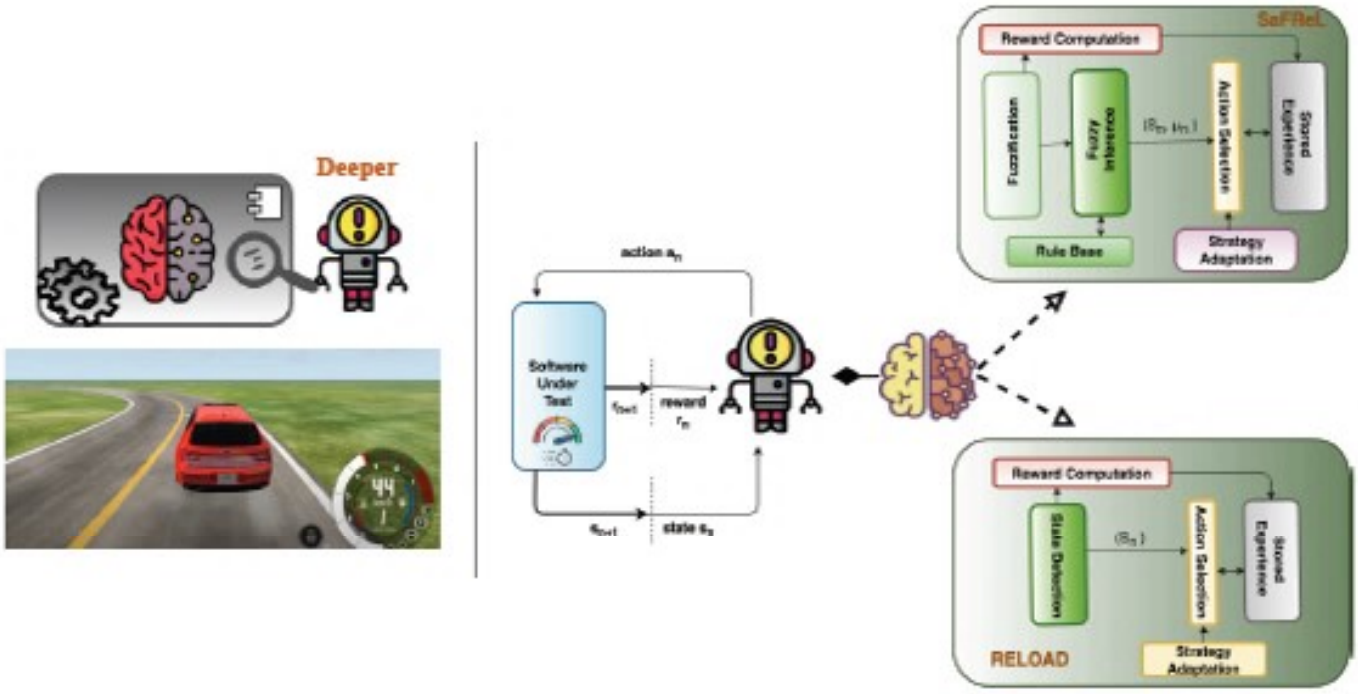
Mälardalen University Doctoral Dissertation 358

Intelligence-Driven Software Performance Assurance

Mahshid Helali Moghadam



Mälardalen University

Mälardalen University Press Dissertations
No. 358

INTELLIGENCE-DRIVEN SOFTWARE PERFORMANCE ASSURANCE

Mahshid Helali Moghadam

2022

School of Innovation, Design and Engineering

ISBN 978-91-7485-549-4
ISSN 1651-4238
Printed by E-Print AB, Stockholm, Sweden

Mälardalen University Press Dissertations
No. 358

INTELLIGENCE-DRIVEN SOFTWARE PERFORMANCE ASSURANCE

Mahshid Helali Moghadam

Akademisk avhandling

som för avläggande av teknologie doktorsexamen i datavetenskap vid
Akademin för innovation, design och teknik kommer att offentligen försvaras
fredagen den 3 juni 2022, 14.00 i Alfa, Mälardalens universitet, Västerås.

