shearing. We show the results for both case study applications in Figure D.6 for image classification and Figure D.7 for object detection. The experimental results mostly confirm the inherent hypothesis, also following the previous results from the first experiments that larger modifications of the source test case lead to a higher violation rate of the follow-up test case. This is true for both case study applications, but the extent to which the effect applies varies with the object detection system being much more susceptible to images rotated even only by small degrees. At the same time, we confirm that a larger number of iterations allows better exploration and approximation of the true error distribution. Where the parameter distribution in the previous experiments showed divergence, mostly for larger parameter values, the focus on the specific MRs in this experiment allows sufficient exploration and good approximation.

The results for image classification clearly show the effect that larger rotations of the original image are more likely to cause a different classification (see Figure D.6a). Here, the bandit does not only learn to select the largest rotation but does accurately approximate the distribution of true faults as given by the exhaustive search baseline. Our results show that for more than 10% of the source test cases a rotation of at least 10 degrees leads to a different image classification. When considering real-world scenarios for the application of image classification systems, a rotation of 10 degrees can likely occur due to tilt or shifts in either the camera or due to external influences on the actual object.

For object detection, the interpretation of the results needs to consider two aspects, which lead to the conclusion that the results can not be directly compared to the image classification case study. First, the original SUT already



(a) Configuration of Rotation transformation (in degrees)

(b) Configuration of Shear transformation (in degrees)

Figure D.7: Object detection: Average violation rate per degree step. The approximated violation rate closely approximates the true violation distribution of the ground truth baseline, i.e. exhaustive search.

has lower performance for the original data set than the image classification SUT. When considering the system to be more imprecise for unmodified data, then it is also likely to be more fragile for modified data. Second, the evaluation metric used, mean average precision, is more fragile than the metric used for image classification. The results show the fragility of the object detection system, as well as the capability to learn to approximate this error distribution over transforming each of the 5000 source test case images only once.

6.4 Discussion

For both case studies in our experiments, our results showed the effectiveness of contextual bandits, as part of Tetraband, to adapt to a prior unknown error distribution in two different case study applications, based on two different neural network architectures and tasks.

From the results, we draw two major conclusions. First, we see a confirmation for the applicability of Tetraband for selecting metamorphic relations using contextual bandits (i.e., Adaptive Metamorphic Testing), as is shown by the close approximation of the true error distribution with limited iterations. Second, our tests reveal robustness weaknesses in the two systems-under-test. Weaknesses in neural networks have been addressed before and are an active research area [56, 57, 58]. The research under the area of *adversarial examples* focuses on finding input perturbations that lead to misbehavior of the model with only minimal or hard-to-detect changes in the input. This approach is different from the setting of our experiment. We select distinct and known image transformations to modify the image without the goal to hide the transformation, which often is the intent of an adversarial example.

7 Conclusion

This paper introduces Adaptive Metamorphic Testing (AMT), a method to control metamorphic testing using contextual bandits. AMT receives a feature

vector representing the source test case and selects a MR to generate a follow-up test case. From the result of evaluating the follow-up test case, whether it reveals a fault, the bandit learns which MRs can be used to exploit weaknesses in the system. At the same time, using state-of-the-art algorithms for contextual bandits, AMT explores non-optimal actions to identify previously unknown weaknesses and to adapt to changing behavior in repeated testing of the same system.

We have evaluated the applicability of AMT using our implementation Tetraband on two image analysis case studies in two distinct tasks. For both case studies, our results showed that Tetraband approximates the true distribution of faults in the SUT with fewer iterations and executions of the SUT than an exhaustive search and more efficiently than random sampling of MRs and their parameters, which is the common best practice. Tetraband learns to select fault-revealing MRs in relation to the source test case while ignoring non-relevant MRs. Furthermore, in the second experiment, Tetraband proved to be effective for the identification of robustness boundaries, that is, exploring the parameterization of individual MRs, which have different impacts on the SUT. Our experiment explored how different degrees of rotation and shear affected the classification result of the transformed image.

