Evolutionary Computation for Software Testing

University of Augsburg

Faculty of Applied Computer Science

The dissertation is submitted for the degree of

Dr. rer. nat.

Lukas Rosenbauer

Gutachter:

Erstgutachter:	Prof. Dr. Jörg Hähner
Zweitgutachter:	Jun.-Prof. Dr. Anthony Stein
Drittgutachter:	Prof. Dr. Bernhard Bauer

Prüfungskommission:

Prof. Dr. Jörg Hähner	Jun.-Prof. Dr. Anthony Stein
Prof. Dr. Bernhard Bauer	Prof. Dr. Gernot Müller
Tag der mündlichen Prüfung:	4.4.2022

Abstract

A variety of products undergo a transformation from a pure mechanical design to more and more software and electronic components. A polarized example are watches. Several decades ago they have been purely mechanical. Modern smart watches are almost completely electronic devices which heavily rely on software. Further, a smart watch offers a lot more features than just the information about the current time (e. g. Bluetooth connectivity or health features).

This change had a crucial impact on how software is being developed. A first attempt to control the rising complexity was to move to agile development practices such as extreme programming or scrum. This rise in complexity is not only affecting the development process but also quality assurance and software testing. If a product contains more and more features then this leads to a higher number of tests necessary to ensure quality standards. Furthermore agile development practices work in an iterative manner which leads to repetitive testing that puts more effort on the testing team. We aimed within the thesis to ease the pain of testing. Thereby we examined a series of subproblems that arise.

A key complexity is the number of test cases. We intended to reduce the number of test cases before they are executed manually or implemented as automated tests. Thereby we examined the test specification and based on the requirements coverage of the individual tests, we were able to identify redundant tests. We relied on a novel metaheuristic called GCAIS which we improved upon iteratively.

Another task is to control the remaining complexity. Testing is often time crucial and an appropriate subset of the available tests must be chosen in order to get a quick insight into the status of the device under test. We examined this challenge in two different testing scenarios.

The first scenario is located in semi-automated testing where engineers execute a set of automated tests locally and closely observe the behaviour of the system under test. We extended GCAIS to compute test suites that satisfy different criteria if provided with sufficient search time. We delivered the metaheuristic as part of a test selection GUI where test engineers can perform a preselection based on their expert knowledge. Mathematically this reduces the search space which is beneficial for the runtime.

The second use case is located in fully automated testing in a *continuous integration* (CI) setting. CI focuses on frequent software build cycles which also include testing. These builds contain a testing stage which greatly emphasizes speed. Thus there we also have to compute crucial tests. However, due to the nature of the process we have to continuously recompute a test suite for each build as the software and maybe even the test cases at hand have changed. Hence it is hard to compute the test suite ahead of time and these tests have to be determined as part of the CI execution. Thus we switched to a computational lightweight *learning classifier system* (LCS) to prioritize and select test cases. We integrated a series of innovations we made into an LCS known as *XCSF classifier system* such as continuous priorities, experience replay and transfer learning. This enabled us to outperform a state of the art artificial neural network which is used by companies such as Netflix. We further investigated how LCS can be made faster using parallelism. We developed generic approaches which may run on any multicore computing device. This is of interest for our CI use case as the build server's architecture is unknown. However, the methods are also independent of the concrete LCS and are not linked to our testing problem.

We identified that many of the challenges that need to be faced in the CI use case have been tackled by *Organic Computing* (OC), for example the need to adapt to an ever changing environment. Hence we relied on OC design principles to create a system architecture which wraps the LCS developed and integrates it into existing CI processes. The final system is robust and highly autonomous. A side-effect of the high degree of autonomy is a high level of automatization which fits CI well. We also gave insight on the usability and delivery of the full system to our industrial partner. Test engineers can easily integrate it with a few lines of code and need no knowledge about LCS and OC in order to use it. Another implication of the developed system is that OC's ideas and design principles can also be employed outside the field of embedded systems. This shows that OC has a greater level of generality.

The process of testing and correcting found errors is still only partially automated. We make a first step into automating the entire process and thereby take an analogy to the concept of self-healing of OC. As a first proof of concept of this school of thought we take a look at touch interfaces. There we can automatically manipulate the software to fulfill the specified behaviour. Thus only a minimalistic amount of manual work is required.

Simplicity is a great virtue but it requires hard work to achieve it and education to appreciate it.
And to make matters worse: Complexity sells better.
- Edsger Wybe Dijkstra [1]

Contents

List of Figures

List of Tables

List of Algorithms

Nomenclature

α	Static degree of autonomy
β	Learning rate for cl.F and cl.ϵ
χ	Crossover probability for XCSF's GA
cl	Classifier of a learning classifier system
cl.ϵ	Mean absolute error estimate of its prediction function
cl.F	Fitness of the classifier
cl.num	Numerosity of a classifier
cl.$p(s)$	Prediction function of the classifier cl
ϵ_0	Error threshold of XCSF
ϵ_I	Initialization value for cl.ϵ
η	Learning rate for the Delta rule
Γ	GSEMO's number of populations
$\mathbf{C}(t)$	Capacitive signal at time t
$\mathcal{T}_i$	Set of all available tests for CI cycle i
μ	GA's mutation probability
ν	Power parameter of XCSF
$\pi(s)$	Policy function which proposes an action for a given state s
$\tau(t)$	Binary touch signal at time t
$\hat{\mathrm{V}}(s)$	Estimated value function
$\mathrm{rk}_i(T)$	Rank assigned to the test case T for CI cycle i
θ_{del}	XCSF's deletion threshold
θ_{GA}	XCSF's GA threshold
θ_{sub}	XCSF's subsumption threshold
$\tilde{s}$	State vector extended by an additional entry holding 1 (at the beginning)
$\mathcal{TS}$	Test suite (set of tests)
$\mathcal{TS}_i$	Test suite chosen at CI cycle i
$\mathcal{TS}_i^{\mathrm{f}}$	Subset of tests of $\mathcal{TS}_i$ which failed
A	Action space of an agent
$a(t)$	Action performed at time t
B	Experience replay buffer
C	Available time budget for testing
$d(T)$	Approximated duration of test case T

F_I	Initialization value for cl.F
l_i	List of prioritized test cases for CI cycle i
m	Number of requirements
N	Number of touch signal samples
n	Number of test cases
p_i	Percentage of found failures at CI cycle i
$P_\#$	Probability of using a # for a condition during covering
$r(t)$	Reward received at time t
$r_i^{\text{fc}}(T)$	Failure count reward function for CI cycle i and test case T
$r_i^{\text{tcf}}(T)$	Test case failure reward for CI cycle i and test case T
$r_i^{\text{trk}}(T)$	Time ranked reward for CI cycle i and test case T
r_{active}	Active robustness
r_{passive}	Passive robustness
S	State space of an agent
$s(t)$	State of the agent at time t
t_{rec}	Recovery time for the system when an utility breakdown is detected
$V(s)$	Value function
$v_i(T)$	Verdict function for a test case T for CI cycle i
V_{ext}	External Variability
V_{int}	Internal Variability
w_i	i-th weight for the local prediction function of cl
AC	Autonomic computing
AI	Artificial intelligence
AIS	Artificial immune system
APX	Approximable computation class
ATCS	Adaptive test case selection problem
BNL	Block nested loop algorithm
BSH	BSH Hausgeräte GmbH
CI	Continuous integration
CPM	Control and power module
CRO	Chemical reaction optimization
CUDA	Compute Unified Device Architecture
CV	Computer Vision

DB Data base

DUT Device under test

EC Evolutionary Computing

EMC Electromagnetic compatibility

ER Experience replay

FPTAS Fully polynomial-time approximation scheme

GA Genetic algorithm

GCAIS Germinal center artificial immune system

GPU Graphical processing unit

GUI Graphical user interface

KISS Keep it simple stupid

KPI Key performance indicator

LCS Learning classifier system

LP Linear program

M Match set of XCSF

ML Machine learning

MLOC Multi-layer observer controller

MSCP Minimum set cover problem

NAPFD Normalized average percentage of faults detected

NLP Natural language processing

NP Complexity class for problems which can be solved non-deterministical in polynomial time

OC Organic Computing

P Complexity class for problems which can be solved deterministical in polynomial time

PTAS Polynomial-time approximation scheme

QA Quality assurance

RL Reinforcement learning

SA Simulated annealing

SEIP Simple evolutionary algorithms with isolated population

SIMD Single instruction multiple data

SuOC System under observation and control

T Test case

TL Transfer learning

TN True negative rate

TP True positive rate

TSMP Test Suite Minimisation Problem

WMSCP Weighted minimum set cover problem

XCS XCS classifier system

XCSF XCSF classifier system

XCSF-ER XCSF with experience replay

XCSF-TL XCSF with transfer learning and experience replay

List of own Publications

- L. Rosenbauer, A. Stein, H. Stegherr, and J. Hähner, "Metaheuristics for the Minimum Set Cover Problem: A Comparison," in *Proceedings of International Joint Conference on Computational Intelligence*, 2020. **Nomination for best poster award.**
- L. Rosenbauer, A. Stein, and J. Hähner, "A Germinal Center Artificial Immune System for Software Test Suite Reduction," in *LIFELIKE Computing Systems Workshop, 8th Edition in the Evolution of the Series of Autonomously Learning and Optimizing Systems (SAOS), at Artificial Life Conference, 16 July 2020, held entirely virtual due to COVID-19 as part of ALIFE 2020, Montréal, Canada*, 2020
- L. Rosenbauer, A. Stein, and J. Hähner, "An Artificial Immune System for Adaptive Test Selection," in *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*, 2020, pp. 2940–2947
- L. Rosenbauer, A. Stein, and J. Hähner, "An Artificial Immune System for Black Box Test Case Selection," in *EvoCop: 21st European Conference on Evolutionary Computation in Combinatorial Optimisation as part of evostar 2021, April 2021, Seville, Spain*, 2021
- L. Rosenbauer, A. Stein, and J. Hähner, "Generic Approaches for Parallel Rule Matching in Learning Classifier Systems," in *Proceedings of the 2020 Genetic and Evolutionary Computation Conference, Cancún, Mexico, July, 2020*, C. A. C. Coello, Ed., 2020
- A. Stein, R. Maier, L. Rosenbauer, and J. Hähner, "XCS Classifier System with Experience Replay," in *Proceedings of the 2020 Genetic and Evolutionary Computation Conference, Cancún, Mexico, July, 2020*, C. A. C. Coello, Ed., 2020
- L. Rosenbauer, A. Stein, R. Maier, D. Pätzel, and J. Hähner, "XCS as a Reinforcement Learning Approach to Automatic Test Case Prioritization," in *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, ser. GECCO '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1798–1806
- L. Rosenbauer, A. Stein, D. Pätzel, and J. Hähner, "XCSF for Automatic Test Case Prioritization," in *Proceedings of the 12th International Joint Conference on Computational Intelligence (ECTA), November 2-4, 2020*, J. J. Merelo, J. Garibaldi, C. Wagner, T. Bäck, K. Madani, and K. Warwick, Eds., 2020. **Nomination for best paper award.**
- L. Rosenbauer, A. Stein, D. Pätzel, and J. Hähner, "XCSF with Experience Replay for Automatic Test Case Prioritization," in *2020 IEEE Symposium Series on Computational Intelligence (SSCI), virtual event, Canberra, Australia, 1-4 December 2020*, H. Abbass, C. A. C. Coello, and H. K. Singh, Eds., 2020
- L. Rosenbauer, D. Pätzel, A. Stein, and J. Hähner, "Transfer Learning for Automated Test Case Prioritization using XCSF," in *EvoApplications: 24th International Conference on the Applications of Evolutionary Computation as part of evostar 2021, April 2021, Seville, Spain*, 2021
- L. Rosenbauer, D. Pätzel, A. Stein, and J. Hähner, "An Organic Computing System for Automated Testing," in *Architecture of Computing Systems – ARCS 2021*, L. Bauer and T. Pionteck, Eds. Cham: Springer International Publishing, 2021. **Best paper award won.**
- L. Rosenbauer, A. Stein, and J. Hähner, "A Genetic Algorithm for HMI Test Infrastructure Fine Tuning," in *Proceedings of the 18th International Conference on Informatics in Control (ICINCO), July 6-8, 2021*, K. Madani, O. Gusikhin, and H. Nijmeijer, Eds., 2021
- L. Rosenbauer, J. Maier, D. Gerber, A. Stein, and J. Hähner, "An Evolutionary Calibration Approach for Touch Interface Filter Chains," in *Proceedings of the 18th International Conference on Informatics in Control (ICINCO), July 6-8, 2021*, K. Madani, O. Gusikhin, and H. Nijmeijer, Eds., 2021. **Best industrial paper award won.**

- M. Kshirsagar, K. Gupt, L. Rosenbauer, C. Ryan, and J. Sullivan, "Hierarchical Clustering Driven Test Case Selection in Digital Circuits," in *Proceedings of the 16th International Conference on Software Technologies - ICSOFT*, L. Maciaszek, H.-G. Fill, and M. van Sinderen, Eds. SciTePress, 2021

1 Introduction

Testing plays a major role in the development of new software products as at it aims at finding errors and thereby ensures quality [21]. Further, it is responsible for a considerable part of the project's total cost [22–24]. Thus innovation in software testing is desirable both from an economical as well as from a qualitative point of view.

Software quality assurance itself is not a single task but rather consists out of various subtasks. These include for example the choice of an appropriate test suite [6] or formal correctness proofs [25]. Thus we aim at solving a series of testing subproblems within this thesis.

We take a focus on use cases that arise from changes in software development itself. There has been a move from linear software development processes such as the *waterfall model* to agile ones [26]. Agile methods have in common that they are iterative and software modules can change several times throughout a project's lifetime. Further, it has become a common practice to test as early as possible and often parallel to software development [27]. Hence testing activities must be adapted and optimized towards an ever changing software.

Constant adaptation and optimization can be found in multiple processes in nature, most prominently evolution. Thus phenomena found in nature have been translated into algorithms in order to make use of these valuable properties. This discipline of computer science is known as *Evolutionary Computation* (EC) [28] which belongs to the broader field of *Artificial Intelligence* (AI).

A systems engineering discipline which makes frequent use of AI and EC is *Organic Computing* (OC) [29]. OC proposes design patterns and methodologies to create embedded systems that can satisfy their goals in an ever changing environment. OC systems can usually be described as robust, adaptive, autonomous and further have certain self-X properties such as self-optimizing or self-learning. Notable examples include smart camera systems [30] or organic traffic control [31].

The aforementioned properties of EC and OC have motivated us to apply these techniques to a series of software testing problems. This has not only led to advances in software testing but also in EC and OC. As a side-effect we improved a machine learning algorithm called *XCS classifier system* (XCS) [32] as well as a family of metaheuristics coined *Germinal Center Artificial Immune Systems* (GCAIS) [33]. Further, we developed an OC system for software testing and thereby employed a first OC application outside of embedded systems.

The research conducted is application driven and we take focus on the reproducibility of our results. Thereby we take special focus on algorithmic descriptions of the methods provided and follow an open source mentality when possible. The intention is to enable practitioners to quickly employ the acquired knowledge.

1.1 Trends in Testing

Software testing has been examined from various perspectives. Within this chapter we intend to give a brief overview about some ideas that we encountered. It is worth mentioning that this is by no means a complete overview.

There are software testing approaches that focus on the concrete product at hand. There are methods that are aimed at *graphical user interfaces* (GUI) [34], embedded systems [35] or distributed systems [36]. These approaches usually are adaptations of more generic verification techniques. For example the aforementioned work on distributed systems [36] heavily relies on *random testing* which is product-independent.

Random testing aims at creating different inputs for a piece of software and to check how the system reacts to these inputs. The main idea of the methodology is that a set of diverse test parameterizations might reveal a high number of errors [37]. The creation of the inputs often involves some form of randomization (thus the name random testing). However, there are variations that are not purely random but take the testing history or the similarity of already chosen inputs into account. Random testing has an active research community as documented in the survey of Huang et al. [37]. Further, some of the random testing methods are rooted in EC.

Whilst random testing relies more on applied math (stochastics and optimization), the approach chosen by the prior mentioned embedded system example is located in software engineering [35]. It uses the software design model to derive an abstract set of tests (also called a *test suite*). These abstract

tests are then translated into concrete tests. For example a formal test description is implemented as an unit test in a specific programming language. These approaches are summarized as *model-based* testing. For a more fine-grained discussion of model-based testing the reader is referred to the survey of Li et al. [38].

These aforementioned techniques have in common that they create a test suite. Another field of software testing aims at evaluating the effectiveness of the tests at hand by introducing bugs to the software to be tested. This can be achieved by creating random software changes which are also known as *program mutations*. The available tests are thereafter applied to the mutated software and their failure revealing capabilities are evaluated. This process may be repeated. Based on the outcome, redundant tests can be identified and removed. Testing methods that roughly follow this process are known as *mutation testing* and for a fine-grained overview we recommend to read the survey of Papadakis et al. [39]. It is worth mentioning that mutation testing can also be used to create new test cases.

Testing can be separated into *black box* and *white box* testing [40]. White box testing summarizes activities where the underlying source code is available and black box testing where it is not. For example, the latter might occur if software is bought from third parties and only binaries are delivered. Hence black box testing activities usually focus on metrics such as requirements coverage or failure revealing capabilities [40]. White box testing techniques can also examine other metrics such as code coverage (e. g. how many lines of code are tested with a given test suite?) [41]. A current trend is not to pursue one goal such as code coverage but several, possibly contradicting, objectives simultaneously [41]. These issues can be modelled as multi-objective optimization problems and EC has lead to a variety of algorithms that fit such problems [42]. Their application to testing problems is known as *search-based software engineering* [43].

The meaning of an individual test can already be defined within the specification of the software to be created. One way to specify tests is to document which system requirements they cover [41]. A single requirement might be covered by several tests and thus there might be redundant tests in the specification. Hence there have been some approaches which focus on identifying redundant ones in the test specification as can be seen in the survey of Yoo and Harman [41]. It is worth mentioning that the underlying problem is NP-hard and known as the *Minimum Set Cover Problem* (MSCP). For the MSCP the quality of an approximation can depreciate if the problem instance's size is large [44] (for algorithms with a polynomial runtime). MSCP as well occurs in other real-world problems and due to its hardness it has been examined intensively by the EC community [7].

Natural language processing (NLP) has also been applied to software testing. A well studied use case is to determine test cases from the system's specification (the text describing it). A systematic overview is given in the survey of Garousi et al. [45]. Another application is the prediction of a manual test's outcome [46]. Thereby expensive manual tests that are likely to pass can be avoided. Li et al. [47] use NLP to create the documentation of unit tests in an automated manner. Sometimes it might occur that two tests detect the same software bug. Failing tests are usually examined manually. Runeson et al. [48] developed a method to identify test reports that mention the same bug in order to reduce the amount of manual labor spent on error analysis.

Computer Vision (CV) has also found its way to testing. For example the behaviour of a GUI can be examined using CV. The state of a GUI is manipulated and an image of the screen is taken and thereafter examined if it meets the expectation [49]. Further, CV can be employed to test the physical behaviour of embedded systems [50].

NLP and CV based methods often make use of AI. It is also worth mentioning that a novel field of software testing researches how AI systems are tested properly [51].

The previously discussed methods work on a test case level (generating new tests, identifying redundant tests, determining a crucial test suite). Another way to examine a piece of software is not linked to using tests, but instead using formal methods. These may rely on a specialized logic which defines a set of rules to use (also known as *axioms*) that are assumed to be true. Based on these rules a mathematical proof is performed. A well-known representative of these systems is the Hoare logic [52]. Formal verification techniques have been successfully applied to different use cases which include parallel programming [53] and hardware development [54]. It is worth mentioning that these proofs can also be performed in an automated manner if certain preconditions are met [55].

(a) BMW E30 series from the 1980s [56].

(b) BMW G20 series from 2019 [57].

Figure 1: BMW 3 series from the 1980s and today.

1.2 Problem Statement

Here we discuss on which part of the wide range of testing we focus. The thesis was initiated by observation that we want to share first. Based on it we infer major challenges for testing.

Our observation is regarding how some products have changed over time. Figure 1 displays the BMW 3-series from the 1980s (E30) and today (G20). The E30 is mostly mechanical, has few electronic components and very little software. On the other hand the newer version heavily relies on software and offers more and more software features that the version from the 80s did not have (e. g. gesture control, navigation or connectivity features). Thus one might infer that the software complexity in cars is rising. We made a similar observation for our industrial partner *BSH Hausgeräte GmbH* (BSH) which is Europe's biggest producer of home appliances. Their products also evolved from mostly mechanical to electronics / software based devices. For example their products have AI features, are connected to the internet, and they are run with a micro computer using a Linux operating system.

Higher numbers of tests are necessary in order to handle this complexity and still be able to deliver a high quality product. The rising number of features is partially handled by employing agile software development methods (BSH software engineers use Scrum) [26]. However, due the iterative nature of agile methods this means that testing must be done in an iterative way as well. Thus the software testing department of BSH started to rely on automated testing in order to shorten the time spent testing (this gives the developers earlier feedback). Test runs are usually triggered both periodically and whenever new software is checked in using an automation tool called Jenkins [58]. Thus prolonged idle times between the creation of the software and its testing can be avoided. Nonetheless due to the rising complexity there are still high numbers of tests to run. For example just the system level tests of a BSH dishwasher have a runtime of more than 48 hours. There is a need to give developers an earlier insight in the current status of the software. We focus within this thesis on determining crucial and redundant test cases to solve this customer pain. We examine two topics to reduce and control this complexity:

- Reduction of the planned tests. We identify and permanently remove redundant tests based on the specification. Thus the complexity can be decreased.
- Selection of a set of important test cases when only limited time is available. This enables us to control the remaining complexity.

We take a special focus on the latter point. There we examine different notions of what an important test case is. This may depend on the situation and from a pure practical point of view some objectives might not be feasible as the data is simply not available. Further the context matters as a fully automated setting leads to different demands as a semi-automated one. We saw that both can be found inside BSH. Hence we developed a set of solutions from which one can choose based on his test setting (automated or semi-automated), what information is available and what important means for the engineer.

1.3 Scientific and Engineering Contributions

The key challenge that we described in the previous subchapter is the handling of a rising numbers of test cases. Thus we developed a series of techniques to tackle this issue for differing scenarios.

Tests are often described by a specification which states what the scope of the test is. We examined if redundant test cases can be found and removed. The underlying optimization problem is NP-hard. We summarized, analysed, and detected issues of some of the approximation algorithms available for the problem. We overcome the issues of the most promising metaheuristic which we examined (GCAIS) by improving its algorithmic structure. Thereby we boosted the method's convergence speed, lowered its memory usage whilst keeping the approximation quality. For our testing application we could reduce the number of costly, manual test cases that need to be performed.

Further, we take a look at partially automated testing. In this scenario testers execute manually selected automated tests locally and closely observe the device under test. The underlying selection problems are deeply linked to the specification problem. Thus we reused and adapted GCAIS to this selection problem. We introduced population initialization techniques which lead to a rich set of solutions that enable testers to find errors earlier and more of them (if compared to other techniques). Further, the errors found are scattered among different features as the metaheuristic employed also optimizes a requirement coverage criteria. In practice we combine it with a manual preselection (based on the tester's expert knowledge) in order to avoid larger search times.

The main innovation of the thesis is located in *continuous integration* (CI). CI is a modern software development approach that focuses on the frequent integration of individual developer's source code. Thereby big forks that are hard to merge can be avoided. However, on the other side the new code has also to be tested in order to avoid that bugs are being introduced. Additionally, there is no unlimited time for testing available (this might endanger an important milestone). Thus the most crucial test cases must be detected. The test suite must be adapted to the newest software changes. We successfully apply a modified XCS to prioritize and select crucial tests. Thereby we developed the following modifications:

- We employ *experience replay* (ER) to improve the system's performance. ER is a technique which stores and reuses data for training. The approach is widespread for neural networks and has just recently gotten in the focus of machine learning methods such as XCS (as can be seen in one of our earlier works [12]). We are the first to successfully employ ER on XCS for a practical problem [15].

- We use a XCS variant for function approximation which is coined *XCSF classifier system* (XCSF) to compute continuous test case priorities. This is enabled by designing a heuristic to choose the priorities based on a test case value function [14].

- We were the first to develop a technique to successfully reuse a test prioritization mechanism learned for one test project to be applied to another one [16].

- We also examine the developed test case prioritization technique using OC metrics, namely its robustness (recovery measure for performance breakdowns). We extend the usual evaluation by also giving insight into the duration of recovery and by introducing a frequency-based approach to measure how often such performance breakdowns occur.

Furthermore we developed two generic LCS parallelization approaches which may run on any multicore computing device and thus may be used for our CI use case as there the hardware of the concrete automatization server is not known.

In order to get the value created to the customer it is necessary to wrap it in a system which interacts with the test bed, reads out available tests and executes the selected one in an automated manner. Thereby we rely on an OC system as it turns out that several of the OC design goals fit the testing use case well.

We further introduce an extension of testing itself which we name *corrective testing*. There we examine the product under test and reconfigure it accordingly until it meets the specified behaviour. Thus we also automate the correction step and thereby take a look at the development loop. We provide a first successful preliminary study based on a concrete example (touch interfaces [19]).

1.4 Structure of the Thesis

The research conducted throughout the PhD studies has been application driven. This can also be found in the structure of the thesis. Each chapter focuses on a specific application. Furthermore the degree of automatisation in the test problem at hand is increasing from chapter to chapter.

We decided to write parts of the thesis as individual as possible in order to enable readers to focus on the application which is in the scope of their interest. We have been inspired to follow this approach by the Organic Computing book which emphasizes this way of writing [29].

For readers unfamiliar with basic concepts from optimization, machine learning, and OC design principles we provided the necessary background to understand the thesis in chapter 2. Therein we focus to explain the aforementioned topics in a brief and simple manner.

The two next chapters are from a EC perspective multi-objective optimization chapters. Chapter 3 deals with the aforementioned identification of redundant test cases based on the specification. Here we explain the underlying NP-hard problem, theoretical insights as well as metaheuristics to approximate it. We introduce our improved method and evaluate it in a series of experiments. In this we integrate some of our previous published work [7, 8].

Chapter 4 discusses how tests can be selected for test engineers who work in a semi-automated way. Thereby we consider several criteria and the underlying mathematical optimization problem is linked to the problem that we examine in the previous chapter. Hence this chapter builds upon the prior one. We present how we adapted GCAIS for the task. We start by presenting our insights gained for a pure requirements based selection [9]. We further discuss an extended variant which takes the testing history into account [10].

In chapters 5 to 7 we discuss how we approached testing within CI. Thereby we proceed a bottom up strategy. We introduce the machine learning approach first, then how runtime might be cut down and afterwards discuss the system architecture which employs the technology.

Chapter 5 discusses the optimization problem first and we introduce how we approximated it using learning classifier systems (chapter 5). In this part of the thesis we integrate our works about XCSF [14], experience replay [12, 15], and transfer learning [16]. We further evaluate the robustness of the methodology developed.

We discuss a generic parallelization approach for LCS in chapter 6. It is generic in terms of the LCS used and the underlying computing system. For the computing system we only assume a multicore processor. This fits our testing use case better as it is generally not known with what hardware it is equipped. We introduce and evaluate easy to implement parallel algorithms which can reduce a LCS's runtime. This chapter is based on [11].

In chapter 7 we show how OC design patterns and architectures can be used to wrap the XCSF based selection technique into a testing system for CI. The system relies on a multi-layered observer-controller architecture and we underline that it has several properties which are of interest from CI point of view (e. g. high degree of autonomy). The chapter relies on [17].

In chapter 8 we describe the idea of corrective testing as a future research direction which takes a look at the development process as a whole. Thereby we give insight into a touch interface component which we test and adapt until it meets the requirements [19].

The thesis is closed with a summary, conclusion and an outlook which takes the entire contribution of the thesis into account.

1.5 Employed Scientific Method

Modern research follows a set of practises which is commonly known as the *scientific method* [59]. For empirical sciences such as physics or machine learning it roughly consists out of the following steps [60]:

1. Define a question.
2. Gather related work.
3. Form a hypothesis that corresponds to the research question.
4. Evaluate the hypothesis using an reproducible experiment.

5. Analyse the experimental data and draw conclusions that may be used for future research.

6. Publish the results.

These steps are usually examined during the peer-reviews of conferences. However, publications can still be hard to reproduce. The nature magazine performed a survey among about $1,500$ scientists from different fields about the reproducibility of the experiments conducted [61]. More than 70 percent tried to reproduce the results of other researches and failed. Some of them even claim that there is a crisis in science due to a lack of reproducibility (step 4 of the aformentioned enumeration).

The questioned scientist also gave insight on the reasons for their failure to reproduce. Some of these issues may not be found in machine learning (e. g. variability of standard reagents). However, issues such as a the lack of the experiment's source code or raw data might very well occur. Thus in order to make it easier for others to reproduce our results we published both the source code and the datasets whenever possible.

We maintain the code and the data in several repositories on Github (based on the subject of the experiments). The Git repositories can be downloaded from here:

`https://github.com/LagLukas`

They are published using open source licenses in order to enable others to freely use the content. In the succeeding chapters we are referencing the corresponding repositories of the experiments. It is worth mentioning that this was not always possible throughout the research conducted as sometimes we had make use proprietary software of an industrial partner which were not allowed to publish.

A side-effect of the open source mentality that we pursue is that it also enables others to build upon our insights. In fact, this is what we ourselves did as we extended the results of Spieker et al. [6] based on the source code and datasets that they published.

2 Background & Prerequisites

Within this chapter we introduce some basic concepts from computer science / mathematics which we deem as essential for understanding parts of this thesis. We start by introducing the notion of *NP-hardness* since several problems within testing turn out to be in this class [41]. Thereby we rely on the book of Jungnickel [62] if not stated otherwise. A part of the thesis' contribution is linked to machine learning and thus we briefly introduce this subfield of AI. The thesis also has a systems engineering contribution that makes use of OC design patterns. Thus we introduce some of the basic architectural concepts from OC (for this we rely on the OC book if not stated otherwise [29]). Organic Computing use cases have their origins in embedded systems. Within this thesis this is not the case and hence we concentrate on the more generic parts.

2.1 NP-hard Optimization Problems

NP-hard optimization problems are linked to decision problems. The latter are questions that are to be answered with *yes* or *no*. A well-known decision problem from mathematics is the *Königsberger bridge problem*. The city of Königsberg was separated into four different parts by the river *Pregel* (as seen in Figure 2). The parts were connected with each other by seven bridges. The question to be answered is: Is there a tour that starts and ends in the same city district and visits each bridge exactly once?

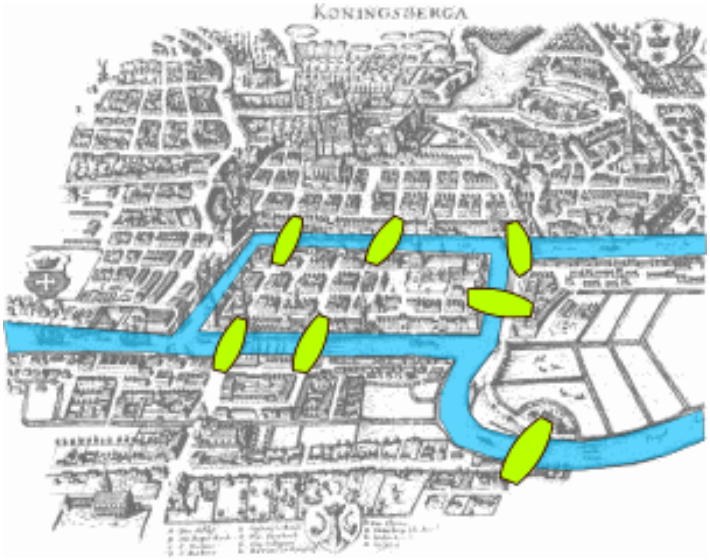


Figure 2: The seven bridges of Königsberg [2].

.

Decision problems can be separated in classes. Problems which can be solved by an algorithm deterministically in polynomial time belong to the class P. In this context polynomial runtime means that the number computation steps of the algorithm are bounded by a function that is a polynomial in the problem size[1]. For our example the problem size would be the number of bridges and the number of city districts. It turns out that there is such an algorithm which can be traced back to Hierholzer [62, p.41-43]. Thus the aforementioned decision problem belongs to P.

Another notable class is NP which stands for *non-deterministical polynomial time*. It summarizes problems where a *yes proof* can be verified in polynomial time, hence $P \subset NP$. There is a subset of problems which are known to be *NP-complete*. These problems have the property that if we would be able to solve them efficiently (that means in polynomial time) then we could solve all problems within

[1]These polynomial boundaries are often expressed using the *Landau notation / big O notation*. If an algorithm needs $f(x)$ then a function $g(x)$ is boundary for its runtime if $\limsup \frac{|f(x)|}{|g(x)|} < \infty$. For example if $f(x) = 4x^2 + x$ then $g(x) = x^2$ is a boundary for $f(x)$. This is abbreviated as $f(x)$ is $O(x^2)$.

NP efficiently. For example if we would switch the role of bridges and city districts in our example then the resulting problem is NP-complete and known as *Hamiltonian circle*.

An open question in mathematics and computer science is if NP is equal to P or not. If that would not be the case then NP complete problems cannot be solved in polynomial time. Hence NP-complete decision problems are regarded as difficult to solve in a reasonable amount of time.

An optimization problem is coined *NP-hard* if the underlying decision problem is NP-complete. For example if we would also consider the length for the Hamiltonian circle then the optimization problem would be to find the shortest route. The question that the underlying decision problem aims to answer is if there is an Hamiltonian circle whose length is smaller than a constant. Naturally if we could solve the optimization problem in polynomial time then we could do the same for the decision problem. Hence NP-hard optimization problems can also be seen as difficult to solve exactly and efficiently.

NP-hard problems have been separated even further in order to categorize how well they may be approximated. These classes rely on the *approximation ratio* of an algorithm A which we define first:

$$\frac{out(A, in)}{OPT(in)} \tag{1}$$

where OPT is the optimal value for the problem instance in and $out(A)$ is the output of A on in. The best value that can be achieved is 1. It is worth mentioning that this is the variant of the approximation ratio for minimization problems.

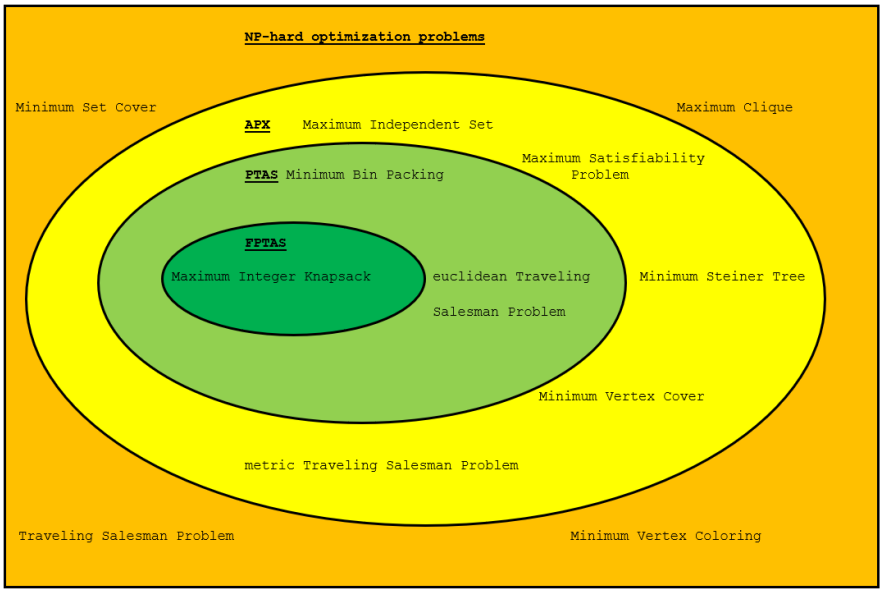


Figure 3: Example separation of some well-known NP-hard optimization problems based on [3, 4].

The definition of the approximation ratio enables us to introduce the following complexity classes (for minimization problems) [4]:

- A NP-hard problem is in *approximable* (APX) if there exists an algorithm that calculates a feasible solution in polynomial time (in terms of the problem size) and the corresponding worst case approximation ratio is bounded by a constant C.
- A NP-hard optimization problem has a *polynomial-time approximation scheme* (PTAS) if there exists an algorithm that produces a feasible solution that is within the factor of $(1 + \epsilon)$ being optimal in polynomial time (in terms of the problem size) where ϵ is an arbitrary postive number.
- A NP-hard optimization problem is in *fully polynomial-time approximation schemes* (FPTAS) if it is in PTAS and the corresponding approximation algorithm's runtime is also polynomial in terms of $\frac{1}{\epsilon}$.

For the classes the relationship $FPTAS \subset PTAS \subset APX$ holds. It is worth mentioning that there is a variety of NP-hard problems that are in APX, PTAS or FPTAS. We give a small overview containing some well-known problems in Figure 3.

2.2 Machine Learning

Machine learning (ML) is a category of AI which focuses on the design of algorithms that improve in an automated manner by experience [63]. These algorithms build a model based on data (the training dataset). This model can be used to propose actions, make predictions or perform classifications. This paradigm differs from more traditional algorithmic methods. Programmers do not explicitly state how the decision is made but define what is to be achieved. They rely on machine learning algorithms to build such a model which fulfills that goal.

Traditionally ML is separated into the following categories [64]:

- **Supervised learning**: Here the training data consists out of tuples of inputs and correct outputs. The goal is to learn a function which maps the input to the correct output. For example an animal classification function which trains on images of animals and predicts the species. Note, the output could also be a continuous value (these tasks are coined regression).
- **Unsupervised learning**: For this category no correct output is a part of training data. Corresponding machine learning methods aim at finding patterns in the data (e. g. customer personas based on a set of customer users) or create features which can be used for other tasks.
- **Reinforcement learning**: This discipline studies agents that interact with an environment. At each time step they are in a state and perform an action. For each action an agent receives a reward. Note, that the next state that the agent will enter is not necessarily exclusively determined by the action performed and the current state but there might also be a random influence. The goal of reinforcement learning is to determine a policy π which maximizes the expected reward received in the long run by proposing an action for each state [65].

It is worth mentioning that there are algorithm families such as LCS that can be used for more than one of these ML categories [66].

There are also other ways to categorize machine learning methodologies. For example if the machine learning algorithm makes use of analogies from the evolution such as LCS then they belong to the field of *evolutionary machine learning* [66].

ML methods can also be separated by the way they learn. *Online* ML methods receive data in a sequential manner and use it instantly for training whereas *offline* ML methods process a fixed set data once for training [64].

Another field of ML that we want to mention is *transfer learning* (TL) which studies the reusage of an already trained model for another similar task [67]. TL has already found its way into practice. In computer vision, object recognition models trained for the image net challenge can be retrained for other object recognition tasks. This functionality has already found its way into standard software libraries such as *Keras* [68]. Similar things are possible for NLP use cases [69].

It is worth mentioning that the aforementioned subfields of ML are by no means a complete list. We confined to the basic classes and the ones that occur within this thesis.

2.3 Observer Controller Architectures

OC has come up with multiple architectures which heavily rely on the *observer controller* design pattern as a building block. The observer controller pattern can be seen as an architectual approach for a control system. This control loop is set on top of a *system under observation and control* (SuOC). Further, sensors and actors are available which enable the control loop to influence the SuOC and measure the effects of the actions taken.

The observer's task is to read out the raw sensor values. It may perform some sort of preprocessing (e. g. in case of sensors that are cameras to convert images to grayscale). Additionally the observer can make initial predictions based on the currently measured values and saved previous ones. The observer packs the cleaned data together with these first predictions and reports them to the controller.

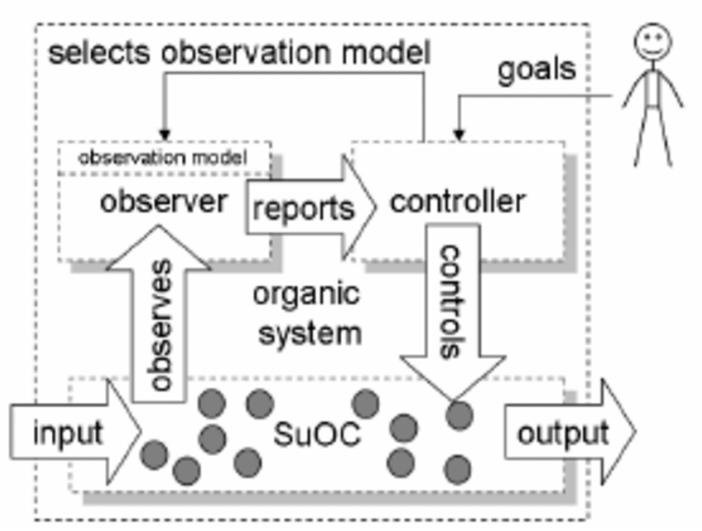


Figure 4: Visualization of the generic observer controller architecture [5].

The controller takes the information received by the observer and given goals into account in order to manipulate the SuOC using the actors. The internal decision mechanism usually relies on an online machine learning technique. Through this the decision mechanism is capable of adapting to a changing environment. The controller can also choose the observation model if the current one is not fitting. Thereby the prediction methods can be changed, different sensors can be read, analysis tools exchanged or simply a sampling frequency may be updated.

OC systems are often distributed and hence several of these observer controller structures (possibly with different SuOCs) may exist in parallel and might communicate with each other to achieve some common global goal. The interaction might lead to a *self-organizing* effect and a certain structure might evolve (which is know as *emergence* in OC). In order to avoid unwanted or harmful self-organisation and emergence, the observer has a detection mechanism to determine this behaviour and the controller can perform countermeasures.

Observer controller can be seen as basic building blocks and can be used to create other architectural concepts. One such approach is the *multi-layer observer controller* (MLOC) architecture.

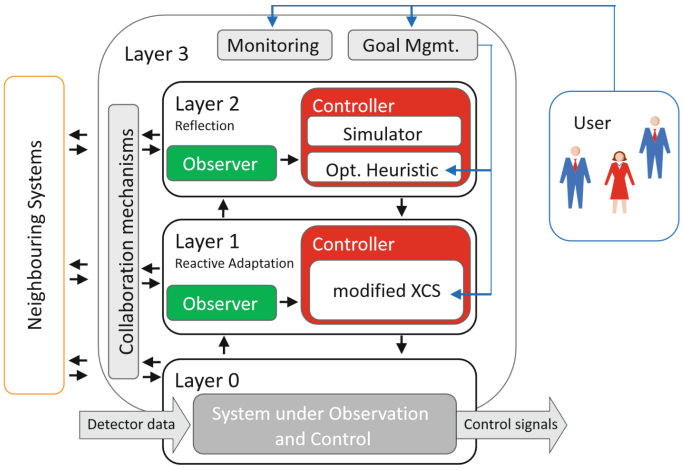


Figure 5: MLOC architecture visualized.

A commonly used MLOC architecture consists of four different layers which are shown in Figure 5. Each layer serves as an abstraction of layers lower in the architecture. The lowest layer, termed

productive layer or layer 0, wraps the SuOC whose state it measures and manipulates, e. g., using a set of sensors and actors.

Layer 1 (reactive adaptation layer) is structured by an observer-controller architecture. Here the sensors and actors usually correspond to actual hardware (e. g. cameras, a robot). In this case a modified *XCS classifier system* (XCS) is used as control mechanism. XCS belongs to the LCS family. LCS learn and maintain a set of rules which determine the next action to be performed.

The *reflection layer* (layer 2) differs from the previously discussed observer controller approaches as it is not meant to measure and influence some physical system, but the rule population of the XCS of layer 1. If situations are detected where the XCS is rather unexperienced (by the observer) then new rules are created either at random or using an optimization heuristic (by the controller). The controller evaluates these rules in a simulation and injects promising rules in XCS' rule basis in order to avoid poor performance for rather new situations.