Fakultetsopponent: Professor Antonia Bertolino, CNR

Akademin för innovation, design och teknik

Abstract

Software performance assurance is of great importance for the success of software products, which are nowadays involved in many parts of our life. Performance evaluation approaches such as performance modeling, testing, as well as runtime performance control methods, all can contribute to the realization of software performance assurance. Many of the common approaches to tackle challenges in this area involve relying on performance models or using system models and source code. Although modeling provides a deep insight into the system behavior, developing a detailed model is challenging. Furthermore, software artifacts such as models and source code might not be readily available at all times in the development lifecycle. This thesis focuses on leveraging the potential of machine learning (ML) and evolutionary search-based techniques to provide viable solutions for addressing the challenges in different aspects of software performance assurance efficiently and effectively.

In this thesis, we first investigate the capabilities of model-free reinforcement learning to address the objectives in robustness testing problems. We develop two self-adaptive reinforcement learning-driven test agents called SaFReL and RELOAD. They generate effective platform-based test scenarios and test workloads, respectively. The output scenarios and workloads help testers and software engineers meet their objectives efficiently without relying on models or source code. SaFReL and RELOAD learn the optimal policies (ways) to meet the test objectives and can reuse the learned policies adaptively in other testing settings. Policy reuse can lead to higher test efficiency and cost savings, for example, when testing similar test objectives or software systems with comparable performance sensitivity.

Next, we leverage the potential of evolutionary computation algorithms, i.e., genetic algorithms, evolution strategies, and particle swarm optimization, to generate failure-revealing test scenarios for robustness testing of AI systems. In this part, we choose autonomous driving systems as a prevailing example of contemporary AI systems. We study the efficacy of the proposed evolutionary search-based test generation techniques and evaluate primarily to what extent they can trigger failures. Moreover, we investigate the diversity of those failures and compare them to existing baseline solutions.

Finally, we again use the potential of model-free reinforcement learning to develop adaptive ML-driven runtime performance control approaches. We present a response time preservation method for a sample type of industrial applications and a resource allocation technique for dynamic workloads in a data grid application. The proposed ML-driven techniques learn how to adjust the tunable parameters and resource configuration at runtime to keep the performance continually compliant with the requirements and to further optimize the runtime performance. We evaluate the efficacy of the approaches and show how effectively they can improve the performance and keep the performance requirements satisfied under varying conditions such as dynamic workloads and the occurrence of runtime events that lead to substantial response time deviations.

ISBN 978-91-7485-549-4
ISSN 1651-4238

Sammanfattning

Det moderna samhället är beroende av mjukvara, vilket innebär att tillförlitlig mjukvaruprestanda i många fall är avgörande. Olika tillvägagångssätt för prestandautvärdering såsom prestandamodellering, testning och prestandaövervakning under körning kan alla bidra till att kvalitetssäkring av prestanda. Många av de vedertagna tillvägagångssätten involverar att förlita sig på prestandamodeller eller att använda systemmodeller och källkodsanalys.

Avancerad modellering kan ge en djup insikt i mjukvarans beteende, men kräver skapande och underhåll av detaljerade modeller. I många utvecklingskontexter finns inte möjlighet att använda sig av sådana modeller-—inte heller källkod är åtkomlig i samtliga fall. Den här avhandlingen fokuserar på att utnyttja potentialen hos maskininlärning (ML) och evolutionära sökbaserade tekniker för att tillhandahålla hållbara lösningar för att hantera utmaningarna kopplade till kvalitetssäkring av mjukvaruprestanda.

Avhandlingen undersöker först modellfri förstärkningsinlärning för effektiv robusthetstestning. Vi utvecklar två självanpassande förstärkningsinlärningsdrivna testagenter: SaFReL och RELOAD. Agenterna genererar plattformsbaserade testscenarioner respektive belastningstester som effektivt uppnår testmålen utan att förlita sig på modeller eller källkod. Dessa smarta testagenter lär sig den optimala policyn (angreppssättet) för att uppfylla specifika testmål och kan därefter återanvända den inlärda policyn adaptivt i andra testsammanhang, till exempel för liknande testmål eller testning av mjukvarusystem med liknande prestandakänslighet, vilket leder till högre testeffektivitet och sparade testresurser.

Därefter utnyttjar vi potentialen hos evolutionära beräkningsalgoritmer,

det vill säga genetiska algoritmer, evolutionsstrategier och partikelsvärmoptimering, för att generera felavslöjande testscenarier för robusthetstestning av AI-system. I denna del väljer vi system för autonoma fordon som exempel på samtida AI-system. Vi studerar evolutionära sökbaserade testgenereringsteknikers effektivitet och utvärderar i första hand i vilken utsträckning de kan provocera systemfel och hur omfattande fel de kan åstadkomma jämfört med befintliga testmetoder.