In conclusion, AMT is effective for selecting MRs and is more time-efficient than exhaustive testing and more effective than the standard approach of pure random sampling. We see the method to be useful in repeated testing scenarios, such as continuous integration, where regression of the SUT can be tested from an initial knowledge about previous fault characteristics. For future work, we plan to investigate the combination of multiple MRs to create follow-up test cases instead of selecting only one MR per iteration.

Acknowledgements

This work is supported by the Research Council of Norway (RCN) through the research-based innovation center Certus, under the SFI program. The experiments were performed on the Abel Cluster, owned by the University of Oslo and Uninett/Sigma2, and operated by the Department for Research Computing at USIT, the University of Oslo IT-department.

Declarations of Interest

None of the authors declares a conflict of interest.

References

[1] Chen, T., Cheung, S., and Yiu, S. *Metamorphic Testing: A New Approach for Generating Next Test Cases.* Technical Report HKUST-CS98-01. Hong Kong: Department of Computer Science, Hong Kong University of Science and Technology, 1998.

[2] Chen, T. Y., Kuo, F.-C., Liu, H., Poon, P.-L., Towey, D., Tse, T. H., and Zhou, Z. Q. "Metamorphic Testing: A Review of Challenges and Opportunities". In: *ACM Computing Surveys* vol. 51, no. 1 (2018).

[3] Barr, E. T., Harman, M., McMinn, P., Shahbaz, M., and Yoo, S. "The Oracle Problem in Software Testing: A Survey". In: *IEEE Transactions on Software Engineering* vol. 41, no. 5 (2015), pp. 507–525.

[4] Murphy, C., Kaiser, G., Hu, L., and Wu, L. "Properties of Machine Learning Applications for Use in Metamorphic Testing". In: *Proceedings of the 20th International Conference on Software Engineering and Knowledge Engineering (SEKE)* (2008), pp. 867–872.

[5] Ding, J., Hu, X.-H., and Gudivada, V. "A Machine Learning Based Framework for Verification and Validation of Massive Scale Image Data". In: *IEEE Transactions on Big Data* vol. 26, no. 3 (2017), pp. 1–1.

[6] Dwarakanath, A., Ahuja, M., Sikand, S., Rao, R. M., Bose, R. P. J. C., Dubash, N., and Podder, S. "Identifying Implementation Bugs in Machine Learning Based Image Classifiers Using Metamorphic Testing". In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 2018, pp. 118–128.

[7] Zhou, Z. Q. and Sun, L. "Metamorphic Testing of Driverless Cars". In: *Communications of the ACM* vol. 62, no. 3 (2019), pp. 61–67.

[8] Zhou, Z. Q., Xiang, S., and Chen, T. Y. "Metamorphic Testing for Software Quality Assessment: A Study of Search Engines". In: *IEEE Transactions on Software Engineering* vol. 42, no. 3 (Mar. 2016), pp. 264–284.

[9] Shahri, M. P., Srinivasan, M., Reynolds, G., Bimczok, D., Kahanda, I., and Kanewala, U. "Metamorphic Testing for Quality Assurance of Protein Function Prediction Tools". In: *2019 IEEE International Conference On Artificial Intelligence Testing (AITest)*. Apr. 2019, pp. 140–148.

[10] LeCun, Y., Bengio, Y., and Hinton, G. "Deep Learning". In: *Nature* vol. 521, no. 7553 (2015), pp. 436–444.

[11] Pouyanfar, S., Sadiq, S., Yan, Y., Tian, H., Tao, Y., Reyes, M. P., Shyu, M.-L., Chen, S.-C., and Iyengar, S. S. "A Survey on Deep Learning: Algorithms, Techniques, and Applications". en. In: *ACM Computing Surveys* vol. 51, no. 5 (2018), pp. 1–36.