The aforementioned layers result in an autonomous but isolated system. A third layer (*collaboration layer*) serves as an interface to neighboring systems (e. g., systems with a similar architecture) as well as the users. Communicating with neighboring systems enables adding collaboration mechanisms such as sharing of parameters or state information. The interface for users usually contains functionality for monitoring the system's performance as well as adjusting the system's goals.

A common assumption is that the presented layers are implemented in a way that if one of them fails (i. e., is not operable any more), the lower ones are still fully functional. This means that failing upper layers only limit the system's capabilities to adapt while not disrupting the capabilities of the lower layers.

3 Test Suite Minimisation

This chapter takes a focus on the specification of tests. With documentation tools such as Polarion [70] it is possible to define requirements, test cases and link the test cases with the requirements. Therefore these tools enable users to get an overview about which requirements are tested and by how many tests.

The test / requirements relation can be displayed as in Table 1. The rows display four tests ($T1$, $T2$, $T3$, $T4$) and the columns six requirements ($R1$, $R2$, ..., $R6$). A brief look at our example table reveals that T1 is redundant (a minimal test suite consists out of $T2$, $T3$ and $T4$). The task of finding and removing these redundant tests is coined *Test Suite Minimisation Problem* (TSMP) [41].

Table 1: Example of test / requirement table. A 1 indicates "covers requirement" and 0 means "does not cover the requirement".

	R1	R2	R3	R4	R5	R6
T1	0	1	1	0	0	0
T2	1	0	0	0	0	0
T3	0	0	1	1	0	0
T4	0	1	0	0	1	1

The Test Suite Minimization Problem is a variant of the *Minimum Set Cover Problem* (MSCP) which is known to be NP-hard [44]. Thereby optimal solutions are difficult to find, but a solution's quality can be evaluated in polynomial time. Approximation methods for TSMP often rely on generic approximation algorithms for MSCP [41, 71]. We focus within this chapter on the following topics:

- We introduce MSCP in a mathematical manner and discuss related work regarding its approximability and computational complexity.
- We give an overview of existing metaheuristics for MSCP and benchmark them on a series of problem instances.
- We choose a search algorithm based on the outcome of the benchmarking. We further improve it algorithmically and apply it to two test specifications of BSH fridges. Thereby we show that we can reduce the number of test cases necessary for the project.

It is worth mentioning that the application of the results that we present here requires a well-written requirements document and test specification. If there are requirements which are not split up and cannot be verified using a single test that is linked against that requirement, we would accidentally remove an important test. In order to avoid this we recommend that the output of a search heuristic should be additionally evaluated manually before any tests are removed.

3.1 Minimum Set Cover Problem

The MSCP is an optimization variant of the *set cover problem* which is a NP-complete decision problem and one of the first NP-hard problems discovered by Karp [62]. It intends to answer the question if there are k sets among a list of given sets S_1,...,S_n that cover all elements of $\cup_{i=1}^{n} S_i$. The goal of the minimization problem is to find the smallest k which achieves this. Note, that the tests correspond to the sets and the requirements to the elements that the sets cover. Mathematically the optimization problem can be described as follows (where the minimal $|M|$ corresponds to k):

$$\begin{aligned} & min\,|M| \\ s.t. \bigcup_{i \in M} S_i &= \bigcup_{i=1}^{n} S_i \\ S_i &\subseteq \{1, 2, ..., m\} \\ M &\subseteq \{1, 2, ..., n\} \end{aligned} \tag{2}$$

One can show that no polynomial time algorithm to approximate MSCP has a constant approximation ratio and thus the problem is not in APX [44] (given $P \neq NP$). Hence its worst case approximation ratio can grow large if the problem size is big. Therefore it can be considered as one of the more challenging NP-hard optimization problems.

The MSCP was one of the first NP-hard problems discovered by Karp [72]. Hence its hardness was examined in a long series of publications. Feige [73] proved that unless all problems in NP can be solved deterministically in $n^{O(log(log(m)))}$ time, no deterministic polynomial-time algorithm can solve the MSCP to within a factor of $ln(m) - \epsilon$ of the optimum for any constant $\epsilon > 0$. Alon et al. [74] showed that, under the weaker hypothesis $P \neq NP$, a polynomial-time approximation within $cln(m)$ is impossible for a certain constant $c > 0$. Dinur and Steurer [44] improved that result by showing that the MSCP cannot be approximated to $(1 - o(m))ln(m)$ given $P \neq NP$. Overall, these results state that for every approximation algorithm for the MSCP there exists a problem instance such that the calculated solution is at least about $ln(m)$ times the optimal value (unless $P = NP$). As these results are about worst case approximation ratios there is still research for an approximation algorithm that might have good results on average [7].

Approximation algorithms also exist outside of EC. Some of them have already been used for the MSCP. We intend to mention some of them briefly. One possible approach is to iteratively add the set to the solution which covers the most elements (which are not already covered by the solution). This greedy algorithm has a logarithmic worst case approximation ratio [4]. Another approach is to transform the MSCP into a integer *linear program* (LP)[2]. The integer constraint can be dropped and a optimal solution for the LP can be determined. However, it may be possible that the solution of the LP is infeasible for the version with the integer constraint. It can be rounded in order to create a feasible solution. There are rounding techniques known to produce solutions which also have a worst case logarithmic approximation rate [4]. It is also worth mentioning that other methods such as branch and bound algorithms exists which can compute optimal solutions. However, these have the downside that their worst case runtime is exponential in the problem size [76].

3.2 Overview of existing Metaheuristics

There have been various approaches from EC to design algorithms that approximate MSCP problems well on average. These works usually evaluate their performance by benchmarking them on various problem instances and analysing the empirical results [7].

We focus on seven different search heuristics which we found by using Google scholar. Several of the works found benchmark their algorithms on the same problem instances (e. g. in [77, 78]). We could determine some methods that were outperformed by others. Further, we could observe that there was a switch in which problem instances are considered. Thus we also include some of the older metaheuristics in order to enlarge our comparison. We also identified two EC algorithms that were examined mathematically and have a guaranteed approximation ratio.

The majority of the algorithms that we present encode solutions as binary vectors of length n. The i-th vector indicates if the i-th set is a part of the solution (1 for yes, 0 for no).

3.2.1 SEIP

Simple evolutionary algorithms with isolated population (SEIP) are a rather new family of algorithms in the field of computational intelligence. It is a population based approach that starts with an empty population. At the beginning an initial solution such as the zero vector $\mathbf{0}$ is chosen and inserted into the population. After that a loop is entered. In each iteration an instance of the population is drawn at random. Then it is copied and mutated. Next, the mutated instance is compared to the ones in the population. If there is an instance superior to the mutated one, it is not inserted. Otherwise the mutated instance is inserted into the population and all instances inferior to the mutated one are deleted from the population. The superior relation is specific to the problem. The loop is exited if a stopping criterion is met, e. g. no iterations are left [79].

[2] A linear program is an optimization problem where a linear function is to be maximized or minimized. Further the problem can have linear (in)equalities as constraints. It also has the constraint that the variables are integers hence it is called a integer linear program. A didactic introduction can be found in the book of Bertsimas and Tsitsiklis [75].

Algorithm 1: Simple evolutionary algorithm with isolated population (SEIP).

```
input  : S_1, S_2, ..., S_n
output: feasible solution
P = {0}
while stopping criterion is not met do
    Choose x from P at random
    Create y from x via mutation
    if ∀ x ∈ P: ¬ superior(x, y) then
        Q = {x ∈ P |superior(y, x)}
        P = (P ∪ {y}) − Q
end
return best solution of P
```

The method is also described as pseudocode in Algorithm 1. SEIP considers a solution $\mathbf{x}$ to be superior to $\mathbf{y}$ if and only if it covers the same number of elements but uses fewer sets. In our implementation we use a bitwise mutation. Hence we flip each bit with a probability of $\frac{1}{n}$ and we stop after a certain number of iterations. It is worth mentioning that Yang et al. [79] showed that the logarithmic border discovered by Dinur and Steurer [44] can be achieved after m steps. Thus SEIP has the best worst case approximation ratio that can be achieved unless $P = NP$ (for an algorithm with polynomial runtime).

3.2.2 Artificial Immune Systems

Artificial immune systems (AIS) are population based heuristics inspired by the immune systems of vertebrates. The population is extended from time to time and the AIS tries to identify bad solutions among the population and deletes them (as an immune system tries to eliminate pathogens).

Joshi et al. [33] designed an AIS called *germinal centre artificial immune system* (GCAIS) for the MSCP. GCAIS allows infeasible solutions in its population. The initial population consists out of the zero vector $\mathbf{0}$. In every iteration all members of the population are mutated (the same way as SEIP). Afterwards the mutated population and the original one are merged. Then all elements of the new population that are *dominated* by other solutions in it are deleted. A solution is said to dominate another one if and only if it either covers more elements and does not cost more or if it covers at least the same number of elements and costs less. The method is described as pseudocode in Algorithm 2.

Algorithm 2: Germinal centre artificial immune system (GCAIS).

```
input  : S_1, S_2, ..., S_n
output: a solution
P = {0}
while stopping criterion is not met do
    P' = {}
    for x in P do
        y = mutate x
        insert y to P'
    end
    P = P ∪ P'
    delete all dominated elements from P
end
return best solution of P
```

3.2.3 GSEMO

The *global simple evolutionary multi-objective optimiser* [80] is similar to GCAIS as it maintains a population of non-dominated solutions during each iteration. In contrast to GCAIS it only mutates one member of the population instead of its entirety. This solution is drawn uniformly at random and mutated the same way. If it is dominated by any solution of the population then it is not inserted. Upon insertion of a non-dominated solution, the population is searched for dominated solutions which are subsequently removed. Thus GSEMO is similar to SEIP as they only differ in the way they regard a solution as superior.

GSEMO may also be parallelized. In order to do so, Γ populations are introduced. On each population an instance of GSEMO is run. Whenever an instance encounters a solution to be inserted to its population, it decides with a probability p if it sends the found solution to all the other populations. The recipients then also update their populations according to the received solution. For the sake of simplicity we call this variant *GSEMO* and describe it in Algorithm 3.

Algorithm 3: Global simple evolutionary multi-objective optimiser (GSEMO).

```
   input  : S_1, S_2, ..., S_n, Γ, p
   output: a solution
 1 P_i = {0} ∀i ∈ {1, 2, ..., Γ}
 2 while stopping criterion is not met do
 3     for i ∈ {1, 2, ..., Γ} do
 4         choose x uniformly at random from P_i
 5         y = mutate x
 6         if y is not dominated by any element of P_i then
 7             insert y to P_i
 8             if random() < p then
 9                 send y to all other populations
10         add all received solutions received to P_i
11         delete all dominated solutions of P_i
12     end
13 end
14 return best solution of all P_i
```

3.2.4 Genetic Algorithm

Genetic algorithms (GAs) are a framework of population based algorithms that roughly consist out of the three operators: selection, crossover and mutation [81]. Solutions are interpreted as chromosomes of an individual. The selection operator chooses a solution from the population. The GA uses it to choose two individuals. Via the crossover operator the two individuals are combined to get two new ones called the children. The children are changed probabilistically using the mutation operator. Afterwards, the GA tries to insert the children into the population but its capacity is limited. Hence sometimes individuals must be removed from the population. This approach is repeated until a stopping criterion is met and the best solution found is returned. We have summarized the basic steps in Algorithm 4.

Various attempts have been made to design a GA for the MSCP [82–84]. Beasley [83] used a binary tournament selection for their GA which draws two individuals from the population at random and takes the one with the least used sets. They further use a one point crossover operator. It creates a new child by drawing a random integer i from $\{1, 2, .., n\}$. For the first child the first i entries of $\mathbf{x}$ and the last $n-i$ of $\mathbf{y}$ are used. For the second the first i entries of $\mathbf{y}$ and the last $n-i$ of $\mathbf{x}$ are used.

The mutation operator of Beasley [83] inverts bits with a certain probability that is inverse monotone to the iteration. As a GA is not guaranteed to produce a feasible solution, they also introduced a greedy heuristic to make infeasible solutions valid. If the population becomes too big, random so-

Algorithm 4: Basic steps of a genetic algorithm (GA).

input : $S_1, S_2, ..., S_n$
output: feasible solution

```
1 P = create_initial_population()
2 while stopping criterion is not met do
3 |   Choose x, y from P via selection
4 |   Create x', y' from x, y via crossover
5 |   Mutate x', y'
6 |   insert x', y' to P
7 |   check size of P
8 end
9 return best solution of P
```

lutions with a cost above average are deleted to keep it in bounds. We use this variant of the GA in our later experiments.

Beasley [83] also developed a method to initialize the population in order to only generate feasible solutions. They iterate over every element to be covered and choose one set at random from the sets that can cover the element. The newly created solution is feasible, but may contain redundant sets. In order to delete redundant sets from the solution, they draw each used set exactly once uniformly at random and check if its elements are covered by other sets of the solution. If so, then the set is deleted from it.

3.2.5 Simulated Annealing

Simulated annealing (SA) is an analogy to physics which, unlike the previous methods, is not population based. The analogy reproduces a hot material that is cooling down. During the cooling the metal atoms have enough time to get into an optimal state by ordering and getting in a stable structure. This process is translated into a local search method. Therefore, a temperature function $T(\cdot)$ is created that decreases over time. The method starts at an initial solution and then tries to jump to another solution in its neighbourhood. The new solution is accepted if it is better (in terms of its cost). If not, it is only accepted with a probability based on the temperature function [85].

Algorithm 5: Simulated Annealing (SA) as pseudocode.

input : $S_1, S_2, ..., S_n$, initial solution $\mathbf{x}$
output: feasible solution

```
1  best = x
2  time = 0
3  while iterations left do
4  |   y = neighbour of x
5  |   if y is better than x then
6  |   |   x = y
7  |   |   best = y
8  |   else
9  |   |   probability = exp(|x|−|y| / T(time))
10 |   |   if random() < probability then
11 |   |   |   x = repair(y)
12 |   end
13 |   time = time + 1
14 end
15 return best found solution
```

There have been several publications about a SA algorithm for the MSCP [86, 87]. The initial solution can be created by the greedy algorithm and there are some degrees of freedom for the neighbourhood operator. The neighbourhood operator that we use drops sets at random. This might lead to an infeasible solution that can be repaired using a greedy algorithm. Afterwards a small search for redundant sets is done to keep the solution as small as possible.

Minotra [87] used the following temperature function:

$$T(t) = \gamma^t T_{initial} \tag{3}$$

where $T_{initial}$ is the starting temperature and γ is a real number between zero and one. In our later experiments we use this version of SA. We summarized the method in Algorithm 5.

3.2.6 Particle Swarm Optimization

Particle swarm optimization is a population based metaheuristic originally developed by Eberhart and Kennedy [88] to solve continuous problems. Each element of the population represents a particle. A particle holds its own current solution and the best solution that it found throughout its search. A particle is conscious of its neighbourhood and knows the current best solution of it. The particle is dragged into both the direction of its own best value and the best value of its neighbourhood. The particle's solution is updated until a stopping criterion is met.

As many optimization problems are discrete, there have been some attempts to adapt PSO for such problems [89, 90]. A new discrete version has been designed by Balaji et al. [78] especially for the MSCP which they call *JPSO*. Thus we focus on that version. It creates the initial solutions of its particles the same way as the genetic algorithm of Beasley [83].

In contrast to the traditional PSO, the JPSO algorithm does not calculate a direction based on several solutions. A particle p_i decides at random if it either moves its current solution $\mathbf{v}_i$ towards a random solution $\mathbf{x}$, the best solution of its neighbourhood $\mathbf{L}_i$, the global best solution $\mathbf{g}$ or towards its own best solution $\mathbf{b}_i$. The target solution is called the attractor. These moves have an analogy to jumps of frogs, thus are called jumps and hence the name *jump particle swarm optimization*. Each jump has the same probability of 0.25. The current solution is merged with the attractor. If the newly generated solution is superior to its own best solution, the best solution of the neighbourhood, or the global one then these are updated. We have summarized the basic behaviour in Algorithm 6.

The merge operator first draws a random number r between 0 and the number of used sets by the particle's current solution $\mathbf{v}_i$. Then we either delete a random set from $\mathbf{v}_i$ or add a random set from the attractor. Each event has the same probability of 0.5. This is repeated r times and costs $O(m)$. During this operation the solution might lose its feasibility. Hence it is repaired greedily if necessary. After this greedy repair the solution might contain redundant sets. Thus, the repair method further iterates over all used sets and checks if its elements are covered by other sets and removes it if so. The overall update of a particle depends on the used neighbourhood operator and random solution generator. It is worth mentioning that [78] do not state how they generate a new random attractor. For our later implementation we assumed that they take the same method as for their population generator. Furthermore they do not describe how to choose the neighbourhood of a solution.

3.2.7 Chemical Reaction Optimization

Chemical reaction optimization (CRO) is a rather new metaheuristic that is still topic of ongoing research in terms of runtime efficiency [91] and on what problems it performs well [92]. Similar to GAs it can be seen as a framework of algorithms that has to be adjusted to the specific problem. There already exists a CRO version for the MSCP which has been successfully tested on the benchmark suite of Beasley [77] such as the algorithm of Balaji et al. [78]. There it also beats the GA of Beasley [83] and often had optimal or close to optimal solutions.

CRO is a population based approach and each member of the population represents a molecule. Each molecule has a potential energy and a kinetic energy. The former is the cost of the solution the molecule holds and the latter describes its willingness to change to a worse solution. The method also holds a central buffer of energy that can exchange energy with the molecules. The entire process resembles more or less a chemical reaction in a box. A molecule can collide with the boxes' wall and

Algorithm 6: Jump particle swarm optimization (JPSO).

```
   input  : S_1, S_2, ..., S_n
   output: feasible solution
 1 P = create_initial_population()
 2 g = best solution among P
 3 while iterations left do
 4     for particle p_i in P do
 5         b = random(0,1)
 6         if b < 0.25 then
 7             attractor = random_solution()
 8         else if b < 0.5 then
 9             // jump towards the particle's best solution
10             attractor = b_i
11         else if b < 0.75 then
12             // jump towards the neighbourhoods best solution
13             attractor = L_i
14         else
15             // jump towards best known solution
16             attractor = g
17         end
18         merge current solution with attractor
19         if current solution better than b_i then
20             b_i = current solution
21         if current solution better than L_i then
22             L_i = current solution
23         if current solution better than g then
24             g = current solution
25     end
26 end
27 return g
```

might change its structure. This on wall collision searches the neighbourhood of the molecule's solution for a new one and changes it according to a certain probability that depends on the molecule and its buffer's energy. This introduces a form of random search to CRO. Molecules can also decompose into two new molecules that are partially based on the previous one which also represents a form of search.

Based on the quality of the two new solutions and their energy levels, they are either accepted into the population or not. If so, the original molecule gets destroyed. Furthermore two molecules can collide and form a new one. This process is called synthesis and has a similar objective like the crossover step in a GA. It combines two solutions in order to get an improved one. Similar to the preceding operators, the acceptance of the new molecule depends on the solution's quality and the energy levels. CRO takes another analogy to chemical reactions as two molecules can collide and bounce away. This is achieved once more by using a neighbourhood operator to perform local search. This operation is called inter-molecular collision. The four operators are called in a loop until a stopping criterion is met. For example, Yu et al. [77] used a fixed number of iterations.

We summarized the abstract behaviour in Algorithm 7. We keep it at this abstract level as a detailed description can be found in [92] and as we focus on the MSCP and not CRO in general in this thesis. There the decomposition and synthesis criteria are also described in great detail. Furthermore, their publication contains precise descriptions of the four operations CRO uses. We focus more on the adaptations of the neighbourhood operator, how the population is initialized, and how infeasible solutions are turned into feasible ones [77]. Thus, we can see a concrete version of CRO for the MSCP. We would like to point out that the previously mentioned work also proposes a version for the weighted minimum set cover problem (there each set has an individual cost).

Algorithm 7: Chemical reaction optimization (CRO).

```
   input  : S_1, S_2, ..., S_m
   output: feasible solution
 1 P = create_initial_population()
 2 while stopping criterion is not met do
 3 |   b = random(0, 1)
 4 |   if b < threshold then
 5 |   |   choose random molecule m from P
 6 |   |   if Decomposition criteria met then
 7 |   |   |   perform decomposition on m
 8 |   |   else
 9 |   |   |   perform an on wall collision on m
10 |   |   end
11 |   else
12 |   |   choose random molecules m1, m2 from P
13 |   |   if Synthesis criteria met then
14 |   |   |   perform synthesis on m1 and m2
15 |   |   else
16 |   |   |   perform an inter-molecular collision on m1 and m2
17 |   |   end
18 |   end
19 end
20 return best solution of P
```

First we have to introduce another notation for the encoding of the solutions. Every other considered algorithm uses a binary encoding for them where the i-th entry indicates if the corresponding set belongs to it. Yu et al. [77] use vectors of dimension m to encode a solution. They enumerate all elements and there the i-th entry indicates by which set the i-th element is covered. It is important to note that this does not lead to an unique representation and a set cover in the other encoding might have several encodings in this one.

A new solution is created by iteration over every element that must be covered. If the i-th element shall be covered then it is detected by which sets it can be covered. One out of these is drawn at random using an uniform distribution.

The neighbourhood operator first deletes the set from the cover which has the worst efficiency among all used ones. The efficiency of a set in a solution is the number of occurrences in the solution. After the set of worst efficiency is removed, the solution might become unfeasible as it can contain uncovered elements. In order to regain feasibility the operator chooses new sets to cover those elements. The i-th set is drawn with the following probability:

$$\frac{s_i}{\sum_{k=1}^{m} s_k} \tag{4}$$

where s_i is the number of elements that the i-th set can cover among the yet uncovered elements.

The CRO framework has a decomposition operator that tries to create two molecules from one. This is done by copying the original molecule twice and performing the neighbourhood operator on each copy ten times. The synthesis operator takes two solutions $\mathbf{x}$, $\mathbf{y}$ and combines them into a new solution $\mathbf{z}$. Here they are combined by choosing entries from each one uniformly at random. Hence:

$$P(\mathbf{z}_i = \mathbf{x}_i) = 0.5 \tag{5}$$

It is worth mentioning that due to the representation, no repair method is necessary. The collision operators do not need to be discussed further as they only rely on the neighbourhood operator.

3.3 Runtime Considerations

Previously we described a series of metaheuristics in an algorithmic way. Thereby we did not discuss important properties such as their runtime. In publications that focus on an empirical evaluation of an approximation algorithm we frequently observed that the metaheuristic was run for a fixed number of iterations [33, 77, 78, 87]. Some of them do not measure the actual runtime and also no theoretical one [33, 77]. In order to give an insight for practitioners, we analysed the worst case cost for an iteration of each algorithm considered. We did the analogous for initialization cost.

Table 2: Cost of an algorithm's *iteration* (iter) and *initialization* (init) in terms of the Landau notation.

	SEIP	GCAIS	GSEMO	
init	$O(n)$	$O(n)$	$O(\Gamma n)$	
iter	$O(n+m)$	$O(\|P\|^2 + Pn)$	$O(\sum_{i=1}^{\Gamma} \|P_i\|^2 + \|P_i\|n)$	
	GA	SA	JPSO	CRO
init	$O(\|P\|mn^2)$	$O(n)$	$O(\|P\|nm^2)$	$O(\|P\|nm)$
iter	$O(mn^2 + \|P\|)$	$O(mn^2)$	$O(\|P\|(n^2m + \chi))$	$O(mn^2)$

We give an overview about the runtime that we determined in our previous work [7] in Table 2. The χ which occurs in the iterational cost of JPSO is defined as follows:

$$\chi = max\{rand, neigh\} \tag{6}$$

where *rand* is the cost of a random solution generation and *neigh* the cost of the used neighbourhood operator. These operators are not stated in the original source [78] and thus we left the cost at this high level (in our later experiments we use the concrete generator and neighbourhood operator described in 3.2.6). Thus we encountered a reproduction problem as mentioned in chapter 1.5.

One observation that we can make is that five out of seven methods have runtime that is not linked to the problem size, but to other algorithmic parameters such as the population size. Some of the algorithms such as GCAIS have an unbounded population which might in turn lead to high runtimes.

The considered algorithms can vastly differ in terms of their cost. There are computational lightweights such as SEIP as well as more intense methods such as JPSO. For some algorithms the

term n^2m is appearing in their runtime. $O(n^2m)$ is the cost of the greedy algorithm mentioned in 3.1 on which many of these methods rely.

Our brief overview about the algorithmic complexities underlines our previous statement that an analysis solely based on the number of iterations such as in [33] is ambiguous as the methods might differ highly in terms of runtime.

3.4 Benchmarking

We extend the insight about the method's runtime by performing an empirical analysis. We decided to do so additionally to our worst case analysis due to the following reasons:

- We only gave an overview on the worst case behaviour. There are widely used algorithms such as Simplex that have bad worst case runtime, but are fast on average [93].
- The Landau notation leads to a coarse-grained runtime approximation since constants are left out. For example the introduced CRO uses the neighbourhood operator ten times (for the decomposition operator) but in the big O notation this costs the same as using it once.
- We did not give any insight of the relationship of the approximation quality and the runtime.

Further we go beyond the problem instances from Beasley's OR library in order to see if there is some wider level of generality for the quality of the metaheuristics considered. Thereby we might identify problem structures that are difficult to solve for some of them since there are instances where every algorithm either has a high runtime or a bad approximation ratio (given $P \neq NP$, see [44]).

Another theoretical result to keep in mind is the *no free lunch theorem* of Wolpert and Macready [94]. It states that all heuristics perform equally well on average on every optimization problem. Hence we also intend to evaluate if these methods that are highly adapted towards MSCP differ much in terms of their performance.

The source code and all problem instances that we considered throughout these experiments can be found here:

`https://github.com/LagLukas/minimum_set_cover`

For our experiments we used a Dell OptiPlex XE3 with 32GB RAM and an Intel i7 8700 processor and it was not used for anything else during the evaluation. Each experiment is run a hundred times.

We stop the execution of an algorithm if one of the following conditions is fulfilled:

- A time budget of one hour is expired.
- The algorithm fails to improve its best solution over 2000 iterations. This means the method cannot find a feasible solution that uses at least one set less than its known best solution. We interpret that as convergence.

For our benchmarking we use two different problem classes. It is worth mentioning that in one of our studies [7] we also considered instances from Beasley's OR library. However, we made similar observations and drew the same conclusions for Beasley's OR library and the problem classes presented here. Hence we leave them out here.

Some of the metaheuristics introduced require hyperparameters. We adopted the hyperparameters for GSEMO from Joshi et al. [33] ($\Gamma = 30$, send probability is set to $\frac{30}{nm}$). We also used the hyperparameters of the original source material for CRO [77] (see Table 3). For the SA we set γ to 0.975 and start with an initial temperature of 256. The GA has a population size of 200. In our JPSO version we use thirty particles and use the 5 nearest neighbours for all neighbourhood operations and we use the $L1$ norm as a metric. It is worth mentioning that SEIP and GCAIS do not have any hyperparameters.

Table 3: Hyperparameters of CRO.

Parameter	Value
Initial population size	10
Initial molecular kinetic energy	1000
Initial central energy buffer size	10000
Collision rate	0.1
Energy loss rate	0.1
Decomposition threshold	10000
Synthesis threshold	1000

3.4.1 Steiner Triple Systems

Fulkerson et al. [95] identified *Steiner triple systems* as especially tough MSCP instances. A Steiner triple system is a family of sets where each set contains exactly three elements. Furthermore, these sets overlap at most by one element. The sets' elements are from a base set B and further all possible two-subsets are distributed over the triplets. Each two-subset of B appears exactly once as a subset of a triple. Hence the elements are evenly distributed among the sets. Together with their equal size, this is the reason why such systems are regarded as tough.

We consider the Steiner triple systems proposed by Fulkerson et al. [95] which are formed on $B = \{1, 2, ..., 27\}$ and $B = \{1, 2, ..., 45\}$. We call these instances *Steiner 27* and *Steiner 45*. It is worth mentioning that the optimal solutions for these MSCP instances are known.

The results of each introduced algorithm are displayed in Table 4. The tables show the average, best approximation ratio, number of iterations and the actual runtime of each considered method. Here GCAIS and the GA produced optimal solutions. The other methods vary in their results and are achieving approximation ratios between 1.1 and 1.6. JPSO cannot improve the initial solutions of its particles for the Steiner triple system of size 27 (as it already converges after 2001 iterations).

The runtime results expose, similar to the theoretical examination, huge differences in terms of runtime. JPSO has by far the highest runtime and SEIP the lowest. The experiment also reveals that a pure evaluation of the number of iterations until an algorithm converges is ambiguous as JPSO has the smallest number of used iterations, but the highest runtime.

Further, the results show the weakness of GCAIS in terms of its runtime: if the sets are rather disjoint and of equal size, the population can become rather huge which leads to a high runtime. The latter is the case for Steiner triple systems which explains that GCAIS has such a high runtime compared to for example CRO or the GA.

Additionally we performed statistical tests to verify if the considered algorithms vary in terms of their iterations, durations and approximation ratios. Our null hypotheses are that they all behave the same way for these magnitudes. We applied Friedman tests to test these hypotheses which all had p-values lower than 10^{-9} which we regard as significant. Thus statistical tests state that the algorithms differ regarding the aforementioned magnitudes.

3.4.2 A bad Case for the Greedy Algorithm

The introduced JPSO, CRO, SA and GA have greedy mechanisms as a part of their search design. There is a class of problem instances where the greedy algorithm has a logarithmic approximation ratio similar to the worst case [3].

The example can be constructed as follows [3]: Let k be an integer bigger than zero. We construct sets S_1,...,S_k that are pairwise disjoint. S_k holds 2^k elements. S_j holds the elements $\{2^j + 1, 2^j + 2, ..., 2^{j+1}\}$ for $j > 1$ and $S_1 = \{1, 2\}$. We construct two additional sets M_1 and M_2. M_1 contains all even numbers and M_2 all odd numbers. Hence we want to cover a total number of $2^{k+1} - 2$ elements and M_1 and M_2 are the biggest sets. Further, half the elements of every S_j are in M_1 and the other half in M_2. The greedy algorithm selects S_j, S_{j-1},..., S_1 instead of the optimal solution consisting out of M_1 and M_2. We display one concrete example for $k = 3$ in Figure 6.

Table 4: Experimental results for Steiner triple systems of size 27 and 45. It contains average values $\pm\sigma$ for the iterations, approximation ratios and runtime (in seconds). The best values of each column are marked bold and the worst are in italics.

Steiner 27	iterations	duration	avg. iteration duration	avg. approx. ratio	best
CRO	2126.0 ± 53.0	1.048 ± 0.16	0.00049 ± 7e-05	*1.644 ± 0.07*	*1.556*
GA	2558.0 ± 478.0	4.342 ± 1.109	0.00172 ± 0.00305	1.078 ± 0.091	**1.0**
GCAIS	2102.0 ± 22.0	474.271 ± 86.936	0.22574 ± 0.0563	**1.0 ± 0.0**	**1.0**
GSEMO	3608.0 ± 1073.0	20.952 ± 8.107	0.00586 ± 0.00077	1.111 ± 0.091	**1.0**
JPSO	**2001.0 ± 0.0**	*701.123 ± 130.386*	*0.35039 ± 0.08644*	1.433 ± 0.082	1.333
SA	2649.0 ± 839.0	14.047 ± 5.01	0.00537 ± 0.02241	1.433 ± 0.063	1.333
SEIP	*4768.0 ± 1250.0*	**0.666 ± 0.242**	**0.00014 ± 3e-05**	1.167 ± 0.131	**1.0**
Steiner 45	iterations	avg. duration	avg. iteration duration	avg. approx. ratio	best
CRO	2166.0 ± 41.0	2.485 ± 0.62	0.00115 ± 0.00038	*1.727 ± 0.119*	*1.6*
GA	2847.0 ± 767.0	25.979 ± 5.508	0.00943 ± 0.01905	1.2 ± 0.144	**1.0**
GCAIS	2631.0 ± 454.0	1951.188 ± 320.956	0.74636 ± 0.23683	**1.033 ± 0.035**	**1.0**
GSEMO	4810.0 ± 2115.0	53.366 ± 28.564	0.01106 ± 0.00182	1.28 ± 0.076	1.133
JPSO	**1833.0 ± 121.0**	*3594.182 ± 15.282*	*1.9693 ± 0.63859*	1.64 ± 0.064	1.533
SA	3678.0 ± 1176.0	88.132 ± 37.294	0.02327 ± 0.00612	2.16 ± 0.126	2.0
SEIP	*7197.0 ± 1473.0*	**2.362 ± 0.706**	**0.00033 ± 3e-05**	1.247 ± 0.063	1.133

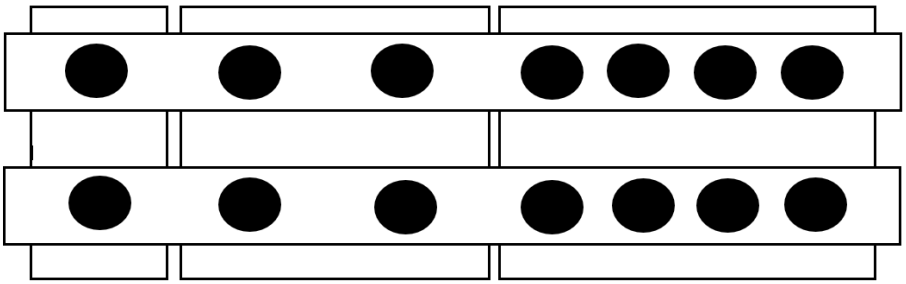


Figure 6: Example of a problem instance where the greedy algorithm has a logrithmic approximation ratio. The rectangles represent sets and the black circles the elements.

For our benchmarking we create several such systems for one instance. To create one instance we draw k from $\{2, 3, .., 5\}$ at random and create the corresponding set system. We repeat this 5 times and each system is disjoint to get our overall instance. Thus, we make it hard for greedy based approaches to break out of a bad solution to the known optimal one.

Table 5 shows the results on those randomly created instances (summarized as random instance). We put all in one table as the problems share the same inner structure and all have the same optimal value of 10. In this case only the GA is capable of finding optimal solutions and the optimal solutions were already in its population from the start. GCAIS, GSEMO, SEIP, and SA also seem to have rather good solutions (their best solutions use 12 instead of 10 sets). Algorithms such as CRO and JPSO that are relying on greedy heuristics achieve rather worse solutions compared to the aforementioned algorithms (due to the design of the problem class). JPSO is also not able to improve the initial solutions of its particles.

Once more JPSO has the highest runtime and SEIP the lowest. The runtime of GCAIS differs a lot on this problem class compared to the Steiner triple systems. In this problem class the available sets highly differ in size which may lead to a smaller population (as it is easier to find a solution that dominates another one).

Here we also performed Friedman tests to verify if the algorithms differ in terms of their iterations, duration and approximation ratios. Once more we could observe p-values lower than 10^{-9}. Thus the statistical tests support the claim that the algorithms differ in the aforementioned magnitudes.

Table 5: Experimental results for the bad cases for the greedy algorithm. It contains average values $\pm\sigma$ for the iterations, approximation ratios and runtime (in seconds). The best values of each column are marked bold and the worst are in italics.

rand	iterations	avg. duration	avg. iteration duration	avg. approx. ratio	best
CRO	2076.0 ± 53.0	1.084 ± 0.405	0.00052 ± 0.00021	*2.0 ± 0.0*	*2.0*
GA	**2001.0 ± 0.0**	2.635 ± 0.919	0.00132 ± 0.00346	**1.163 ± 0.084**	**1.0**
GCAIS	2302.0 ± 252.0	5.346 ± 1.357	0.00236 ± 0.00027	1.288 ± 0.06	1.25
GSEMO	2648.0 ± 200.0	9.208 ± 2.573	0.0035 ± 0.00021	1.288 ± 0.06	1.25
JPSO	**2001.0 ± 0.0**	*492.657 ± 113.087*	*0.24621 ± 0.06516*	2.013 ± 0.092	1.875
SA	2297.0 ± 514.0	9.221 ± 4.08	0.00403 ± 0.00175	1.3 ± 0.065	1.25
SEIP	*5813.0 ± 1258.0*	**1.049 ± 0.388**	**0.00018 ± 4e-05**	1.45 ± 0.222	1.25

3.5 Improving GCAIS

In our previous MSCP benchmarking we could recognize that GCAIS (Algorithm 2) had an optimal or close to optimal approximation ratio across several problem instances which is not the case for the other metaheuristics considered. This is also the case for several instances from Beasley's OR library [7]. On these instances GCAIS and GSEMO were capable of outperforming several of the other heuristics such as the GA. However, for some instances this comes at a cost. The runtime may be high if the sets are of similar size and only slightly overlap (e. g. as in the Steiner triple system evaluation). This can be traced back to large populations [96]. Within this subchapter we address this issue by introducing a simple datastructure and a population boundary based on an equivalence relation [8].

3.5.1 Population Boundaries

The domination relation which we described for the base variant of GCAIS is also known as *Pareto-domination* [97]. The relation is defined for an arbitrary number of objectives and not only two as for the MSCP (number of sets used, number of elements covered). A solutions is regarded as better if it is better in at least one objective and not worse in the remaining.

The base variant of GCAIS only distinguishes solutions by Pareto-domination which might be too coarse-grained (as in the example of the Steiner triple systems). We group the solutions based on their concrete values of their objectives. We regard two solutions as equivalent if they have the same values. For example we regard all MSCP solutions that cover ten elements whilst using two sets as equivalent.

We create a population boundary P_b for each equivalence class. Whenever the limit is exceeded then we drop as many random solutions as necessary to return an acceptable level. Hence the total population has a maximum capacity of $n * m * P_b + 1$ (there is only one solution that uses zero sets and covers zero elements).

3.5.2 Skyline Computation

The set of non-dominated solutions is known as the *Pareto-frontier* [97]. Its efficient computation has been examined by the database community [98, 99] as the Pareto-frontier contains the most "interesting" data points (set of best possible choices). There it is known as the computation of the *Skyline*.

A simplistic approach is to compare each solution with each other which is known as the *block nested loop* (BNL) algorithm. If the method is employed for the MSCP then it has a cost of $O(|P|^2)$. There is also a divide and conquer approach and prefiltering method which both have a runtime of $O(|P|log(|P|))$ [98, 99]. Other methods such as the research of Endres et al. [100, 101] take special focus on large amounts of data and efficient parallelizations on database nodes.

We explicitly exploit the fact that the MSCP has only two objectives. We save the population in a table. An entry i holds all solutions that use i sets. This has the side-effect that we can check the population boundary rather easily. We split the computation of the Skyline in GCAIS in two steps:

- Insertion of the mutated solutions.
- Removal of dominated solutions and possible deletion within in the population.

Whenever a newly created solution is to be inserted into the population we check if a corresponding table entry already exists. If not then we create a new entry (solely consisting of the new solution). If there already is an entry, we check how many elements the solutions cover. If the existing entry's solutions cover more elements then the new solution is not inserted. If they cover the same number of elements then the solution is added to the entry's ones. The third case is when the new solution covers more elements. In this case the entry's solutions are removed and the novel one is inserted. Each of these operations has a constant cost and hence we can insert a solution in $O(1)$. Thereby we can insert the mutated population in $O(|P|)$.

The population might contain dominated solutions after the insertion operation. For example the entry using three sets might cover more elements than the entry using four sets (this case is not considered within our insertion approach). Thus a population repair method is necessary in order to have population that forms a Pareto-frontier.

The repair method iterates over the table indexes $\{1, 2, ..., n\}$ (in an ascending order). We hold the start entry of index 1 in a variable. Whenever we encounter an entry that covers less elements it is deleted. If we encounter an entry that covers more elements then we save the entry (in form of its index) to the variable and continue as mentioned before. A succeeding entry's solutions that are dominated by the old entry's solutions are also Pareto-dominated by the variable's new value. After this operation the population contains no dominated solution. It is straightforward to see that this computation costs $O(n)$. It is worth mentioning that if $m \ll n$ then one could also index the table by the number of elements covered (this would lead to a cost of $O(m)$). Further, the Skyline computation alone does not change what the metaheuristic is computing, only how it is done (in contrast to the population boundaries). However, by employing the table approach we can insert and compute the Pareto-frontier in linear time.

For the sake of simplicity we summarized the insertion and Skyline procedure in Algorithm 8. *table* denotes the look up table and the entry using i sets is accessed using $table[i]$. The attribute *cov* describes how many elements the solutions of the entry cover. Hence the equivalence relation can be evaluated easily. In our later implementation we hold the solutions of an entry in a simple list.

3.5.3 Alternative Approach

It is worth mentioning that the issue that we encountered for GCAIS has also been discovered by Joshi et al. [96]. However, they run into the problem whilst examining the multi-objective knapsack problem and not MSCP.

Joshi et al. [96] did not solve the issue by using population boundaries or improving the way how the Skyline is computed. Instead they did not use Pareto-domination, but *epsilon-dominance*. The relation separates the objective space into hypercubes of edge length *epsilon*. If two solutions are compared which are not in the same hypercube, then the ordinary Pareto-Dominance is applied. If it is not the case, a score for each solution is computed. It is the sum of all objectives that are to be maximized minus the sum of all objectives that are to be minimized. The solution with higher score dominates the solution with the lower one.

If the concrete multi-objective optimization problem corresponds to the MSCP, one solution $\mathbf{x}$ epsilon-dominates a solution $\mathbf{y}$ if and only if the following condition holds (given both solutions are in the same hypercube):

$$||\mathbf{y}||_1 - ||\mathbf{x}||_1 + |\bigcup_{\mathbf{x}_i=1} S_i| - |\bigcup_{\mathbf{y}_i=1} S_i| > 0 \tag{7}$$

Thereby epsilon-dominance is more strict within a hypercube (it is more likely to sort out solutions). The idea is that this stricter dominance relation might lead to a smaller population which also has a positive effect on the runtime.

Algorithm 8: Insertion and Skyline procedure for GCAIS using a look-up table.

```
   input : mutated solutions P'
 1 // insertion
 2 for x in P' do
 3 |   covered_x = requirements covered by x
 4 |   if table has no entry |x| then
 5 |   |   table[|x|].cov = covered_x
 6 |   |   set the solutions of table[|x|] to x
 7 |   else if table[|x|].cov == covered_x then
 8 |   |   append x to the solutions of table[|x|]
 9 |   |   if table[|x|] holds more than P_b elements then
10 |   |   |   delete a random element of table[|x|]
11 |   else if table[|x|].cov < covered_x then
12 |   |   table[|x|].cov = covered_x
13 |   |   set the solutions of table[|x|] to x
14 end
15 // repair the table
16 i = 1
17 for k in {2, ...n} do
18 |   if table[k].cov ≤ table[i].cov then
19 |   |   delete table[k]
20 |   else if table[k].cov > table[i].cov then
21 |   |   i = k
22 end
```

3.5.4 Evaluation on Industrial Data

The MSCP instances we have evaluated previously were purely theoretical. We switch the focus on two MSCP instances that correspond to TSMP instances. The test specifications that we are examining within this experiment are manually executed system tests for fridges. These are of special interest as expensive manual labor must be invested to execute these (a BSH engineer in Germany costs about 100 Euros per hour).

We filtered out test cases of the two datasets that exclusively cover a requirement (these must be executed). The first dataset (Fridge-1) contains 15 test cases for 68 requirements. The second dataset (Fridge-2) contains 89 test cases for 133 requirements.

We compare our look-up table version of GCAIS with the base variant proposed by Joshi et al. [33]. Additionally we explore if the positive effects of epsilon-dominance are also valid for MSCP. Further in the analysis of the theoretical instances we saw that GSEMO performs reasonable well in terms of runtime and approximation quality. Hence we include this metaheuristic into the experiment.