Slutligen använder vi återigen modellfri förstärkningsinlärning för att utveckla adaptiva ML-drivna tillvägagångssätt för prestandastyrning, dvs. en metod för att säkerställa svarstider. Vi utvärderar metoden för två exempelapplikationer, nämligen ett industriell styrsystem och en resursallokering i beräkningsnät. De presenterade ML-drivna teknikerna lär sig hur man justerar parametrar och resurskonfigurationener under körning för att kontinuerligt uppfylla prestandakraven med optimal marginal. Vi visar hur effektivt metoderna kan förbättra systemprestanda när den operativa miljön förändras, t.ex. varierande last och resursbegränsningar.

Abstract

Software performance assurance is of great importance for the success of software products, which are nowadays involved in many parts of our life. Performance evaluation approaches such as performance modeling, testing, as well as runtime performance control methods, all can contribute to the realization of software performance assurance. Many of the common approaches to tackle challenges in this area involve relying on performance models or using system models and source code. Although modeling provides a deep insight into the system behavior, developing a detailed model is challenging. Furthermore, software artifacts such as models and source code might not be readily available at all times in the development lifecycle. This thesis focuses on leveraging the potential of machine learning (ML) and evolutionary search-based techniques to provide viable solutions for addressing the challenges in different aspects of software performance assurance efficiently and effectively.

In this thesis, we first investigate the capabilities of model-free reinforcement learning to address the objectives in robustness testing problems. We develop two self-adaptive reinforcement learning-driven test agents called SaFReL and RELOAD. They generate effective platform-based test scenarios and test workloads, respectively. The output scenarios and workloads help testers and software engineers meet their objectives efficiently without relying on models or source code. SaFReL and RELOAD learn the optimal policies (ways) to meet the test objectives and can reuse the learned policies adaptively in other testing settings. Policy reuse can lead to higher test efficiency and cost savings, for example, when testing similar test objectives or software systems with comparable performance sensitivity.

Next, we leverage the potential of evolutionary computation algorithms, i.e., genetic algorithms, evolution strategies, and particle swarm optimization, to generate failure-revealing test scenarios for robustness testing of AI

systems. In this part, we choose autonomous driving systems as a prevailing example of contemporary AI systems. We study the efficacy of the proposed evolutionary search-based test generation techniques and evaluate primarily to what extent they can trigger failures. Moreover, we investigate the diversity of those failures and compare them to existing baseline solutions.

Finally, we again use the potential of model-free reinforcement learning to develop adaptive ML-driven runtime performance control approaches. We present a response time preservation method for a sample type of industrial applications and a resource allocation technique for dynamic workloads in a data grid application. The proposed ML-driven techniques learn how to adjust the tunable parameters and resource configuration at runtime to keep the performance continually compliant with the requirements and to further optimize the runtime performance. We evaluate the efficacy of the approaches and show how effectively they can improve the performance and keep the performance requirements satisfied under varying conditions such as dynamic workloads and the occurrence of runtime events that lead to substantial response time deviations.

To my family

Acknowledgments

When I started my industrial PhD in Sweden, I knew that it is a unique opportunity leading me to learn many new things about research on the intersection of software engineering and machine learning, with regard to industry perspectives in the meantime. However, this journey did not only teach me new research materials but also coached me and gave me valuable lessons that changed my life. Looking back, I see this journey as an amazing experience that helped me know myself and my life better.

This path, with all its wonderful moments, would not have reached the end without all people who helped during my PhD. First, I should thank sincerely my supervisors: Prof. Markus Bohlin, Dr. Mehrdad Saadatmand, Dr. Markus Borg, and Prof. Björn Lisper. I thank my main supervisor, Markus Bohlin, for all his continuous support, encouragement, and for believing in me. I am very grateful to Mehrdad for all his guidance, for supporting and making way for my PhD research within RISE projects, and for being a supportive project/program manager. I deeply thank Markus Borg for all his energizing support and encouragement, helpful guides, compassion, and the productive research collaborations that he created and invited me to. In the end, I would like to thank Björn for all his guidance, tactful comments, and feedback on my research, and for always being patient in answering my questions.