[12] Langford, J. and Zhang, T. "The Epoch-Greedy Algorithm for Multi-Armed Bandits with Side Information". In: *Advances in Neural Information Processing Systems 20 (NIPS 2007)*. 2007, pp. 817–824.

[13] Zhou, L. "A Survey on Contextual Multi-Armed Bandits". In: *arXiv preprint arXiv:1508.03326* (2016).

[14] Agarwal, A., Hsu, D., Kale, S., Langford, J., Li, L., and Oct, L. G. "Taming the Monster: A Fast and Simple Algorithm for Contextual Bandits". In: *International Conference on Machine Learning*. 2014, pp. 1638–1646.

[15] Lattimore, T. and Szepesvari, C. *Bandit Algorithms*. en. Vol. Revision: 8b22b8b6131c37e388d5e3b2eecf0b4ff5d7db92. https://banditalgs.com/, 2019.

[16] Sutton, R. S. and Barto, A. G. *Reinforcement Learning: An Introduction*. 2nd. MIT Press, 2018.

[17] Li, L., Chu, W., Langford, J., and Schapire, R. E. "A Contextual-Bandit Approach to Personalized News Article Recommendation". In: *International Conference on World Wide Web (WWW)*. 2010, pp. 661–670.

[18] Lu, T., Pal, D., and Pal, M. "Contextual Multi-Armed Bandits". In: *Proceedings of the 13th International Conferenceon Artificial Intelligence and Statistics (AISTATS)*. 2010, pp. 485–492.

[19] Tang, L., Rosales, R., Singh, A., and Agarwal, D. "Automatic Ad Format Selection via Contextual Bandits". In: *Proceedings of the 22nd ACM International Conference on Conference on Information & Knowledge Management*. 2013, pp. 1587–1594.

[20] Baskiotis, N., Sebag, M., Gaudel, M.-C., and Gouraud, S.-D. "EXIST: Exploitation/Exploration Inference for Statistical Software Testing". In: *On-Line Trading of Exploration and Exploitation, NIPS 2006 Workshop*. 2006.

[21] Loth, M., Sebag, M., Hamadi, Y., and Schoenauer, M. "Bandit-Based Search for Constraint Programming". In: *International Conference on Principles and Practice of Constraint Programming*. 2013, pp. 464–480.

[22] Balafrej, A., Bessiere, C., and Paparrizou, A. "Multi-Armed Bandits for Adaptive Constraint Propagation". In: *International Joint Conference on Artificial Intelligence* (2015), pp. 290–296.

[23] Ontañón, S. "Combinatorial Multi-Armed Bandits for Real-Time Strategy Games". In: *Journal of Artificial Intelligence Research* vol. 58, no. 1 (Mar. 2017), pp. 665–702.

[24] Segura, S., Fraser, G., Sanchez, A. B., and Ruiz-Cortes, A. "A Survey on Metamorphic Testing". In: *IEEE Transactions on Software Engineering* vol. 42, no. 9 (2016), pp. 805–824.

[25] Sun, L. and Zhou, Z. Q. "Metamorphic Testing for Machine Translations: MT4MT". In: *2018 25th Australasian Software Engineering Conference (ASWEC)*. Nov. 2018, pp. 96–100.

[26] Segura, S., Troya, J., Durán, A., and Ruiz-Cortés, A. "Performance Metamorphic Testing: A Proof of Concept". In: *Information and Software Technology* vol. 98 (June 2018), pp. 1–4.

[27] Johnston, O., Jarman, D., Berry, J., Zhou, Z. Q., and Chen, T. Y. "Metamorphic Relations for Detection of Performance Anomalies". In: *2019 IEEE/ACM 4th International Workshop on Metamorphic Testing (MET)*. 2019, pp. 63–69.