We adopted the hyperparameters for GSEMO from chapter 3.4. We employ a population boundary of $P_b = 200$ and no boundary at all.[3] We examine the values 0, 5, 10 and 15 for epsilon. Note that a value of 0 indicates that raw Pareto-Dominance is used.

We evaluate the algorithm's performance using three different *key performance indicators* (KPI):

- **speed up**: The speed up measures the relative runtime of an algorithm A compared to an algorithm B:

$$\text{speed up(A, B)} = \frac{\text{runtime of B}}{\text{runtime of A}} \tag{8}$$

  A speed up of two means that A is twice as fast as B.

- **memory savings**: The memory savings is also a relative measure. We use the algorithms population sizes as a basis for it. The number of solutions that the algorithm maintains scales

[3] We retrieved that value in a small hyperparameter study. We actually determined that the magnitude of the variable has no high impact on the approximation quality. It is solely important that the boundary exists.

with the memory consumption and is mainly responsible for the memory usage of the considered metaheuristics. We compare the maximum population sizes encountered throughout the search. It is defined as follows:

$$\text{mem save(A, B)} = \frac{\text{max population size of B}}{\text{max population size of A}} \tag{9}$$

A value of ten indicates that A only uses ten percent of the memory used by B.

- **estimated approximation ratio**: An optimal solution for the two problem instances is not known and an exhaustive search is not feasible (for the Fridge-2 dataset there are 2^{89} different solutions). Hence we use an estimation of the optimal value $\widehat{OPT}$:

$$\text{est. approx. rate} = \frac{out(A)}{\widehat{OPT}} \tag{10}$$

We estimate the optimal value using the considered metaheuristics' output solutions.

For the memory savings and the speed up we compare the metaheuristics with the base variant of GCAIS (i. e. algorithm B in KPI definitions). Thereby we can examine if our algorithmic changes have a measureable positive effect. It is worth mentioning that the KPIs should not be evaluated in an isolated way, but in context with each other. For example an algorithm that just proposes to use all tests will have an excellent speed up and memory saving, but it will do pretty bad on the approximation rate KPI. On the other hand an exhaustive search leads to the best approximation rate but it will have a very bad speed up. Thus a method that performs decent in all three categories is desirable.

The TSMP instances are smaller than the theoretical MSCP instances. Hence we only give each algorithm a search time of ten minutes. Further, if we do not measure an improvement throughout a hundred iterations, we regard that as convergence. We repeat each search a hundred times and use the same computer as in chapter 3.4. The corresponding source code is available here:

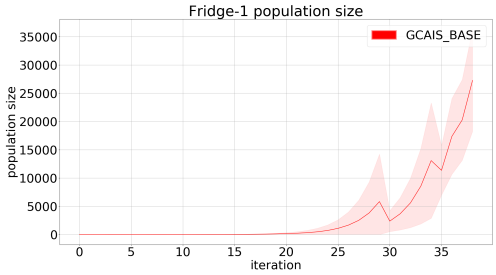
`https://github.com/LagLukas/gcais_test_suite_reduction`

The results of our experiments are displayed in Table 6. Our first dataset (Fridge-1) is rather easy to solve for the considered methods compared to our second one. The epsilon-dominance variants, the bounded variant and the base variant of GCAIS always achieve the best possible results. Also, GSEMO comes close to this. However, the combination of a bounded population and epsilon-dominance is rather detrimental as these versions produce worse solutions than the version without it. We can see certain differences in the memory usage and the speed up. The bounded version of GCAIS only uses about a tenth of the memory of its base variant and is about fourteen times faster. Yet GESMO is even faster but only achieves close to the best results and requires more memory.

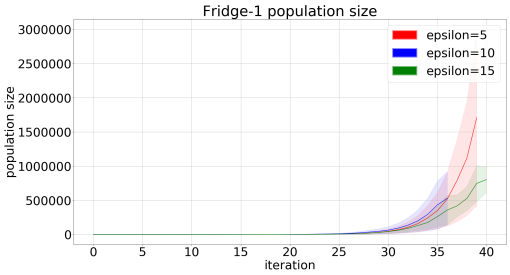
Our other dataset (Fridge-2) is tougher to solve for the considered metaheuristics as only the bounded GCAIS variant without epsilon-dominance always achieves optimal results. Once more the results show that this variant can drastically cut down memory usage and runtime. GSEMO has an even shorter runtime and memory usage but on the other hand only achieves approximation rates of about three. These differences between GSEMO and our bounded version of GCAIS are due to GSEMO's convergence to an inferior solution. GCAIS does not get stuck (as it always finds optimal solutions) and thus the population continues to grow as does the runtime.

On both datasets we could observe that in our case the epsilon-dominance has a detrimental effect on the population size and therefore on the runtime. Combined with a bounded population these effects disappear but the method is unable to find as good solutions as the GCAIS with population boundaries and no epsilon-dominance. Hence we could not observe the same positive effects of the usage of epsilon-dominance as [96] did for the Knapsack problem. The pure bounded version always achieved the best results and achieved high values in our other KPIs as well.

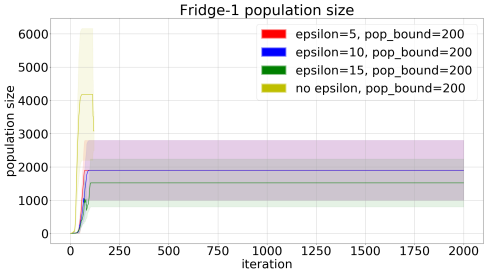
Most of the observed differences can be explained by taking a look at the population growth and size which we visualized in Figures 7 and 8. The base variant and pure epsilon-dominance variants of GCAIS show an exponential growth for Fridge-1 and on the other dataset we can observe a similar



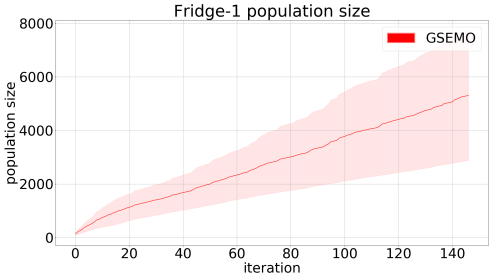
(a) GCAIS base variant



(b) GCAIS with epsilon-dominance

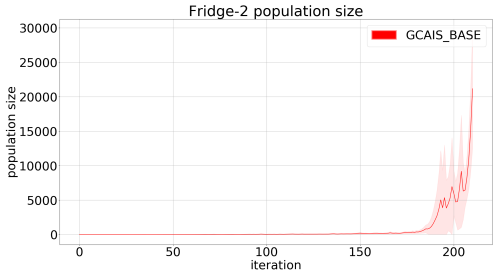


(c) GCAIS with bounded population

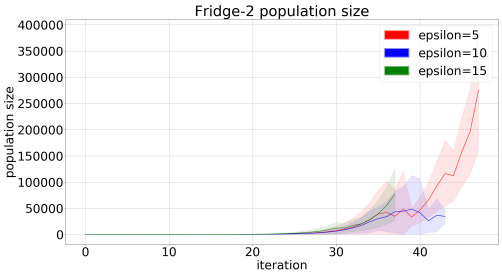


(d) GSEMO

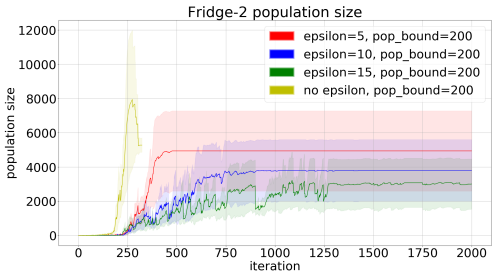
Figure 7: Population sizes $\pm\sigma$ for the Fridge-1 dataset.



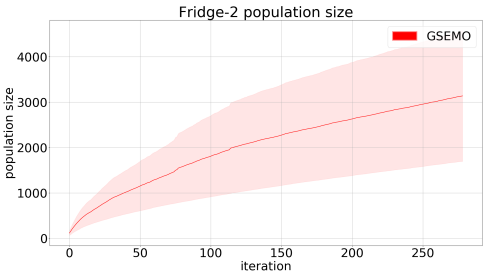
(a) GCAIS base variant



(b) GCAIS with epsilon-dominance



(c) GCAIS with bounded population



(d) GSEMO

Figure 8: Population sizes $\pm\sigma$ for the Fridge-2 dataset.

Table 6: KPIs for the experimental results (averaged values ± standard deviation σ). A - character indicates that the parameter was not used. We marked the best values of algorithms that always found optimal solutions bold. The best values of individual KPIs are marked in italics. The horizontal line separates our approaches to the ones we are comparing to.

Fridge-1	P_b	epsilon	mem save	speed up	est. approx. rate
GCAIS	200	5	22.77 ± 0.0	1.28 ± 0.19	1.29 ± 0.2
GCAIS	200	10	22.77 ± 0.0	1.54 ± 0.19	1.34 ± 0.17
GCAIS	200	15	*28.47 ± 0.0*	2.11 ± 0.16	1.38 ± 0.16
GCAIS	200	-	**10.35 ± 0.0**	**14.14 ± 0.02**	***1.0 ± 0.0***
GCAIS	-	5	0.03 ± 0.03	0.77 ± 0.16	***1.0 ± 0.0***
GCAIS	-	10	0.03 ± 8.95	0.75 ± 0.16	***1.0 ± 0.0***
GCAIS	-	15	0.03 ± 16.27	0.73 ± 0.15	***1.0 ± 0.0***
GCAIS_BASE	-	-	1.0	1.0	***1.0 ± 0.0***
GSEMO	-	-	1.66 ± 0.78	*16.81 ± 0.13*	1.03 ± 0.05
Fridge-2	P_b	epsilon	mem save	speed up	est. approx. rate
GCAIS	200	5	8.49 ± 0.0	0.32 ± 0.38	3.62 ± 0.46
GCAIS	200	10	10.9 ± 0.0	0.51 ± 0.48	3.82 ± 0.47
GCAIS	200	15	*12.49 ± 0.0*	0.84 ± 0.33	3.96 ± 0.3
GCAIS	200	-	**5.11 ± 0.01**	**3.58 ± 0.08**	***1.0 ± 0.0***
GCAIS	-	5	0.14 ± 3.4	0.88 ± 0.14	2.7 ± 0.13
GCAIS	-	10	0.18 ± 3.14	0.88 ± 0.14	2.66 ± 0.28
GCAIS	-	15	0.23 ± 2.47	0.96 ± 0.23	2.8 ± 0.19
GCAIS_BASE	-	-	1.0	1.0	1.0 ± 0.02
GSEMO	-	-	9.89 ± 0.01	*47.33 ± 0.01*	2.94 ± 0.15

observation for epsilon equal to 5. For the other two variants the runtime ran out and thus we do not fully see an exponential growth. GSEMO's population size grows linearly for Fridge-1 and more or less logarithmically for Fridge-2. Our bounded GCAIS version has, as expected, a constant population size (after several iterations). The jumps in the graphs are due to newly found solutions that dominate other solutions in the population which get deleted. These different growths are one of the causes for the differences in speed up as all of the considered algorithms have a runtime which depends on this magnitude.

The growths in terms of population size can be explained by taking a look at the structure of our datasets and the problem itself. Our test specifications consist out of test cases that have similar sizes and only slightly overlap in terms of the requirements which they cover. Also, the two dimensions (covered requirements and used tests) are integers and there are only limited valid values. Thus there can be many solutions that cover the same number of requirements and use the same tests and it is hard to find solutions which dominate large portions of the population. If the tests highly differ in their size it would be easier to find dominating solutions which leads to smaller populations. Hence we make similar observations as for the Steiner triple systems.

Next to our visual evaluation and the discussion of the raw values of Table 6, we perform additional statistical testing to confirm our observations. We test each KPI and each dataset individually. Our null hypothesis is that the algorithms do not differ on one dataset regarding one KPI. This can be verified using a Friedman test. On all different null hypotheses we observed p-values below 10^{-10} which we regard as significant. Thus we conclude that the algorithms differ in terms of the KPIs.

Overall we are able to reduce the size of the test suites by over 30 percent.

3.5.5 Evaluation on Beasley's OR Library

On our two MSCP instances that correspond to test specifications we could show that our algorithmic changes to GCAIS improved the runtime, memory usage and in the case of Fridge-2 even the approximation ratio. However, we have to keep in mind that two datasets form a limited observation window. Thus we also examine problem instances of Beasley's OR library.

We examine the *scpe* instances. These differ from our test specifications as they are much larger instances. Based on the results of Table 6 we confine to examine the base variant of GCAIS, the version with population boundaries (and no epsilon-dominance), and GSEMO. It is also worth discussing why we do not consider the variety of metaheuristics that we introduced in chapter 3.2. We already compared the base variant of GCAIS with CRO, the GA, SEIP etc. on several of Beasley's instances in an earlier work [7]. There we could observe that GCAIS and GSEMO have the best performance (regarding the approximation ratio).

Table 7: Experimental results for Beasley's OR library. The best values are marked bold. Each KPI is displayed ± standard deviation σ.

KPI	algorithm	scpe1	scpe2	scpe3	scpe 4	scpe5
mem_save	GSEMO	1.62 ± 0.15	1.62 ± 0.13	1.39 ± 0.2	1.1 ± 0.3	1.91 ± 0.15
mem_save	bounded GCAIS	**4.23 ± 0.01**	**4.54 ± 0.01**	**4.68 ± 0.02**	**4.31 ± 0.02**	**4.54 ± 0.01**
speed_up	GSEMO	**2.56 ± 0.15**	**2.22 ± 0.15**	1.69 ± 0.28	1.26 ± 0.41	**3.37 ± 0.15**
speed_up	bounded GCAIS	1.73 ± 0.18	1.36 ± 0.24	**2.56 ± 0.14**	**2.04 ± 0.2**	1.87 ± 0.21
approx rate	GSEMO	1.46 ± 0.16	1.53 ± 0.1	1.49 ± 0.12	1.43 ± 0.14	1.53 ± 0.1
approx rate	bounded GCAIS	**1.06 ± 0.04**	**1.01 ± 0.03**	**1.09 ± 0.03**	**1.04 ± 0.05**	**1.08 ± 0.04**
approx rate	GCAIS_BASE	1.11 ± 0.04	1.08 ± 0.06	1.11 ± 0.05	1.07 ± 0.05	1.14 ± 0.06

We displayed the experimental results in Table 7. Once more the population boundaries for GCAIS lead to a cut down in terms of memory and runtime. Further, they reveal that our adapted version of GCAIS did not lose its capability to find close to optimal or optimal solutions on these more theoretical instances. In all cases our version was even superior to the base variant. However, in three out of five cases GSEMO was once more faster than our bounded version as it once more converged towards a suboptimal solution.

We verified our observations about the bounded GCAIS' superiority in terms of memory usage and approximation quality using one-sided Wilcoxon signed-rank tests. The p-values were below 0.05 which we regard as significant.

Further, on these datasets the population of the base variant of GCAIS does not grow as much as during our evaluation of the industrial datasets. This explains why the memory savings are lower. The smaller populations thus lead to a smaller runtime which unfolds in smaller speed ups for the other algorithms. The reason for the smaller populations is that GCAIS detects dominating solutions more easily, which keep the population in bounds. Hence we think that the problem structure of test specifications differs from these more theoretical MSCP instances. This can be seen as a sign that GCAIS benefits from our algorithmic changes even if the MSCP instance does not correspond to a test specification.

3.6 Chapter Summary

Within this chapter we tried to answer the question if we are capable of determining redundant test cases solely based on the test specification. We defined redundancy based on the requirements that the tests cover. Hence we intended to find a minimal subset of test cases that cover all requirements. This is in fact a well studied problem throughout mathematics and computer science which is known as the Minimum Set Cover Problem.

The MSCP is a NP-hard optimization problem. We first discussed the mathematical state of the art and presented major results from complexity theory which underline the hardness of the problem. Unless $P = NP$, there cannot be an algorithm with polynomial runtime that has a worst case approximation ratio which is better than logarithmic in the problem size.

The limited possible worst case approximation capabilities have motivated computer scientists to design a variety of metaheuristics that perform well on average. We introduced several of these methods. We further showed that some of them have runtimes which are not polynomial in the problem size but in dynamic algorithmic hyperparameters. In our experiments we pointed out that this might lead to high runtimes (possibly even exponential ones). On the other hand the heuristics produced high quality solutions. This is the case for a class of metaheuristics called GCAIS.

We improved the algorithmic structure of GCAIS by lowering its theoretical runtime. Further, we introduced a population boundary based on an equivalence relation. Each equivalence class has a fixed capacity.

We evaluated our GCAIS variant empirically and observed that we not only cut down runtime and memory usage but we were also capable of improving the approximation quality. Thereby we relied on traditional MSCP instances as well as test specifications from BSH which indicates that our changes are of interest for general MSCP instances. On the test instances we could determine over thirty percent redundant test cases. Hence the approach is worth considering when a proper test specification is available. If the specification is too coarse grained (in terms of the requirements that cannot be covered by a single test), this may lead to the removal of necessary tests. Thus we recommend to use this method combined with a test engineer who additionally examines the proposed test cases.

4 Test Selection for semi-automated Testing

Test cases which are executed in an automated manner are a piece of software. Experienced developers are capable of implementing vast numbers of tests. For higher test levels such as system tests, usually only a limited number of hardware prototypes are available. Testing is often a time crucial task and hence only a limited time budget is available. Thus a set of crucial test cases is to be determined which takes the available execution time into account. This problem is known as *test case selection* [41].

There is no fixed mathematical definition of what *crucial* means in this context. This is rather application specific. The survey of Yoo and Harman [41] documents various objectives such as code coverage, requirements coverage or failure revealing capabilities. Sometimes several criteria have to be considered simultaneously and hence test selection becomes a multi-objective optimization problem.

The problem is often handled in a generic way as out of the box multi-objective optimizers such as NSGA-II are used for the task [40, 41, 102]. Within this chapter we examine if *Germinal Center Artifical Immune Systems* (GCAIS) are also fit for this task. We are motivated to use it as several of the testing objectives such as all coverage metrics are variants of the set covering problem on which GCAIS performs well (see chapter 3).

One of the internal test departments of BSH needed a solution to select test cases based on requirements coverage. Hence we start the chapter with an analysis of a bi-objective version of the test selection problem (requirements coverage and execution time). We thereby also present a technique to reuse previous results (in order to further reduce the runtime). Later we extend the list of objectives by introducing a failure probability goal in order to improve the failure revealing capabilities of the resulting test suite.

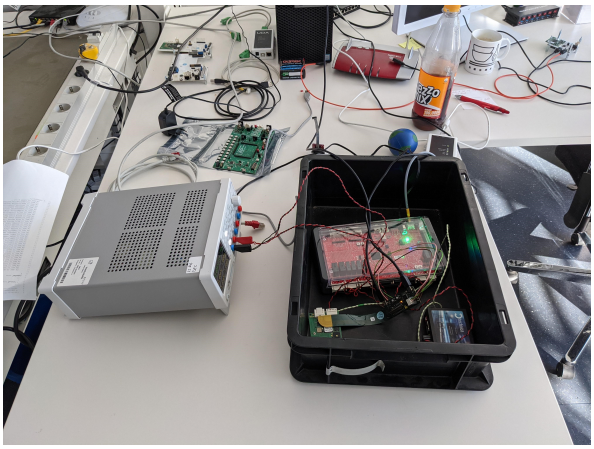


Figure 9: Example of a local test bed in BSH.

The aforementioned test engineers prefer to work in a semi-automated way. Their *device under test* (DUT) is usually directly located at their workplace (e. g. as in Figure 9). The corresponding test cases are available as source code using some test framework. They execute tests locally and simultaneously observe the physical DUT if a strange behaviour occurs which is not checked by the current test case.

These test engineers also do not always execute all available test cases. They usually perform some sort of preselection of the tests based on their expert knowledge or on recommendations from their fellow software engineer colleagues. This reduced test suite might still be large and some aid in further confining the test suite is necessary.

In our later experiments we leave out this manual preselection in order to not introduce a form of bias into our experiments. Hence we consider the entire set of available tests to search for.

4.1 Requirements based Selection

The first variant of test selection focuses on the objectives requirements coverage and execution time. We try to find a test suite of maximal coverage which has a limited execution time. Formally we can describe it as follows:

$$\begin{aligned} max\ |\bigcup_{T\in\mathcal{TS}} T| \\ s.t.\sum_{T\in\mathcal{TS}} d(T) \leq C \\ \mathcal{TS} \subseteq \{T_1, T_2, ..., T_n\} \\ T_i \subseteq \{1, 2, ..., m\} \end{aligned} \tag{11}$$

where $d(T)$ is the approximated duration of a test case T and C is an execution time budget.

This test selection variation is closely related to the *weighted minimum set cover problem* (WMSP) where a set cover of minimal cost is desired. The MSCP is a special case of the WMSP as each set has a cost of one. The WMSP differs from this test selection version as WMSP has the constraint that each element (requirement) must be covered and the cover's total cost must be minimized.

The theoretical results regarding MSCP that we presented in chapter 3.1 are also valid for WMSCP (WMSCP is not in APX and the best achievable worst case approximation ratio is logarithmic in the problem size for polynomial time algorithms [44]). It is straightforward to see that this test case selection variant is also NP-hard[4].

Further, the metaheuristics and algorithms presented in chapter 3.1 can also be used for WMSCP. Most of the metaheuristics that we discussed have also been benchmarked on WMSCP instances [77, 78, 87] or have been examined mathematically regarding the problem [79].

4.1.1 Adjustments of GCAIS

We use GCAIS to solve the following auxiliary problem:

$$max\ |\bigcup_{T\in\mathcal{TS}} T|,\ \ min \sum_{T\in\mathcal{TS}} d(T) \tag{12}$$

Hence the budget constraint is dropped and the test suite's execution time becomes another objective which is to be minimized.

After GCAIS has been applied to the aforementioned auxiliary problem, it has produced a population of solutions. All of these solutions are feasible for the problem described in equation 12, but some of them may be infeasible for the actual test selection problem at hand (there might be solutions which exceed the budget C).

We can use a table-based approach as introduced in chapter 3.5 to find feasible solutions. The table differs here as entry indexes correspond to the requirements covered and the attribute *exec* corresponds to the approximated execution time of the entry's solutions. We greedily search the entries for the solutions of highest coverage whose execution time is within the budget. The usage of the table leads to a runtime of $O(m)$ instead of $O(|P|)$.

The runtime consideration is also the reason why we changed the table indexes from the cost to the requirements covered. For the MSCP the cost axis (number of used sets) is purely discrete and only $n + 1$ values are possible. However, the tests' execution times can highly vary and are generally real numbers. This leads to combinatorially more possible execution times of individual test suites which would in terms lead to a higher search time.

We introduce another innovation for GCAIS by tweaking the population initialization. In the original variant the starting population solely consists out of the zero vector $\mathbf{0}$. We initialize the

[4] A solution of the decision problem connected to equation 11 can clearly be verified in polynomial time. If there would be an polynomial time algorithm that produces an optimal solution, we could combine it with a binary search. The binary search can be used to find the minimal time budget C where the optimal algorithm proposes a solution which covers all requirements. This solution would also be optimal for WMSCP.

Algorithm 9: Greedy search for a solution that is feasible for test case selection problem described in 11.

```
input : table, time budget C
for k in {m, ...0} do
    if table[k].exec ≤ C then
        return of a solution of table[k]
end
```

population randomly for different time budgets. The intention of the heuristic is that the Pareto-frontier produced by GCAIS has elements for a variety of time budgets and requirement coverages. The original variant might converge towards a Pareto-frontier of high requirements coverage (to produce feasible MSCP solutions) and thus could be unable to provide solutions for the full bandwidth of time budgets.

Algorithm 10: Random solution creation.

```
input : time constraint c
retries = 0
sol = empty solution
cost = 0
while retries < 5 do
    T = random unused test
    if cost +d(T) < c then
        add T to sol
        cost = cost + d(T)
        retries = 0
    else
        retries = retries + 1
    end
end
```

We describe the routine to create a novel solution in Algorithm 10. Whenever we intend to create a solution whose execution time does not exceed a predefined limit c, we draw a random test case which is not yet a part of the solution under construction. When we fail to add a new test case several times, the method terminates and the solution is returned (here 5 retries).

We perform an equidistant separation of the time axis to choose the time budgets for which we want to generate the initial solutions. For our later experiments we use a mesh of 5 percent. Hence we create solutions with a maximum execution time of 0, 5,..., 100 percent of the total execution time of all tests.

4.1.2 Experimental Setup

We rely again on datasets from BSH Hausgeräte GmbH. We reuse the Fridge-1 and Fridge-2 dataset. Further we introduce another fridge dataset (Fridge-3). We assign them the execution times of historical tests in order to generate a realistic setting.

We once more rotate the algorithms against which we compare GCAIS using our population initialization approach. In order to measure its effects we employ GCAIS with the standard population initialization. Additionally we consider the following methods:

- *Random selection* : A common practice in testing is to choose a test suite at random [103]. We choose as many tests randomly until the given time budget is exhausted (corresponds to Algorithm 10). Thus we can also examine how much improvement is due to GCAIS.

- *SEMO* : This is a variant of GSEMO which basically only uses one population. Further information can be found in the book of Neumann and Witt [97, p.36]. The method is parameter free.
- *SEIP* : As mentioned earlier this metaheuristic can also be used for weighted set covering problems [79] and thus also for this test selection variation.

We reuse the population boundary of 200 from our MSCP experiments for GCAIS. We also reuse the OptiPlex XE3 computer.

We give each bio-inspired algorithm a total time budget of 5 minutes. If the quality of the Pareto-frontier does not change over 1000 iterations, we interpret that as convergence.

The source code / data that corresponds to these experiments is available here:

`https://github.com/LagLukas/adaptiveTestSelection`

4.1.3 Evaluation

We evaluated the time budgets 0.01, 0.02,..., 1 of the total execution time (of all tests) for each algorithm and dataset. We visualized the results in Figure 10. If an algorithm has a value of 0 for a time budget, it was unable to produce a solution for the time budget.

Within these experiments we also examine the effect of our random population initialisation. If the vanilla variant (using only the zero vector) is employed then this leads to a rather poor Pareto-frontier. For a very low budget no solutions can be offered. However, with our initialization we can overcome this problem. Furthermore, we can observe that these low-budget solutions offer a higher coverage than the purely random created ones. Hence GCAIS is able to refine these low budget solutions. For higher time budgets both forms of population initialization achieve equivalent results.

Throughout all three datasets both GCAIS variants achieve full coverage with the lowest time budget (compared to the other three methods). Thus, from a pure WMSCP perspective, these are the best choice. The methods also reach high coverage levels fast. Further, both have a rather low variance compared to other approaches.

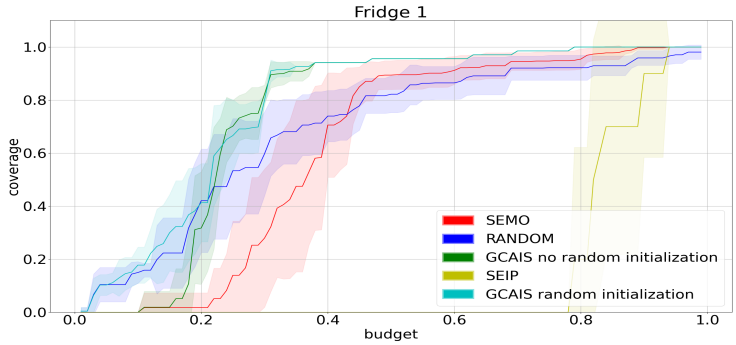
The other two evolutionary algorithms have varying performance on the datasets. For example, SEIP performs well on the Fridge-3 data, but poorly on the other two. Often SEIP and SEMO cannot offer solutions in their population for lower time budgets (e. g. Figure 10 (b)).

The pure random selection becomes more and more inferior if the available time budget is rising. There is a striking performance difference between both GCAIS variants and the random selection if the time budget rises above 0.2. The random selection is, in most cases, better than SEMO on all three datasets. Similar effects can be observed for the first two datasets and SEIP.

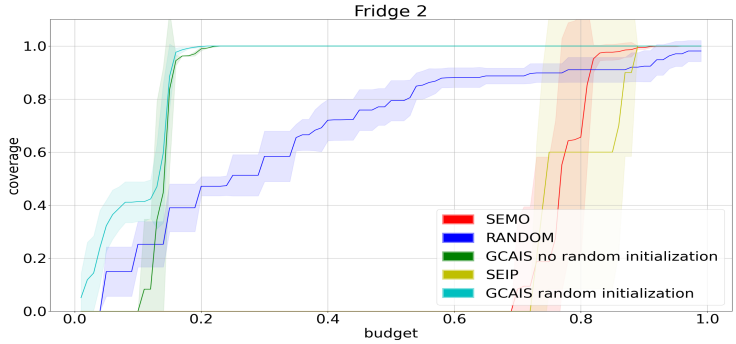
Our visual evaluation has led to the hypothesis that GCAIS using our population initialization is in most cases the best choice for ATCS on all three datasets. However, we deem a pure visual examination as insufficient. Thus we perform additional statistical tests. We decided to go for one-sided paired Wilcoxon rank tests. We test the null hypothesis "algorithm x performs better than GCAIS using Algorithm 10 on dataset y" for each considered algorithm x and dataset y. Thus the alternative hypothesis is that GCAIS using our random initialization is better. We summarized the p-values in Table 8. They are in all but one case significant. The tests confirm that our GCAIS variant is better than SEIP, SEMO and the pure random selection. Further, we are also better than the vanilla variant in 2 out of 3 cases. Even though we cannot reject the null hypothesis on the Fridge-3 dataset this does not imply that the null hypothesis can be confirmed. It is worth mentioning that we are close to being significant since the p-value is only 0.058.

Table 8: P-values for one-sided paired Wilcoxon tests which compare GCAIS using our population initialization with the vanilla variant, SEIP, SEMO and the random selection on all three datasets. Significant ones are marked bold.

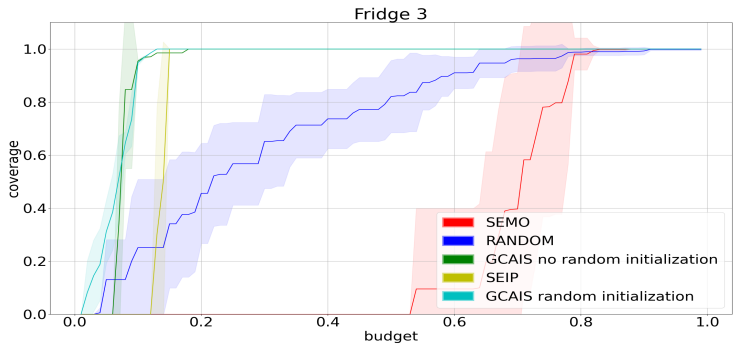
	SEIP	SEMO	RANDOM	vanilla GCAIS
Fridge 1	**0.0**	**0.0**	**0.005**	**0.0**
Fridge 2	**0.0**	**0.0**	**0.0**	**0.001**
Fridge 3	**0.001**	**0.001**	**0.001**	0.058



(a)



(b)



(c)

Figure 10: Mean achieved coverages $\pm\sigma$ for varying time budgets. The budgets are relative to the execution time of all available tests.

4.1.4 Incorporating Immune Memory

The immune system has another crucial property: it remembers past diseases in order to be prepared for upcoming similar diseases. This attribute has been translated to an optimization mechanism. Joshi et al. [104] injected one of the best found solutions from a previous, related problem to the new one. Thereby they converged faster to high quality solutions. Joshi et al. [104] evaluated the performance of the memory approach on MSCP instances which gives rise to the idea that these positive results might also be valid here. This might be of use as requirements may change throughout a projects lifetime [105] and so might the tests [27].

We simulate the change of requirements and tests. We drop requirements and test cases uniformly at random but proportional to the problem size. We follow the same approach to simulate new test cases and requirements. For example if there are currently m requirements and the drop proportion is ℵ then we draw a random integer ℶ from $[0, ℵm]$ and drop ℶ randomly chosen requirements from the available ones. We set the proportion for adding new tests / requirements to 0.05 and for dropping them to 0.025. Thus we emulate a test suite that is more and more growing. We assign newly created requirements uniformly at random to the new test cases. Further, an old requirement is added to the existing test with a probability of 0.05 (to simulate requirement refinements). We gathered historical test case runtimes which we used as bases for the simulated test cases' execution times. We simply draw a random past execution time uniformly at random. For each dataset we create a time-series of length 40.

In chapter 4.1.3 we observed that the random initialization can lead to superior solutions. Thus we examine the effects of memory on GCAIS using the initialization described in Algorithm 10.

First we intend to investigate if the immune memory improves the runtime. Thereby we once more rely on the speed up by computing the quotient of the runtime of the method without and with memory. We display the speed ups in Table 9. The usage of immune memory cuts down between 10 and 25 percent of the execution time.

Table 9: Averaged measured speed ups and relative population size $\pm\sigma$.

	Fridge 1	Fridge 2	Fridge 3
speed up	1.25 ± 0.08	1.75 ± 0.19	1.27 ± 0.14
realtive population size	0.79 ± 0.14	0.67 ± 0.08	0.75 ± 0.17

Table 9 additionally contains the *relative* population sizes per iteration. We compare the population size of the base variant with our memory approach. Hence a value lower than 1 indicates that the memory variant has a smaller population size and a value higher than 1 tells us that the population of the memory variant is larger. We can observe that the memory variant's population is in all three scenarios smaller. This explains why our improved algorithm is faster since the population size has a major impact on GCAIS' runtime.

We also examined if the usage of memory worsens the quality of the found solutions. We evaluated the possible relative time budgets 10, 20,..., 100 percent of the total execution time. We employed once more a Wilcoxon test to investigate if the immune memory worsens the approximation. We could observe a p-value below 0.05 and thereby we infer that this is not the case.

We additionally compared memory variant of GCAIS with SEIP, SEMO and the random selection on these simulated time series. There we made observations and conclusions that are highly similar to the ones discussed in chapter 4.1.3 (GCAIS produces the statistically superior solutions, SEIP / SEMO fail to produce feasible solutions for a variety of time budgets). We discuss these results in detail in [9].

4.2 Incorporating a Failure Objective

Our initial variant solely focused on requirements coverage and execution time. Thereby we could create test suites that examine a variety of features. However, one of the main tasks of testing is to reveal errors. Within this subchapter we examine if we can model a objective which helps us revealing errors.

We rely on the testing history which is generally available as testing is often done repetitively. We use the historical data to model the failure probability for an individual test case T. We use a simplistic measure by relying on the empirical probability. Hence we approximate the failure revealing capability of a test as follows:

$$P(T \text{ fails}) = \frac{f_T}{N_T} \tag{13}$$

where f_T is the number of executions where T failed and N_T is the total number of executions of the test.

We use this probability measure to introduce an additional optimization objective for a test suite $\mathcal{TS}$:

$$FP(\mathcal{TS}) = \frac{\sum_{T \in \mathcal{TS}} P(T \text{ fails})}{|\mathcal{TS}|} \in [0, 1] \tag{14}$$

which measures the mean failure probability of the test suite.

The extended optimization problem has the following form:

$$\begin{aligned} \max |\bigcup_{T \in \mathcal{TS}} T|, \max FP(\mathcal{TS}) \\ s.t. \sum_{T \in \mathcal{TS}} d(T) \leq C \\ \mathcal{TS} \subseteq \{T_1, T_2, ..., T_n\} \\ T_i \subseteq \{1, 2, ..., m\} \end{aligned} \tag{15}$$

We once more intend to solve the problem indirectly by defining an auxiliary problem:

$$\max |\bigcup_{T \in \mathcal{TS}} T|, \max FP(\mathcal{TS}), \min \sum_{T \in \mathcal{TS}} d(T) \tag{16}$$

Some of the objectives are in opposition to each other. For example if more tests are added to enlarge the coverage then the execution time is also increased. Furthermore, if the coverage and average fault probability should be enlarged simultaneously, an optimization algorithm has to add tests with a high likelihood to fail that also verify previously uncovered requirements. If only tests are added that nearly never fail but cover a lot of requirements, then the average fault probability will decline. Further, if a test suite solely focuses on tests with a high likelihood to fail but only tests a few features then this will lead to a rather small coverage.

4.2.1 Extending the Random Initialization

We encountered issues with the failure revealing capabilities of the GCAIS using the random population initialization described in Algorithm 10. We additionally initialize the population along a *greedy* axis and not only a *random* axis. We coin this initialization technique *multi-axis initialization*.

The greedy initialization starts with an empty solution (no test case used). Then it iteratively adds the next test case with the highest failure probability, makes a copy of the solution and injects it to the population. It stops adding tests when there is no test left that has a positive failure probability.

We keep the initialization of the random axis as in chapter 4.1.1. Thus we use 5 retries and a mesh of five percent. The extended initialization is described in Algorithm 11.

4.2.2 Experimental Setup

We once more use datasets from BSH Hausgeräte GmbH. We rely on two oven projects and a dishwasher project. All three projects are from a system test level. We have insight on several test runs which are also known as *test sessions*. The three datasets also differ in the number of test cases, amount of failures, as well as the number of executions (verdicts). We give an overview in Table 10. It is worth mentioning that the number of test cases changed throughout the project's lifetime. Thus the verdicts are not a multiple of the test cases or the test sessions.

Table 10: Examined datasets.

	Oven 1	Oven 2	Dishwasher
test sessions	39	36	45
test cases	486	477	1499
verdicts	22,350	17,349	186,195
failed	10.94%	3.41%	3.44%

Algorithm 11: Multi-axis initialization.

input : test cases, n, fault probabilities, time budgets c_k, max retries r
output: a population of solutions

```
P = {}
// create random solutions
for each time budget c_k do
    x = 0
    count = 0
    while count ≤ r and execution time of x ≤ c_k do
        draw test case i
        if time left to add i then
            add i to x
            count = 0
        else
            count += 1
        end
    end
    insert x to P
end
x = 0
// create greedy solutions
while ∃i ∈ {0, 1, ..., n − 1} : x[i] = 0 ∧ P(i-th test case fails) > 0 do
    index = arg max_{i∈{0,1,...n−1}} P(i-th test case fails)
    x[index] = 1
    insert a copy of x to P
end
return P
```

Table 11: P-values for one-sided Wilcoxon tests. The columns represent a dataset and the rows an algorithm to compare with. The entry of row x and column y is the p-value of the null hypothesis "the solutions of algorithm x find more errors than GCAIS's solutions on dataset y".

	Oven 1	Oven 2	Dishwasher
NSGA-II	0.0	$6.26e-274$	$1.70e-10$
Random	$1.19e-110$	0.0	$8.98e-25$
SEMO	$1.18e-81$	$1.41e-262$	0.04

We simulate the real use case based on the dataset at hand. We act as if we would have to choose a test suite for the i-th session for a fixed time budget C. Afterwards we have to choose a solution from the Pareto-frontier whose execution time is below C. For the requirements-based selection we did a greedy search according to the coverage. Here we consider both requirements coverage and the mean failure probability. Formally we look for the test suite $\mathcal{TS}$ which maximises the following term:

$$FP(\mathcal{TS}) + \frac{|\bigcup_{T \in \mathcal{TS}} T|}{m} \tag{17}$$

The division of the requirements coverage by m is used to normalize the values. This selection criterion has the side-effect that we can reuse the table approach for GCAIS (thereby we give the method only information about the expected execution time and the value of the selection criterion described in equation 17).

Multi-objective optimization heuristics for test selection are usually only compared with pure random selections [40, 102]. However, we deem this as too little and thereby also rely on NSGA-II which was successfully employed for test selection for embedded systems [40,102]. We use the variant of Lachmann et al. [40] (including the hyperparameters) as they examined it on larger datasets. We additionally once more rely on a pure random selection as well as on SEMO.

For these experiments we focus on how well the produced solutions find errors, how early, how many and how they are distributed among the requirements.

We regard a non changing Pareto-frontier over 100 iterations as convergence. We also reuse the Dell OptiPlex XE3 computer from our earlier experiments. The published source code and data is available here:

`https://github.com/LagLukas/moa_testing`

4.2.3 Failure Revealing Capabilities

The goal of testing is to find errors and not to prove their absence. Hence we examined how well the found test suites perform in this task. We compared the solutions of SEMO, NSGA-II and the random selection on every considered dataset with GCAIS' solutions. In order to compare the found test suites we rely on statistical tests. We employ a series of one-sided Wilcoxon tests and use a significance level of 0.05. We use the statistical tests to evaluate the null hypothesis of the form "the solutions of algorithm x find more errors than GCAIS's solutions on dataset y". We display the corresponding p-values in Table 11. All of them are below the significance level and hence we reject every null hypothesis and infer that the computed test suites of GCAIS are better in finding errors.

We switch our focus to analysing how high the time budget needs to be until the first error is detected. We display the minimum, maximum, quartiles and average execution times that were necessary to find a first fault in Table 12. There we can see that, in three experimental settings, GCAIS produces solutions that have the lowest time budget on average.

We can also see a large range for the time budgets needed to detect the first error. This variety of values may disturb the mean value. Thus we decided to examine the median for each algorithm and dataset. There we can observe that GCAIS has the lowest median. SEMO and the pure random selection lead to similar results on the Oven 1 dataset (regarding the median). We also added the first and third quartiles into Table 12 in order to describe the time budget distribution more accurately. Once more we can make the observation of rather low values for GCAIS. Furthermore the necessary time budget seems to be rather stable as in many cases the first quartile, the median, and the third

Table 12: Overview of the time budget needed to reveal the first error. The table contains the mean values $\pm\ \sigma$, medians, maxima and minima. The best values are marked bold.

		Oven 1	Oven 2	Dishwasher
GCAIS	mean	**0.145±0.28**	**0.296±0.39**	**0.024±0.06**
	third quartile	**0.05**	**0.36**	**0.01**
	median	**0.05**	**0.05**	**0.01**
	first quartile	0.05	**0.04**	**0.01**
	max	0.65	0.64	0.3
	min	0.02	0.02	**0.01**
NSGA-II	mean	0.316 ± 0.36	0.415 ± 0.45	0.239 ± 0.25
	third quartile	0.48	1.0	0.55
	median	0.1	0.1	0.05
	first quartile	0.1	0.05	0.05
	max	0.69	**0.41**	0.75
	min	0.02	**0.01**	**0.01**
RANDOM	mean	0.182 ± 0.27	0.313 ± 0.38	0.056 ± 0.02
	third quartile	0.19	0.41	0.05
	median	**0.05**	0.1	0.05
	first quartile	0.05	0.05	0.05
	max	**0.63**	0.74	**0.15**
	min	0.04	**0.01**	0.05
SEMO	mean	0.222 ± 0.32	0.35 ± 0.37	0.062 ± 0.04
	third quartile	0.35	0.48	0.05
	median	**0.05**	0.185	0.05
	first quartile	**0.04**	0.08	0.05
	max	0.8	0.59	0.25
	min	**0.01**	**0.01**	**0.01**

Table 13: P-values for one-sided Wilcoxon tests. The columns represent a dataset and the rows an algorithm to compare with. The entry of row x and column y is the p-value of the null hypothesis "algorithm x's solutions find the first error earlier than GCAIS's solutions on dataset y".

	Oven 1	Oven 2	Dishwasher
NSGA-II	$6.52e-46$	$5.95e-13$	$6.46e-06$
Random	$7.56e-11$	$9.89e-06$	$2.28e-17$
SEMO	$1.03e-60$	$5.80e-55$	$2.14e-50$

Table 14: p-values for one-sided Wilcoxon tests. The columns represent a dataset and the rows an algorithm to compare with. The entry of row x and column y is the p-value of the null hypothesis "algorithm x's solutions detect more broken features than GCAIS's solutions on dataset y".