Along with my industrial PhD life, I have been a researcher at the Digital Platforms unit within the Industrial Systems department at RISE Research Institutes of Sweden and working with wonderful colleagues on different research projects. Hereby, I would like to, first, thank my unit manager, Larisa Rizvanovic, for being a care-taking, patient, and attentive manager, and then all my great colleagues at the Västerås office. In particular, I like to thank Ali Balador, my former colleague and one of my kind friends from the very first. Thank you, Ali, for being an inspiring and knowledgeable senior researcher

colleague. Then, I thank Abbas, my kind-hearted colleague/PhD brother. Hanging out with you was always fun, Abbas. Next, I like to express my sincere thank you to my care-taking colleagues/friends who we were working together in common projects: Sima, Niclas, Alireza, Samaneh, Pasqualina, and Tomas. Thank you for being nice, supportive and kindly.

I would like to also thank all the wonderful people at MDU, in particular Prof. Marjan Sirjani and Prof. Masoud Daneshtalab, for the research collaborations and the course lectures that I had the pleasure of being involved in. Meanwhile, thanks to ITS ESS-H industrial research school, in particular Prof. Maria Lindén, Prof. Mats Björkman, and Susanne Fronnå, for the management; all the schoolmates; people at the software testing laboratory research group, particularly Per and Daniel, for all interesting friendly and fun discussions; and all my MDU friends.

The work presented in this thesis has been mainly funded by a couple of European research projects running at RISE, namely TESTOMAT (The Next Level of Test Automation), IVVES (Industrial-grade Verification and Validation), AIDoRt (AI-augmented automation for efficient DevOps, a model-based framework for continuous development At RunTime in cyber-physical systems), and InSecTT (Intelligent Secure Trustable Things); and Swedish Knowledge Foundation (KK stiftelsen) through ITS ESS-H industrial school at Mälardalen University. Furthermore, I would like to also express my sincere gratitude to all my co-authors and thank all the Master students who have worked with me for their theses: Golrokh, Erblin, Bassam, Sara, and Ali. Wish you great success in your future careers!

Last but not least, my very patient husband, Behnam, thank you from the bottom of my heart for all your patience, understanding, and being there for me during this journey. My parents, *Maman Baba*, and my kindest brother, Ali, many thanks to you for always supporting me in every aspect of my life. This exciting adventure would not have reached here without the support of my wonderful close friends: my kind and supportive ones in Västerås—Mahshid, Aida, Afshin, Neda, Fatemeh, Atefeh, Mohammad(s); my compassionate and care-taking ones in Stockholm—Shafagh and Niloufar; and the rest of my amazing friends: Marzieh, Jalal, Sara A., Hazhir, Ali Z., Iliar, Rozhan, and my friends at game nights. Thanks to all of you!

Mahshid Helali Moghadam
Västerås, April 2022

List of publications

Papers included in the thesis[1]

Paper A *Machine Learning to Guide Performance Testing: An Autonomous Test Framework*, Mahshid Helali Moghadam, Mehrdad Saadatmand, Markus Borg, Markus Bohlin, and Björn Lisper. The 12th IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), IEEE 2019.

Paper B *An Autonomous Performance Testing Framework Using Self-Adaptive Fuzzy Reinforcement Learning*, Mahshid Helali Moghadam, Mehrdad Saadatmand, Markus Borg, Markus Bohlin, and Björn Lisper. Software Quality Journal, 1-33, Springer 2021.

Paper C *Performance Testing Using a Smart Reinforcement Learning-Driven Test Agent*, Mahshid Helali Moghadam, Golrokh Hamidi, Markus Borg, Mehrdad Saadatmand, Markus Bohlin, Björn Lisper, and Pasqualina Potena. IEEE Congress on Evolutionary Computation (CEC), IEEE 2021.

Paper D *Machine Learning Testing in an ADAS Case Study Using Simulation-Integrated Bio-Inspired Search-Based Testing*, Mahshid Helali Moghadam, Markus Borg, Mehrdad Saadatmand, Seyed Jalaleddin Mousavirad, Markus Bohlin, and Björn Lisper. Technical report, Mälardalen University, 2022 (submitted for journal publication).

[1] The included articles have been reformatted to comply with the thesis layout.

Paper E *Efficient and Effective Generation of Test Cases for Pedestrian Detection – Search-based Software Testing of Baidu Apollo in SVL*, Hamid Ebadi, Mahshid Helali Moghadam, Markus Borg, Gregory Gay, Afonso Fontes, and Kasper Socha. IEEE International Conference on Artificial Intelligence Testing, IEEE 2021.