[28] Akgün, Ö., Gent, I. P., Jefferson, C., Miguel, I., and Nightingale, P. "Metamorphic Testing of Constraint Solvers". In: *Principles and Practice of Constraint Programming*. Ed. by Hooker, J. Vol. 11008. LNCS. 2018, pp. 727–736.

[29] Gotlieb, A. and Botella, B. "Automated Metamorphic Testing". In: *Proceedings 27th Annual International Computer Software and Applications Conference* (2003), pp. 34–40.

[30] Mayer, J. and Guderlei, R. "An Empirical Study on the Selection of Good Metamorphic Relations". In: *30th Annual International Computer Software and Applications Conference*. 2006, pp. 475–484.

[31] Kanewala, U. and Bieman, J. M. "Using Machine Learning Techniques to Detect Metamorphic Relations for Programs without Test Oracles". In: *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)* (2013).

[32] Kanewala, U., Bieman, J. M., and Ben-Hur, A. "Predicting Metamorphic Relations for Testing Scientific Software: A Machine Learning Approach Using Graph Kernels". In: *Software Testing, Verification and Reliability* vol. 26, no. 3 (May 2016), pp. 245–269.

[33] Pei, K., Cao, Y., Yang, J., and Jana, S. "DeepXplore: Automated Whitebox Testing of Deep Learning Systems". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM Press, 2017, pp. 1–18.

[34] Ma, L., Liu, Y., Zhao, J., Wang, Y., Juefei-Xu, F., Zhang, F., Sun, J., Xue, M., Li, B., Chen, C., Su, T., and Li, L. "DeepGauge: Multi-Granularity Testing Criteria for Deep Learning Systems". In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering - ASE 2018*. 2018, pp. 120–131.

[35] Xie, X., Ho, J. W., Murphy, C., Kaiser, G., Xu, B., and Chen, T. Y. "Testing and Validating Machine Learning Classifiers by Metamorphic Testing". In: *Journal of Systems and Software* vol. 84, no. 4 (2011), pp. 544–558.

[36] Chan, W. K., Ho, J. C. F., and Tse, T. H. "Finding Failures from Passed Test Cases: Improving the Pattern Classification Approach to the Testing of Mesh Simplification Programs". en. In: *Software Testing, Verification and Reliability* vol. 20, no. 2 (2010), pp. 89–120.

[37] Yang, S., Towey, D., and Zhou, Z. Q. "Metamorphic Exploration of an Unsupervised Clustering Program". In: *Proceedings of the 4th International Workshop on Metamorphic Testing*. 2019, pp. 48–54.

[38] Mekala, R. R., Magnusson, G. E., Porter, A., Lindvall, M., and Diep, M. "Metamorphic Detection of Adversarial Examples in Deep Learning Models with Affine Transformations". In: *Proceedings of the 4th International Workshop on Metamorphic Testing*. 2019, pp. 55–62.

[39] Saha, P. and Kanewala, U. "Fault Detection Effectiveness of Metamorphic Relations Developed for Testing Supervised Classifiers". In: *2019 IEEE International Conference On Artificial Intelligence Testing (AITest)*. Apr. 2019, pp. 157–164.

[40] Cai, K.-Y., Gu, B., Hu, H., and Li, Y.-C. "Adaptive Software Testing with Fixed-Memory Feedback". en. In: *Journal of Systems and Software*. The Impact of Barry Boehm's Work on Software Engineering Education and Training vol. 80, no. 8 (Aug. 2007), pp. 1328–1348.

[41] Zhou, Z. Q., Sinaga, A., Susilo, W., Zhao, L., and Cai, K.-Y. "A Cost-Effective Software Testing Strategy Employing Online Feedback Information". en. In: *Information Sciences* vol. 422 (Jan. 2018), pp. 318–335.

[42] Mayer, J. and Guderlei, R. "On Random Testing of Image Processing Applications". In: *2006 Sixth International Conference on Quality Software (QSIC'06)*. Oct. 2006, pp. 85–92.