	Oven 1	Oven 2	Dishwasher
NSGA-II	$1.38e-309$	$3.38e-12$	0.1751
Random	0.0	$9.50e-200$	$1.09e-109$
SEMO	$5.31e-245$	$5.02e-85$	$1.399e-43$

quartile are the same. For the other examined methods this is not the case and their distributions are more diverse. We consider these close quartiles an indicator for the robustness of the solutions produced by GCAIS.

We deem a discussion solely based on central tendencies such as mean values or medians as insufficient and once more decided to use statistical tests to take a deeper look at our results. We reuse a significance level of 0.05 and rely on one-sided Wilcoxon tests. We examine the null hypothesis "algorithm x's solutions need a lower time budget to find a first error than GCAIS's solutions on dataset y" for each considered algorithm and dataset. We show the corresponding p-values in Table 13. The p-values are below our significance level and we reject all null hypotheses and accept the alternative hypothesis. Thus we infer that solutions found by GCAIS are capable of finding the first error earlier than NSGA-II, a pure random selection, and SEMO. It is worth mentioning that the comparably good results of GCAIS are not only due to the initialization. An additional comparison between our immune system and the initialization showed that GCAIS also performs better in that scenario.

We additionally examined the objective failure probability $F(\cdot)$ and there we could see that in all but one combination the GCAIS approach is significantly better than the other approaches. This might be seen as an indicator for why GCAIS has been found to be superior in detecting errors on the considered datasets.

4.2.4 Detection of Broken Features

In the previous experiments we solely focused on the detection of failures. It lacks the link to the requirements covered by the tests. For example if two features are broken and one test suite leads to a lot of failed tests exclusively for one feature and another one has a failed test for each broken feature then the first test suite would be better even though it would have failed to identify all broken features. Thus within this evaluation we focus on the evaluation of how well the considered methods recognize broken features.

We examine hypotheses of the form "the solutions of algorithm x detect more broken features than GCAIS's solutions on dataset y" using a series of one-sided Wilcoxon tests and a significance level of 0.05. The p-values are displayed in Table 14. We can reject the null hypothesis in 8 out of 9 cases and accept the alternative hypothesis (GCAIS detects more broken features). However, on our dishwasher dataset the p-value for NSGA-II is rather low (about 0.1751) but still not significant which means that we cannot reject the null hypothesis. Thus we decided to perform an additional two-sided Wilcoxon test to check if on average both methods perform equally well on this dataset and this statistical test indicated that this is the case.

We decided to investigate the difference between GCAIS and NSGA-II on the dishwasher dataset more deeply. We plotted the average difference of the percentage of detected broken features in Figure 11. The x-axis displays the test session index, the y-axis the relative time budget and the z-axis the difference in detected broken features. We can see a clear superiority of GCAIS for high budgets and very low budgets (less than 10 percent). NSGA-II is ahead for time budgets of about 20 percent for very early test sessions. For succeeding test sessions this gap is tightening. After session index 5 GCAIS becomes significantly superior. Hence if more test outcomes are available then GCAIS's performance increases on this dataset.

Our previous evaluation exclusively focused on the question if GCAIS performs better than the other methods. We also intend to give a total overview instead of only this relative consideration.

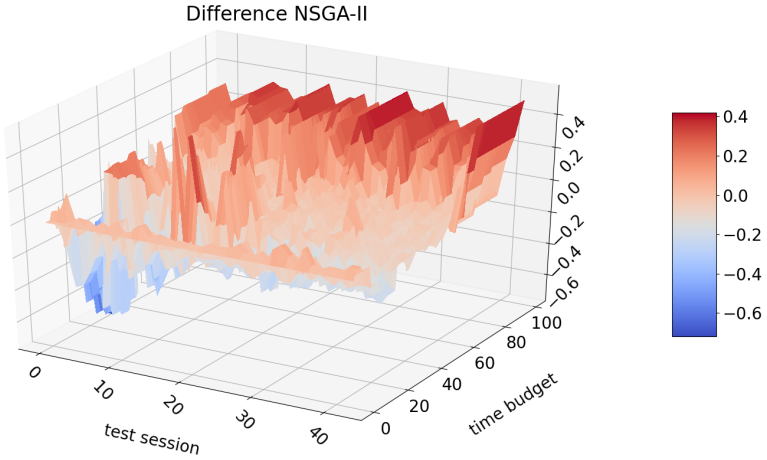
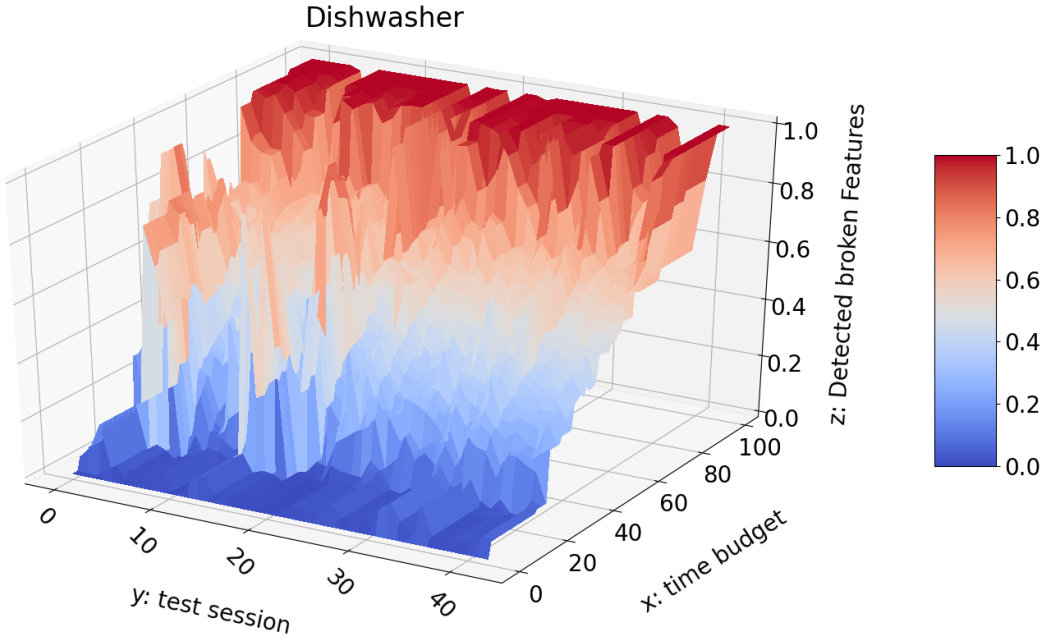
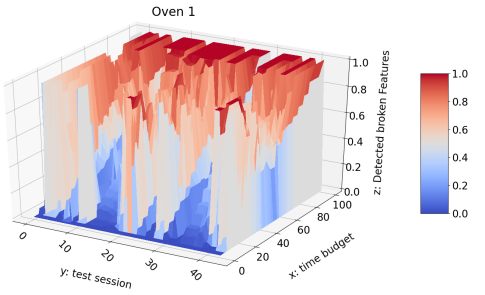


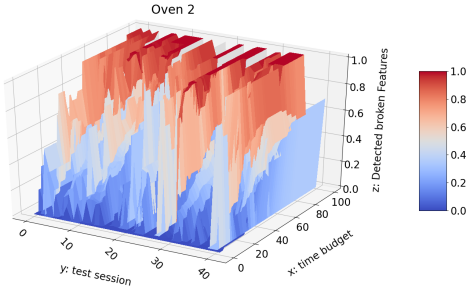
Figure 11: Difference in the percentage of found broken features between GCAIS and NSGA-II on the dishwasher dataset. A positive value indicates that GCAIS found more broken features (a negative one indicates the opposite).



(a) Percentage of found broken features using GCAIS on the dishwasher dataset.



(b) Percentage of found broken features using GCAIS on the oven 1 dataset



(c) Percentage of found broken features using GCAIS on the oven 2 dataset.

Figure 12: Percentages of found broken features across the considered datasets.

Thus we also visualized the raw numbers. Figure 12 a) shows the performance on the dishwasher dataset. Here the performance is generally slowly rising with an increasing time budget. There are a few outliers that became apparent where GCAIS finds even more broken functionalities, especially at the start of testing. In this phase there are the most faults in our dataset which explains this observation. Figures 12 b) and c) display the performance on the two oven datasets. In this setting generally more errors occurred overall and also in a higher number during later stages of testing. This explains why there are several test sessions where we detected high numbers of broken requirements. The succeeding test sessions often show less issues found since between the two sessions the previously found oven software bugs have been fixed. It is worth mentioning that the two oven projects share some generic components hence they show some similarity in their error behaviour.

4.3 Chapter Summary

Within this chapter we moved from test suite minimisation to test case selection. We showed that both problems are partially linked to set covering problems. We composed critical test suites that fulfill up to three different criteria: small execution time, requirements coverage and failure revealing capabilities.

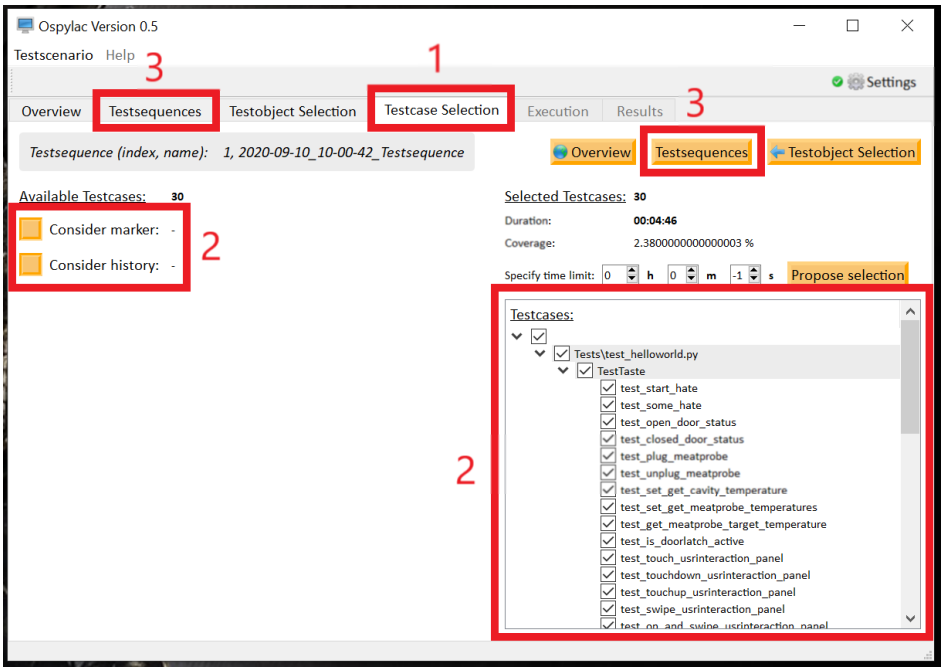


Figure 13: Test selection GUI *ospylac*.

We compared our approach with a variety of generic search heuristics and metaheuristics that have been successfully applied to test selection. There our approach could prevail and provide solutions for a variety of settings. This can be partially traced back to the population initialization techniques that we introduced. Thereby we showed that GCAIS is feasible for this use case.

The chapter mostly focused on algorithmic description of the approaches introduced and their experimental evaluation. However, this leaves one question open: How did we deliver the value generated to the customer? A student called Lukas Huber developed a *graphical user interface* (GUI) coined *ospylac* which also offers methods developed to the test engineers.

We provide an example screenshot of the GUI for one test project in Figure 13. It has a menu for test selection (1) and options for manually prefiltering the test cases at hand (2). One can consider specially marked test cases, tests that failed a certain number of times or can manually select / deselect test cases. The chosen test cases can be summarized as *test sequences* (3). Our GCAIS approach is used when a tester uses the *propose selection* option. Then GCAIS also takes the preselection of the

tester into account. Thereby the final product can be seen as a guided search which also has a positive effect on the runtime[5]. The GUI is currently in use in the oven test team of BSH.

[5]The preselection confines the search space which is beneficial for the algorithmic complexity.

5 Test Case Prioritization and Selection in CI

Continuous integration (CI) is a development practice which has its roots in *extreme programming* which is an agile software development process [106]. CI focuses on frequent builds of the software which is under development. In order to achieve this the source code is maintained at a common code basis and can be assembled in an automated way using a build server (e. g. Jenkins [58]). One of the main goals is to integrate the works of individual programmers frequently (usually at least daily [107]). Thereby lengthy integrations of large chunks of code can be avoided. This also reduces the number of bugs that are introduced to the code basis from integration to integration. Hence CI aims at improving software quality.

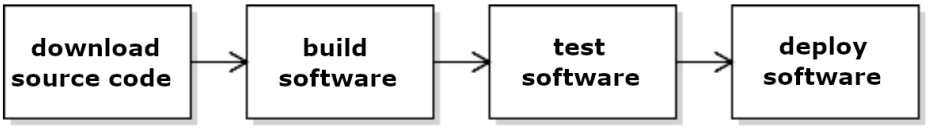


Figure 14: Simplified CI pipeline.

A software build starts with the download of the source code. Afterwards the software itself is built which is followed by a testing stage. If the stage is passed then the software may be deployed. We visualize these steps in Figure 14. These steps form a so called pipeline each run of which is called a CI cyle. It is worth mentioning that the pipeline usually also contains detailed reporting functionalities in order to give an insight about the software at hand. Further, this enables non-software developers to get an overview on the project.

Here we mainly focus on the testing stage. Within it, tests from several test levels may run (e. g. unit tests, integration tests or even system tests). CI emphasizes speed and hence a set of crucial tests is to be executed in order to get a quick insight about bugs that might be in the software itself. This also means that there is a limited search time to determine these tests.

A novel approach is to employ *reinforcement learning* (RL) to compose these test cases in a CI setting. Spieker et al. [6] were the first to achieve this. They employed an artificial neural network to compute the test suites based on the testing history. This is done by prioritizing the tests according to their execution time and likelihood to fail. The tests are chosen according to the prioritization and the time budget. This straightforward approach has motivated major companies such as Netflix to use the approach in practice [108].

We examined if learning classifier systems such as XCSF can be used for this task as well. Hence we differ from Spieker et al. [6] as we use another machine learning technique. We also gradually improved and adapted the method to the task itself and this chapter gives insight how we achieved that.

5.1 Problem Description

During each CI cycle i there is a fixed time budget C available. For every test case T there is an estimated duration $d(T)$ known. The task is to assign a rank $\mathrm{rk}_i(T)$ to each test which corresponds to the test's priority. This is done at the beginning of each CI cycle. Note, that ranks are not unique and it is possible that two test cases get the same rank.

After the rank assignment step, a schedule is created which takes into account the ranks and the available time budget C as follows: 1) The available tests are first sorted by their ranks in descending order. 2) Then as many tests are scheduled from the beginning of the sorted list as long as the estimated overall duration does not exceed the budget C. 3) If it is not possible to schedule all tests having equal rank, the remaining time budget is filled up with randomly chosen tests of the same rank. 4) The resulting schedule is summarized in a list.

Let $l_i(T)$ be the index of a scheduled test T. The elements of the list determine the test suite $\mathcal{TS}_i$ to be executed during CI cycle i.

After the test suite has been executed, the results can be collected. Let $\mathcal{TS}_i^{\mathrm{f}}$ be the test cases of $\mathcal{TS}_i$ that failed. Furthermore, let $\mathcal{TS}_i^{\mathrm{t,f}}$ be the set of failed tests if all available tests would have been

executed. Thus the percentage of found failures p_i can be computed as follows:

$$p_i = \frac{\mathcal{TS}_i^{\mathrm{f}}}{\mathcal{TS}_i^{\mathrm{t,f}}} \tag{18}$$

A commonly used metric to evaluate the quality of a test prioritization is the *normalized average percentage of faults detected* (NAPFD) [109]:

$$\mathrm{NAPFD}(\mathcal{TS}_i) = p_i - \frac{\sum_{T \in \mathcal{TS}_i^{\mathrm{f}}} l_i(T)}{|\mathcal{TS}_i^{\mathrm{f}}| \cdot |\mathcal{TS}_i|} + \frac{p_i}{2 \cdot |\mathcal{TS}_i|} \tag{19}$$

Its values range from 0 to 1 and a high value is desired. Its advantage over p_i itself is that it takes the prioritization into account (by using l_i). If many passing test cases receive a high priority, i. e., have a low index l_i in the schedule list, then the NAPFD metric decreases. Vice versa, if the schedule contains a large number of failing tests with high priority (i.e., low indexes l_i), then this results in a higher NAPFD value.

Using the definition of NAPFD we are now able to define the *adaptive test case selection problem* (ATCS) as:

$$\begin{aligned} &\max && \mathrm{NAPFD}(\mathcal{TS}_i) \\ &\text{subject to} && \sum_{T \in \mathcal{TS}_i} d(T) \leq C \\ & && \mathcal{TS}_i \subseteq \mathcal{T}_i \end{aligned} \tag{20}$$

where $\mathcal{T}_i$ denotes the entire set of all available tests during cycle i. Thus the goal is to find an optimal subset $\mathcal{TS}_i$ of $\mathcal{T}_i$.

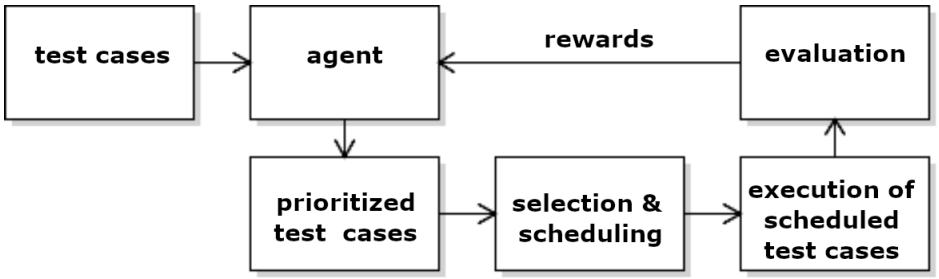


Figure 15: Workflow for solving ATCS using RL.

Similar to Spieker et al. [6], we intend to solve ATCS using RL which leads to the workflow described in Figure 15. The following paragraphs define three reward functions that Spieker et al. devised [6] as well as the state and action spaces for our proposed XCSF-based RL agent.

A reasonable idea is to try using the NAPFD metric as a reward function for an agent. However, in practice this would force us to always execute all tests (p_i is needed for its computation) which we explicitly want to avoid. Hence Spieker et al. [6] have proposed the following reward functions:

- *failure count reward*

$$r_i^{\mathrm{fc}}(T) = |\mathcal{TS}_i^{\mathrm{f}}| \tag{21}$$

- *test failure reward*

$$r_i^{\mathrm{tcf}}(T) = \begin{cases} 1 - v_i(T) & , T \in \mathcal{TS}_i \\ 0 & , \text{otherwise} \end{cases} \tag{22}$$

- *time ranked reward*

$$r_i^{\mathrm{trk}}(T) = |\mathcal{TS}_i^{\mathrm{f}}| - v_i(T) \cdot \sum_{\substack{t_k \in \mathcal{TS}_i^{\mathrm{f}}, \\ \mathrm{rk}(T) < \mathrm{rk}(t_k)}} 1 \tag{23}$$

where $v_i(T)$ denotes the binary verdict of a test case T during cycle i. A 1 indicates "test passed" or "test not executed due to time restriction". A 0 stands for "test failed". In the following, we denote the reward received at time t simply as $r(t)$.

The failure count reward gives each test case the same reward, namely, the number of failed tests of the current schedule. The test failure reward is more fine-grained as it rewards test cases individually based on their verdict (a reward of 1 if failed, 0 otherwise). The time ranked reward also takes the prioritization into account as it punishes a passed test by the number of failed tests having a lower priority; furthermore, passed tests with a low rank receive a relatively high reward.

The problem's state space S is defined as follows:

$$S \coloneqq [0, C] \times \{0, 1\}^k \times [0, 1] \tag{24}$$

A state (a test case) contains the *approximated duration* (a real number between 0 and C), the *testing history* (a binary vector of length k, each entry being the test's verdict of the respective CI cycle), and the *time of the last execution relative to the entire testing history* (a real number between 0 and 1). The hyperparameter k indicates how many previous verdicts of the current test case the agent is aware of. If there are not k test results available yet (e. g., at the start of training), the missing entries are filled with zeroes. States are vectors of dimension $k + 2$; we denote the state at time t by $s(t)$.

The action space A is $\mathbb{R}$ as each agent maps a test case described by the current state $s(t)$ to a scalar rank which reflects its priority. We define $a(t)$ as the action chosen at time t.

It is worth mentioning that the RL interpretation of ATCS differs from the traditional *temporal difference learning* (TD) scenario. In TD, sequences of states, actions and rewards are typically observed in the following order:

$$s(1), a(1), r(1), s(2), a(2), r(2), \ldots \tag{25}$$

However, for ATCS, all available tests are ranked (i. e., a state observed and an action executed) first one by one. After that, a corresponding schedule is executed based on whose outcome rewards are distributed (see Figure 2). Hence here we observe the following sequence:

$$s(1), a(1), s(2), a(2) \ldots, s(n), a(n), r(1), r(2) \ldots r(n) \tag{26}$$

In the ATCS RL problem, an episode corresponds to a single CI cycle with the terminal state being the last available test case of the cycle. Furthermore, the length of an episode as well as the states encountered are determined by the available tests and not by the agent's actions. It is also worth mentioning that the learning environment for the ATCS problem can be considered non-stationary, i. e., in general it does not stay constant throughout the episodes. This is due to the fact that, between two cycles, the software to be tested or the available tests may change.

5.2 XCSF Classifier System

Learning Classifier Systems (LCSs) comprise a family of evolutionary, rule-based machine learning algorithms. We focus on the XCSF classifier system [110] that is designed to be a general purpose function approximator. An XCSF consists of a population of rules, a locally acting learning mechanism for those, as well as a globally acting evolutionary algorithm that optimizes the rules' localities, i. e., the subspace of S each rule covers. A rule participates in the decision process if a certain set of rule-specific conditions is fulfilled by the given state. For example, a condition could be to assert if an attribute lies in an interval. One such rule is called a *classifier* cl. A classifier has several learning parameters which determine its influence in the system's collective decisions making process.

We use XCSF to approximate a state-value function $\mathrm{V}(\cdot)$ for the following policy π:

$$\pi(s) = \hat{\mathrm{V}}(s) \tag{27}$$

where $\hat{\mathrm{V}}(\cdot)$ is an approximation of $\mathrm{V}(\cdot)$. This policy follows a simple heuristic: If a test case (i. e., a state) has a high value, it should have a high priority. $\hat{\mathrm{V}}(\cdot)$ estimates the reward that will be received if that policy is applied.

Each classifier cl models a so called *prediction function* cl.$p(s)$. It is the classifier's local estimation of the value function $V(\cdot)$. We model cl.$p(s)$ as a linear function as proposed by Wilson [110]:

$$\text{cl}.p(s) = w_0 + \sum_{i=1}^{k+2} w_i \cdot s_i \tag{28}$$

where w_i are real-valued weights which are initialized randomly and $k+2$ denotes the state space's dimensionality as before.

A classifier tracks how often the rule has been applied which is called its *experience* which we denote as cl.experience. Further, it tracks an integer attribute called numerosity (cl.num) which indicates how many times an identical classifier has been created (identical in terms of conditions[6]). Additionally, it keeps record of its prediction accuracy relative to its local niche of overlapping rules which is called *fitness* cl.F. Another quality parameter of a classifier is the prediction error cl.ϵ which is the mean absolute error estimate of its prediction function.

Whenever XCSF tries to approximate the value function for a state $s(t)$, it searches the rule population for classifiers whose conditions are satisfied by $s(t)$. These classifiers are collected in a *match set* M. This process is known as *matching*. If the match set is too small, new classifiers that match the given point are created at random. This creation method is called *covering*. Based on the classifiers within M XCSF computes the estimation of the function at s (in our case the estimation of the value function):

$$\hat{\text{V}}(s) = \frac{\sum_{\text{cl}\in M} \text{cl}.p(s) \cdot \text{cl}.F}{\sum_{\text{cl}\in M} \text{cl}.F} \tag{29}$$

The approximation is coined *system prediction* in the LCS community and corresponds to the next action since we follow the policy described in equation 27.

XCSF uses a gradient–based update rule for the prediction functions cl.$p(s)$ which is known as *modified delta rule* [110]:

$$\Delta w_i = \frac{\eta}{\|\tilde{s}\|_2^2}(r - \text{cl}.p(s))\tilde{s}_i \tag{30}$$

where $\|\cdot\|_2$ denotes the Euclidean norm and η is the learning rate. The weights of cl.$p(\cdot)$ are updated by adding Δw_i. The vector $\tilde{s}$ is the same as the state vector s but extended with a leading 1 to fit the intercept weight w_0, i. e.,

$$\tilde{s} = (1, s_1, s_2, ..., s_{k+2})^T, \tag{31}$$

and r denotes the reward received for the chosen action in s. Hence we try to approximate r based on s using cl.$p(\cdot)$.

Note that in step t only the classifiers of the match set that corresponds to $s(t)$ are updated. The remaining learning parameters of the classifiers cl $\in M$ such as cl.F and cl.ϵ are updated as usually done in XCS.

Within this thesis we follow a *need to know* policy regarding the description of our modified XCSF. Thus we limit explanations to the parts that are necessary to understand how we solved the use case at hand. For a more fine-grained explanation we refer the reader to [32,110] for more details on XCSF.

XCSF periodically applies to its match set a *steady-state niche genetic algorithm* (GA)[7] that is used to optimize the classifiers' conditions, i. e. find an optimal partitioning of the state space S. The heuristic selects parent classifiers from the match set M and reproduces them by applying crossover and mutation on their conditions. Mutation introduces a form of random search into the optimization (to avoid the GA from getting stuck at a local optimum) whilst a crossover operator tries to create a set of new conditions based on the conditions of the two parents. Parental classifiers are selected by their fitness values cl.F via a fitness proportionate selection.

The population of XCSF has a fixed capacity. If the population contains too many classifiers after new classifiers have been introduced to it (e. g., by the GA or by the covering mechanism),

[6] For other LCS such as XCS it is also required that the classifiers propose the same action. However, XCSF has no real action, only a dummy one. Hence this constraint is satisfied by construction.

[7] A steady-state GA does not exchange the entire population after an iteration, but only a part. Further, niche means in that context that it only looks at classfiers that are responsible for a certain subspace of S.

XCSF deletes classifiers at random but proportional to their quality. We call this deletion mechanism *pruning*.

As detailed by the workflow described in Figure 15, we split the functionality of XCSF in a separate action selection (Algorithm 12) and batch update learning method (Algorithm 13). The action selection also saves the encountered match sets and states. At the end of the CI cycle the match sets faced throughout the episode are used to incrementally update the classifier population (Algorithm 13). The update method in line 2 of Algorithm 13 applies the delta rule (cf. Eq. 30) to the weights of the classifiers' prediction functions $\mathrm{cl}.p(s)$ and further updates the other learning parameters. Additionally the GA is run periodically in the batch update.

Algorithm 12: action selection of XCSF

input : state $s(t)$
output: action $a(t)$
1 M = matching(population, $s(t)$)
2 $a(t) = \hat{\mathrm{V}}(s) = \frac{\sum_{\mathrm{cl}\in M} \mathrm{cl}.p(s(t))\cdot \mathrm{cl}.F}{\sum_{\mathrm{cl}\in M} \mathrm{cl}.F}$
3 match_sets.append(M)
4 states.append($s(t)$)
5 return $a(t)$

Algorithm 13: XCSF batch update learning

input: rewards of CI cycle
1 // iterate over encountered states, match sets, rewards
2 **for** $j = 1$ **to** *length(rewards)* **do**
3 update match_sets[j] using states[j] by rewards[j]
4 **if** *GA should run* **then**
5 run GA on match_sets[j]
6 **end**
7 match_sets = {}
8 states = {}
9 prune population

5.3 Experience Replay

Experience replay (ER) is a technique that stores past experiences of the form $(s(t), a(t), r(t), s(t+1))$ in a buffer in order to later use them for training. It originally has been developed to speed up learning in RL settings using the well-known *Q-learning* method [111]. In 2015 Mnih et al. found the ER technique to be crucial to prevent deep Q-networks from oscillations or divergences in the weight updates [112].

Even though ER has proven useful for deep learning it has found little attention in the LCS community. We performed a first study in 2020 which examined the effects of ER on XCS classifier system [12]. We evaluated two classes of problems:

- **Multi-step problems**: These learning tasks are outside of the LCS community known as *sequential decision problems*. For that problem class the agent's utility does not depend on single actions but on the whole sequence of actions chosen.
- **Single-step problems**: Class of learning tasks which can often be rendered as online supervised learning tasks (e. g. online regression) [66].

Within the study we evaluated a series of rather theoretical machine learning problems which belong to the aforementioned classes. We made the observation that ER is beneficial for single-step problems and detrimental for multi-step problems.

Our XCSF-based approach focuses on approximating a value function. Hence the learning task can also be seen as a single-step problem which gives rise to the hypothesis that ER might be beneficial for this practical use case. Hence we also developed an ER version of XCSF for this task.

According to the update rule given in (30), we only need to store experiences of the form $(s(t), r(t))$ for which we use a buffer B of fixed capacity. Whenever the buffer's limit is reached, the oldest entry is deleted in favor of the most recent experience.

Since, in the ATCS setting, rewards can be calculated only after finishing an entire CI cycle (i. e., an episode), new experiences are stored in B only at the end of each CI cycle; this implies that during a CI cycle, B does not change.

The ER-extended XCSF is trained by drawing batches of a fixed size. Each experience $(s(j), r(j))$ is drawn from the buffer with probability

$$\frac{j}{\sum_{i=1}^{t} i} \tag{32}$$

This way, newer experiences are given a higher chance to be drawn which is a form of *prioritized* ER that focuses on the reuse of more recent data. We prefer this prioritization as due to software changes older entries in B are more likely to be deprecated. We call the resulting method *XCSF-ER*.

Algorithm 14: Action selection of XCSF-ER

input : state $s(t)$
output: action $a(t)$
M = matching(population, $s(t)$)
$a(t) = \hat{\mathrm{V}}(\mathrm{s}) = \frac{\sum_{\mathrm{cl}\in M} \mathrm{cl}.p(s(t))\cdot \mathrm{cl}.F}{\sum_{\mathrm{cl}\in M} \mathrm{cl}.F}$
states.append($s(t)$)
return $a(t)$

Algorithm 15: ER and buffer update for XCSF-ER

input: rewards of CI cycle, encountered_states
for $j = 1$ **to** *length(rewards)* **do**
  insert (states[j], rewards[j]) to B
end
if *ER should run* **then**
  draw batch b from B based on (32)
  for *(s, r) in b* **do**
    match_set = matching(population, s)
    update match set using r
    if *GA should run* **then**
      run GA on match_set
  end
states = {}
prune population

Our updated action selection method is illustrated by Algorithm 14. In contrast to XCSF's usual action selection procedure, we only store the encountered states. The buffer update and experience replay method are described in Algorithm 15. The latter procedure is called at the end of every CI cycle. It is worth mentioning that experience replay is only executed periodically (e. g., every third CI cycle) to avoid overfitting. The GA, usually run periodically as well (based on the average time since the matching classifiers took part in a GA iteration), was simply moved into the ER procedure.

5.4 Transfer Learning

The research presented yet focuses on developing a test prioritization agent from scratch on individual projects. However, the formal description of ATCS is rather generic as states solely rely on information such as previous failures or execution times. These are not linked to a specific product such as a car, web application or a home appliance. Hence one might argue that a LCS develops general rules. These might be reused for other projects as well. This leads to the hypothesis that we introduce within this subchapter:

Hypothesis 1. *XCSF-ER benefits from the reuse of previously trained classifier population in terms of performance for ATCS.*

Or in other words: Is transfer learning for test case prioritization possible?

The TL methods mentioned in chapter 2 have been employed for artificial neural networks. However, there is very little research in terms of TL for LCS available. A first study has been performed by Iqbal et al. [113] who developed a rule extraction method for XCS; their technique identifies a subset of the rules part of a solution for low-dimensional binary problems which may be useful for higher dimensional problems of the same problem family. In another work, Li et al. [114] develop transformations of the classifiers' conditions for certain binary problems (e. g. multiplexers) that make it possible to reuse them in other contexts. Both works focus on rather theoretical toy learning tasks. However, our problem does not fit to the aforementioned approaches (as the structure of our state space and action space differs). Further we differ from their approaches as the dimension of their problem space changes and ours stays the same. These facts motivated us to develop a novel transformation of the classifier population.

Our approach is to simply reset certain classifier quality parameters (experience, numerosity, fitness and prediction error) to their initial default values (see Algorithm 16). We transform the entire population and hence reuse all classifiers.

Algorithm 16: Classifier conversion for transfer learning.

input : classifier cl
output: transformed classifier
1 cl.experience = 0
2 cl.num = 1
3 cl.F = F_I
4 cl.ϵ = ϵ_I
5 return cl

Both the fitness and experience values have a large impact on which classifiers are deleted and which ones are kept whenever the population is being pruned. By resetting them, we enable the system to quickly get rid of rules detrimental to the new scenario although having been good for the previous one. If a previous rule turns out to be appropriate for the new scenario as well, it will regain high fitness values quickly. We coin an XCSF-ER that includes this mechanism *XCSF-TL*.

It is worth stressing that we do not adapt the classifier's prediction weights since we use the same action space and these prediction functions are used to compute the actions. Hence we regard them as the main knowledge of the population that we explicitly intend to keep. Furthermore, our approach is not necessarily exclusively linked to ATCS but might also be applied to other problems where the state and action space stay the same between two problem instances.

Our method expects that the dimensionality of the state space of the receiving XCSF and the sending XCSF is the same. The dimensionality of the state space may seem to be variable (it has a dimensionality of $k + 2$, see (24)) but empirical results for both the neural network based [6] and our LCS-based approaches [13–15] indicate that, for each kind of RL approach, there is a certain testing history length k yielding good results. Thus, we assume an arbitrary but fixed k for ATCS for each considered dataset and neglect a change of dimensionality.

5.5 Experiments

For these experiments we focus on three different topics. 1) The evaluation of our XCSF-ER variant and its comparison with the state of the art artificial neural network of Spieker et al. [6] as well as our previous raw XCSF variant [14] 2) An evaluation of our population transformation in order to see if TL is possible for this use case 3) A robustness evaluation.

5.5.1 Evaluation of ER

To evaluate our XCSF-ER against our XCSF variant, we use three different industrial datasets which were originally examined by Spieker et al. [6]. The first two (*ABB paint control* and *ABB IOF/ROL*) are from a Scandinavian robot company whereas the third is the *Google shared dataset of test results* (GSDTSR). An overview of the datasets' structure is provided in Table 15. While each contains the test results of more than 300 cycles, they all vary in the number of test cases and the percentage of failed tests.

Table 15: Examined industrial datasets for the ATCS.

	paint control	IOF/ROL	GSDTSR
CI cycles	312	320	336
test cases	114	2 086	5 555
verdicts	25 594	30 319	1 260 617
failed	19.36%	28.43%	0.25%

We compare the proposed XCSF-ER against the neural network approach of Spieker et al. [6] and our previous XCSF-based agent [14]. It is worth mentioning that we started our research on ATCS using a XCS based approach [13] followed by the XCSF method described in chapter 5.2. The main results of these studies are shown in the Appendix A.

For the neural network and XCSF-based agents we can rely on the respective original implementations and hyperparameters [6,14]. The github repository that combines these implementations with the implementation of XCSF-ER can be found here:

`https://github.com/LagLukas/xcsf_er`

For the problem setting in this work, and according to the used definition of the state space S, classifiers' conditions are a mix of interval-based parts (for the real-valued numbers the state vector $s(t)$) and ternary encoded parts (for the binary numbers of the state vector). For the GA we use a roulette wheel selection based on the classifiers' fitness. Hence a classifier is drawn at random but its probability to be drawn is equal to the proportion of its fitness in the entire population. For the crossover of the ternary conditions we apply a one point crossover and for the intervals and weights of cl.$p(\cdot)$ an arithmetic one. The one-point crossover chooses an integer i from $\{1, 2, ..., k\}$ uniformly at random. The first set of conditions is created by taking the first i conditions from the first classifier and the last $k - i$ conditions from the second one. For the second set of conditions to be created this is reversed. The arithmetic crossover computes new conditions from the old ones by taking λ percent of the first parent's conditions and $1-\lambda$ percent of the second parent's conditions, for example $\mathbf{x} = \lambda\mathbf{y} + (1-\lambda)\mathbf{z}$ if $\mathbf{y}$ and $\mathbf{z}$ are condition vectors (we used $\lambda = 0.6$). For the mutation of the ternary conditions we follow the widespread approach of Butz and Wilson [115] (i. e., iterate over the bits and flip each with a probability of $\frac{1}{q}$ where q is the number of bits) and for the interval-based conditions and weights of cl.$p(\cdot)$ we apply a random mutation (i. e., choosing an entirely random interval). We don't perform any form of subsumption as it is often the case in the literature [66].

Our buffer has a maximum capacity of 12 000 and we draw batches of size 2 000. Updates using ER are performed every third cycle. The initial weights of cl.$p(\cdot)$ during covering and mutation are drawn uniformly at random from $[-10, 10]$. The remaining parameters are adopted from our previous study [14][8].

[8] $\eta = 0.1$, $N = 2000$, $\alpha = 0.15$, $\beta = 0.15$, $\nu = 5$, $\theta_{GA} = 25$, $\mu = 0.025$, $\epsilon_I = 0$, $F_I = 0$, $\theta_{del} = 20$, $\theta_{sub} = 20$, $\chi = 0.75$, $\epsilon_0 = 0.01$, $P_{\#} = 0.33$ (notation of Butz and Wilson [115])

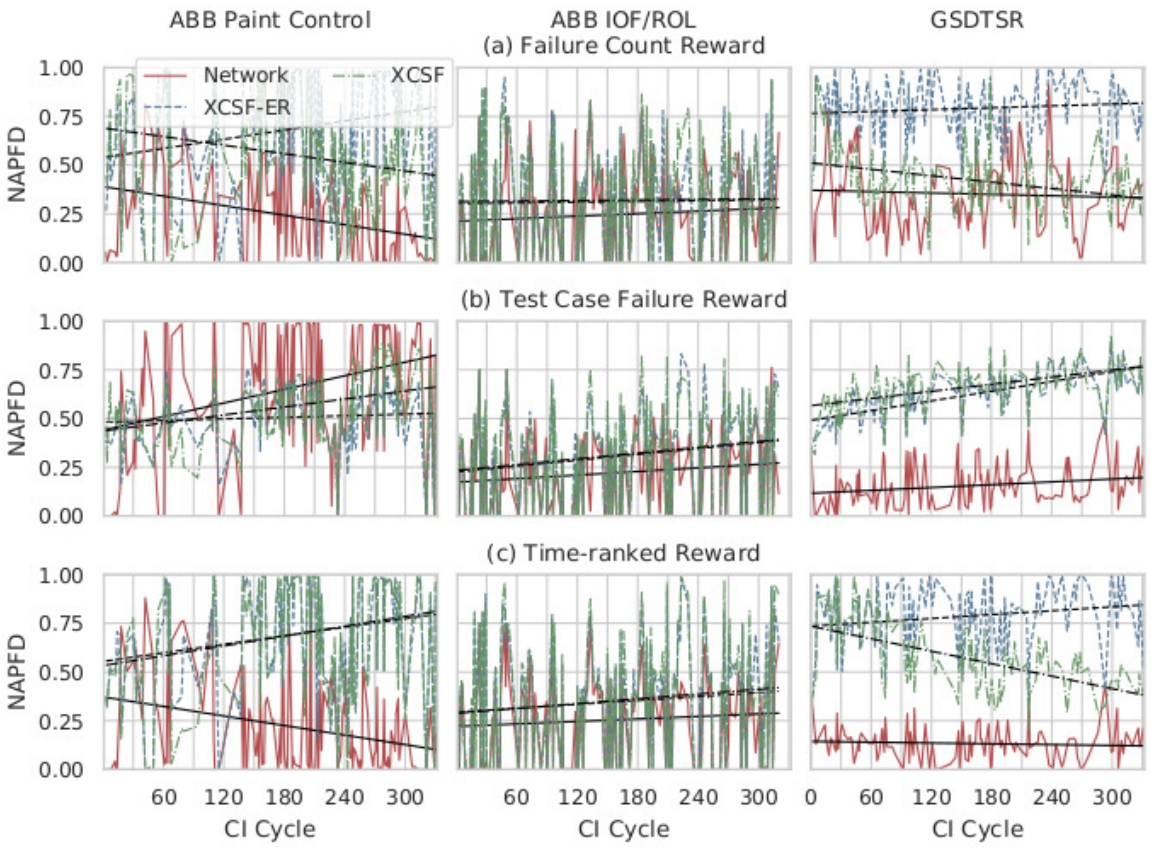


Figure 16: Comparison of XCSF-ER with XCSF and the neural network.

We selected an appropriate value for the history length k by means of conducting a preliminary hyperparameter study. We evaluated $k \in \{2, 3, ..., 10\}$ on the paint control dataset using the time ranked reward. At that, XCSF-ER achieved the best results for a value of $k = 6$ which we are using throughout the following experiments.

We have run our experiments for 30 i. i. d. repetitions and present the averaged results. Since the datasets contain the results of all the test cases for the respective number of CI cycles, we are able to measure the performance of the methods in terms of the NAPFD metric while simulating the actual use case by only providing the agent with the results of the tests that it chose to run. Furthermore, we reproduce the setting of Spieker et al. [6] and give every agent a time budget which corresponds to 50 percent of the runtime of all tests.

The results of running our approach on the three datasets using each of the three reward functions from chapter 5.1 are displayed in Figure 16 where each column represents one of the datasets and each row corresponds to one of the reward functions. The plots also contain trendlines for each agent's results. Note that there is still a rather high amount of variance which is mainly due to the changes in the underlying software reflected in the datasets which naturally have a strong impact on the test results.

For the first column (ABB paint control), ER shows a positive effect for the failure count reward: XCSF-ER is superior to the other agents (in terms of the trendline). However, for the test case failure reward ER appears to be rather detrimental and it is outperformed by the other two agents. For the time ranked reward, there is no observable effect of ER on the performance. Nonetheless, both XCSF and XCSF-ER achieve higher NAPFD values compared to the neural network using this reward function.

The trendlines of both LCSs are superior to the one of the neural network for the ABB IOF/ROL dataset (Column 2). Visually we can observe neither positive nor negative effects of ER on XCSF.

On the Google data it can be seen that ER has a negative effect if the test case failure reward is applied. However, if the other two reward functions are used then ER is clearly beneficial for XCSF. Furthermore, both LCSs exceed the network on this dataset.

As we can see, for both the neural network- and the pure XCSF-based method, a reward function that is beneficial for one dataset may be detrimental to another one. For example, when considering the neural network, the failure count reward function works best on the GSDTSR dataset whereas the performance decreases over time if it is applied to the ABB Paint Control dataset (we make a similar observation when combining the XCSF-based solution with the time ranked reward). This is unsatisfactory from a practical point of view since an appropriate reward function should be known a priori. However, for XCSF-ER we could not observe this issue: Both the time ranked reward as well as the failure count reward are suitable choices for all three datasets considered.