Paper F *Adaptive Runtime Response Time Control in PLC-Based Real-Time Systems Using Reinforcement Learning*, Mahshid Helali Moghadam, Mehrdad Saadatmand, Markus Borg, Markus Bohlin, and Björn Lisper. The 13th IEEE/ACM International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), IEEE/ACM 2018.

Paper G *Makespan Reduction for Dynamic Workloads in Cluster-Based Data Grids Using Reinforcement Learning-Based Scheduling*, Mahshid Helali Moghadam and Seyed Morteza Babamir. Journal of Computational Science, 24, 402-412, Elsevier 2018.

Additional papers, not included in the thesis

1. *RWS-L-SHADE: An Effective L-SHADE Algorithm Incorporation Roulette Wheel Selection Strategy for Numerical Optimisation*, Seyed Jalaleddin Mousavirad, Mahshid Helali Moghadam, Mehrdad Saadatmand, Ripon Chakrabortty, Gerald Schaefer, and Diego Oliva. The 25th International Conference on the Applications of Evolutionary Computation (evoapplications), Springer 2022.

2. *Deeper at the SBST 2021 Tool Competition: ADAS Testing Using Multi-Objective Search*, Mahshid Helali Moghadam, Markus Borg, and Seyed Jalaleddin Mousavirad. The IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST), IEEE/ACM 2021.

3. *Automated Performance Testing Based on Active Deep Learning*, Ali Sedaghatbaf, Mahshid Helali Moghadam, and Mehrdad Saadatmand. The 2nd ACM/IEEE International Conference on Automation of Software Test, ACM/IEEE 2021.

4. *Towards a Verification-Driven Iterative Development of Software for Safety-Critical Cyber-Physical Systems*, Marjan Sirjani, Luciana Provenzano, Sara Abbaspour Asadollah, Mahshid Helali Moghadam, and Mehrdad Saadatmand. Journal of Internet Services and Applications, 12(1), Springer 2021.

5. *A Population-Based Automatic Clustering Algorithm for Image Segmentation*, Seyed Jalaleddin Mousavirad, Gerald Schaefer, Mahshid Helali Moghadam, Mehrdad Saadatmand, and Mahdi Pedram. The Genetic and Evolutionary Computation Conference (GECCO) Companion, ACM 2021.

6. *LSTM-AM-ABC: An LSTM-based Plagiarism Detection via Attention Mechanism and a Population-based Approach for Pre-Training Parameters with imbalanced Classes*, Seyed Vahid Moravvej, Seyed Jalaleddin Mousavirad, Mahshid Helali Moghadam, and Mehrdad Saadatmand. The 28th International Conference on Neural Information Processing (ICONIP), Springer 2021.

7. *Poster: Performance Testing Driven by Reinforcement Learning*, Mahshid Helali Moghadam, Mehrdad Saadatmand, Markus Borg, Markus Bohlin, and Björn Lisper. The 13th IEEE International Conference on Software Testing, Verification and Validation, IEEE 2020.

8. *From Requirements to Verifiable Executable Models using Rebeca*, Marjan Sirjani, Luciana Provenzano, Sara Abbaspour Asadollah, and Mahshid Helali Moghadam. Software Engineering and Formal Methods. SEFM 2020 Collocated Workshops. Springer 2020.

9. *Machine Learning-Assisted Performance Testing*, Mahshid Helali Moghadam. The 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), ACM 2019. **Winner of Silver Prize at ACM Student Research Competition (SRC)**

10. *Learning-Based Self-Adaptive Assurance of Timing Properties in a Real-Time Embedded System*, Mahshid Helali Moghadam, Mehrdad Saadatmand, Markus Borg, Markus Bohlin, and Björn Lisper. The 11th

IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW 2018), IEEE 2018.

11. *Learning-Based Response Time Analysis in Real-Time Embedded Systems: A Simulation-Based Approach*, Mahshid Helali Moghadam, Mehrdad Saadatmand, Markus Borg, Markus Bohlin, and Björn Lisper. The 1st IEEE/ACM International Workshop on Software Qualities and Their Dependencies (SQUADE), ACM/IEEE 2018.

12. *Adaptive Service Performance Control Using Cooperative Fuzzy Reinforcement Learning in Virtualized Environments*, Olumuyiwa Ibidunmoye, Mahshid Helali Moghadam, Ewnetu Bayuh Lakew, and Erik Elmroth. The 10th IEEE/ACM International Conference on Utility and Cloud Computing (UCC), ACM/IEEE 2017.