[43] Guderlei, R. and Mayer, J. "Towards Automatic Testing of Imaging Software by Means of Random and Metamorphic Testing". In: *International Journal of Software Engineering and Knowledge Engineering* vol. 17, no. 06 (Dec. 2007), pp. 757–781.

[44] Xu, L., Towey, D., French, A. P., Benford, S., Zhou, Z. Q., and Chen, T. Y. "Enhancing Supervised Classifications with Metamorphic Relations". In: *Proceedings of the 3rd International Workshop on Metamorphic Testing - MET '18*. 2018, pp. 46–53.

[45] Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. "ImageNet Large Scale Visual Recognition Challenge". In: *International Journal of Computer Vision* vol. 115, no. 3 (2015), pp. 211–252.

[46] Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C.-y., and Berg, A. C. "SSD: Single Shot MultiBox Detector". In: *European Conference on Computer Vision*. Vol. 9905. LNCS. 2016, pp. 21–37.

[47] He, K., Zhang, X., Ren, S., and Sun, J. "Identity Mappings in Deep Residual Networks". In: *European Conference on Computer Vision*. Vol. 9908. LNCS. 2016, pp. 630–645.

[48] Iandola, F. N., Han, S., Moskewicz, M. W., Ashraf, K., Dally, W. J., and Keutzer, K. "SqueezeNet: AlexNet-Level Accuracy with 50x Fewer Parameters and <0.5MB Model Size". In: *arXiv preprint arXiv:1602.07360* (2016).

[49] Krizhevsky, A., Nair, V., and Hinton, G. "The CIFAR-10 Dataset". In: *online: http://www. cs. toronto. edu/kriz/cifar. html* (2014).

[50] Huang, J., Rathod, V., Sun, C., Zhu, M., Korattikara, A., Fathi, A., Fischer, I., Wojna, Z., Song, Y., Guadarrama, S., and Murphy, K. "Speed/Accuracy Trade-Offs for Modern Convolutional Object Detectors". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017, pp. 3296–3297.

[51] Lin, T.-Y., Dollar, P., Girshick, R., He, K., Hariharan, B., and Belongie, S. "Feature Pyramid Networks for Object Detection". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2017, pp. 936–944.

[52] Lin, T.-Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., Dollár, P., and Zitnick, C. L. "Microsoft COCO: Common Objects in Context". In: *European Conference on Computer Vision*. Vol. 8693. LNCS. 2014, pp. 740–755.

[53] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. "OpenAI Gym". In: *arXiv:1606.01540 [cs]* (June 2016). arXiv: 1606.01540 [cs].

[54] Dud, M. and Langford, J. "Doubly Robust Policy Evaluation and Learning". In: *Proceedings of the 28th International Conference on Machine Learning*. 2011, pp. 1097–1104.

[55] Sharif Razavian, A., Azizpour, H., Sullivan, J., and Carlsson, S. "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*. 2014, pp. 806–813.

[56] Goodfellow, I., Lee, H., Le, Q. V., Saxe, A., and Ng, A. Y. "Measuring Invariances in Deep Networks". In: *Advances in Neural Information Processing Systems*. Vol. 22. 2009, pp. 646–654.

[57] Carlini, N. and Wagner, D. "Towards Evaluating the Robustness of Neural Networks". In: *IEEE Symposium on Security and Privacy*. 2017, pp. 39–57.

[58] Biggio, B. and Roli, F. "Wild Patterns: Ten Years after the Rise of Adversarial Machine Learning". In: *Pattern Recognition* vol. 84 (Dec. 2018), pp. 317–331.

Authors' addresses

Helge Spieker Simula Research Laboratory, Martin Linges vei 25, 1364 Fornebu, Norway, helge@simula.no

Arnaud Gotlieb Simula Research Laboratory, Martin Linges vei 25, 1364 Fornebu, Norway, arnaud@simula.no