Table 16: P-values for H_0: XCSF $\geq$ XCSF-ER using the time-ranked reward for XCSF-ER (significant entries are bold).

reward function XCSF	paint control	IO/ROL	GSDTSR
failure count	**0.00681**	0.15045	**0.0**
test case failure	**0.0**	0.08591	**0.0**
time ranked	0.47241	0.55662	**0.0**

Table 17: P-values for H_0: XCSF $\geq$ XCSF-ER using the failure count reward for XCSF-ER (significant entries are bold).

reward function XCSF	paint control	IO/ROL	GSDTSR
failure count	**0.00269**	0.52413	**0.0**
test case failure	**0.0**	0.40465	**0.0**
time ranked	0.34184	0.88968	**0.0**

Table 18: P-values for H_0: neural network $\geq$ XCSF-ER using the time-ranked reward for XCSF-ER (significant entries are bold).

reward function network	paint control	IO/ROL	GSDTSR
failure count	**0.0**	**0.00072**	**0.0**
test case failure	0.2248	**0.0**	**0.0**
time ranked	**0.0**	**0.00121**	**0.0**

Table 19: P-values for H_0: neural network $\geq$ XCSF-ER using the failure count reward for XCSF-ER (significant entries are bold).

reward function network	paint control	IO/ROL	GSDTSR
failure count	**0.0**	**0.01779**	**0.0**
test case failure	0.14179	**0.0001**	**0.0**
time ranked	**0.0**	**0.02691**	**0.0**

Since trendlines might be disturbed due to outliers, a pure visual examination of the results may be insufficient. Thus we also perform statistical tests. We decided to use a series of paired Student-t tests and we denote null hypotheses as H_0. We use them to compare the averaged results of individual agents which we displayed in Figure 16. Student-t tests have the precondition that the data has to be normally distributed which we ensured with a series of Shapiro-Wilk tests. A significance level of 0.05 was used for all tests.

We first evaluated if the failure count or the time rank reward function is better suited for XCSF-ER. Here, statistical tests could not give a precise answer as no p-value was below the significance

level of 0.05 for any of the three datasets. Thus we compare the network and XCSF with XCSF-ER using these two reward functions. It is worth mentioning that the statistical tests confirmed that the failure count reward and time ranked reward are a better choice than the test case failure reward for ABB paint control and GSDTSR.

Table 16 contains the results of the comparison of XCSF-ER using the time ranked reward function and each possible combination of XCSF and reward function. The analogous case is displayed in Table 17 for the failure count reward function. In both tables our agent is statistically significantly superior in 5 of 9 cases. For the remaining cases the statistical tests cannot confirm which approach is superior to the other (IOF/ROL dataset and paint control dataset using time ranked reward). Hence the statistical tests support our visual observations regarding XCSF.

We display the analogous camparison with the neural network in Tables 18 and 19. Here XCSF-ER is significantly superior in 8 out of 9 cases. In the remaining case the p-value is rather low and thus there is a tendency that XCSF-ER might be better. Once more we cannot say which of the time ranked and failure count reward functions is more suitable in direct comparison.

Overall, our results confirm that ER can increase XCSF's performance. We can also recommend to use either the failure count reward or the time ranked reward in practice for our agent. Furthermore, this enables us to close the gap between the neural network and the LCS since on all three datasets our system is superior or equal in performance with a suitable but fixed reward function. The latter is not possible for a pure XCSF since each reward function leads to unsatisfactory results on at least one dataset.

5.5.2 Evaluation of TL

A conclusion from chapter 5.5.1 was that the failcount and the time rank reward are best suited for XCSF-ER. Hence we confine to these two reward functions within this part of the thesis.

The source code used for the experiments can be found here:

`https://github.com/LagLukas/transfer_learning`

We further evaluate a novel dataset from BSH coined *SMBV1* within these experiments[9]. The dataset corresponds to a product which is in an early state of development and hence only information about 40 CI cycles is available. It contains $165,073$ verdicts of up to $1,632$ test cases. About 0.58 percent of the executions failed.

Here we aim at evaluating the effects of our population transformation as defined in Algorithm 16. Hence we use the same hyperparameters for both LCSs (the ones from chapter 5.5.1, we also average the results over thirty runs). This enables us to examine the transformation's impact as the systems only differ in the fact that one uses the transformation and the other does not.

Our first question deals with the choice of the base dataset for our transfer learning approach. A common practice in natural language processing and computer vision is to reuse a trained model that performed reasonably well on a large dataset [69,116]. This would point to the usage of the GSDTSR dataset since it is the largest and the previous LCSs performed well on it [14,15]. We examined this hypothesis by using each available dataset as a training basis for XCSF and by applying the pretrained model on the other remaining datasets. We displayed the averaged NAPFD values achieved in Table 20 (using the time rank reward). On the SMBV1 dataset, all pretrained models perform similar. On the other datasets, the model pretrained on the GSDTSR dataset works best. Thus results show that, on average, the GSDTSR dataset is the best choice. We confirmed this by applying a series of one-sided Wilcoxon tests whose p-values were below our significance level of 0.05 for both the paint control and the IOF/ROL datasets. On the SMBV1 dataset we used a two-sided Wilcoxon test whose p-value indicated that the agents indeed perform equally (again using a significance level of 0.05). Hence in the following experiments we are using the model pretrained on the GSDTSR dataset.

Now we switch our focus to a comparison with the model without TL. We consider the paint control, IOF/ROL and SMBV1 dataset (since we used the GSDTSR for pretraining the model).

We display the quotient of the NAPFD values of XCSF-TL and the base version in Table 21. There we can see that the usage of TL boosts the performance by up to 5.5 percent. Furthermore, we cannot

[9] We also compared XCSF-ER and the neural network of Spieker et al. [6] on this dataset and could statistically confirm that XCSF-ER is the better choice there [16].

Table 20: Average NAPFD $\pm\sigma$ achieved for different datasets as knowledge bases. The columns correspond to the examined pretrained model and the rows to the examined dataset.

applied to \ *pretrained on*	paint control	IOF/ROL	GSDTSR	SMBV1
paint control	-	0.78 ± 0.26	0.79 ± 0.27	0.78 ± 0.26
IOF/ROL	0.5 ± 0.32	-	0.51 ± 0.34	0.48 ± 0.32
GSDTSR	0.69 ± 0.17	0.74 ± 0.17	-	0.66 ± 0.2
SMBV1	0.5 ± 0.0	0.5 ± 0.0	0.5 ± 0.0	-

Table 21: NAPFD ratio between the XCSF-TL and the XCSF-ER variant not using it.

	paint control	IOF/ROL	SMBV1
failcount	1.0121	1.0550	1.0264
time rank	1.0341	1.0210	1.0001

observe negative effects in any of the nine experimental set ups. The lowest performance increase can be observed for the SMBV1 dataset when trained using the time rank reward. There, TL improves the performance only by about 0.01 percent. Nonetheless, the experimental results are in favor of our hypothesis. We once more perform additional statistical tests to examine our hypothesis since the averaged NAPFD values contain a certain level of variance. We again use one-sided Wilcoxon tests and test the hypothesis that the XCSF-ER without TL is superior to XCSF-TL. The p-values are shown in Table 22. We once more used a significance level of 0.05 and all p-values are below it. We can reject all null hypotheses and thus infer that XCSF-ER benefits from TL on the considered datasets.

We also intend to give an overview about the corresponding time series of each agent and not only about aggregated results. Thus we plotted the averaged NAPFD values for each CI cycle, agent, and dataset in Figures 17 to 19. Note, that there is a certain variance in performance due to the changes in the underlying environment (e. g. due to new software errors or fixed ones).

In Figure 17 we can observe close to no effect of TL for the time rank reward. We can only see a small performance increase at the start and around CI cycle 32. For the other cycles the performance is equal. However, for the failcount reward we can clearly recognize the advantage of TL in terms of NAPFD. The vanilla XCSF needs about 35 CI cycles to catch up.

For the IOF/ROL dataset (Figure 18) we can see no performance boost at the start of the experiment. However, TL dampens occuring performance breakdowns, especially for the failcount reward (e. g. for CI cycles between 200 and 250). The same effects appear even more distinct on the paint control scenario (Figure 19).

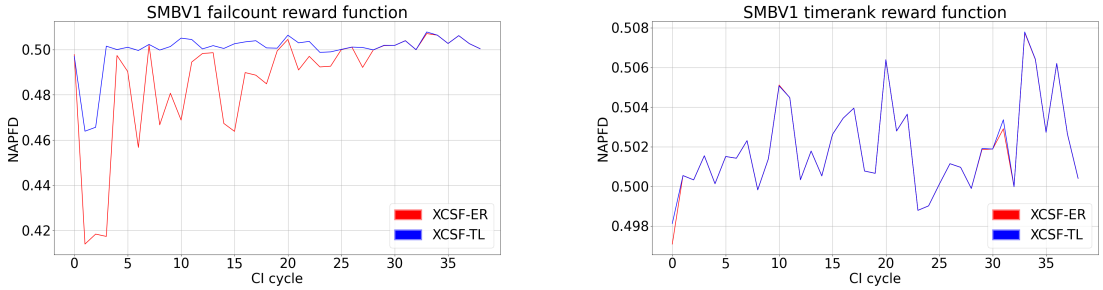
In general, the visual evaluation fits our statistical one. The Wilcoxon test filters out values of equal performance and only considers data points where the observations differ. In the majority of these differing data points our TL approach has higher NAPFD values which explains the p-values below the significance level.

5.5.3 Evaluation of Robustness

In the previous experiments we evaluated how well our algorithmic changes to XCS deal with ATCS in terms of NAPFD and how good they perform against the artificial neural network of Spieker et al. [6]. One additional observation that we could make is that there is a high variation in terms of

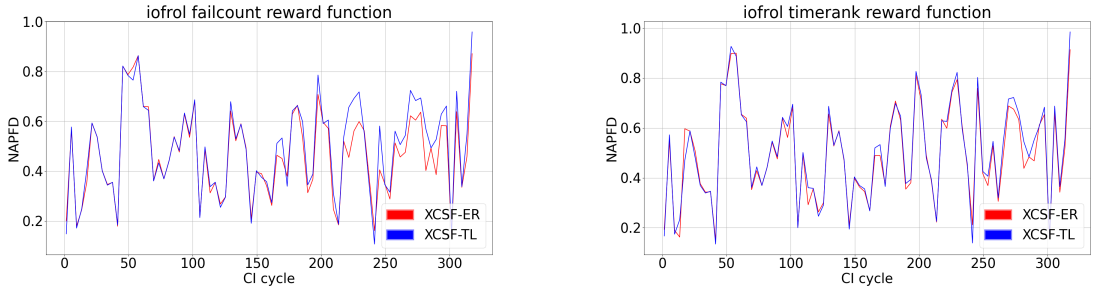
Table 22: P-values for the null hypothesis that TL is detrimental for the performance in terms of NAPFD.

	paint control	IOF/ROL	SMBV1
failcount	1.46e-29	4.86e-73	2.29e-524
time rank	1.28e-87	6.91e-24	0.001346



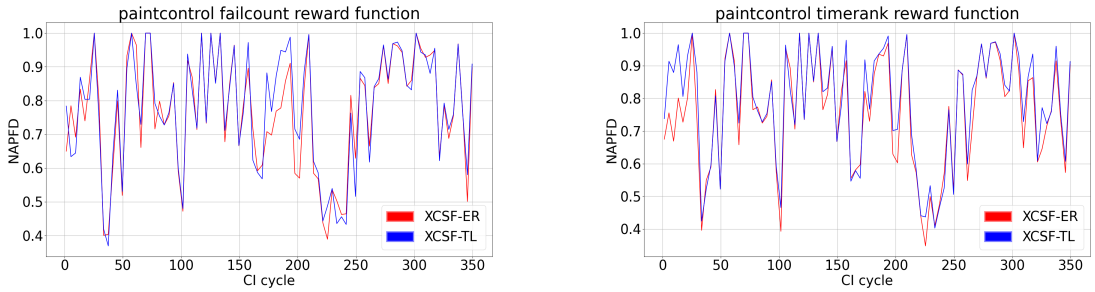
(a) failcount (b) time rank

Figure 17: Averaged achieved NAPFD values for the SMBV1 dataset.



(a) failcount (b) time rank

Figure 18: Averaged achieved NAPFD values for the IOF/ROL dataset.



(a) failcount (b) time rank

Figure 19: Averaged achieved NAPFD values for the paintcontrol dataset.

NAPFD for the agents considered (see Figure 16). This variation may be traced back to changes in the underlying software code basis, the test cases at hand or the test bed itself.

OC has come up with two robustness metrics to measure how well a method can cope with the influence of disturbances δ:

- **passive robustness**:
$$r_{\text{passive}} = \frac{\partial U}{\partial \delta} \tag{33}$$
- **active robustness**:
$$r_{\text{active}} = \frac{\partial U}{\partial t} \tag{34}$$

where U denotes an utility function (in our case NAPFD). Note that the active robustness is only measured when the utility is below a predefined threshold.

These metrics are inspired by physical disturbances which usually can be measured and are differentiable. Furthermore the utility function has to be differentiable as well. For NAPFD this is not the case since it is defined on discrete sets (the prioritized test cases and their outcome) which thereby are closed sets and the mathematical notion of differentiability requires a function to be defined on open sets.

Passive robustness measures the impact of the disturbance on the utility by using the partial derivative (with respects to δ). Thereby it describes the change in utility if the magnitude of the disturbance changes. However, due to the aforementioned mathematical issues and the fact that we cannot measure the magnitude of a disturbance (e. g. what is the magnitude of a software bug?) we leave out this KPI.

Similar problems arise when the active robustness is considered, as our time axis is discrete (the CI cycles). However, OC has come up with an approximation to overcome these issues:

$$\frac{\Delta U}{t_{\text{rec}}} \tag{35}$$

where ΔU is the maximal drop in utility and t_{rec} is the time the system needs to recover. Note t_{rec} is the length of the span which starts with the performance break-in and stops when the utility is above the threshold again. Similar to the gradient described in equation 34 the approximation measures the recovery speed. The estimation enables us to have a measure for it and thereby we focus on this KPI.

In order to have an evaluation over wider time horizon we consider the GSDTSR, paint control and ifrol datasets (these contain results of a few hunderd CI cycles and the previously considered SMBV1 only of about 40) for this examination. We use XCSF-ER for the GSDTSR dataset and XCSF-TL for the remaining two. We observed highly similar results for both the timerank and failcount reward and in order to not repeat ourselves we confine here to the timerank reward. Nonetheless, the results for the failcount reward are listed in Appendix B.

We once more perform 30 repetitions and give the agent a time budget of 50 percent of total execution time. There we reuse the hyperparameters from the previous experiments.

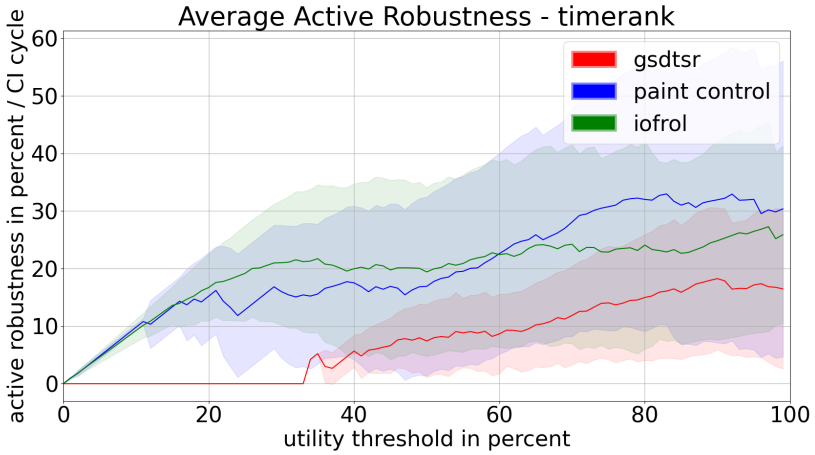


Figure 20: Estimation of the active robustness.

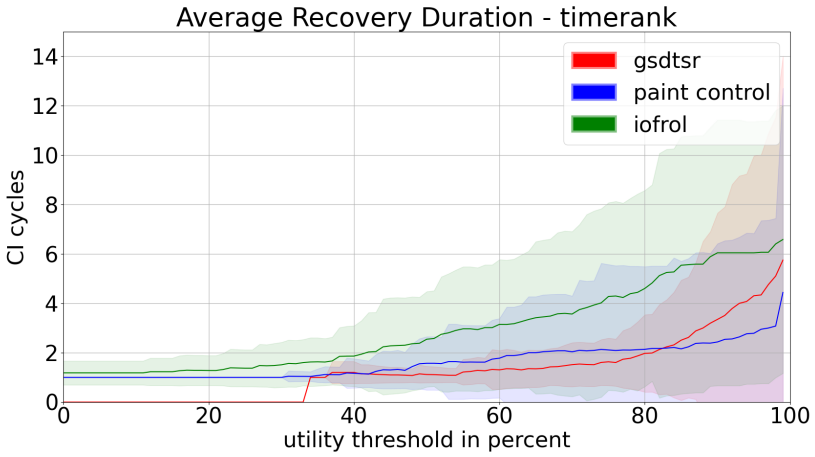


Figure 21: Estimation of the average recovery time.

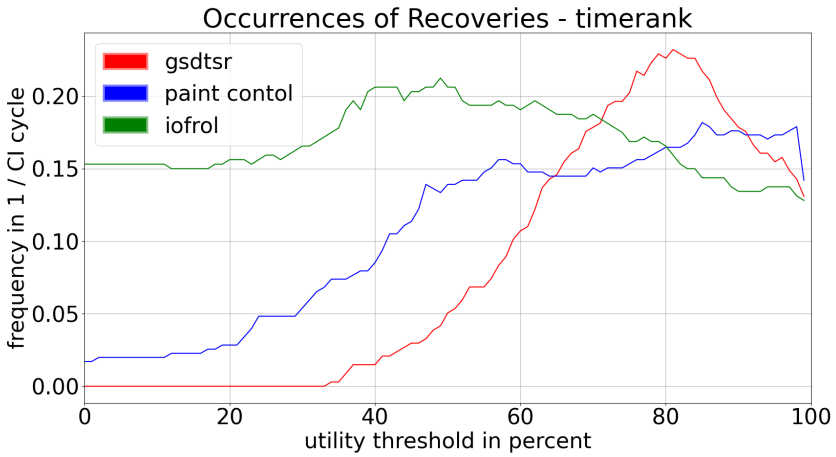


Figure 22: Event frequency of utility break-ins.

For our evaluation we consider a variety of utility thresholds (ranging from 0 to 100 percent of NAPFD). We present the averaged results ± standard deviation σ in Figure 20. As can be expected, both robustness and its variance generally increases with rising utility thresholds. Notably there is a reduction in the active robustness for the paint control dataset if the utility threshold rises above 80 percent. In this range the NAPFD values are several times slightly below this value but recover in one CI cycle which in turn reduces the active robustness (the ΔUs are very small). For GSDTSR we set the active robustness to zero until a threshold of about thirty percent since for these values we could not observe any performance break downs below that level. On the first blink the measured active robustness (recovery speed) may seem low, but one may keep in mind that our utility function only has a small limited range (0 to 100 percent). Thereby our system is capable to recover rather quickly.

Figure 20 also indicates that our system performs differently for each considered dataset. We evaluated this hypothesis using a Friedman test that was significant (using a significance level of 0.05) and thus we infer that the structure of the project to test also has an impact on our test system's performance.

The previous evaluation focused on how fast the system reaches a predefined utility threshold after a performance break down. In Figure 21 we switch the focus on how long it takes to recover in such situations. The plot shows that, for all considered datasets, the performance recuperates after 1-3 CI cycles if the threshold is below 50 percent. For higher thresholds the required CI cycles increase but are still on average single-digit. It is also worth mentioning that our system was always capable of recovering. Furthermore we can observe once more that the three datasets lead to different recovery times which we once more can confirm using a Friedman test and the aforementioned significance level.

After elaborating how fast the system recuperates and how long these events take, we switch our focus towards how frequently such events occur. We measured the frequencies (in $\frac{1}{\text{CI Cycle}}$) for different thresholds in Figure 22. For all three considered datasets we can observe that the frequencies increase with the thresholds until reaching a threshold value in the range of 70 to 80 percent. After that, the frequencies start to decrease since multiple small events that were counted separately for lower thresholds tend to be merged into fewer, longer events as the threshold is increased (which is in line with our evaluation of the recovery times).

We can observe differences in the frequencies for utility thresholds lower than 50 percent. The highest frequencies can be observed for the iofrol dataset, a lower one for the paintcontrol dataset and the lowest for GSDTSR. For the latter there is often a frequency of zero (no event at all). These differences are not at random and an additional significant Friedman test underlines this. Thus the cause is linked to the internal structure of the datasets. One difference between GSDTSR respectively paintcontrol and the iofrol dataset is the frequency of the test runs. For the first two projects, tests were run on a daily basis (sometimes even several times) and for iofrol usually only once per month. Hence two succeeding iofrol software versions can be considered as more different if compared to GSDTSR or paintcontrol and it is naturally more difficult to create precise test case prioritization based on historical data. However, the frequencies are still rather small and if we also take the high active robustness values and short durations of the recovery processes into account then one can see that our approach can be regarded as robust for these three datasets.

5.6 Additional Discussion of the Approach

In the previous parts of the chapter we focused on a detailed problem and algorithmic description followed by an evaluation. Within this part we want to discuss the methods classification as an RL algorithm and why we chose such an approach.

We try to solve ATCS by approximating a state-value function in an online way. These regression tasks can also be classified as *supervised learning*. Furthermore ATCS is not modelled as Markov decision process which is the usual problem formulation in RL [65]. However, it is worth mentioning that RL is not limited to Markov decision problems [117]. Whitehead and Lin [117] solely require a definition of an agent and its environment (which is given here). We try to approximate a state-value function which is often in the focus of RL. For example Q-Learning tries to find an optimal state-value function which solves one of the well-known Bellman equations [118].

Overall there are arguments for a classification of the method as RL or supervised learning. We think it boils down to how one weighs these arguments. For the mere sake of consistency we followed previous works and classified it as a RL problem whilst we understand both opinions. We think nonetheless that from a academic point of view the question was worth discussing.

The second topic that we want to elaborate here is why we decided to go for an LCS. The original work of Spieker et al. [6] employed not only a neural network, but also Q-Learning (which was outperformed by the former). Thus there has already been an experimental evaluation of two widespread methods. Out of pure curiosity we tried out an alternative approach (XCS classifier system) in order to find out how this evolutionary approach performs on this novel problem. There we lacked behind on some problem instances compared to Spieker et al. [13]. XCS differs from XCSF as it has a constant output (discrete actions). One of the reviewers gave us the critical feedback to employ an XCSF to compute continuous actions which we tried out. There we only had to figure out how to use it and which function shall be approximated. As we can see in our evaluation the critique was well placed. Thus one might say that our approach was, similar to the techniques, evolutionary.

5.7 Related Work

Clearly our work is related to the publication of Spieker et al. [6] who were the first to come up with an RL approach for the use case.

From a machine learning perspective this a LCS chapter. We are not the first trying to change the algorithmic structure of XCSF in order to improve its overall performance. Recently interpolation-based enhancements have achieved a certain level of success on a variety of toy problems but also on a first practical use case (distributed traffic control) [119]. The usage of interpolation does not exclude ER and thereby the work presented might be extended by it. There are even the first approaches using interpolation to create new synthetic experiences which might further push performance [120]. However, the possible threat of these methods is that interpolation usually requires some form of spatial closeness / continuity [121] which is not necessarily the case here.

Furthermore metaheuristics might also be used for the task as can be seen in chapter 4. The major difference is in terms of runtime. In our metaheuristics chapter we measured runtimes of up to two hours until a test suite has been computed as the number of iterations until convergence is detected might be high. However, for XCSF that is not the case. The computation of a test case priority is independent of the test corpus' size (see chapter 5.2) and thereby the test suite's computational cost is linear in the test corpus size. This translates to small runtimes in real time (we often measure only a few seconds for our final system). We deemed the metaheuristic's high cost as a no-go for the CI context (especially as the test corpus underlies frequent changes and there might be several CI cycles a day which affects the test suite to choose). We deemed a computational lightweight such as an LCS as more feasible.

The approach discussed further differs from multi-objective approaches as we focus on the quality of the underlying prioritization and failure revealing capabilities which is computed based on the outcome of a test session. In practice these test results are often stored in standardized *junit xml* files which originated from the Java unit test framework *Junit* [122]. These files are also created by other major test frameworks such as *pytest* [123] or *Google test* [124]. The report generation is also supported by build servers such as Jenkins [125]. Hence in a CI environment the information necessary to use the RL approach is easy to retrieve. Additional coverage based methods are either limited to white-box testing (as the underlying source code must be available) or the test cases must be linked to the requirements (in case of requirements coverage). The latter can lead to a considerable effort for automated testing since the test cases (in form of software, e. g. a pytest test case) must be linked to these requirements and these links must be updated upon test and requirements refinement. Hence the RL approach can be seen as a rather generic approach and the multi-objective ones as more specific methods. Thus a potential user of the technology should consider which preconditions are valid for their project to test.

5.8 Chapter Summary

Within this chapter we evaluated if LCS can be used to prioritize and select test cases in a CI environment. We presented a modified XCSF variant which we used to employ a heuristic for prioritizing the test cases. We further pushed XCSF's performance by incorporating experience replay. We coined the result *XCSF-ER*. We benchmarked XCSF-ER against a state of the art artificial neural network for the task [6] which is also used by Netflix [108]. Thereby we could not only empirically show that our method has an equivalent performance, but is in most cases superior in terms of performance.

An issue that both the artificial neural network and the raw XCSF faced is that the correct reward function is only known a posteriori which is problematic in practice as a favorable reward function should be known from the start of the project. Our XCSF-ER is capable to overcome this issue and we are able to recommend two reward functions a priori.

We further asked ourselves the question if acquired knowledge for one test project can be reused for another one. This is of importance as enterprises usually do not only develop one product. We developed a simplistic population transformation and showed experimentally that transfer learning for this test use case is possible. It is worth mentioning that we know of no work previous to ours that employs transfer learning for test case prioritization.

We extended the evaluation of our approach by incorporating active robustness measurements which is a OC KPI to estimate how fast a method / system recovers if a disturbance causes a performance breakdown. We extended this OC analysis by additionally considering the average recovery time. Further, we measured how frequently such performance breakdowns occur which we deem as necessary since a pure recovery speed / time analysis leaves out the important information how often these events occur. We could show that for a bandwidth of performance goals our approach has high recovery speeds, short recovery durations and events occur fairly seldom.

6 Runtime Efficiency Considerations

In chapter 5 we exclusively focused on designing an LCS which computes test suites that satisfy testing metrics. Another vital task is to calculate the test suite in a reasonable amount of time due to the following two reasons:

- The problem at hand is a time critical problem. Too long search times might erase the advantage of a crucial test suite with low execution time.
- The test suite is not constant but must be continuously recomputed and it is not known when new CI builds are started. This makes computing the test suite ahead of time hard to realize.

We exclusively focus on the matching step as empirical studies indicate that it is the computationally most expensive step [126–129]. Further, theoretical runtime analysis underlines this fact [130].

In order to achieve runtime cutdowns we employ parallelism. A key constraint of our problem domain is that we do not know the target hardware architecture where our later testing system might run. We do not know if it has a *graphical processing unit* (GPU) or with what kind of processor it is equipped. This forces us to design an approach that is hardware independent. We therefore make the minimalistic assumptions that the underlying computing device has a multicore-processor and an interface to program it. These are conditions that are valid for a vast range of desktop processors (e. g. all of Intel's i-series processors and AMD Ryzen processors have several cores) and even for cheaper platforms such as Raspberry Pi this is the case. Further, threading libraries are a standard in major programming languages such as C++, C#, Java or Python.

The goal of being hardware-independent is not limited to our use case. It is of general interest when designing parallel programs. Mattson et al. [131] defined it as a goal when designing a parallelization strategy. Independence from the underlying hardware enables the ease reuse of the approach on other architectures. Mattson et al. [131] further demand that programs should be *comprehensible* which they interpret as easy to maintain and test. We also take this goal into account by designing a simplistic parallelization approach. Furthermore, one must also take the original goal of any parallelization into account: Creating a speed up for the problem at hand. Thus the design of a parallelization can be seen as a multi-objective optimization problem. This naturally means that there might be tradeoffs between the objectives, for example if we improve the hardware independence, we might slow down the program.

The matching process is for most LCS highly similar [66, 130]. Thus we define algorithms on a high level which can be reused for a variety of LCS. In order to have a more general overview on the parallelization to be introduced we focus on simulated states and the most widespread LCS (XCS classifier system). The simulation enables an easy evaluation of different hyperparameters which have an influence on the runtime of the matching process such as the size of the match set or the dimension of the problem space.

6.1 Related Work

Parallel programming is not only a field of research for computer scientists. It often involves the development of specialized hardware such as GPUs which may later be used not only for graphics programming, but also for different tasks like neural networks [132]. GPUs are also used for Google's machine learning framework tensorflow and there have also been first steps of exploiting GPUs to speed up XCS [128, 129]. The latter requires an Nvidia GPU that can be programmed via *compute unified device architecture* (CUDA) which has been designed to be used within Fortran, C and C++. CUDA uses its own parallel programming approach which requires knowledge of the hardware structure of the GPU that is used. Hence a CUDA program that is optimized towards one type of GPU might be a bad choice for another one [133]. Also, the CUDA implementation must be optimized towards the type of LCS that is used. However CUDA has not only been applied to XCS, but also to other LCS such as BioHEL [134].

Another approach for XCS is to exploit *single instruction multiple data (SIMD)* instructions for matching [126]. Further bitfields are used to reduce the memory needed to save classifiers. SIMD instructions can be seen as special commands for vector operations. The used instruction set is widely

available for AMD and Intel processors, but on the other hand it requires to use the original XCS with ternary conditions. In this setting a state is a binary vector and the conditions are a vector where each entry has either the value 0, 1 or #. 0 and 1 indicate that the corresponding entry of the state must be 0 or 1 and # is defined as a "don't care" symbol. The latter entry accepts both 0 and 1. This may speed up the matching for the original formulation of XCS, but not in all problems states a purely binary and there are various extensions of XCS such as the XCSF that we use for testing. A C approach has been used for the implementation. Another approach for XCS does not involve parallel programming at all. Gilles et al. [127] exploited XOR commands to speed up the matching of XCS.

It is worth mentioning that optimizing machine learning algorithms is a lucrative task and has come into the focus of several companies. For example *Apache TVM* is an open source project for optimizing neural networks for a target hardware [135]. The project is supported by major companies such as Nvidia, Amazon, Microsoft, Intel, ARM and AMD. Other closed source companies such as AU Zone offer entire SDKs to develop and optimize neural networks for a target architecture [136]. However, we were unable to find similar technology for LCS.

In summary, existing parallelization attempts for LCS are designed towards a specific hardware family (e. g. Nvidia GPUs) or certain desktop CPUs. There is no generic approach that offers some level of abstraction of the underlying hardware. Further expert knowledge is required for the concrete implementation (e. g. sufficient knowledge of CUDA or the processor instruction set). The approaches exclusively focus on the speed up objective. These approaches achieve vast successes and can provide high speed-ups (90 for the SIMD approach [126], 12 and 18 for the different GPU approaches [128,129]). However, their hardware dependencies render them unfeasible for our use case. Hence we saw the need for an easy to implement hardware-independent approach.

6.2 Algorithmic Design

When a parallel program is designed, an initial step is usually to identify which resources are shared between threads and where race conditions can occur [131]. We identified the match set as a possible shared resource. Threads may search the population for matching classifiers and sporadically insert the found ones in the match set. We realize the match set as a list. We have identified two strategies in order to overcome the aforementioned race condition:

1. A lock can be used to only allow one thread at a time to access the match set.
2. Each thread gets a local match set and after the entire population has been searched the local match sets are merged.

A *lock* is a standard mechanism which can be used to protect shared resources as it only allows one thread at at time to acquire it [137]. If the match set is rather big, then the second approach leads to fewer accesses of the lock than the first. The merge step of the second method might lead to many threads trying to write to the global match set at the same time.

In order to calculate the match set, the entire population must be traversed. Each element should only be checked once. Hence, the population should be split in order to get chunks of equal size for each thread. Thus we introduce a task model for the matching based on a separation of the population. We describe our top level behaviour in Algorithm 17. For the created tasks we examine both solutions for race conditions later on. The first approach is described in Algorithm 18 and the other one in Algorithm 19.

The designed algorithms do not rely on any specific version of LCS and, with traditional methods of object oriented programming, reusability of the matching method can be assured. For example, by using inheritance for classifiers and by calling the method of the abstract base class inside the algorithm. Furthermore, we examined the programming languages C++, C#, Java, R and Python and all of these have a threading and a lock mechanism which enables a developer to implement the designed algorithms directly. Hence no source code in another programming language is necessary that must be wrapped for usage.

Algorithm 17: Creation of tasks for the matching.

input : Population P, state s, n_threads
output: Match Set

```
match_set = { }
chunk_size = floor(|P|/n_threads)
for i = 0; i< n_threads; i++ do
    thread i find matching for P[i * chunk_size] to P[min((i + 1) * chunk_size, |P|)]
end
return match_set
```

Algorithm 18: Matching of a task using a lock whilst inserting a new classifier.

input : Population P, state s, start, end

```
for i = start; i< end; i++ do
    if s fullfills conditions of P[i] then
        lock()
        insert P[i] to match_set
        unlock()
end
```

Algorithm 19: Matching of a task using a lock and local lists.

input : Population P, state s, start, end

```
local_list = { }
for i = start; i< end; i++ do
    if s fullfills conditions of P[i] then
        insert P[i] to local_set
end
lock()
merge local_list with global match_set
unlock()
```

6.3 Experiments

The duration of a matching method depends on several parameters. If the population or the dimensionality of the state vector becomes larger, the sequential filter process' runtime is rising as its complexity is linear in the population size and the state space's dimension. The runtime of a parallel program also depends on the number of used threads and the number of race conditions that occur. The latter correlates in our case with the size of the resulting match set and the former is examined to determine how well the program scales if more resources are available. Thus we examine the influence of these four magnitudes on the performance in terms of the algorithm's speed up (compared to the sequential process).

For our XCSF for ATCS the dimension is fixed. Further, the match set size is induced by the dataset and the current population at hand. In order to have widened view on our parallelization we consider a simulation and the vanilla XCS (where these issues do not occur). If we create a classifier that shall not match the given state, we draw an index i uniformly at random and set the i-th condition of the classifier to the opposite of the state. Further, the non-matching classifiers are distributed uniformly over the population.

Overall we consider population sizes from $10,000$ to $150,000$ and increase the size by $10,000$. We use the dimensions $3, 6, 9, ...24$. We use an Intel i7 processor with up to 12 virtual cores for our experiments and use $2, 4, 6, ..., 12$ threads for our experiments to examine the scalability of our algorithms. We simulate that 1, 2, 3, ... 30 percent of the population match the given state. This leads to $21,600$ different experimental settings. We perform 100 iterations for each combination of the described magnitudes.

We once more published the source code that corresponds to the experiments:

`https://github.com/LagLukas/para_matching`

6.3.1 Scalability

The scalability of a parallel program is its ability to increase its speed up when more resources such as threads or cores are added [137]. As multicore processors can highly vary in the number of cores, this is an important question for us to examine. In Figure 23 and Table 23 we aggregate our results into one dimensional data (grouped by the number of threads). On the x-axis is the number of used threads and on the y-axis is the speed up in comparison to the sequential matching. For each of our two parallel algorithms we visualize the average speed up and the speed ups within two times the standard deviations[10]. In our experiments utilizing only few threads, the algorithm with a simple lock exceeds Algorithm 19, but starting from about eight threads the matching method with local lists turns out to overtake the lock-based approach. Furthermore there seems to be no more growth in terms of performance after more than ten threads for the method using a simple lock. Algorithm 19 seems to scale better and even with the maximum number of parallel threads for our processor there still seems to be no saturation. For both methods the variance increases with a higher number of used threads.

Table 23: Aggregated speed ups (rounded to the third decimal, best values highlighted) and their standard variance.

Threads	2	4	6	8	10	12
mean simple lock	**1.905**	**2.102**	**2.254**	**2.386**	2.438	2.436
σ simple lock	0.084	0.185	0.234	0.269	0.274	0.280
mean local list	1.732	1.913	2.15	2.376	**2.582**	**2.774**
σ local list	0.085	0.163	0.209	0.241	0.290	0.327

[10] We decided to display the results $\pm 2\sigma$ as the data is normal distributed and the two sigma rule indicates that we thereby show about 95 percent of the values that occur.

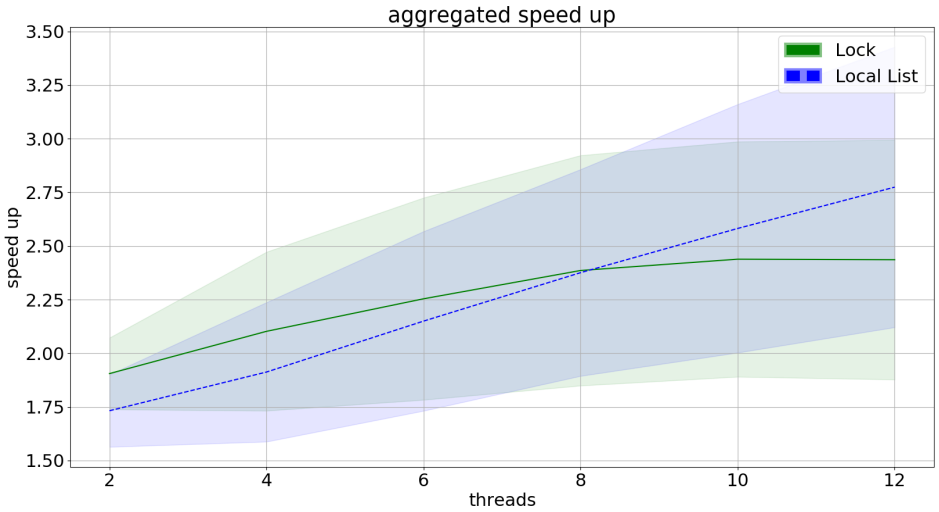


Figure 23: Speed up $\pm 2\sigma$ of both algorithms compared to sequential matching. Aggregated over all considered dimensions, population sizes, and sizes of the match set.

For this first overview we aggregated a lot of results into one graph and table. It leaves out a more fine-grained evaluation of the influence of the magnitudes considered. The number of race conditions that occur have an impact on the program's runtime as we briefly mentioned previously. In our case the number of race conditions correlates with the number of threads and the relative match set size. The higher the threads the more threads might try to acquire the match set lock. If more classifiers match the given situation then more of them need to be inserted.

We directly compare both methods by plotting the difference in terms of speed up in Figure 24 and the corresponding variance in Figure 25. The approach using local lists is becoming more and more superior with a rising number of race conditions (more threads and matching classifiers). On the other hand, if only a few threads are available and the match sets tend to be rather small, the usage of Algorithm 18 can lead to slightly better results on average, but there is also a certain degree of variance in the results. This points to the need of statistical testing. For this an ANOVA analysis is usually suitable, but it requires a homogeneity of the variances and that is not the case here. Hence, we decided to compare each group individually using an unpaired two-sided Student t-test. This requires normally distributed data which we could verify using a Shapiro Wilk test. We test the null hypothesis that the use of a simple lock is superior to the use of local lists for each group. We reject the null hypothesis at a significance level of 0.05. We visualized these results in Figure 26. Here we can see that, for most cases, we cannot reject the null hypothesis. We can verify our hypothesis that our method with local lists becomes more and more superior when the number of race conditions is rising. The usage of local lists is already superior if the match set is about 13 percent the size of the population when we fully exploit our processor.

Tables which contain the corresponding raw numbers are in the Appendix C. In short the speed ups for both algorithms are always above 1.5 and peak at about 3.6. Hence we are always better than the sequential variant.

Overall, out of our scalability experiments we can state that both parallel algorithms lead to a certain speed up. We could verify that when the number of available threads and the approximate match set's size are rising, there is a tendency that the usage of local lists becomes superior. Note, we did not use the most powerful multicore processor. Under the assumption that this effect continues for processors with more cores such as server CPUs, then this algorithm can be superior for even smaller match sets if the maximum number of parallel threads is used. On the other hand, for smaller multicore processors, the usage of a simple lock also leads to a certain speed up. Hence one can choose one of our algorithms depending on the number of available cores whilst still having a certain abstraction from the concrete hardware that is used and having an independence from which kind of LCS is used.

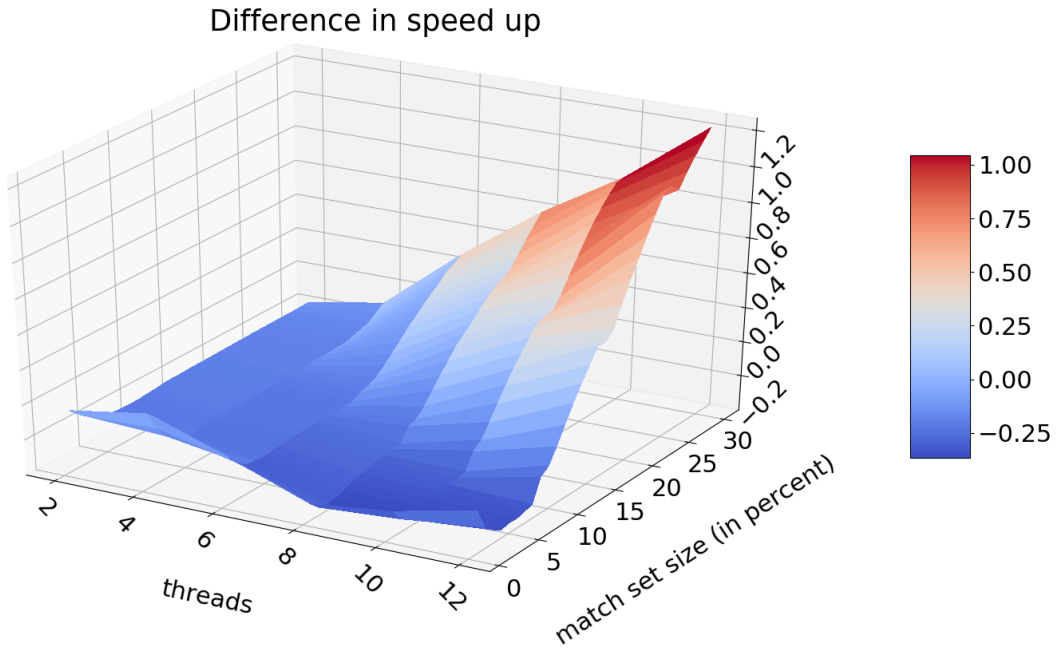


Figure 24: Difference in speed up between Algorithm 19 (local lists) and Algorithm 18 (simple lock) varying over threads and relative match set size.

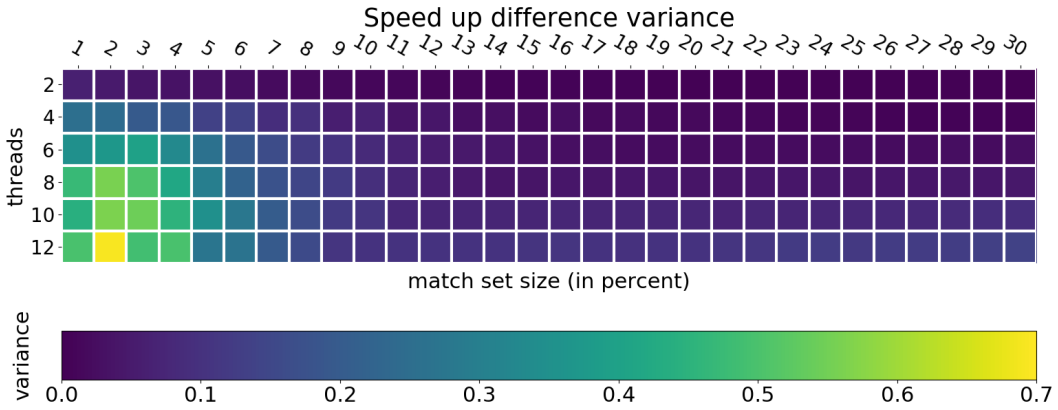


Figure 25: Variance of difference in speed up between Algorithm 19 (local lists) and Algorithm 18 (simple lock).

6.3.2 Variation of the Problem Size

LCS problems can differ in the dimension of the state and the population size may also vary. Hence we focus on the evaluation of our results grouped by the population size and the dimension of the state. Here we are using the maximum number of threads.

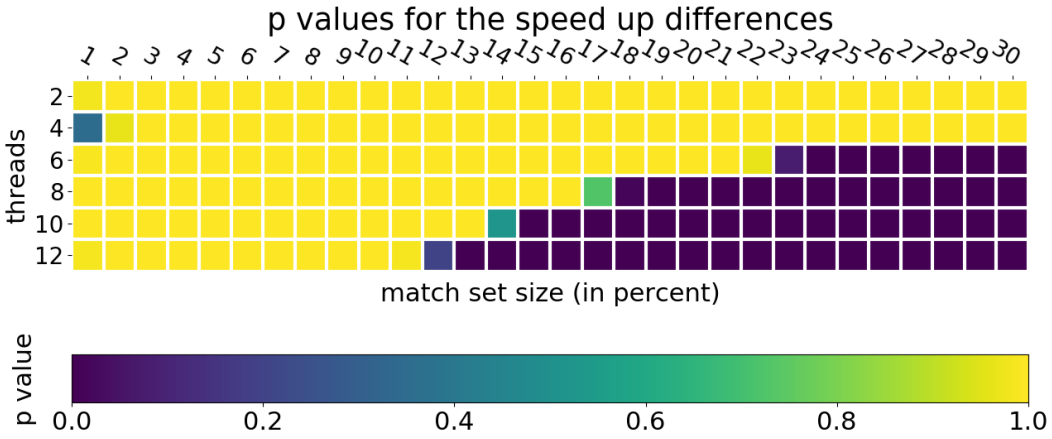


Figure 26: P values of the null hypothesis that Algorithm 18 (simple lock) is superior to Algorithm 19 (local list).

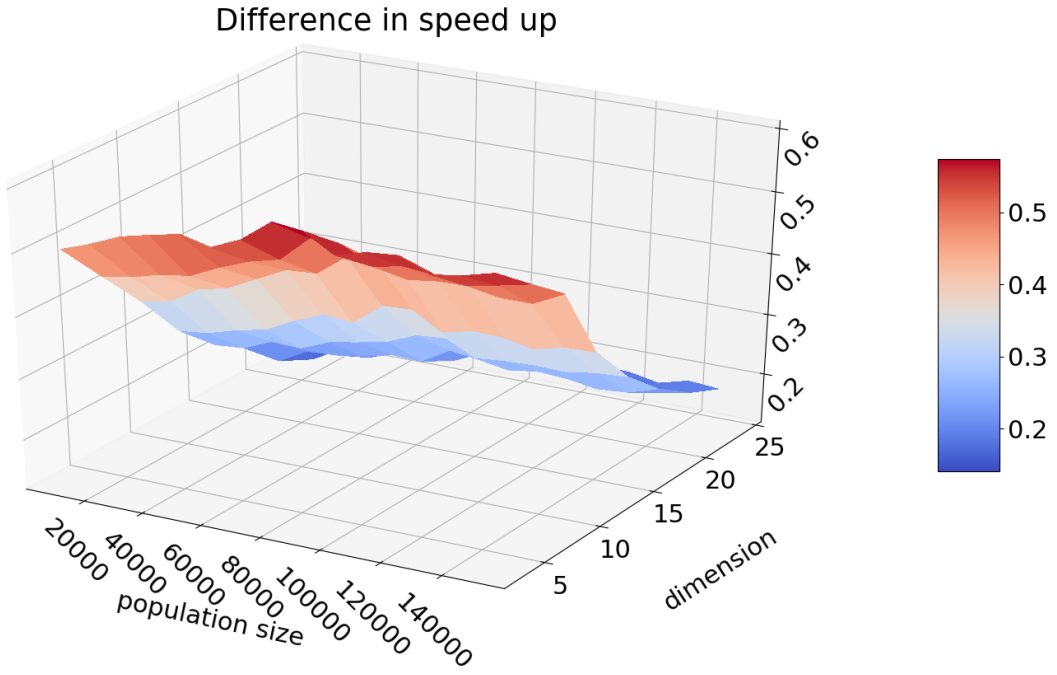


Figure 27: Difference in speed up between Algorithm 19 and Algorithm 18 varying over population size and dimension.

In terms of raw speed ups the results for both algorithms vary between 2.4 and 3. The total overview is once more in Appendix C. Generally both methods benefit from rising population sizes but a rising dimension slows them down. The latter has a higher impact on the lock based approach.

We once more focus on examining the difference between the two introduced methods. The comparison can be seen in Figure 27 and the corresponding variance in Figure 28. There, the method with a simple lock is always inferior to the other one, but the results are not purely deterministic and contain some variance. Thus, we once more prefer additional statistical tests. In order to be consistent to our previous evaluation, we use Student t-tests again. Our null hypothesis for every group is again that Algorithm 18 is superior to Algorithm 19 (significance level of 0.05). Here we could reject the null hypothesis for every group. Thus in this experiment the use of local lists is superior to the use of a simple lock if we fully exploit our processor.

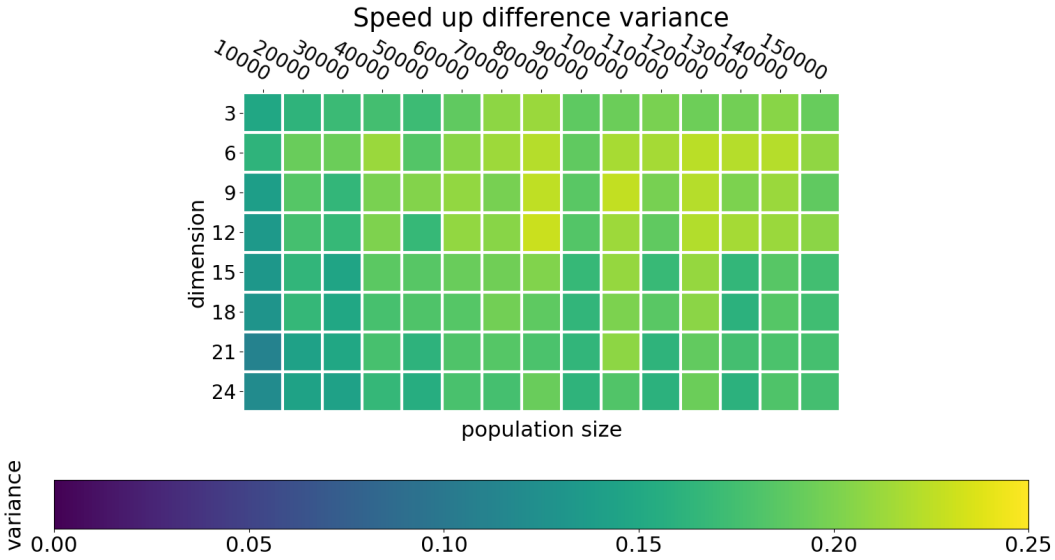


Figure 28: Variance of the difference in speed up between Algorithm 19 and Algorithm 18.

6.3.3 Results in the Context of LCS

Most publications about parallelizing LCS focus on the matching step [126, 128, 134]. Abedini et al. [128] performed some measurements about its proportion and, depending on the problem, it roughly takes between 20 and 80 percent of the total runtime of XCS. Hence, depending on the problem to be solved, there is still a reasonable sequential part of the program. We could achieve speed ups between 2.4 and 3.1 when we fully exploit our CPU. Thus, if we use this approximation of the matching's runtime relative to the total runtime of XCS we would save worst case 11.6 percent of the total runtime and best case 54.19 percent of the total runtime by applying Algorithm 19. This might even be improved if a processor with more cores is used.

In parallel programming, Amdahl's law [138] defines how much runtime improvement for a program is possible. It states if a program has a part that can be parallelized and costs sequentially p time and the other parts cannot or will not be parallelized and cost q time, then the runtime will always be at least q. On the other hand the maximum time that we can save is p. If we put our former numbers in context of Amdahl's law, we achieve a runtime saving of at worst 58.33 and at best 67.74 percent of p.

6.4 Chapter Summary

Several parallelization attempts for LCS exist. All that we found focus on the matching step as it is known to be the computationally most expensive step of this class of machine learning methods. These achieve vast runtime reductions in terms of speed up. However, on the shady side they require a specific hardware accelerator and the expert knowledge to program it. This renders the approaches unfeasible for our testing use case and generally neglects other parallel programming objectives.

We presented two parallel matching algorithms. These are easy to implement in most languages, only having the preconditions that the underlying system has a multicore processor and that the used programming language offers basic functionality for parallel programming. Further our matching algorithms are suitable for most LCS. This enables fast integration into existing LCS.

Our level of abstraction comes at a cost. Approaches that use a specific hardware accelerator, LCS, and work with a system level programming language achieve higher speed ups as they can fully exploit the underlying hardware. However, if we plug our measured speed ups into the estimated matching runtimes of Abedini et al. [128], we can still cut down between 58.33 and 67.74 percent of the matching's runtime for XCS.

We examined the scalability of our algorithms as well. For one of our approaches we could not observe a saturation in terms of speed up with our processor. Our experiments further showed that our approach with a simple lock is suitable for smaller multicore CPUs and if a larger CPU is available, we recommend the usage of the algorithm using local lists. Also we only used a CPU from the medium price sector. We expect our algorithms to perform even better if a superior processor is used. Furthermore, the hardware independence enables us to employ the methods for our CI testing use case.

7 An Organic Computing System for CI

The previous chapter leaves one important issue open. How do we deliver the value created by our RL approach to the customer? From our experience testers are not necessarily software developers or machine learning experts. Hence they might lack the skills to build a software that exploits our ML techniques for their testing process.

Based on the previous discussions regarding CI and testing one can infer that the challenges faced in software testing are similar to the ones identified by the OC community. In testing we also have to deal with an ever changing environment (the software and the test cases at hand are changing) where we have to continuously adapt in order to achieve our goals. OC intends to overcome this challenge by giving agents / subsystems large degrees of freedom in their decision making process. Agents autonomously make local decisions in order to fulfill common goals and are capable of adapting to environmental changes. In testing each CI pipeline can be seen as a subsystem (for different test levels or similar products). Together with the high automatisation that is necessary for a CI process we deem these OC design approaches as desirable.

Within this chapter we discuss how we adapted the MLOC architecture from chapter 2.3. We go layer by layer and show how we used the existing design pattern and where we changed it. We further give insight about other OC ideas which came into the focus of our interest.

7.1 Productive Layer

The SuOC is the DUT which may be anything between a software component and a full product (e. g. a car or a web application). It is interfaced via a test framework such as *Google test* [124]. Engineers write or generate tests that are available as source code. These automated tests are maintained in a repository and can be executed using the corresponding testing framework. It conceptually makes sense to move the overall process of rule generation to this layer (and thus out of layer 1). In context of the original MLOC template, values can often be observed in a sample rate of some clock time, for example every 300 ms. However, here we differ since our time unit is not motivated by physics or embedded systems. Our time axis is the CI cycle.

7.2 Reactive Layer

In layer 1, the observer senses the available test cases and at the end of the CI cycles the results of those executed. Further, it collects CI metadata to give the controller a more detailed knowledge of the available test cases. For each test, it stores its

- **test history**: A vector of previous test outcomes, for example [failed, failed, passed, failed].
- **last execution**: A test might not be executed every CI cycle. The observer thus stores the time step (i.e. the CI cycle) the test has been performed last.
- **execution times**: Encountered durations of the test case which can be used to approximate the test case's duration.

as described in the state space of the test agent (in chapter 5.2).

The observer can estimate the test case's duration based on the execution time. Together with the test history, this signal can be used to estimate whether it is a short test that often fails or whether it is a rather long test that usually passes. The last execution can be used to guide exploration: For example, it allows to check whether a previously passing test that has not been run for quite a while still passes or has begun to fail more frequently.

Our reactive layer contains another abstraction mechanism: it generalizes from the given testing framework. We deem this as necessary since there are different testing frameworks for specific test levels. For example National Instrument's *test stand* [139] is designed to be used on system test level and is capable of integrating various hardware for manipulating and examining the DUT. On the other hand, Google test [124] is developed to be a raw software unit testing framework. The abstraction from the test engine enables the reusability of the system across several test levels and makes its components independent of the test level.

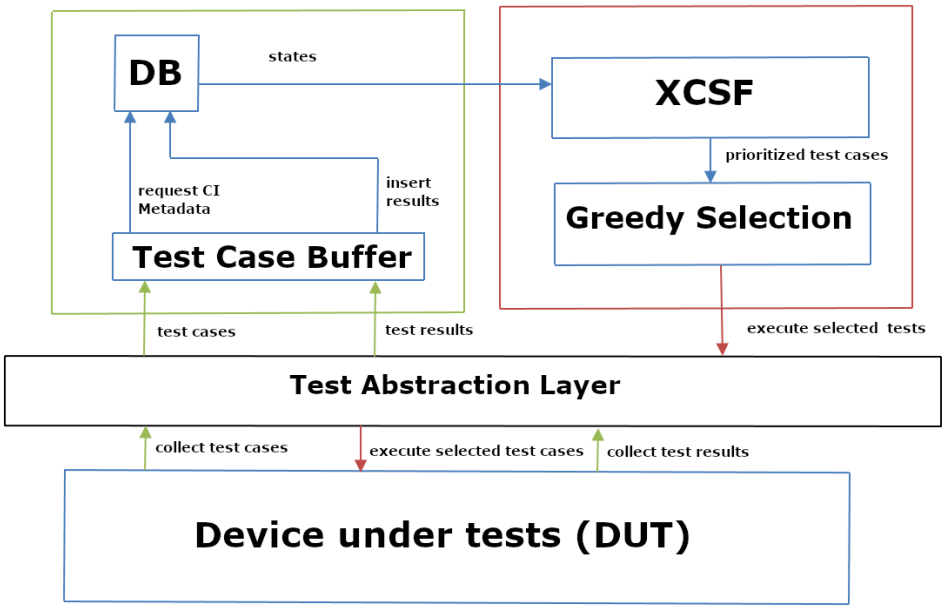


Figure 29: Interaction of layer 1 with the DUT using the test abstraction layer. The green box represents the observer and the red box the controller. Together they form layer 1.

The acquired information about the tests is given to the controller which uses it to compute an appropriate test suite that takes the available execution time into account. We perform our selection using our XCSF-TL variant as an agent and once more follow the workflow described in Figure 15. Hence we first prioritize all tests and then collect the test cases greedily according to the prioritization and the time budget. Afterwards we execute the selected tests.

After executing the test cases, their results as well as additional CI metadata (durations, CI cycle) are retrieved and the observer combines the information and uploads them to a *database* (DB). This makes the data reusable for the succeeding CI cycle. We visualized the entire flow of data and separation of the building blocks of our layer 1 in Figure 29. For the sake of simplicity, we regard the DUT and the test engine as one block.

7.3 Reflection Layer

As mentioned in chapter 2.3, layer 2 usually creates rules for situations where the system lacks experience. We slightly deviate from the template in that our system does not create new rules solely using a metaheuristic and that it does not evaluate them in a simulation. We also create classifiers using the TL and ER approach described in chapter 5.2. Additionally we exploit idle times of the CI build servers that are not under full load then we use the free time to train the system. Thereby we just run another CI cycle. We regard this as more favorable than performing simulations since we prefer real data over simulated data.

In our system, learning is solely performed using ER and TL, both of which are part of layer 2. TL is additionally used when the system is created the first time as we transfer knowledge from a pretrained XCSF instead of creating a new population entirely at random.

The system might further benefit from other testing activities. For example, before shipping software to customers, *all* tests are run. The data created in the process can be used for additional training and evaluation of the quality of the proposed rules. This data has the advantage that the complete knowledge of the outcome of all tests is available (other than during our system's runtime, where only a subset of tests is executed at any time).

We summarized the behaviour of the reflection layer schematically in Figure 30. The match set observer checks if the size of the match set (set of rules that fit to the given state) is too small. If this is the case, it requests matching rules from neighbouring systems (using the collaboration layer)

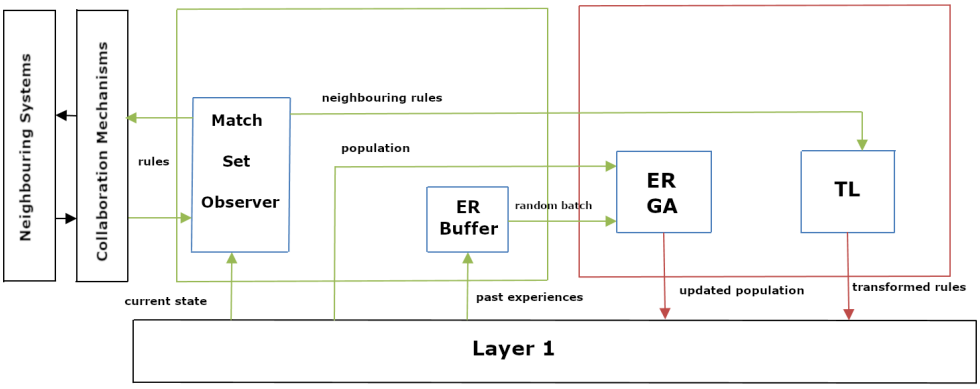


Figure 30: Interaction of layer 2 with layer 1 and neighbouring systems. The green box represents the observer and the red box the controller. Together they form layer 2.

and injects them into the TL component of the controller which inserts the transformed rules into the match set (hence giving them to layer 1). The ER buffer saves past experiences and periodically sends a random batch to the controller in order to refine the rule basis. It is worth mentioning that the third mechanism attributed to layer 2 (exploitation of idle times) is not part of Figure 30. This is due to us following a *keep it simple stupid* (KISS) approach and simply running complete CI cycles during idle times which results in the system not having knowledge of whether the CI cycle's purpose is training or actual application.

7.4 Collaboration Layer

The collaboration layer enables the testers to set two goals. 1) the time budget for the test suite 2) A reward function for the XCSF-based agent.

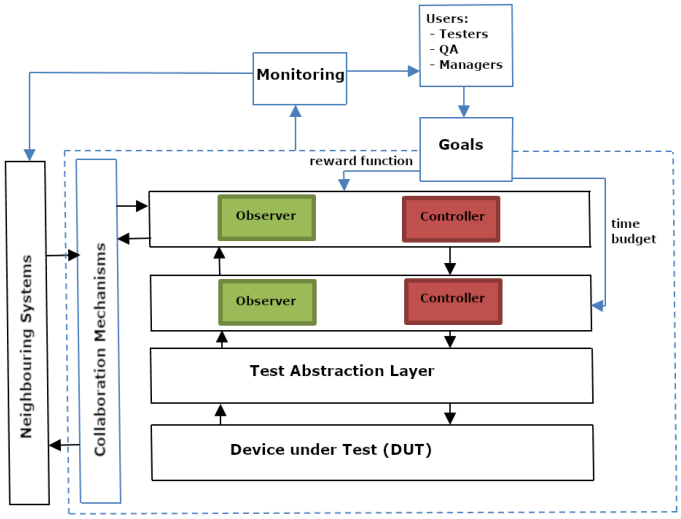


Figure 31: Interaction of the collaboration layer with the remaining parts of the system.

Test engineers are not the only users of the autonomous testing system as described here. Software developers, *quality assurance* (QA), and project managers often need knowledge of the status quo of testing as well. Hence the monitoring of systems such as ours is a part of some kind of overall project reporting. In order to meet this requirement, our system's logging is integrated into a global monitoring system which is realized as a dashboard app.

Collaboration with other systems can be enabled using some form of message-based communication or shared memory. We decided to use the latter since neighbouring systems solely exchange rules. We maintain the populations of individual XCSFs in a software artifact managment system called *Jfrog Artifactory* [140].

We summarized the interaction of the collaboration layer in Figure 31. One can see there that collaboration is limited to the reflection layer. Additionally the reflection layer is the only section whose goals can be set. Further, it displays the stakeholders and our change to a global monitoring system[11]. The content outlined by the dashed lines represents an individual agent which may work autonomously.

It is also worth discussing what a *neighbouring* system in our context is. From a mathematical point of view, neighbouring means closeness in terms of some distance or similarity measure. We instead use closeness from a testing and development perspective. We search the populations of other XCSFs of different variants of the same product[12] and different test levels of it. This simple heuristic limits the search time whilst enabling the reuse of rules. If the reflection layer requests matching rules, an exhaustive search through the neighbourhood is performed after which the rules found are transformed as described in Algorithm 16 and then provided to the reactive layer.

7.5 Autonomy & Self-Organization

In OC, a certain degree of *autonomy* is desired in order to deal with the encountered complexity of the task. A system is regarded as autonomous if its decisions are solely based on an internal control mechanism. Neither full autonomy is desired as the system would become uncontrollable nor full external control is wished (this would lead to a maintainance overhead). This goal is also desirable from a CI perspective since this DevOps practice requires a high amount of automatisation which can be seen as a side-effect of a system with a high degree of autonomy.

OC measures of autonomy relate the number of parameters which are set by the internal control mechanism to the number of parameters set by external influences (e. g. testers that change some goals). In our current implementation the configuration space consists out of the test cases that can be run, the time budget and the reward function to be used.

The test suite to be run is computed by our modified XCSF. If n tests are available then the chosen test suite can be encoded as a bit vector of length n (the i-th entry indicating the i-th test shall be executed or not). Note, that this encoding is common in the testing context [41]. The time budget can be encoded as a floating point variable of constant length c_b and the reward function using an ID which also has a constant size c_r (e. g., if the reward functions are maintained in some enum datatype). Both of these constants are also measured in bits.

In OC, the *static degree of autonomy* α is defined as follows:

$$\alpha = \frac{V_{int} - V_{ext}}{V_{int}} \tag{36}$$

where:

- V_{int} corresponds to the internal variability of the system which in our case corresponds to the number of tests n.
- V_{ext} is the external variability which corresponds to the number of external control variables (here, c_r and c_b) and internal parameters that are changed externally (e. g., by corrective measures).

α has a maximum value of one and a high value indicates a high degree of autonomy.

The externally changed internal parameters are in our use case the test cases manually set or excluded by the test engineers. We could observe that this has, up to now, never been done by our testers. Thus we can compute α as follows:

$$\alpha = \frac{n - c_r - c_b}{n} \tag{37}$$

[11] In the original MLOC variant the monitoring focuses on one isolated agent.

[12] For example Bosch and Siemens home appliances are from the same producer and a Bosch home appliance often shares many things with a Siemens one.

Within the projects of our industrial partner we often observed several hundreds of test cases, sometimes even several thousands. Hence n is by far the dominating factor which leads to a value of α close to one. Therefore, we employed a system with a high degree of autonomy which enables user control, allowing to disable the autonomy in extreme situations. It is worth mentioning that, next to the static degree of autonomy, there exists the *dynamic degree of autonomy* measure [141]. It differs from the static one by only taking the bits into account that are applied at a fixed time t. Within our current projects we could not observe yet that the testers often changed the time budget and the reward function was only set at the start. Thus we can make similar observations for the dynamic degree of autonomy.

In OC autonomy is linked to self-organization [29] (as it often occurs for autonomous, distributed systems). There are multiple ways to define self-organization. One focuses on the number of control mechanisms k, their distribution among the j agents and the agents' degree of autonomy. As seen in the prior paragraph, our system is highly autonomous. Self-organization in an OC context requires at least a partially distributed system (which is true for ours). Each of our agents has two control mechanisms by design (one for the reactive layer and one for the reflection layer), hence $k = 2j$. Systems with these properties are said to be *strongly self-organizing* [29].

Another way of measuring self-organization can be applied if no full knowledge of the designed system is available. In such scenarios, if the communication between the individual elements is observed, communication graphs are constructed and evaluated. However, since we have detailed knowledge of our system, we were able to use the aforementioned, more straightforward approach [29].

It is worth mentioning that self-organisation itself is not a quality indicator. There are both examples of self-organisation that lead to undesired effects or help to achieve a system goal [29]. Here the different agents can only communicate by exchanging experienced rules which we showed is beneficial for the system's performance [16]. Hence we regard this form of self-organisation as rather positive.

7.6 Getting the OC System to the Customer

After performing the systems engineering task of proposing an architecture as well as developing the underlying ML approach, the concrete software has to be written and delivered to the customer. In concrete we did this for the *MASON* test framework of BSH Hausgeräte GmbH. The test framework is built on top of pytest, for test versioning Git (combined with Github) is in use and Jenkins is employed to enable the CI process.

We implemented the presented architecture in Python. We maintain the build software in a company-wide Python package index[13]. This enables test engineers to use the standard package installer called *PIP* to retrieve the software. We distribute it under the name *test_abstraction_layer* and to automatically install the newest version an user only has to execute the following command:

```
pip install test_abstraction_layer
```

Thereby the testers can use the installation tool that they are familiar with. It is worth mentioning that the software is platform independent and can be used on Linux and Windows.

The software can be used as a command line tool which we gave the name *Q_auto* as analogy to Q of the James Bond movies. Q is the scientist that equips agents such as James Bond with the necessary tools to perform their job. This is more or less the same that our software does as it works on, next to the agent role itself, the tasks of cleaning and storing the CI data, using the test framework etc.. The tool can be called as follows:

```
Q_auto <<pipeline_name>> <<CI_cycle>>
```

where:

- **pipeline_name**: The unqiue name of the pipeline, e.g. sw_test_product42.
- **CI_cycle**: The current CI cycle (an integer).

[13]This means you cannot install it from the public package index, but only as a BSH employee.

The pipeline name and CI cycle are mainly used to store the test's outcome in an unique way. Further the XCSF's classifier population is versioned using the pipeline name and CI cycle.

It is worth mentioning that the command line tool also supports additional arguments which correspond to possible settings of the underlying test framework. These commands are just handed over to a call of the test framework and thereby we enable the testers to use the options that they want (e. g. set some test log levels).

Additional necessary configuration information such as the reward function to be used or the time budget are saved in an .ini file of the test git repository. The testing department within BSH specified a location for the file in order to fit their testing process. It is automatically loaded from there. It specifies the time budget and the reward function.

The test engineers within BSH use Jenkins to enable their CI process. Jenkins enables users to specify the steps of a pipeline using a script (similar to batch or shell scripts). This script language enables the integration of our system in the CI process as follows:

1. Copy paste the .ini file template to the git test repository (perhaps adapt the time budget).
2. Run the pip command in order to install the software to the pipeline's test environment.
3. Run the Q_auto command.

Hence an easy integration is achieved by solely copying a file and executing two commands.

In order to give this usage information to the customer we relied on GitHub which is a web application for version control that uses Git. GitHub has the feature to host a documentation as a web page which we used in order to share the documentation. We used a Sphinx documentation as it supports different designs and has a out-of the box search functionality. We provide an example snapshot in Figure 32.

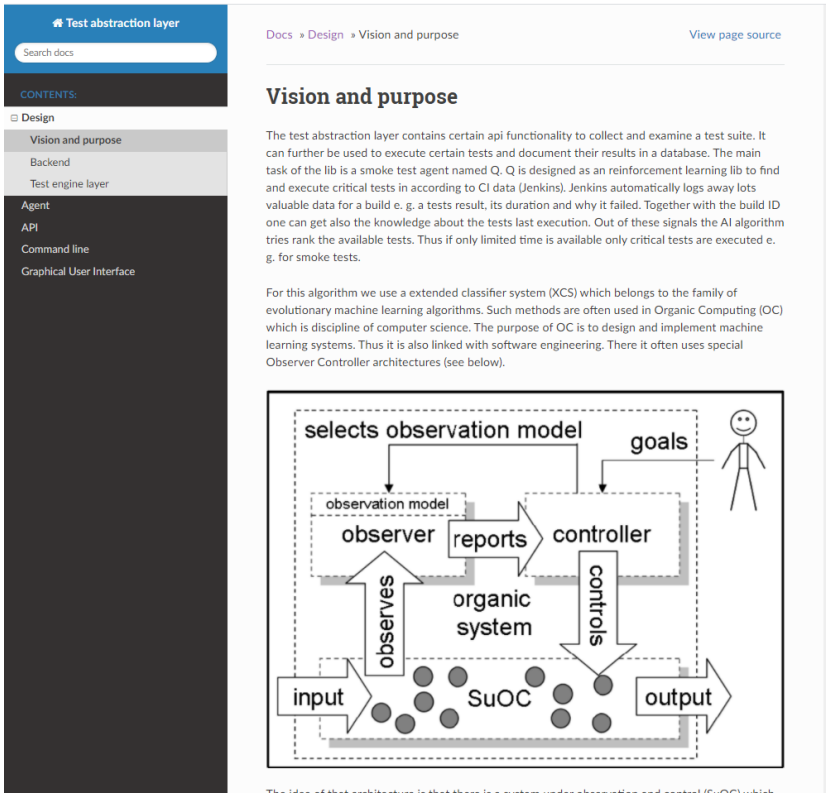


Figure 32: Example snapshot of the first page of our software documentation. On the left side is the table of contents and a searchbar.

The hosted webpage furthermore has the advantage that the newest documentation is always hosted and thereby confusion due to deprecated documentations that are stored locally are avoided.

The documentation is further automatically updated whenever we build a new version of the software (which we also do using a CI process).

We briefly want to mention that we did not publish the source code which implements the system described in this section since the code contains strictly confidential BSH data (e. g. database credentials or other login data). Corporate rules forbid this.

We do not want to confine to showing the documentation and installation procedure. We also want to give a brief, rather informal insight on how the system performs in practice. Therefore we display the performance of our three BSH pilot projects: A dishwasher user interface (DC UI), the system test of a dishwasher (DC ST) and an oven *control and power module* (CPM).

The CPM has tests for about 3 hours (123 tests), the DC ST contains tests for 14 hours (1,869 tests) and the DC UI for about 16 hours (about 40,000 tests). It is worth outlining that the UI has so many tests as it checks various screens in 26 different languages. There the tests are rather short (in contrast to the other two).

We cannot evaluate the system's performance using NAPFD as it requires the knowledge of the outcomes of all possible tests (and not just of the executed ones). Thus we need another measure that may be applied in practice. We measure it using a KPI that we coin the *failure velocity* at cycle i:

$$\frac{\mathcal{TS}_i^{\mathrm{f}}}{C \times \mathcal{TS}_i} \tag{38}$$

which computes the percentage of failed tests (among the executed ones) divided by the employed time budget (C is normalized by the execution time of all tests). Hence it takes an analogy to velocity. However, we do not measure how far we went per time but how many failing tests our system can find in the given time. Thus the KPI takes both objectives into account.

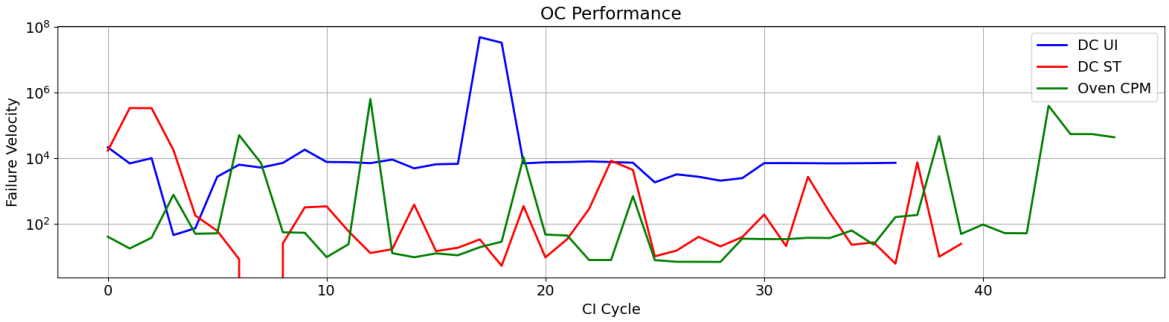


Figure 33: Failure velocity for the three BSH pilot projects.

Figure 33 displays the failure velocity for the three aforementioned projects. It is worth mentioning that the individual graphs end earlier as no newer results are available for them. Further we can see variations over the up to 40 CI cycles observed and upon performance breakdowns the system is capable to recover as we could see in the robustness experiments in chapter 5. Further we can see that the project's structure has an impact on the system's performance which is something that we also observed in the previous evaluation. Here we decided to use a logarithmic scale for our KPI as it is not normalized and might have high values (see DC UI) which on the other hand slightly dampens the visible learning effect of the system. Further in the middle of the timeline the testers decided to downgrade the available testing time (roughly by half) and the system is still capable to have higher values in terms of our KPI. The BSH testers further experimented with the time budget on DC ST by making it smaller and even set it there for a few cycles to a minute (which lead to the break-downs in cycle 8 and 9). Afterwards it was increased again and the system recuperated. However, what one should take from the plot is that the OC system is capable to detect failing tests and is also shows some robustness on this first samples. We keep it at this level as there is definitely a need for a long-term evaluation after the system ran several months.

7.7 Related Work

The work presented directly exploits OC methods which can be traced back to the first observer controller architecture described by Richter et al. [5]. We make use of the refined multi layer observer controller architecture [142]. Our MLOC adaptation differs from the one described by Tomforde et al. [142] as we do not rely on a simulation in the reflection layer and that we perform the entire learning process in that layer. Further we restricted communication with other agents on the reflection layer (exchange of rules). In the OC book [29] it is further proposed to monitor the agents in isolation. However, we further integrated the monitoring in BSH global system in order to ease the review of the big testing picture. The most striking difference is that we employ the MLOC pattern for a non-embedded systems use case.

OC concepts itself are related to another field coined *autonomic computing* (AC) [143]. The AC movement started out as an IBM project in 2003 and focused on solving growing complexity in large-scale datacenters. AC has simultaneously come up with the idea to build adaptive systems which have certain self-x capabilities. The main AC architecture is known as *monitor, analyse, plan, execute* (MAPE) and highly overlaps with observer controller approaches. The monitor and analyse parts are more or less the observer whilst the plan and execute components can be seen as a controller. For a closer discussion of the main differences we refer the reader to [29, 144].

7.8 Chapter Summary

Within this chapter we moved from a pure machine learning point of view to a systems engineering perspective. We outlined that the problems that the test use case implies are similar to the ones that OC tries to solve on an abstract level. Hence we employed OC methods to model a system for this use case which makes use of the ML techniques developed. Thereby we moved beyond the typical embedded system use cases that OC tries to solve. Thus we underlined OC concepts are not necessarily bound to these kinds of systems.

The test system itself relies on the multi-layer observer controller architecture which is a standard pattern in OC [29]. We described how we adopted this approach and where we changed some things. We went layer by layer and elaborated where we put which ML technique that we developed and how we aligned it with the testing use case.

We further discussed the system's degree of autonomy and self-organization capabilities. We underlined that the architecture leads to no detrimental self-organization and furthermore has a high degree of autonomy which fits well to CI since this software development approach requires a high degree of automatization. The latter is a mere side-effect of a high degree of autonomy.

We also gave insight about the concrete software that we delivered to the customer. There we underlined the encapsulation of the system. A test engineer requires no knowledge of OC or ML in order to use it. In order to integrate our software into a pipeline a tester only has to copy paste two lines of code into his pipeline script. Further, a configuration file must be maintained together with the tests itself (there the tester can specify the time budget). Thus we created a black box system which can be quickly integrated in existing CI pipelines within the company. Additionally we gave a first brief insight about the system's performance in practice.

8 Towards Corrective Testing

The previous chapters heavily focused on a test case perspective and how to refine test suites. However, if an issue with the product is detected it still has to be reported to the development team. The engineers then react and try to fix the issue. A proposed fixed version is retested and in case of an unsuccessful repair attempt this entire loop has to be performed again. We visualized this in Figure 34. It would be more elegant if the issue could be fixed automatically.

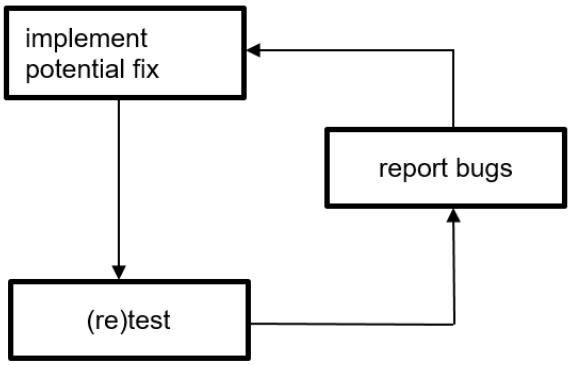


Figure 34: Traditional workflow of testing.

In BSH Home Appliances many software components are configurable and have to fulfill a specified behaviour. This is for example the case for the generic touch interface software used within BSH. It is a configurable signal processing component which detects touch events based on measured capacities. As it is a part of an embedded system it must satisfy regulations of any electric device. This includes for example *electromagnetic compatibility* (EMC) [145]. This set of laws states that a device must function properly even if it is disturbed by electromagnetic noise (to a certain level). We examined if we could successfully automate the entire aforementioned trial and repair loop. Thus a form of *corrective testing* would be created. Thereby we model the task as an optimization problem and rely on a genetic algorithm to solve it. Hence the chapter provides a preliminary study if corrective testing is possible.

8.1 Problem Description

BSH uses capacitive touch which is the most widespread technology with a market share of 74% in 2017 [146]. Capacitive touch roughly works the same way as a plate condensator. The touch sensor is one plate and the human finger the other one. If the finger comes closer to the sensor then the capacity increases. If the finger moves away then the capacity decreases. This capacity signal may be used to detect if the user tapped the touch screen or in other words: A touch event occured.

The *touch interface* (TI) consists out of a microcontroller to scan the capacity on the panel (by computing a score value) and then process the measured signal. Based on it computes if there has been a touch event or not (see Figure 35).

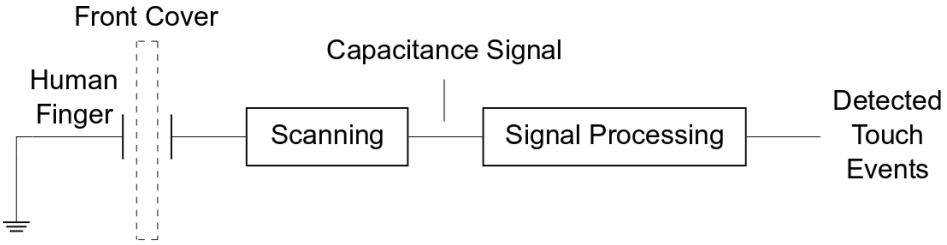


Figure 35: Working principle of a capacitive touch device.

The signal processing part of BSH consists out of a pipeline of operators. These operators have in total 15 different parameters which have to be configured. The BSH touch is a proprietary technology and thus we are not allowed to give a deeper insight.

At time t the TI senses a capacitive signal $\mathbf{C}(t)$. The TI then processes it and is configured with a parameterization $\mathbf{p} \in \mathbf{Z}^{15}$. Note, the parameters are integers as a form of digital signal processing is employed. Further let $b(\mathbf{C}, \mathbf{p})$ be the binary output of the filterchain (touch yes or no). Let $\tau(t)$ be the actual physical touch.

Our test scenario introduces various forms of noise onto $\mathbf{C}(t)$ and leads to a time series of measurements. Let N be the sample size.

A given configuration $\mathbf{p}$ should be capable to detect the introduced touch events which can be measured as follows:

$$TP(\mathbf{p}) = \frac{|\{0 \leq t \leq N | \tau(t) = 1 \wedge \tau(t) = b(\mathbf{C}(t), \mathbf{p})\}|}{|\{0 \leq t \leq N | \tau(t) = 1\}|} \tag{39}$$

The measure TP is an adaption of the true positive rate to the use case.

Additionally the configured filter should detect correctly if there was no touch event at all:

$$TN(\mathbf{p}) = \frac{|\{0 \leq t \leq N | \tau(t) = 0 \wedge \tau(t) = b(\mathbf{C}(t), \mathbf{p})\}|}{|\{0 \leq t \leq N | \tau(t) = 0\}|} \tag{40}$$

TN corresponds to the true negative rate.

Our objective function is based on the two aforementioned measures and can be computed as follows:

$$\frac{TP(\mathbf{p}) + TN(\mathbf{p})}{2} \tag{41}$$

The factor 2 is used for a normalization and thus its values range from 0 (worst) to 1 (best). If a value of 1 is achieved then the calibrated filter pipeline does not miss a single touch event and it correctly recognizes if no touch is introduced onto the device. If a touch event is not detected or a touch event is falsely recognized, the value declines. Thus the objective function is to be maximized. It is worth mentioning that the objective function is also coined *fitness* function in evolutionary computation [97] and is not be confused with fitness of a classifier of an LCS. We follow this naming convention for the rest of this chapter.

8.2 Employed Genetic Algorithm

Genetic algorithms are metaheuristics which rely on a population of solutions. In every iteration they refine the population by selecting two solutions, combining them (which is known as crossover) and then changing them randomly with a certain probability (which is known as *mutation*) [81]. This is done to create two solutions which are then injected to the population. If a predefined population size is exceeded then two solutions are deleted. This process is repeated until a stopping criterion is reached. Here we stop when a fixed search time is exhausted. A corresponding pseudocode is displayed in Algorithm 20.

For our GA we use a k-tournament selection. Thus we draw k random solutions from the population and choose the one with the highest fitness. This is repeated twice in order to get two solutions $\mathbf{x}$ and $\mathbf{y}$ that will be used for the crossover operation. For the latter we use a one-point crossover to generate two new solutions. The operator chooses an integer r from $\{1, 2, .., 15\}$ uniformly at random which serves as a breakpoint. The first child $\tilde{\mathbf{x}}$ receives the first r entries from $\mathbf{x}$ and the last $15 - r$ entries from $\mathbf{y}$:

$$\tilde{\mathbf{x}}_i = \begin{cases} \mathbf{x}_i & i \leq r \\ \mathbf{y}_i & i > r \end{cases} \tag{42}$$

For the second child $\tilde{\mathbf{y}}$ this is reversed (first r entries from $\mathbf{y}$, remaining entries from $\mathbf{x}$).

We apply a creep mutation. Each element gets a new value with a probability of μ. Further, an elitist deletion mechanism is used and thus the elements with the worst fitness are deleted.

We initialize our population entirely at random, but we take the datatype of each filter parameter into account. For example if a parameter is modelled as an unsigned integer which is saved in a byte then we draw a value from 0 to 255 uniformly at random.

Algorithm 20: GA as pseudocode

```
P = create_initial_population()
while stopping criterion is not met do
    Choose x, y from P via selection
    Create x̃, ỹ from x, y via crossover
    Mutate x̃, ỹ using μ
    insert x̃, ỹ to P
    if population capacity exceeds limit then
        perform deletion
end
return one of the best solutions of P
```

8.3 Evaluation

For these experiments we rely both on data acquired in our laboratory as well as on simulated one. Within our first tryouts we observed that the GA is highly sensitive regarding its hyperparameters (for this problem). Thus we start with a dedicated hyperparameter study. We run the optimization in a simulation as the signal processing component is pure software and can also easily be run on a normal computer (and not a microcontroller). Thus whenever the GA proposes a new configuration we run the signal processing with the measured data and proposed parameterization. This has the advantage that the higher clockrate of a desktop computer can be exploited (which is beneficial for the experiment's duration).

We repeat every experiment that we conduct thirty times and represent averaged results. Furthermore we give the GA a search time of two hours. For our experiments we used a Dell OptiPlex XE3 with 32GB RAM and an Intel i7 8700 processor and it was not used for anything else during the evaluation.

8.3.1 Hyperparameter Study

As mentioned before the hyperparameters have a crucial impact on the GA [28]. Thus we decided to evaluate the effect of different hyperparameter combinations on the GA's performance. For k we consider $\{5, 10, ..., 35\}$, for the population sizes $|P|$ $\{250, 500, ..., 1500\}$ and for the mutation probability μ $\{0.01, 0.02, ..., 0.06\}$.

We use an ideal, noise-free simulated signal $\mathbf{C}(t)$. This is done by creating touch events of different lengths. We also introduce phases of different length where no touch has been introduced. The signal is a sequence of 100 touch ON and touch OFF phases. It starts and ends with an OFF phase. The ON phase has a random duration of between 30 and 80 time stamps. We use the 10 starting and 10 ending time steps to let the signal level rise from OFF to ON and from ON to OFF respectively. This is modelled as a linear function, for example for the rising signal:

$$\mathbf{C}(t) = OFF + \frac{ON}{10} * (t - t_0) \tag{43}$$

where t_0 is the start of increasing signal slope. The length of the OFF phase is also created randomly and ranges from 50 to 500 time steps. Thus longer sections where no one interacts with the TI can be simulated. For our test dataset we perform the same generation procedure but only create 10 touch ON and touch OFF phases.

In our later experiments we rely on different data. Thus the hyperparameters that we choose within this study are not overfitted to one of the later datasets.

We give a short overview about the fitness values achieved in Table 24. We created the table based on the average values achieved for each hyperparameter combination. It displays minima, maxima, quartiles, mean values and the standard deviation on the training and test dataset. We can observe a high variety for the achieved fitness values on both datasets (the minima and maxima differ up to 25 percent). This underlines the sensitivity of our GA with respect to its hyperparameters. The majority of the fitness range is covered by half of the considered combinations as we can see by the displayed

Table 24: High level fitness overview. Values rounded to the fifth digit.

	train dataset	test dataset
Maximum	0.94895	0.94961
Minimum	0.68822	0.59832
Mean	0.86883	0.82119
Standard deviation	0.07271	0.06730
1. Quartile	0.81703	0.77135
Median	0.89924	0.83441
3. Quartile	0.92848	0.87174

Table 25: Top five hyperparmeter combinations and their median fitness.

k	μ	$\|P\|$	fitness train	fitness test
35	0.05	750	0.94895	0.93637
30	0.03	750	0.94875	0.93647
30	0.03	250	0.94874	0.93655
30	0.06	1000	0.94839	0.94961
25	0.05	1250	0.94040	0.94788

quartiles. The fitness values of the remaining combinations are closer. This variability can also be seen in the standard deviation which is between 6.7 and 6.9 percent. On the test dataset the mean and the median are similar but on the training dataset there is a gap of 5 percent between the two due to outliers.

The prior analysis lacks the link to the combinations' location in the hyperparameter space. We focus on the top five training fitness combinations due to the high dimensionality (mutation probability, population size, tournament k, fitness value). We displayed these in Table 25. On these we can see that a high fitness value on the training dataset can lead to high values on the test dataset. For example the parameter combination with the highest test fitness is also within the table. The worst of our top five combinations is still better than the aforementioned 3.Quartiles which further underlines the GA's sensitivity. Generally our GA benefits from a high k and thus a longer search through the population to find a parent is being performed. For the mutation probability we can observe medium and high values. For the population size we can notice nearly the full considered range of values. It is worth mentioning that on the top five worst hyperparameter combinations the population sizes and mutation probabilities are low and the tournament k is high. The weak performance might be traced back to a non-diverse population. Based on Table 25 we fixed k to 30 and visualized the fitness

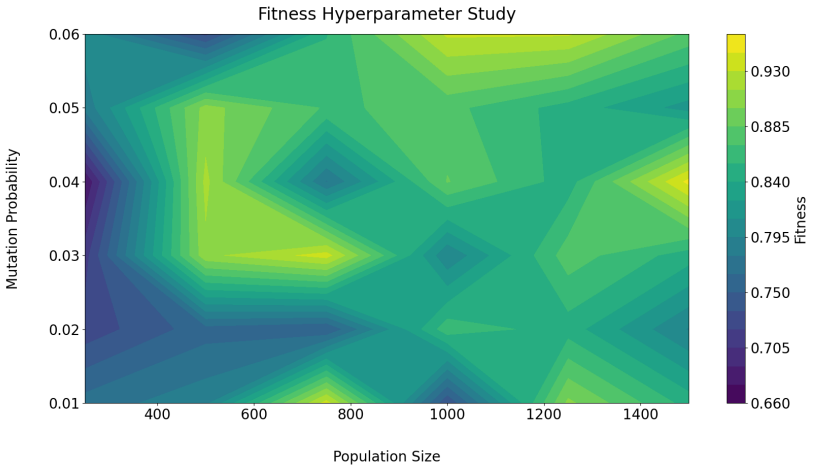
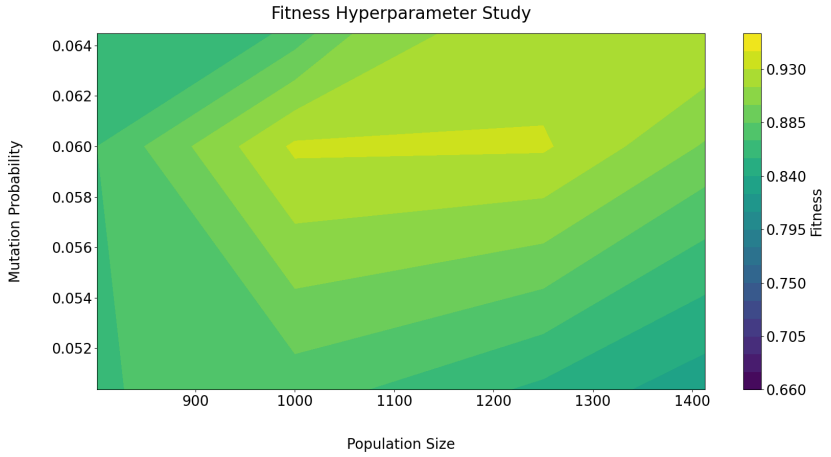


Figure 36: Fitness values on the test dataset for tournament $k = 30$.



(a) Fitness region on the top.

(b) Fitness region on the right.

Figure 37: Fitness at the corner regions.

landscape (on the test dataset) in Figure 36 (as the best test set performance is achieved with $k = 30$ and a good performance on the training dataset can also be observed). There is a small sweetspot for population sizes of 500 to 750. Furthermore we can identify higher fitness plateaus on the right and on the top of the plot. We decided to analyse these regions further. The corresponding plots are shown in Figures 37 (a) and (b). We can see that both regions are isolated local optima and no gain in fitness can be observed if we go further in either direction.

Based on our hyperparameter study we set k to 30, the mutation probability to 0.06 and the maximum population size to 1000 for the succeeding experiments. We decided to use this combination as it achieved the highest fitness value on the test dataset and is close to the best performance on the training dataset (there is only a difference of 0.00056).

8.3.2 Simulated Noise

We extend our setting from the hyperparameter study by introducing noise to the simulated signal. We decided to use *white noise* which is common in signal processing [147] and time series analysis [148].

We employ the noise in our simulation by generating Gaussian distributed random variables $\epsilon(t)$ with mean zero and standard deviation σ and adding them to $\mathbf{C}(t)$. Thus we observe the following noisy signal:

$$\mathbf{C}(t) + \epsilon(t) \tag{44}$$

Table 26: Results for different Gaussian noise and EMC noise. Iteration related metrics are rounded to integers. The remaining measures are rounded to the fifth digit.

	$\sigma = 10$	$\sigma = 15$	$\sigma = 20$	$\sigma = 25$	EMC
fitness of manual calibration (test dataset)	0.96512	0.97109	0.96450	0.96792	0.97081
median fitness GA (test dataset)	0.97763	0.98954	0.99141	0.99249	0.97560
deviation fitness GA (test dataset)	0.15849	0.14793	0.00635	0.08988	0.00851
fitness of manual calibration (train dataset)	0.96495	0.96383	0.96356	0.96013	0.76109
median fitness GA (train dataset)	0.97987	0.98987	0.99127	0.99164	0.97174
deviation fitness GA (train dataset)	0.00401	0.00400	0.00107	0.00131	0.00579
median of iterations until human level	129	142	75	67	67
standard deviation of iterations until human level	81	154	68	65	68

We employ different sigmas $\{10, 15, 20, 25\}$. Hence we consider four different training datasets here. It is worth mentioning that the noise-free signal $\mathbf{C}(t)$ ranges from 0 to 100 in our simulation. Each training and test dataset contains the same number of ON and OFF phases as in the hyperparameter study. Furthermore the durations of each phase are created in the same way.

We display the achieved results in columns 2 to 5 of Table 26. We display the different median fitness values achieved (on both train and test dataset). Additionally the table holds the fitness values achieved by the manual calibration. We can see that if we consider the median value the GA approach always leads to better results on both the training and the test dataset. However, one should keep in mind that the GA does not always have a constant output as can be derived from the non-zero standard deviation. Thus we additionally perform a Wilcoxon ranksum test in order to verify our hypothesis. It is worth mentioning that this statistical test has no preconditions that need to be checked. We measured a p-value below 10^{-30} which we regard as significant and thus we infer that in this experiment the GA's parameters lead to higher fitness values than the manually determined ones.

Table 26 additionally contains information about the number of iterations necessary until the evolutionary algorithm surpasses the human approach's performance on the datasets. Generally we can observe that the GA needs a rather low number of iterations until it exceeds the manual configuration. The magnitude depends on the considered noise level which we confirmed using an additional Friedman test (with a significance level of 0.05).

8.3.3 Electromagnetic Noise

Within our last experiment we switched our focus on real data collected from our lab where we examined a cooktop prototype. We acquired it by setting up a robot cell and noise generator which interacted with the touch interface (introducing various forms of noise, performing touch events and observing the TI). We focused on a noise family which is known as *injected current* in electrical engineering. We decided to concentrate on this noise class as the touch development team regards it as the most challenging one. For details on this test we refer the reader to the corresponding standard [149].

Our EMC dataset consists out of $129,759$ samples and we used the first 90 percent for training and the remaining 10 percent for validation. It contains 202 ON and OFF phases.

We display the results in the column "EMC" of Table 26. Again we can observe that our GA leads to more precise parameters on both the training and the test dataset. We once more perform an additional Wilcoxon ranksum test to verify this hypothesis. We examine the nullhypothesis "The manual configuration is superior to the configurations found by the GA on the training and the test dataset". We computed a p-value of less than 10^{-8} which we regard as significant. Thus we reject the nullhypothesis and accept the alternative hypothesis that the GA leads to more precise calibrations. It is worth mentioning that this low p-value is not only due to the performance gap on the training dataset. If we would examine both datasets in isolation then we could still observe a p-value of less than 10^{-6} on the training dataset and of about 0.01 on the test dataset.

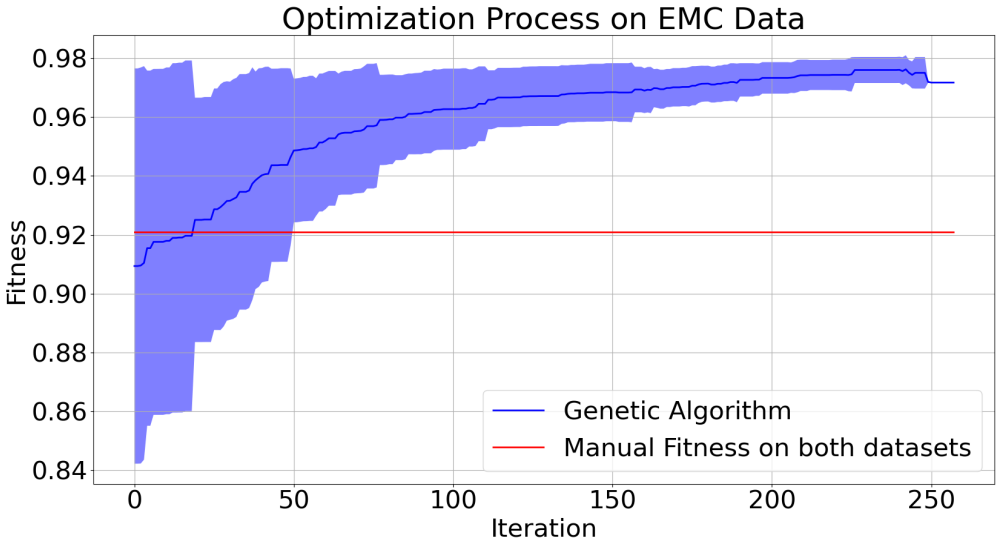


Figure 38: Learning process of the GA plus minus standard deviation on the EMC data. The plot further contains the fitness of the manual calibration on both the training and validation datasets.

We already mentioned a considerable performance gap between the manual configuration and the ones found by our GA if the training dataset is considered. There is a sequence of touch events in the dataset with high noise levels that the TI using the manually determined parameters fails to recognize. However, this is not the case if we employ the parameters found by our GA (the TI is capable of detecting these events using the GA's calibration).

For our EMC experiment we can see once more that the GA outperforms the human at this task rather quickly. The median iterations necessary to achieve this is less than one hundred. However, we decided to give a more fine-grained overview on the learning process in Figure 38. It displays the average fitness plus minus its standard deviation at each step. We additionally added the performance of the manually determined calibration (on the training- and test dataset) to the plot.

We can observe a large variation at the beginning which is gradually declining over time. This is due to the creation of several hundred random solutions for initialization of the population. This might lead to starting solutions that already have decent fitness values. Generally the GA already starts at rather high fitness values (of about 90 percent) which it can gradually improve. Furthermore we can see that there is some variation in the number of iterations performed in the two hours (as there is a fitness drop after about 240 iterations which does not happen if one run of our elitist GA is evaluated in isolation). Even though the GA has a search time of two hours it only performs up to 250 iterations per search. This is due to the computational cost of the fitness function. Whenever we want to evaluate a solution we have to process about 117,000 samples of the training dataset. For each sample the signal processing component must be executed. Thus an evaluation of the fitness function costs several seconds execution time.

8.3.4 Qualitative Remarks

In our experiments we could show that our method is capable of outperforming a manual approach and we could achieve recognition rates of 97 to 99 percent on the respective validation set. However, we did not discuss if these seemingly high values can be considered as good or not. Thus we take a closer look at the performance achieved.

For most samples we observed the behaviour shown in Figure 39. The plot displays the output of the configured filter chain as well as the touch events detected using the force sensor which was attached on to the robot in our lab (to measure $\tau(t)$). In this example we can see that our configuration detects touch events correctly but prolongs them. The examined filterchain contains a *low pass filter* to tackle noise [150]. This filter element has the side effect that it delays the signal slightly which

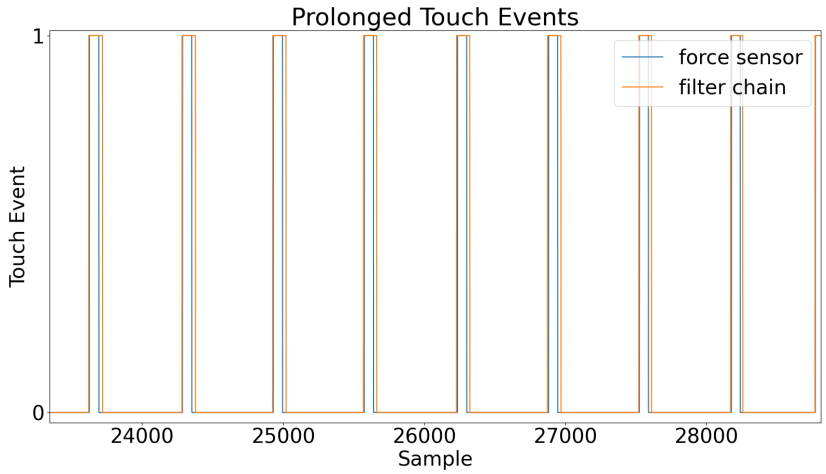


Figure 39: Example of prolonged touch events due to a low pass filter (on the EMC data).

explains this observation. Our fitness level works at the sample level which explains why we do not achieve a recognition rate of 100 percent. We sample fast (at about 10 ms) and thus we deem these prolonged touch events as acceptable. In fact the highest prolongation that we found was about 150 ms.

It is worth mentioning that the parameterization provided by the GA cannot deal with unlimited noise. If the noise level becomes too high then we observed that our configuration may miss out touch events. However, we made similar observations with the manual calibration that fulfills the industry standard [149]. Thus we focused on a setting that satisfies the industry standard within the experiments.

8.4 Related Work

The use case presented can also be seen as a calibration task. There is already a variety of work on calibrations using metaheuristics available [151–154]. Many of them rely on GAs. Due to their success we decided to use one of them. A curious reader of previous chapters might formulate the question why we did not use a germinal center artificial immune system here. The heuristic is designed to work on solutions that are encoded as binary vectors (entries that are either 0 or 1) which is just not the case here.

GAs have also been employed for various signal processing tasks such as active noise control or early forms of speech recognition [155]. Their work shares some similarity as they use a GA to fine-tune single filter components, but these are only using a few configurable parameters (up to 5). We optimize several filter components in parallel which have to interact with each other. Optimizing several interacting elements is a challenging task as shown in the study of Doerr et al. [156] who focus on fine-tuning several PID controllers.

The idea of corrective testing is deeply related to the OC concept of *self-healing* [29]. A self-healing system can recognize a flaw and perform corrective measures. A key distinction is rooted in the system boundary. Here the corrective action has an external source and not an internal one. Another difference is in the time when corrective measures take place. Here it is within the development stage and for self-healing systems it is throughout the system's lifetime.

8.5 Chapter Summary

Within this chapter we opened up the research direction of corrective testing. Thereby we understand that testing goes beyond only finding the errors. It should also repair the product at hand and hence reduce the stress which is put on the development team.

Based on our industry experience we saw that many software components are frequently reused and only need to be reconfigured to fit for example a new hardware or use case. An example for this

the BSH touch software which is used in all of its appliances ranging from cooktops over washing machines to coffee machines. We modelled the testing and configuration of the touch device as an optimization problem and employed a GA to the task.

In our experiments we could observe that GA was capable of detecting bad configurations and was able to manipulate the parameterization until the touch interface was functioning as desired and could fulfill industry standards. Furthermore, the configurations performed better than the ones detected manually. Thus we see this small study as a first proof of concept that corrective testing is possible. It is worth mentioning that the project itself is continued by BSH corporate innovation.

9 Conclusion

Within this chapter we briefly revisit the thesis and discuss its results. Furthermore an outlook about future research directions and engineering applications is given.

9.1 Summary

We started our thesis by making an observation about how products have changed throughout the last decades. We could observe a move from mechanical components to more and more software and electronical ones. This leads to more challenges for software development, especially in software testing. These novel features cause an increase in the number of tests that need to be specified, implemented and executed. The thesis focused on how this complexity can be handled using artificial intelligence.

We started our journey by examining test specifications. These contained the information which requirements are covered by a test case. We aimed at identifying redundant tests which are the ones whose requirements are already covered by other tests. It turned out that the underlying optimization problem is the minimum set cover problem which is known to be NP-hard. Thus we looked at the technical state of the art regarding MSCP approximation algorithms. We reimplemented and analysed a series of corresponding metaheuristics. Thereby we encountered vast differences in terms of runtime and approximation quality (some might even have an exponential runtime). We chose the most promising one (GCAIS) and improved its algorithmic structure to overcome its problems. Thereby we could reduce its runtime and memory consumption drastically. We employed our improved variant to test specifications for costly manual tests of BSH Home Appliances. Thereby we could identify a considerable number of redundant tests which do not need to be executed.

We continued by taking a look at semi-automated testing. There test engineers are closely observing the system under test (in our case embedded systems). Thereby they rely on a subset of all automated tests. We developed a multi-criteria selection approach which builds upon the GCAIS from the test specification use case. We deemed this as desirable since one of the objectives is mathematically a variation of the MSCP on which GCAIS performs so well. We employed the metaheuristic to find small test suites that are capable to detect errors and cover a variety of functionality. Algorithmically we improved its initialization approach by introducing a multi-axis initialization which leads to a diverse starting population. We evaluated our modified approach in a series of experiments using industrial data and there we could prevail against a variety of other approaches including a NSGA-II variant specialised for the problem at hand. However, the search time was high (due to the number of tests) which we deem as problematic for a fully automated approach. In order to overcome this issue we developed a test case selection GUI where the tester may perform a preselection (based on their expert knowledge). GCAIS assists the selection by computing a test suite on the reduced search space. The aforementioned GUI has been delivered to the BSH oven testing team who work in a semi-automated way.

We shifted our focus to automated testing as part of a continuous integration process. CI has become a crucial part of agile software development and focuses on frequent integration of each developer's source code in order to ensure quality. Thereby a set of crucial test cases has to be computed which has a bounded execution time (in order to have a quick insight into the software at hand). This can be achieved by prioritizing and selecting test cases accordingly. A novel paradigm to achieve this is to employ reinforcement learning and a state of the art method is an artificial neural network approach which has also found its way into major companies such as Netflix. The used neural network, datasets and experiments are publicly available. We evaluated LCS as an alternative approach and relied on the aformentioned datasets and source code. This we could compare, ceteris paribus, with the neural network. We iteratively improved our LCS approach and achieved the following milestones (in terms of machine learning):

1. A first attempt using XCS which showed that LCSs have potential for the use case.

2. We moved to continuous actions (priorities) with the usage of XCSF. In order to enable this we developed a simplistic heuristic for the agent's policy which is based on the approximation of a value function. This improved the performance significantly.

3. An open point of the pure XCSF approach was that the correct reward function is not known a priori which makes it hard to use it in practice. We overcame the issue by incorporating experience replay and based on that we are able to recommend a reward function.

4. We were the first to be capable to reuse the test prioritization knowledge gained for one project to another. We achieved this by developing a straightforward transfer learning approach for XCSF's population.

The final LCS empirically outperforms the neural network which was originally proposed. We also evaluated the first industrial data from BSH and could see that our approach also outperforms the neural network on this dataset.

We moved beyond traditional KPIs for test case prioritization and took a look at quality measures from OC. We evaluated the approach's active robustness which is a form of recovery speed that is measured when the performance breaks down. We further examined the duration until the agent has recovered. We extended the set of OC KPIs by introducing a frequency measure in order to evaluate how often these events take place. All the aforementioned measures depend on a target performance level and we evaluated all possible values and could identify a vast range of utilities where our agent is robust, the frequency is low and its recovery durations are low.

We further analysed how LCS can be made faster in terms of runtime. For our use case we do not know the target hardware architecture. Hence we developed and evaluated two parallelization algorithms which may run on any computing device which has a multicore processor. We focused on the computationally most intense step, the matching, and could achieve runtime cuts for a vast range of problem structures. We were able to reduce the cost of computing a priority to less than 20 ms which enables us to quickly compute a test suite. The approach can be used for most types of LCSs.

A machine learning approach alone does not generate any value. It must be wrapped and delivered as an usable piece of software. A key part of any software is its architecture which must carefully consider the challenges and requirements that arise from the problem at hand. We observed that due to agile processes the software underlies constant changes and so do the tests. Hence the agent lives in an ever changing environment and the prioritization must be continuously adapted. OC is a systems engineering discipline that focused on use cases which face these challenges. It has mainly focused on embedded systems and not testing. However, many of the concepts such as the design approaches are independent from embedded systems. This enabled us to rely on these ideas. We wrapped our agent in a slightly adapted MLOC architecture in order to get our innovation to the tester. We further gave insight in how we built and delivered the final software product to the test engineers in BSH. It can be used as a command line tool which can be installed using a one-line command. Further, it is configured using an .ini file which specifies general things such as the available execution time. The tester requires no knowledge about RL or OC. Thereby we delivered an easy to use product. As a side-effect we showed that concepts of OC are not limited to embedded systems.

We further took a first step beyond traditional testing which only concerns itself with a test case level. We examined if we could already include corrective measures in order to enhance the degree of automatization. For a first proof of concept we examined the generic BSH touch software. We were able to automatically test and correct the examined product (a cooktop) until it behaved as specified.

9.2 Outlook

From a pure scientific point of view the open source mentality of this thesis enables other researches to benchmark their novel approaches against the ones presented. However, it also enables others to extend and improve our methodologies. We determined the following possible directions:

- There has not yet been a parallelization of GCAIS. Further Skyline research might be employed to perform a successful parallelization. For example the divide and conquer approach mentioned by Börzsönyi [99] can serve as a basis. In the long run this might reduce the search time.

- Stein [119] showed in a series of experiments that interpolation can be beneficial for LCS such as XCS(F). This might be the case for ATCS as well if the correct form of interpolation is used. There are also the first steps to combine interpolation with ER in order to boost the learning process [157]. A similar approach could lead to advances here.

- The performance of the XCSF variant presented is limited by the amount of information which the signal holds (in our case the test case metadata). The performance might be enhanced by feeding XCSF with additional information (e. g. software version control history to give an insight about which software components are affected).

It is further worth mentioning that BSH corporate innovation has taken over the touch interface project where we made a first proof of concept regarding corrective testing. They plan to enhance the solution to further include other criteria such as the accuracy of the position (of the touch event detected).

From an engineering perspective we are currently rolling out our OC system into the global test process of BSH. We started by integrating the system into three pilot projects and based on the insight of these first attempts we iteratively extend and improve the software at hand. Thus the future focus is more on the pure engineering part.

10 Acknowledgement

I received valuable input, feedback and help from various persons. Here I want to express my gratitude to the following people from my life as an engineer:

- *Bernhard Kobl* for giving me the chance to pursue this project and supporting the idea.
- My friend and mentor *Torsten Eckardt* for coming up with the idea of focusing on testing and leading me through the project from an industrial point of view.
- *Maximilian Beitinger, Florian Brandstetter* and *Daniel Roger* for being willing to try out the CI software and giving my valueable feedback and feature requests.
- *Dr. Asier Lacasta Soto* for giving me initial research directions.
- My former boss *Bernd Heinisch* and my current boss *Daniele Dall'Acqua* who gave me the necessary time to pursue the presented research. Especially the latter helped me to improve my presentation skills.
- *Dr. Martin Lešo* for giving me periodic feedback on the status of my research.
- *Daniel Gerber* and *Johannes Maier* for our common work on the touch calibration research.

I also want to thank the following people from my life as a scientist:

- My supervisors *Prof. Dr. Jörg Hähner* and *Jun.-Prof. Dr. Anthony Stein* for teaching me how research in computer science is conducted and leading me through this journey from an academic point of view.
- *David Pätzel* for giving me critical feedback and improvement proposals for all CI papers. Thereby I learned a lot about how publications are properly written.

References

[1] E. W. Dijkstra, "On the nature of Computing Science," https://www.cs.utexas.edu/users/EWD/transcriptions/EWD08xx/EWD896.html, 1984, [Online; accessed 5-April-2022].

[2] Giusca, Bogdan, "Königsberg Bridges," https://commons.wikimedia.org/wiki/File:Konigsberg_bridges.png, 2005, [Online; accessed May 14, 2021].

[3] T. Hagerup, "Algorithmen für NP-harte Probleme," 2017, "University of Augsburg lecture".

[4] D. P. Williamson and D. B. Shmoys, *The Design of Approximation Algorithms*. Cambridge University Press, 2011.

[5] U. Richter, M. Mnif, J. Branke, C. Müller-Schloer, and H. Schmeck, "Towards a generic observer/controller architecture for organic computing," in *INFORMATIK 2006 – Informatik für Menschen, Band 1*, C. Hochberger and R. Liskowsky, Eds. Bonn: Gesellschaft für Informatik e.V., 2006, pp. 112–119.

[6] H. Spieker, A. Gotlieb, D. Marijan, and M. Mossige, "Reinforcement learning for automatic test case prioritization and selection in continuous integration," *CoRR*, vol. abs/1811.04122, 2018.

[7] L. Rosenbauer, A. Stein, H. Stegherr, and J. Hähner, "Metaheuristics for the Minimum Set Cover Problem: A Comparison," in *Proceedings of International Joint Conference on Computational Intelligence*, 2020.

[8] L. Rosenbauer, A. Stein, and J. Hähner, "A Germinal Center Artificial Immune System for Software Test Suite Reduction," in *LIFELIKE Computing Systems Workshop, 8th Edition in the Evolution of the Series of Autonomously Learning and Optimizing Systems (SAOS), at Artificial Life Conference, 16 July 2020, held entirely virtual due to COVID-19 as part of ALIFE 2020, Montréal, Canada*, 2020.

[9] L. Rosenbauer, A. Stein, and J. Hähner, "An Artificial Immune System for Adaptive Test Selection," in *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*, 2020, pp. 2940–2947.

[10] L. Rosenbauer, A. Stein, and J. Hähner, "An Artificial Immune System for Black Box Test Case Selection," in *EvoCop: 21st European Conference on Evolutionary Computation in Combinatorial Optimisation as part of evostar 2021, April 2021, Seville, Spain*, 2021.

[11] L. Rosenbauer, A. Stein, and J. Hähner, "Generic Approaches for Parallel Rule Matching in Learning Classifier Systems," in *Proceedings of the 2020 Genetic and Evolutionary Computation Conference, Cancún, Mexico, July, 2020*, C. A. C. Coello, Ed., 2020.

[12] A. Stein, R. Maier, L. Rosenbauer, and J. Hähner, "XCS Classifier System with Experience Replay," in *Proceedings of the 2020 Genetic and Evolutionary Computation Conference, Cancún, Mexico, July, 2020*, C. A. C. Coello, Ed., 2020.

[13] L. Rosenbauer, A. Stein, R. Maier, D. Pätzel, and J. Hähner, "XCS as a Reinforcement Learning Approach to Automatic Test Case Prioritization," in *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, ser. GECCO '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1798–1806.

[14] L. Rosenbauer, A. Stein, D. Pätzel, and J. Hähner, "XCSF for Automatic Test Case Prioritization," in *Proceedings of the 12th International Joint Conference on Computational Intelligence (ECTA), November 2-4, 2020*, J. J. Merelo, J. Garibaldi, C. Wagner, T. Bäck, K. Madani, and K. Warwick, Eds., 2020.

[15] L. Rosenbauer, A. Stein, D. Pätzel, and J. Hähner, "XCSF with Experience Replay for Automatic Test Case Prioritization," in *2020 IEEE Symposium Series on Computational Intelligence (SSCI), virtual event, Canberra, Australia, 1-4 December 2020*, H. Abbass, C. A. C. Coello, and H. K. Singh, Eds., 2020.

[16] L. Rosenbauer, D. Pätzel, A. Stein, and J. Hähner, "Transfer Learning for Automated Test Case Prioritization using XCSF," in *EvoApplications: 24th International Conference on the Applications of Evolutionary Computation as part of evostar 2021, April 2021, Seville, Spain*, 2021.

[17] L. Rosenbauer, D. Pätzel, A. Stein, and J. Hähner, "An Organic Computing System for Automated Testing," in *Architecture of Computing Systems – ARCS 2021*, L. Bauer and T. Pionteck, Eds. Cham: Springer International Publishing, 2021.

[18] L. Rosenbauer, A. Stein, and J. Hähner, "A Genetic Algorithm for HMI Test Infrastructure Fine Tuning," in *Proceedings of the 18th International Conference on Informatics in Control (ICINCO), July 6-8, 2021*, K. Madani, O. Gusikhin, and H. Nijmeijer, Eds., 2021.

[19] L. Rosenbauer, J. Maier, D. Gerber, A. Stein, and J. Hähner, "An Evolutionary Calibration Approach for Touch Interface Filter Chains," in *Proceedings of the 18th International Conference on Informatics in Control (ICINCO), July 6-8, 2021*, K. Madani, O. Gusikhin, and H. Nijmeijer, Eds., 2021.

[20] M. Kshirsagar, K. Gupt, L. Rosenbauer, C. Ryan, and J. Sullivan, "Hierarchical Clustering Driven Test Case Selection in Digital Circuits," in *Proceedings of the 16th International Conference on Software Technologies - ICSOFT*, L. Maciaszek, H.-G. Fill, and M. van Sinderen, Eds. SciTePress, 2021.

[21] E. Dustin, J. Rashka, and J. Paul, *Automated Software Testing: Introduction, Management, and Performance*. USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[22] G. Fraser and F. Wotawa, "Redundancy Based Test-Suite Reduction," in *Fundamental Approaches to Software Engineering*, M. B. Dwyer and A. Lopes, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 291–305.

[23] Y. Yu, J. A. Jones, and M. J. Harrold, "An Empirical Study of the Effects of Test-Suite Reduction on Fault Localization," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 201–210.

[24] H. Hsu and A. Orso, "MINTS: A general framework and tool for supporting test-suite minimization," in *2009 IEEE 31st International Conference on Software Engineering*, 2009, pp. 419–429.

[25] V. D'Silva, D. Kroening, and G. Weissenbacher, "A Survey of Automated Techniques for Formal Software Verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 7, pp. 1165–1178, 2008.

[26] D. Todaro, *The Epic Guide to Agile: More Business Value on a Predictable Schedule with Scrum*. R9 Publishing LLC, 2019.

[27] C. Larman and B. Vodde, *Scaling Lean & Agile Development: Thinking and Organizational Tools for Large-Scale Scrum*, 1st ed. Addison-Wesley Professional, 2008.

[28] T. Back, D. B. Fogel, and Z. Michalewicz, *Handbook of Evolutionary Computation*, 1st ed. GBR: IOP Publishing Ltd., 1997.

[29] C. Müller-Schloer, H. Schmeck, and T. Ungerer, *Organic Computing - A Paradigm Shift for Complex Systems*, 1st ed. Berlin, Heidelberg: Springer-Verlag, 2011.

[30] S. Rudolph, "Mutual Influences in Self adaptive and Autonomously Learning Systems," Ph.D. dissertation, University of Augsburg, 01 2020.

[31] H. Prothmann, H. Schmeck, S. Tomforde, J. Lyda, J. Hähner, C. Müller-Schloer, and J. Branke, "Decentralised Route Guidance in Organic Traffic Control," *Proceedings - 2011 5th IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2011*, 10 2011.

[32] S. W. Wilson, "Classifier Fitness Based on Accuracy," *Evolutionary Computation*, vol. 3, no. 2, pp. 149–175, 1995.

[33] A. Joshi, J. Rowe, and C. Zarges, "An Immune-Inspired Algorithm for the Set Cover Problem," in *Parallel Problem Solving from Nature – PPSN XIII*. Cham: Springer International Publishing, 2014, pp. 243–251.

[34] A. Rauf, A. Jaffar, and A. Shahid, "Fully Automated GUI Testing and Coverage Analysis using Genetic Algorithms," *International Journal of Innovative Computing, Information and Control*, vol. 7, 06 2011.

[35] J. Zander, I. Schieferdecker, and P. J. Mosterman, *Model-Based Testing for Embedded Systems*, 1st ed. USA: CRC Press, Inc., 2011.

[36] B. K. Ozkan, R. Majumdar, F. Niksic, M. T. Befrouei, and G. Weissenbacher, "Randomized Testing of Distributed Systems with Probabilistic Guarantees," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, Oct. 2018.

[37] R. Huang, W. Sun, Y. Xu, H. Chen, D. Towey, and X. Xia, "A Survey on Adaptive Random Testing," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[38] W. Li, F. Le Gall, and N. Spaseski, "A Survey on Model-Based Testing Tools for Test Case Generation," in *Tools and Methods of Program Analysis*, V. Itsykson, A. Scedrov, and V. Zakharov, Eds. Cham: Springer International Publishing, 2018, pp. 77–89.

[39] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, and M. Harman, "Chapter Six - Mutation Testing Advances: An Analysis and Survey," in *Advances in Computers*, A. M. Memon, Ed. Elsevier, 2019, vol. 112, pp. 275 – 378.

[40] R. Lachmann, M. Felderer, M. Nieke, S. Schulze, C. Seidl, and I. Schaefer, "Multi-Objective Black-Box Test Case Selection for System Testing," in *Proceedings of the Genetic and Evolutionary Computation Conference*, ser. GECCO '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 1311–1318.

[41] S. Yoo and M. Harman, "Regression Testing Minimization, Selection and Prioritization: A Survey," *Softw. Test. Verif. Reliab.*, vol. 22, no. 2, p. 67–120, 2012.

[42] R. Azzouz, S. Bechikh, and L. Ben Said, *Dynamic Multi-objective Optimization Using Evolutionary Algorithms: A Survey*. Cham: Springer International Publishing, 2017, pp. 31–70.

[43] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-Based Software Engineering: Trends, Techniques and Applications," *ACM Comput. Surv.*, vol. 45, no. 1, Dec. 2012.

[44] I. Dinur and D. Steurer, "Analytical Approach to Parallel Repetition," in *Proceedings of the Forty-sixth Annual ACM Symposium on Theory of Computing*, ser. STOC '14. New York, NY, USA: ACM, 2014, pp. 624–633.

[45] V. Garousi, S. Bauer, and M. Felderer, "NLP-assisted software testing: A systematic mapping of the literature," *Information and Software Technology*, vol. 126, p. 106321, 2020.

[46] H. Hemmati and F. Sharifi, "Investigating NLP-Based Approaches for Predicting Manual Test Case Failure," in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, 2018, pp. 309–319.

[47] B. Li, C. Vendome, M. Linares-Vásquez, D. Poshyvanyk, and N. A. Kraft, "Automatically Documenting Unit Test Cases," in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2016, pp. 341–352.

[48] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of Duplicate Defect Reports Using Natural Language Processing," in *29th International Conference on Software Engineering (ICSE'07)*, 2007, pp. 499–510.

[49] T.-H. Chang, T. Yeh, and R. C. Miller, *GUI Testing Using Computer Vision*. New York, NY, USA: Association for Computing Machinery, 2010, p. 1535–1544.

[50] R. Ramler and T. Ziebermayr, "What You See Is What You Test - Augmenting Software Testing with Computer Vision," in *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2017, pp. 398–400.

[51] G. Numan, *Testing Artificial Intelligence*. Cham: Springer International Publishing, 2020, pp. 123–136.

[52] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming," *Commun. ACM*, vol. 12, no. 10, p. 576–580, Oct. 1969.

[53] R. M. Keller, "Formal Verification of Parallel Programs," *Commun. ACM*, vol. 19, no. 7, p. 371–384, Jul. 1976.

[54] R. Drechsler, *Advanced Formal Verification*. USA: Kluwer Academic Publishers, 2004.

[55] A. Chudnov, N. Collins, B. Cook, J. Dodds, B. Huffman, C. MacCárthaigh, S. Magill, E. Mertens, E. Mullen, S. Tasiran, A. Tomb, and E. Westbrook, "Continuous Formal Verification of Amazon s2n," in *Computer Aided Verification*, H. Chockler and G. Weissenbacher, Eds. Cham: Springer International Publishing, 2018, pp. 430–446.

[56] Stricker, Rudolf, "BMW M3 E30," https://de.m.wikipedia.org/wiki/Datei:BMW_M3_E30_front_20090514.jpg, 2009, [Online; accessed January 31, 2021].

[57] Migl, Alexander, "BMW G20," https://commons.wikimedia.org/wiki/File:BMW_G20_IMG_0167.jpg, 2019, [Online; accessed January 31, 2021].

[58] J. F. Smart, *Jenkins: The Definitive Guide*. Beijing: O'Reilly, 2011.

[59] B. Gower, *Scientific Method: A Historical and Philosophical Introduction*. Taylor & Francis, 2012.

[60] S. Crawford and L. Stucki, "Peer Review and the Changing Research Record," *J. Am. Soc. Inf. Sci.*, vol. 41, no. 3, p. 223–228, Mar. 1990.

[61] M. Baker, "1,500 scientists lift the lid on reproducibility," 2016, [Online; accessed 28-January-2021].

[62] D. Jungnickel, *Graphs, Networks and Algorithms*, 4th ed. Springer Publishing Company, Incorporated, 2013.

[63] T. M. Mitchell, *Machine Learning*, 1st ed. USA: McGraw-Hill, Inc., 1997.

[64] A. Géron, *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*, 1st ed. O'Reilly Media, Inc., 2017.

[65] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018.

[66] R. J. Urbanowicz and W. N. Browne, *Introduction to Learning Classifier Systems*, 1st ed. Springer Publishing Company, Incorporated, 2017.

[67] L. Torrey and J. Shavlik, "Transfer learning," *Handbook of Research on Machine Learning Applications*, 2009.

[68] F. Chollet, "Keras - Transfer learning & fine tuning," https://keras.io/guides/transfer_learning/, 2020, [Online; accessed 23-February-2021].

[69] A. Baevski, S. Edunov, Y. Liu, L. Zettlemoyer, and M. Auli, "Cloze-driven Pretraining of Self-attention Networks," http://arxiv.org/abs/1903.07785, 2019.

[70] Siemens, "Software Lifecycle Under Control," https://polarion.plm.automation.siemens.com/, 2020, [Online; accessed 1-Feburary-2021].

[71] R. Noemmer and R. Haas, "An Evaluation of Test Suite Minimization Techniques," in *Software Quality: Quality Intelligence in Software and Systems Engineering*, D. Winkler, S. Biffl, D. Mendez, and J. Bergsmann, Eds. Cham: Springer International Publishing, 2020, pp. 51–66.

[72] R. Karp, "Reducibility Among Combinatorial Problems," *Complexity of Computer Computations*, vol. 40, pp. 85–103, 01 1972.

[73] U. Feige, "A Threshold of Ln N for Approximating Set Cover," *J. ACM*, vol. 45, no. 4, pp. 634–652, Jul. 1998.

[74] N. Alon, D. Moshkovitz, and S. Safra, "Algorithmic Construction of Sets for K-restrictions," *ACM Trans. Algorithms*, vol. 2, no. 2, pp. 153–177, Apr. 2006.

[75] D. Bertsimas and J. Tsitsiklis, *Introduction to Linear Optimization*, 1st ed. Athena Scientific, 1997.

[76] C. Mannino and A. Sassano, ""Solving Hard Set Covering Problems"," *Operations Research Letters*, vol. 18, no. 1, pp. 1 – 5, 1995.

[77] J. J. Q. Yu, A. Y. S. Lam, and V. O. K. Li, "Chemical Reaction Optimization for the Set Covering Problem," in *2014 IEEE Congress on Evolutionary Computation (CEC)*, July 2014, pp. 512–519.

[78] S. Balaji and N. Revathi, "A New Approach for Solving Set Covering Problem Using Jumping Particle Swarm Optimization Method," *Natural Computing*, vol. 15, no. 3, pp. 503–517, Sep. 2016.

[79] Y. Yu, X. Yao, and Z.-H. Zhou, "On the Approximation Ability of Evolutionary Optimization with Application to Minimum Set Cover," *Artificial Intelligence*, vol. s 180–181, 11 2010.

[80] O. Giel and I. Wegener, "Evolutionary Algorithms and the Maximum Matching Problem," in *Proceedings of the 20th Annual Symposium on Theoretical Aspects of Computer Science*, ser. STACS '03. Berlin, Heidelberg: Springer-Verlag, 2003, p. 415–426.

[81] J. H. Holland, "Genetic Algorithms," *Scientific American*, vol. 267, no. 1, pp. 66–73, 1992.

[82] Wen-Chih Huang, Cheng-Yan Kao, and Jorng-Tzong Horng, "A Genetic Algorithm Approach for Set Covering Problems," in *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*, June 1994, pp. 569–574 vol.2.

[83] J. Beasley and P. Chu, ""A Genetic Algorithm for the Set Covering Problem"," *European Journal of Operational Research*, vol. 94, no. 2, pp. 392 – 404, 1996.

[84] K. S. Al-Sultan, M. F. Hussain, and J. S. Nizami, ""A Genetic Algorithm for the Set Covering Problem"," *Journal of the Operational Research Society*, vol. 47, no. 5, pp. 702–709, May 1996.

[85] R. Kruse, C. Borgelt, C. Braune, F. Klawonn, C. Moewes, and M. Steinbrecher, *Computational Intelligence Eine methodische Einführung in Künstliche Neuronale Netze, Evolutionäre Algorithmen, Fuzzy-Systeme und Bayes-Netze*, 2nd ed., ser. Computational Intelligence. Wiesbaden: Springer Vieweg, 2015.

[86] L. W. Jacobs and M. J. Brusco, "Note: A local-search heuristic for large set-covering problems," *Naval Research Logistics (NRL)*, vol. 42, no. 7, pp. 1129–1140, 1995.

[87] D. Minotra, "A Study of Heuristic-Algorithms for Set-Covering Problems," 06 2008.

[88] R. Eberhart and J. Kennedy, "A new Optimizer using Particle Swarm Theory," in *MHS'95. Proceedings of the Sixth International Symposium on Micro Machine and Human Science*, Oct 1995, pp. 39–43.

[89] B. Xue, S. Nguyen, and M. Zhang, “”A New Binary Particle Swarm Optimisation Algorithm for Feature Selection”,” in *Applications of Evolutionary Computation*, A. I. Esparcia-Alcázar and A. M. Mora, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 501–513.

[90] H. Nezamabadi-pour, M. R. Shahrbabaki, and M. M. Farsangi, “Binary Particle Swarm Optimization: Challenges and new Solutions,” *The CSI Journal on Computer Science and Engineering*, vol. 6, no. 1, 2008.

[91] H. Stegherr, A. Stein, and J. Haehner, “Parallel Chemical Reaction Optimisation for the Utilisation in Intelligent RNA Prediction Systems,” in *ARCS Workshop 2019; 32nd International Conference on Architecture of Computing Systems*, May 2019, pp. 1–8.

[92] A. Lam and V. Li, “Chemical Reaction Optimization: A tutorial,” *Memetic Computing*, vol. 4, 03 2012.

[93] K.-H. Borgwardt, “The Average number of pivot steps required by the Simplex-Method is polynomial,” *Zeitschrift für Operations Research*, vol. 26, no. 1, pp. 157–177, Dec 1982.

[94] D. H. Wolpert and W. G. Macready, “No Free Lunch Theorems for Optimization,” *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 1, pp. 67–82, April 1997.

[95] D. R. Fulkerson, G. L. Nemhauser, and L. E. Trotter, *Two computationally difficult set covering problems that arise in computing the 1-width of incidence matrices of Steiner triple systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1974, pp. 72–81.

[96] A. Joshi, J. E. Rowe, and C. Zarges, “Improving the Performance of the Germinal Center Artificial Immune System Using epsilon-Dominance: A Multi-objective Knapsack Problem Case Study,” in *Evolutionary Computation in Combinatorial Optimization*, G. Ochoa and F. Chicano, Eds. Cham: Springer International Publishing, 2015, pp. 114–125.

[97] F. Neumann and C. Witt, “,” *Bioinspired Computation in Combinatorial Optimization: Algorithms and their Computational Complexity, Natural Computing Series, ISBN 978-3-642-16543-6. Springer-Verlag Berlin Heidelberg, 2010*, 01 2010.

[98] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang, “Skyline with presorting,” in *Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405)*, 2003, pp. 717–719.

[99] S. Börzsönyi, D. Kossmann, and K. Stocker, “The Skyline Operator,” in *Proceedings of the 17th International Conference on Data Engineering*. USA: IEEE Computer Society, 2001, p. 421–430.

[100] M. Endres and F. Weichmann, “Index Structures for Preference Database Queries,” in *Flexible Query Answering Systems*, 05 2017, pp. 137–149.

[101] M. Endres and W. Kießling, “Parallel Skyline Computation Exploiting the Lattice Structure,” *Journal of Database Management*, vol. 26, pp. 18–43, 10 2015.

[102] A. Arrieta, S. Wang, A. Arruabarrena, U. Markiegi, G. Sagardui, and L. Etxeberria, “Multi-Objective Black-Box Test Case Selection for Cost-Effectively Testing Simulation Models,” in *Proceedings of the Genetic and Evolutionary Computation Conference*, ser. GECCO ’18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1411–1418.

[103] J. W. Duran and S. C. Ntafos, “An Evaluation of Random Testing,” *IEEE Transactions on Software Engineering*, vol. SE-10, no. 4, pp. 438–444, 1984.

[104] A. Joshi, J. Rowe, and C. Zarges, “On the Effects of Incorporating Memory in GC-AIS for the Set Cover Problem,” in *MIC 2015: The XI Metaheuristics International Conference*. University of Lille 1, 2015, 11th Metaheuristics Internation Conference (MIC 2015) ; Conference date: 07-06-2015 Through 10-06-2015.

[105] N. Nurmuliani, D. Zowghi, and S. Powell, "Analysis of Requirements Volatility during Software Development Life Cycle," in *2004 Australian Software Engineering Conference. Proceedings.*, 2004, pp. 28–37.

[106] K. Beck and C. Andres, *Extreme Programming Explained: Embrace Change (2nd Edition)*. Boston: Addison-Wesley Professional, 2004.

[107] M. Fowler, "Continuous Integration," https://www.martinfowler.com/articles/continuousIntegration.html, 2006, [Online; accessed 21-February-2021].

[108] Netflix, "Lerner — using RL agents for test case scheduling," https://netflixtechblog.com/lerner-using-rl-agents-for-test-case-scheduling-3e0686211198, 2019, [Online; accessed 21-February-2021].

[109] X. Qu, M. B. Cohen, and K. M. Woolf, "Combinatorial Interaction Regression Testing: A Study of Test Case Generation and Prioritization," in *2007 IEEE International Conference on Software Maintenance*, Oct 2007, pp. 255–264.

[110] S. Wilson, "Classifiers that Approximate Functions," *Natural Computing*, vol. 1, pp. 1–2, 04 2002.

[111] L.-J. Lin, "Reinforcement Learning for Robots Using Neural Networks," USA, 1992, uMI Order No. GAX93-22750.

[112] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-Level Control through Deep Reinforcement Learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 02 2015.

[113] M. Iqbal, W. N. Browne, and M. Zhang, "Reusing Building Blocks of Extracted Knowledge to Solve Complex, Large-Scale Boolean Problems," *IEEE Transactions on Evolutionary Computation*, vol. 18, no. 4, pp. 465–480, 2014.

[114] X. Li and G. Yang, "Transferable XCS," in *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, ser. GECCO '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 453–460.

[115] M. V. Butz and S. W. Wilson, "An Algorithmic Description of XCS," in *Advances in Learning Classifier Systems*, P. Luca Lanzi, W. Stolzmann, and S. W. Wilson, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 253–272.

[116] E. Mohamed, K. Sirlantzis, and G. Howells, "Application of Transfer Learning for Object Detection on Manually Collected Data," in *Intelligent Systems and Applications*, Y. Bi, R. Bhatia, and S. Kapoor, Eds. Cham: Springer International Publishing, 2020, pp. 919–931.

[117] S. D. Whitehead and L.-J. Lin, "Reinforcement learning of non-Markov decision processes," *Artificial Intelligence*, vol. 73, no. 1, pp. 271–306, 1995, computational Research on Interaction and Agency, Part 2.

[118] C. J. C. H. Watkins and P. Dayan, "Technical Note: Q -Learning," *Mach. Learn.*, vol. 8, no. 3–4, p. 279–292, May 1992.

[119] A. Stein, "Interpolation-Assisted Evolutionary Rule-Based Machine Learning - Strategies to Counter Knowledge Gaps in XCS-Based Self-Learning Adaptive Systems," doctoralthesis, Universität Augsburg, 2019.

[120] W. B. P. von Pilchau, A. Stein, and J. Hähner, "Bootstrapping a DQN replay memory with synthetic experiences," in *Proceedings of the 12th International Joint Conference on Computational Intelligence (IJCCI 2020), November 2-4, 2020*, J. J. Merelo, J. Garibaldi, C. Wagner, T. Bäck, K. Madani, and K. Warwick, Eds., 2020.

[121] J. Meinguet, "Multivariate interpolation at arbitrary points made simple," *Zeitschrift für angewandte Mathematik und Physik ZAMP*, vol. 30, no. 2, pp. 292–304, Mar 1979.

[122] S. Bechtold, S. Brannen, J. Link, M. Merdes, M. Philipp, J. de Rancourt, and C. Stein, "JUnit 5 User Guide," https://junit.org/junit5/docs/current/user-guide/, 2020, [Online; accessed 2-March-2021].

[123] H. Krekel, "Pytest Framework," https://docs.pytest.org/en/stable/, 2020, [Online; accessed 26-January-2021].

[124] Google, "Google Test," https://github.com/google/googletest, 2020, [Online; accessed 26-January-2021].

[125] Jenkins, "Junit Plugin," https://plugins.jenkins.io/junit/, 2020, [Online; accessed 2-March-2021].

[126] X. Llorá and K. Sastry, "Fast Rule Matching for Learning Classifier Systems via Vector Instructions," in *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 1513–1520.

[127] G. Ènè and M. Pèroumalnaik, "Speedup Character-Based Matching in Learning Classifier Systems with Xor," in *Proceedings of the 12th Annual Conference Companion on Genetic and Evolutionary Computation*, ser. GECCO '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 1879–1884.

[128] M. Abedini, M. Kirley, R. Chiong, and T. Weise, "GPU-accelerated eXtended Classifier System," in *2013 IEEE Symposium on Computational Intelligence and Data Mining (CIDM)*, April 2013, pp. 293–300.

[129] P. L. Lanzi and D. Loiacono, "Speeding Up Matching in Learning Classifier Systems Using CUDA," in *Learning Classifier Systems*, J. Bacardit, W. Browne, J. Drugowitsch, E. Bernadó-Mansilla, and M. V. Butz, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 1–20.

[130] J. Drugowitsch, *Design and Analysis of Learning Classifier Systems - A Probabilistic Approach*. Springer Publishing Company, Incorporated, 01 2008, vol. 139.

[131] T. Mattson, B. Sanders, and B. Massingill, *Patterns for Parallel Programming*, 1st ed. Addison-Wesley Professional, 2004.

[132] K.-S. Oh and K. Jung, "GPU implementation of neural networks," *Pattern Recognition*, vol. 37, no. 6, pp. 1311 – 1314, 2004.

[133] D. B. Kirk and W.-m. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010.

[134] M. A. Franco, N. Krasnogor, and J. Bacardit, "Speeding up the Evaluation of Evolutionary Learning Systems Using GPGPUs," in *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 1039–1046.

[135] TVM, "Apache TVM: An End to End Machine Learning Compiler Framework for CPUs, GPUs and accelerators," https://tvm.apache.org/, 2020, [Online; accessed 17-March-2021].

[136] AUZone, "DeepView 2.0," https://www.embeddedml.com/, 2020, [Online; accessed 17-March-2021].

[137] V. Kumar, *Introduction to Parallel Computing*, 2nd ed. USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[138] T. Ungerer, *Parallelrechner und parallele Programmierung*. Spektrum Akademischer Verlag, 1997.

[139] NationalInstruments, "TestStand," https://www.ni.com/en-us/shop/software/products/teststand.html, 2020, [Online; accessed 26-January-2021].

[140] Jfrog, "Jfrog Artifactory," https://jfrog.com/artifactory/, 2020, [Online; accessed 26-January-2021].

[141] H. Schmeck, C. Müller-Schloer, E. Çakar, M. Mnif, and U. Richter, *Adaptivity and Self-organisation in Organic Computing Systems*. Basel: Springer Basel, 2011, pp. 5–37.

[142] S. Tomforde, H. Prothmann, J. Branke, J. Hähner, M. Mnif, C. Müller-Schloer, U. Richter, and H. Schmeck, *Observation and Control of Organic Systems*. Basel: Springer Basel, 2011, pp. 325–338.

[143] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[144] S. Tomforde, B. Sick, and C. Müller-Schloer, "Organic Computing in the Spotlight," *CoRR*, vol. abs/1701.08125, 2017.

[145] EU, "Directive 2014/30/EU," https://ec.europa.eu/growth/single-market/european-standards/harmonised-standards/electromagnetic-compatibility_en, 2014, [Online; accessed 7-December-2020].

[146] G. Walker, "A Review of Technologies for Sensing Contact Location on the Surface of a Display," *Journal of the Society for Information Display*, vol. 20, no. 8, pp. 413–440, 2012.

[147] V. Tuzlukov, *Signal Processing Noise*, ser. Electrical Engineering & Applied Signal Processing Series. CRC Press, 2018.

[148] N. Cressie and C. Wikle, *Statistics for Spatio-Temporal Data*. Wiley, 2015.

[149] IEC, "61000-4-6: Testing and Measurement Techniques – Immunity to conducted Disturbances, induced by radio-frequency Fields," https://www.iecee.org/dyn/www/f?p=106:49:0::::FSP_STD_ID:18793, 2009, [Online; accessed 3-December-2020].

[150] K. D. Rao and M. N. S. Swamy, *Digital Signal Processing: Theory and Practice*, 1st ed. Springer Publishing Company, Incorporated, 2018.

[151] M. Arakawa, Y. Yamashita, and K. Funatsu, "Genetic Algorithm-based Wavelength Selection Method for Spectral Calibration," *Journal of Chemometrics*, vol. 25, no. 1, pp. 10–19, 2011.

[152] M. Shafii and F. De Smedt, "Multi-Objective Calibration of a distributed hydrological Model (WetSpa) using a Genetic Algorithm," *Hydrology and Earth System Sciences*, vol. 13, no. 11, pp. 2137–2149, 2009.

[153] F. Millo, P. Arya, and F. Mallamo, "Optimization of Automotive Diesel Engine Calibration using Genetic Algorithm Techniques," *Energy*, vol. 158, pp. 807 – 819, 2018.

[154] Wu Zhizhou, Sun Jian, and Yang Xiaoguang, "Calibration of VISSIM for Shanghai Expressway using Genetic Algorithm," in *Proceedings of the Winter Simulation Conference, 2005.*, 2005.

[155] K. F. Man and K. S. Tang, "Genetic Algorithms for Control and Signal Processing," in *Proceedings of the IECON'97 23rd International Conference on Industrial Electronics, Control, and Instrumentation (Cat. No.97CH36066)*, vol. 4, 1997, pp. 1541–1555 vol.4.

[156] A. Doerr, D. Nguyen-Tuong, A. Marco, S. Schaal, and S. Trimpe, "Model-based policy search for automatic tuning of multivariate PID controllers," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, 2017, pp. 5295–5301.

[157] W. Pilar von Pilchau, A. Stein, and J. Hähner, "Bootstrapping a DQN Replay Memory with Synthetic Experiences," pp. 404–411, 01 2020.

Appendices

A Solving ATCS using XCS(F)

This appendix chapter visually displays the performance in terms of NAPFD which we achieved using XCS and XCSF (and comparing them with the neural network of Spieker et al. [6]) [13, 14] in Figure 40. The succeeding tables hold p-values for several Student t-tests which compare the performance of the individual agents with each other. Student t-test have precondition that the data is normal distributed. We verified this using Shapiro-Wilk tests (with a significance level of 0.05).

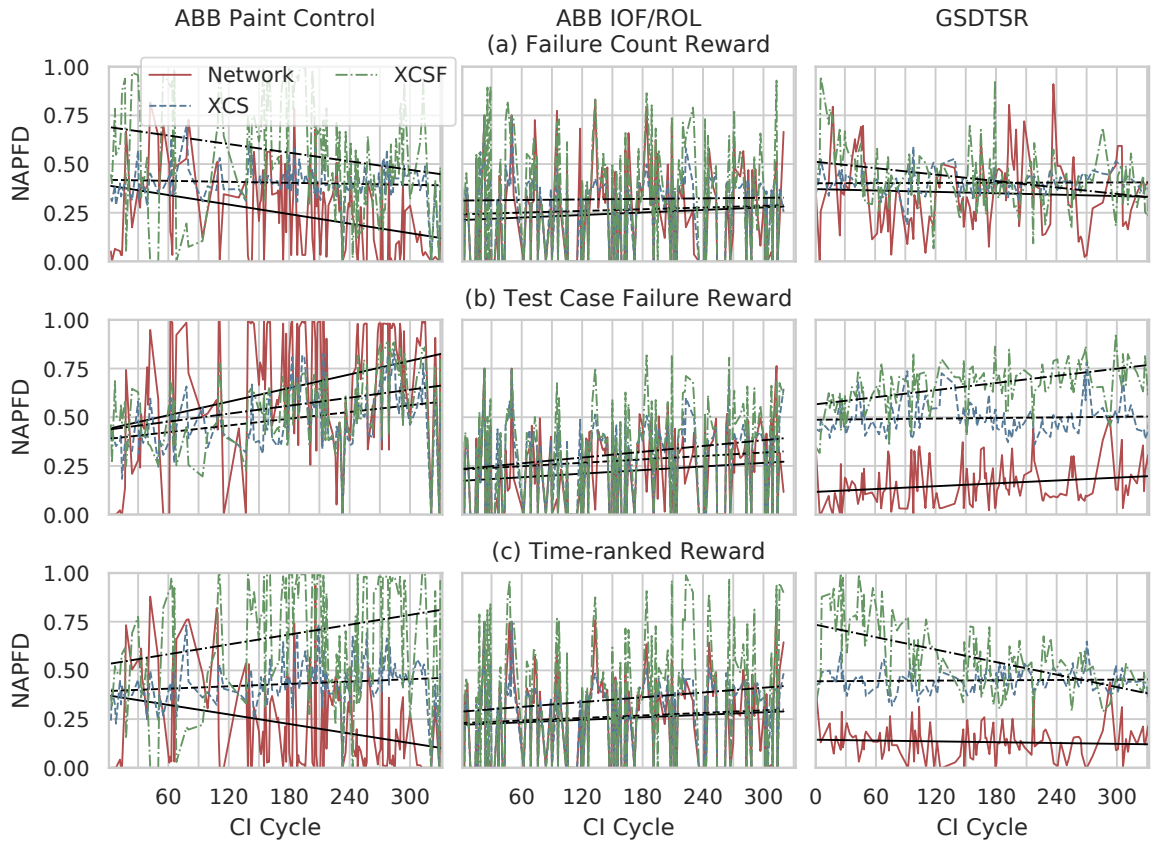


Figure 40: Comparison of XCSF with XCS and the neural network of Spieker et al. [6]. The figure considers three different datasets as described in Table 15. Visually XCS is already better than the neural network in several combinations of reward function / dataset. XCSF improves performances even more as it uses continuous actions. However an open issue is still that the performance vastly differs from reward function to reward function.

Table 27: P-values for paired Student t-tests. Values below 0.05 are marked bold. The null hypothesis is that XCS is superior to XCSF.

	Paint Control	IOF/ROL	GSDTSR
r^{fc}	**0.00523**	0.25042	**0.0002**
r^{tcf}	0.30446	0.32404	**0.00091**
r^{trk}	**0.00192**	0.14235	**0.00001**

Table 28: P-values for paired Student t-tests. Values below 0.05 are marked bold. The null hypothesis is that the neural network is superior to XCSF.

	Paint Control	IOF/ROL	GSDTSR
r^{fc}	**0.0**	**0.01473**	**0.0**
r^{tcf}	0.99999	**0.00015**	**0.0**
r^{trk}	**0.0**	**0.00078**	**0.0**

Table 29: P-values for paired Student t-tests. Values below 0.05 are marked bold. The null hypothesis is that the neural network is superior to XCS.

	Paint Control	IOF/ROL	GSDTSR
r^{fc}	**0.0000**	0.3847	**0.0030**
r^{tcf}	0.9999	**0.0012**	**0.0000**
r^{trk}	**0.0000**	0.4227	**0.0000**

B Robustness for Failcount Reward

This appendix section shows the visual results for the robustness of the CI agent (if the failcount reward is employed). This consists out of the estimation of the active robustness (Figure 41), the average recovery time (Figure 42) and the frequency of recovery events (Figure 43). Thereby we consider different utility thresholds (0-100 percent).

It is worth mentioning that the visual evaluation of these results as well as the statistical ones lead to the same results as for the timerank reward (which we discussed in detail in chapter 5).

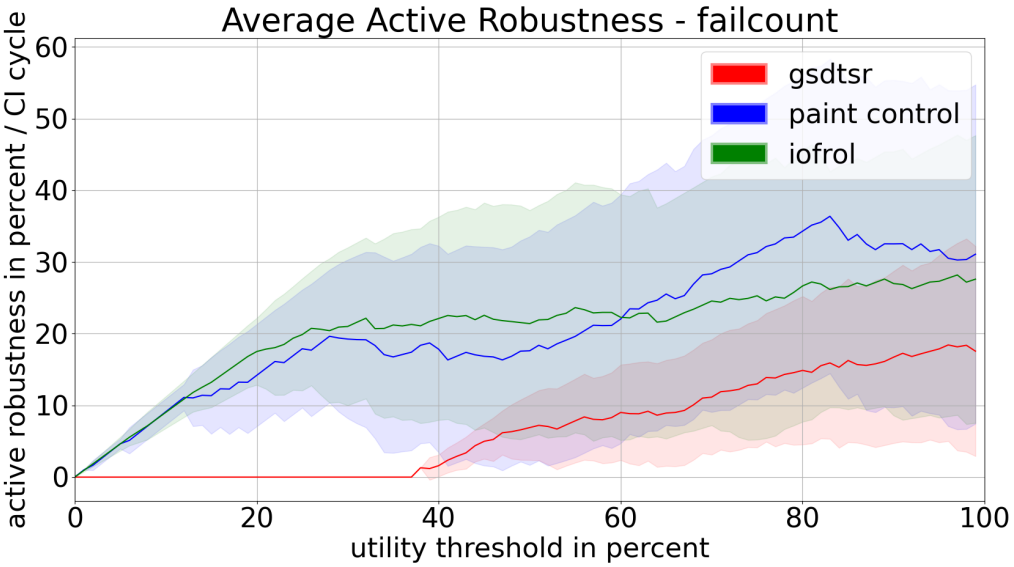


Figure 41: Estimation of the active robustness.

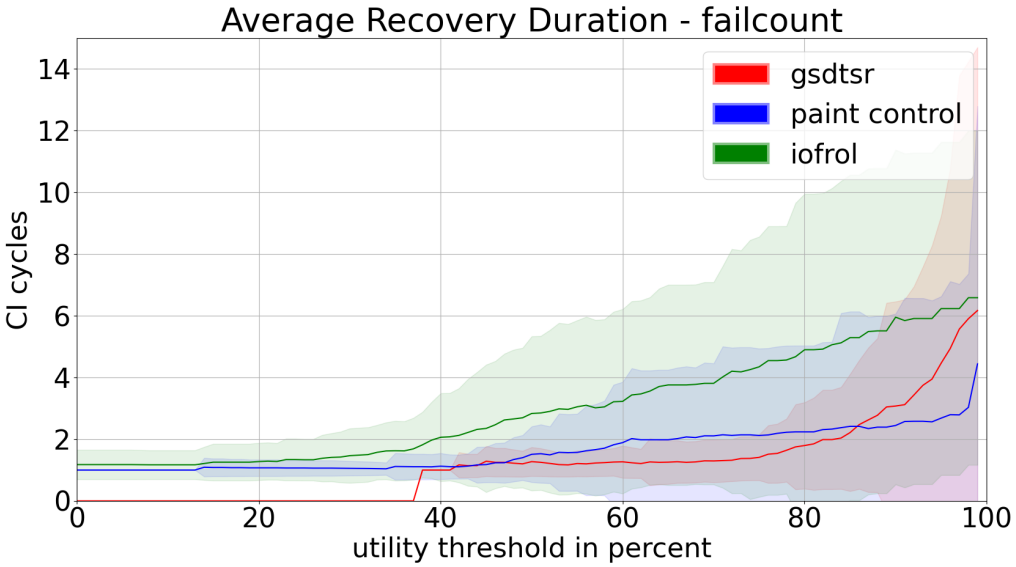


Figure 42: Estimation of the average recovery time.

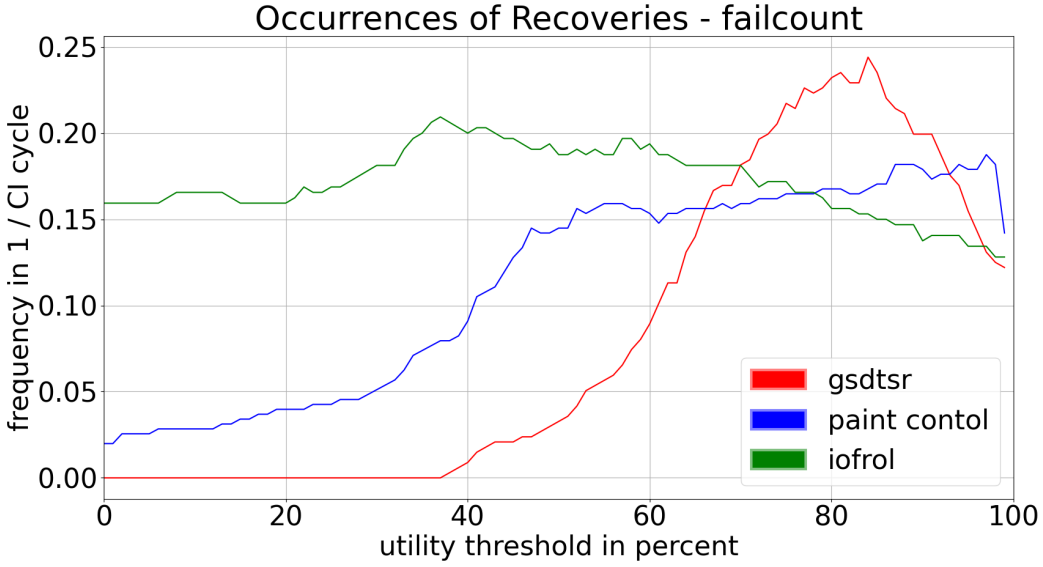


Figure 43: Event frequency of utility break-ins.

C Speed Ups for XCS

The chapter gives insight about the speed ups achieved for various combinations of match size, number of threads, dimension and population sizes. Thereby it contains additional information for the evaluation of the parallelization approaches defined in chapter 6.

Tables 30 and 31 display the speed ups for different combinations of population size and dimension (results aggregated over the different match sizes considered and using the maximum number of threads of the processors = 12). Tables 32 and 33 give insight about the speed ups for different combinations of threads and match sizes (results aggregated over all considered dimensions and population sizes).

Table 30: Speed up $\pm$ 2σ (rounded to the third decimal) of the algorithm using local lists (Algorithm 19) for varying population size and dimension. The best and worst values of a column is highlighted in bold.

$size \backslash dim$	3	6	9	12	15	18	21	24
10000	**2.811 ± 0.205**	**2.717 ± 0.214**	**2.618 ± 0.185**	**2.559 ± 0.176**	**2.512 ± 0.165**	**2.471 ± 0.162**	**2.42 ± 0.133**	**2.401 ± 0.137**
20000	2.926 ± 0.215	2.84 ± 0.247	2.748 ± 0.225	2.678 ± 0.218	2.634 ± 0.198	2.566 ± 0.191	2.521 ± 0.159	2.496 ± 0.158
30000	2.995 ± 0.212	2.888 ± 0.241	2.793 ± 0.212	2.717 ± 0.201	2.66 ± 0.182	2.619 ± 0.174	2.575 ± 0.169	2.539 ± 0.149
40000	3.014 ± 0.213	2.915 ± 0.247	2.823 ± 0.236	2.764 ± 0.232	2.707 ± 0.213	2.663 ± 0.194	2.614 ± 0.188	2.584 ± 0.173
50000	3.015 ± 0.209	2.906 ± 0.225	2.843 ± 0.243	2.761 ± 0.203	2.727 ± 0.21	2.675 ± 0.2	2.628 ± 0.171	2.595 ± 0.164
60000	3.007 ± 0.223	2.929 ± 0.249	2.854 ± 0.247	2.797 ± 0.246	2.73 ± 0.223	2.681 ± 0.199	2.636 ± 0.193	2.611 ± 0.185
70000	**3.071 ± 0.246**	**2.986 ± 0.26**	2.896 ± 0.249	2.839 ± 0.243	**2.778 ± 0.23**	2.723 ± 0.216	2.679 ± 0.201	2.641 ± 0.18
80000	3.069 ± 0.242	2.98 ± 0.259	**2.905 ± 0.266**	**2.841 ± 0.263**	2.774 ± 0.228	2.717 ± 0.206	2.672 ± 0.193	2.64 ± 0.196
90000	3.06 ± 0.219	2.97 ± 0.236	2.889 ± 0.224	2.824 ± 0.222	2.749 ± 0.199	2.701 ± 0.183	2.659 ± 0.181	2.625 ± 0.172
100000	3.044 ± 0.235	2.963 ± 0.26	2.888 ± 0.262	2.821 ± 0.245	2.766 ± 0.234	2.722 ± 0.215	**2.687 ± 0.215**	2.644 ± 0.19
110000	3.039 ± 0.236	2.963 ± 0.252	2.88 ± 0.245	2.806 ± 0.235	2.746 ± 0.205	2.719 ± 0.205	2.669 ± 0.179	2.634 ± 0.17
120000	3.039 ± 0.231	2.958 ± 0.263	2.892 ± 0.266	2.825 ± 0.252	2.77 ± 0.233	**2.724 ± 0.225**	2.671 ± 0.199	**2.65 ± 0.199**
130000	3.045 ± 0.227	2.968 ± 0.265	2.888 ± 0.241	2.839 ± 0.255	2.754 ± 0.202	2.705 ± 0.182	2.683 ± 0.194	2.639 ± 0.165
140000	3.034 ± 0.228	2.961 ± 0.26	2.882 ± 0.25	2.832 ± 0.247	2.768 ± 0.214	2.722 ± 0.2	2.67 ± 0.189	2.65 ± 0.19
150000	3.035 ± 0.227	2.956 ± 0.25	2.882 ± 0.243	2.821 ± 0.239	2.755 ± 0.204	2.718 ± 0.193	2.676 ± 0.188	2.647 ± 0.182

Table 31: Speed up $\pm$ 2σ (rounded to the third decimal) of the algorithm using a simple lock (Algorithm 18) for varying population size and dimension. The best and worst values of a column is highlighted in bold.

$size \backslash dim$	3	6	9	12	15	18	21	24
10000	**2.31 ± 0.092**	**2.288 ± 0.112**	**2.259 ± 0.095**	**2.281 ± 0.094**	**2.289 ± 0.1**	**2.285 ± 0.097**	**2.264 ± 0.09**	**2.272 ± 0.104**
20000	2.408 ± 0.11	2.403 ± 0.14	2.38 ± 0.142	2.393 ± 0.135	2.407 ± 0.132	2.393 ± 0.142	2.376 ± 0.128	2.36 ± 0.131
30000	2.458 ± 0.128	2.439 ± 0.146	2.397 ± 0.118	2.419 ± 0.133	2.413 ± 0.109	2.42 ± 0.125	2.405 ± 0.129	2.386 ± 0.138
40000	2.465 ± 0.137	2.459 ± 0.178	2.417 ± 0.164	2.456 ± 0.172	2.461 ± 0.161	2.461 ± 0.16	2.448 ± 0.165	2.429 ± 0.161
50000	2.457 ± 0.135	2.428 ± 0.141	2.413 ± 0.164	2.43 ± 0.132	2.463 ± 0.159	2.467 ± 0.162	2.445 ± 0.153	2.432 ± 0.148
60000	2.46 ± 0.155	2.433 ± 0.162	2.401 ± 0.173	2.446 ± 0.174	2.456 ± 0.162	2.46 ± 0.169	2.453 ± 0.169	2.439 ± 0.172
70000	**2.504 ± 0.169**	**2.479 ± 0.167**	**2.44 ± 0.149**	**2.492 ± 0.168**	**2.494 ± 0.161**	**2.504 ± 0.178**	**2.491 ± 0.168**	**2.469 ± 0.172**
80000	2.467 ± 0.181	2.43 ± 0.185	2.414 ± 0.187	2.458 ± 0.196	2.471 ± 0.177	2.47 ± 0.169	2.457 ± 0.166	2.455 ± 0.19
90000	2.463 ± 0.158	2.423 ± 0.142	2.4 ± 0.148	2.438 ± 0.143	2.442 ± 0.136	2.461 ± 0.146	2.451 ± 0.148	2.438 ± 0.152
100000	2.474 ± 0.153	2.44 ± 0.175	2.416 ± 0.192	2.464 ± 0.181	2.482 ± 0.186	2.49 ± 0.186	2.489 ± 0.201	2.465 ± 0.173
110000	2.457 ± 0.163	2.431 ± 0.181	2.402 ± 0.154	2.437 ± 0.144	2.458 ± 0.131	2.48 ± 0.167	2.467 ± 0.146	2.449 ± 0.151
120000	2.453 ± 0.159	2.431 ± 0.185	2.413 ± 0.178	2.458 ± 0.191	2.48 ± 0.188	2.489 ± 0.189	2.476 ± 0.182	2.468 ± 0.19
130000	2.479 ± 0.168	2.448 ± 0.179	2.418 ± 0.159	2.481 ± 0.178	2.465 ± 0.13	2.477 ± 0.14	2.486 ± 0.157	2.466 ± 0.157
140000	2.461 ± 0.182	2.432 ± 0.184	2.412 ± 0.174	2.462 ± 0.178	2.471 ± 0.155	2.49 ± 0.167	2.471 ± 0.169	2.464 ± 0.172
150000	2.45 ± 0.157	2.417 ± 0.166	2.389 ± 0.136	2.45 ± 0.174	2.456 ± 0.144	2.471 ± 0.154	2.468 ± 0.165	2.465 ± 0.17

Table 32: Speed up ± 2σ (rounded to the third decimal) of the algorithm using local lists (Algorithm 19) for varying number of threads and match set size. The best and worst values of a column is highlighted in bold.

relative match set size \ threads	2	4	6	8	10	12
1	**2.481 ± 0.063**	**3.09 ± 0.246**	**3.214 ± 0.305**	**3.31 ± 0.376**	**3.374 ± 0.405**	**3.47 ± 0.475**
2	2.392 ± 0.05	2.939 ± 0.215	3.105 ± 0.308	3.218 ± 0.422	3.255 ± 0.465	3.35 ± 0.604
3	2.315 ± 0.041	2.764 ± 0.164	2.971 ± 0.325	3.034 ± 0.369	3.076 ± 0.428	3.071 ± 0.387
4	2.242 ± 0.033	2.633 ± 0.155	2.811 ± 0.268	2.859 ± 0.311	2.897 ± 0.351	2.914 ± 0.378
5	2.184 ± 0.032	2.497 ± 0.111	2.651 ± 0.217	2.697 ± 0.228	2.71 ± 0.275	2.671 ± 0.201
6	2.11 ± 0.026	2.388 ± 0.112	2.514 ± 0.166	2.538 ± 0.173	2.57 ± 0.221	2.555 ± 0.192
7	2.049 ± 0.022	2.261 ± 0.077	2.405 ± 0.137	2.43 ± 0.14	2.428 ± 0.159	2.418 ± 0.141
8	1.983 ± 0.019	2.196 ± 0.082	2.285 ± 0.104	2.311 ± 0.117	2.323 ± 0.13	2.304 ± 0.113
9	1.918 ± 0.016	2.103 ± 0.051	2.185 ± 0.087	2.213 ± 0.096	2.206 ± 0.096	**2.303 ± 0.12**
10	1.866 ± 0.013	2.039 ± 0.059	2.103 ± 0.07	2.126 ± 0.076	**2.128 ± 0.087**	2.353 ± 0.126
11	1.814 ± 0.012	1.957 ± 0.033	2.023 ± 0.057	2.041 ± 0.058	2.166 ± 0.086	2.407 ± 0.136
12	1.772 ± 0.01	1.897 ± 0.038	1.955 ± 0.046	1.972 ± 0.048	2.202 ± 0.093	2.45 ± 0.14
13	1.725 ± 0.009	1.833 ± 0.023	1.89 ± 0.04	**1.963 ± 0.049**	2.245 ± 0.096	2.497 ± 0.145
14	1.69 ± 0.008	1.784 ± 0.024	1.843 ± 0.035	1.985 ± 0.046	2.286 ± 0.099	2.543 ± 0.145
15	1.648 ± 0.007	1.742 ± 0.018	1.786 ± 0.029	2.032 ± 0.052	2.334 ± 0.101	2.587 ± 0.152
16	1.619 ± 0.007	1.706 ± 0.021	**1.742 ± 0.024**	2.074 ± 0.056	2.377 ± 0.108	2.637 ± 0.157
17	1.588 ± 0.006	1.66 ± 0.016	1.72 ± 0.023	2.105 ± 0.054	2.412 ± 0.105	2.642 ± 0.146
18	1.557 ± 0.006	1.636 ± 0.018	1.756 ± 0.024	2.144 ± 0.057	2.453 ± 0.107	2.687 ± 0.157
19	1.525 ± 0.005	1.6 ± 0.012	1.793 ± 0.025	2.185 ± 0.059	2.494 ± 0.109	2.74 ± 0.166
20	1.507 ± 0.005	1.581 ± 0.015	1.826 ± 0.027	2.206 ± 0.055	2.518 ± 0.109	2.79 ± 0.173
21	1.481 ± 0.004	1.54 ± 0.011	1.854 ± 0.027	2.237 ± 0.054	2.521 ± 0.109	2.83 ± 0.184
22	1.461 ± 0.004	1.517 ± 0.011	1.888 ± 0.028	2.276 ± 0.058	2.567 ± 0.122	2.871 ± 0.19
23	1.438 ± 0.004	1.493 ± 0.009	1.917 ± 0.029	2.307 ± 0.059	2.61 ± 0.119	2.922 ± 0.205
24	1.42 ± 0.004	1.47 ± 0.009	1.947 ± 0.028	2.337 ± 0.058	2.656 ± 0.129	2.965 ± 0.214
25	1.401 ± 0.004	**1.449 ± 0.008**	1.979 ± 0.031	2.356 ± 0.054	2.699 ± 0.138	2.978 ± 0.194
26	1.384 ± 0.003	1.469 ± 0.008	2.014 ± 0.032	2.389 ± 0.064	2.734 ± 0.141	2.99 ± 0.215
27	1.369 ± 0.003	1.496 ± 0.009	2.037 ± 0.031	2.431 ± 0.067	2.77 ± 0.143	3.027 ± 0.223
28	1.358 ± 0.003	1.521 ± 0.009	2.071 ± 0.034	2.471 ± 0.074	2.794 ± 0.157	3.06 ± 0.229
29	1.338 ± 0.003	1.545 ± 0.009	2.09 ± 0.033	2.494 ± 0.072	2.817 ± 0.168	3.086 ± 0.243
30	**1.327 ± 0.003**	1.573 ± 0.009	2.121 ± 0.035	2.532 ± 0.079	2.828 ± 0.169	3.109 ± 0.25

Table 33: Speed up ± 2σ (rounded to the third decimal) of the algorithm using a simple lock (Algorithm 18) for varying number of threads and match set size. The best and worst values of a column is highlighted in bold.

relative match set size \ threads	2	4	6	8	10	12
1	**2.533 ± 0.062**	**3.071 ± 0.267**	**3.353 ± 0.396**	**3.622 ± 0.569**	**3.632 ± 0.479**	**3.627 ± 0.52**
2	2.494 ± 0.051	3.031 ± 0.273	3.303 ± 0.434	3.578 ± 0.697	3.585 ± 0.658	3.629 ± 0.781
3	2.456 ± 0.041	2.927 ± 0.233	3.207 ± 0.475	3.405 ± 0.641	3.437 ± 0.663	3.395 ± 0.587
4	2.415 ± 0.033	2.831 ± 0.226	3.054 ± 0.391	3.226 ± 0.525	3.256 ± 0.552	3.271 ± 0.619
5	2.371 ± 0.031	2.716 ± 0.159	2.9 ± 0.306	3.049 ± 0.373	3.058 ± 0.427	3.016 ± 0.345
6	2.305 ± 0.025	2.623 ± 0.16	2.762 ± 0.228	2.872 ± 0.268	2.905 ± 0.335	2.903 ± 0.343
7	2.248 ± 0.021	2.495 ± 0.106	2.646 ± 0.186	2.746 ± 0.214	2.748 ± 0.248	2.753 ± 0.255
8	2.185 ± 0.018	2.431 ± 0.11	2.522 ± 0.138	2.615 ± 0.177	2.63 ± 0.198	2.627 ± 0.202
9	2.118 ± 0.016	2.332 ± 0.068	2.417 ± 0.116	2.506 ± 0.14	2.504 ± 0.143	2.495 ± 0.095
10	2.062 ± 0.013	2.264 ± 0.077	2.336 ± 0.095	2.411 ± 0.111	2.414 ± 0.13	2.511 ± 0.076
11	2.005 ± 0.011	2.18 ± 0.043	2.25 ± 0.076	2.317 ± 0.086	2.359 ± 0.068	2.477 ± 0.068
12	1.957 ± 0.01	2.118 ± 0.049	2.186 ± 0.061	2.246 ± 0.071	2.36 ± 0.053	2.424 ± 0.062
13	1.909 ± 0.009	2.051 ± 0.028	2.12 ± 0.054	2.163 ± 0.046	2.327 ± 0.047	2.375 ± 0.06
14	1.873 ± 0.008	2.0 ± 0.032	2.074 ± 0.048	2.169 ± 0.032	2.288 ± 0.045	2.337 ± 0.058
15	1.829 ± 0.007	1.959 ± 0.023	2.012 ± 0.04	2.17 ± 0.03	2.257 ± 0.043	2.301 ± 0.058
16	1.8 ± 0.007	1.921 ± 0.028	1.967 ± 0.033	2.146 ± 0.029	2.229 ± 0.044	2.275 ± 0.06
17	1.768 ± 0.006	1.869 ± 0.021	1.917 ± 0.027	2.117 ± 0.028	2.205 ± 0.04	2.282 ± 0.057
18	1.736 ± 0.005	1.845 ± 0.024	1.939 ± 0.022	2.098 ± 0.027	2.184 ± 0.041	2.279 ± 0.045
19	1.703 ± 0.005	1.809 ± 0.016	1.954 ± 0.019	2.079 ± 0.027	2.165 ± 0.042	2.216 ± 0.038
20	1.685 ± 0.005	1.789 ± 0.02	1.947 ± 0.019	2.055 ± 0.025	2.178 ± 0.043	2.153 ± 0.034
21	1.657 ± 0.004	1.744 ± 0.014	1.927 ± 0.017	2.047 ± 0.025	2.173 ± 0.031	2.098 ± 0.034
22	1.636 ± 0.004	1.72 ± 0.015	1.916 ± 0.018	2.035 ± 0.025	2.154 ± 0.029	2.04 ± 0.033
23	1.612 ± 0.004	1.696 ± 0.011	1.895 ± 0.017	2.025 ± 0.025	2.114 ± 0.024	1.997 ± 0.035
24	1.594 ± 0.004	1.67 ± 0.012	1.886 ± 0.016	2.014 ± 0.025	2.084 ± 0.023	1.955 ± 0.038
25	1.574 ± 0.004	1.648 ± 0.01	1.874 ± 0.016	2.014 ± 0.024	2.051 ± 0.02	1.949 ± 0.038
26	1.558 ± 0.003	1.653 ± 0.008	1.867 ± 0.017	2.013 ± 0.021	2.019 ± 0.018	2.003 ± 0.034
27	1.542 ± 0.003	1.668 ± 0.006	1.858 ± 0.016	1.997 ± 0.018	1.991 ± 0.017	1.97 ± 0.028
28	1.53 ± 0.003	1.677 ± 0.006	1.846 ± 0.017	1.973 ± 0.015	1.964 ± 0.016	1.937 ± 0.025
29	1.509 ± 0.003	1.667 ± 0.005	1.842 ± 0.016	1.944 ± 0.014	1.94 ± 0.015	1.906 ± 0.024
30	**1.496 ± 0.003**	**1.662 ± 0.005**	**1.839 ± 0.017**	**1.924 ± 0.013**	**1.943 ± 0.014**	**1.876 ± 0.024**