13. *A Multi-Objective Optimization Model for Data-Intensive Workflow Scheduling in Data Grids*, Mahshid Helali Moghadam, Seyyed Morteza Babamir, and Meghdad Mirabi. The IEEE 41th Conference on Local Computer Networks Workshops, IEEE 2016.

Contents

I

Thesis

Chapter 1

Introduction

1.1 Software Performance Assurance

With the growing trend of using computer systems in our daily life, various parts of our life are dependent on the services provided by the software. Quality assurance—with regard to different quality characteristics—is an essential step of the software development life cycle [1, 2, 3], and is of great importance, particularly in the domains where any types of failures might cause major damages to businesses, vital infrastructures, or the environment. The well-known ISO/IEC 25010 standard [4] provides a general model defining various quality characteristics for software products. The quality of a software product expresses to which degree the software meets the quality requirements (needs) of the stakeholders.

Performance, which is one of the main characteristics in different classifications of the quality attributes [4, 5, 6], is an important one which needs to be tested and assured to guarantee the success of software products. It generally describes how well the system accomplishes its functionality. In conventional software systems, performance requirements mainly describe the time- and resource-bound constraints on the behavior of the software system, which are expressed using performance metrics such as response time, throughput, and resource utilization. For example, from a user's perspective, performance could mean that in addition to the functional correctness, the software should also have an adequate response time. Regardless of the

domain, the software with poor performance can lead to huge financial loss. For instance, enterprise applications (EAs) [7, 8] with internet-based user interfaces (UIs) such as e-commerce websites, banking, retailing, and airline reservation systems, are examples whose success is strictly subject to performance assurance. For instance, studies show strong correlations between the load time of a web page and user satisfaction as well as conversion rate; slow loading can considerably lead to the loss of customers and money [9]. Therefore, these systems are often highly required to be robust enough against varying execution conditions [10].

Nowadays, in addition to conventional software systems, AI-enabled systems—the systems incorporating machine learning (ML) components—are starting to emerge in many domains and the trend of using ML is growing very fast. The European Commission (EC) defines AI systems as "software (and possibly also hardware) systems designed by humans that, given a complex goal, act in the physical or digital dimension by perceiving their environment through data acquisition, interpreting the collected structured or unstructured data, reasoning on the knowledge, or processing the information, derived from this data and deciding the best action(s) to take to achieve the given goal" [11]. According to the EC's *Ethics Guidelines for Trustworthy AI* [12], a trustworthy AI system must be lawful, ethical, and robust. ML components are not explicitly programmed, they are intended to learn from data and experience instead—called *Software 2.0* [13]. Then, the quality assurance methodology for AI systems [14] might have extra focuses and require different approaches from the conventional software systems in some aspects. For instance, in AI systems, a part of the requirements is seen as implicitly encoded in the data and accordingly the challenge of under-specificity might be common in requirements definitions of these systems. Nonetheless, regarding AI-enabled systems, it is also highly expected to assure the ability of the system in particular to control the risk of hazardous events. *Robustness* is one of the main focuses of performance requirements of the AI systems as well [14, 15]. The robustness of ML-driven systems indicates how well the system can perform when it is exposed to the inputs that are different but similar to those ones in the training data, and it covers both environmental uncertainty, e.g., environmental conditions, and system-level variability, e.g., sensor components failures [14].

Performance Assurance. System-level performance testing and runtime

performance control are two essential directions of performance engineering that contribute to the realization of system performance assurance. In this thesis, runtime performance control refers to the approaches based on adjusting the tunable parameters of the application or adjusting the resource configuration in a way adaptive to varying runtime conditions, to keep the performance requirements satisfied [16, 17] and in some cases towards one step further, i.e., optimize the performance in the runtime [18]. Keeping the performance requirements continually satisfied is a challenge for the systems exposed to fluctuating runtime conditions and is of high interest for various software systems running on different platforms. There are several works on proposing adaptive runtime performance control techniques for cloud services [19, 17, 20], load balancer and web applications [21], and also real-time systems [22, 23] to optimize the performance and keep the performance satisfying the requirements.

Performance testing is often considered a system testing type which is done on the whole (fully integrated) product to make sure it satisfies the desired requirements [24]. It is often accomplished to meet the primary objectives as i) evaluating performance metrics of the system, ii) detecting functional problems emerging under certain execution conditions, iii) detecting violations of performance requirements [25]. Therefore, one direction of performance testing is in the form of robustness (stress) testing [26] to evaluate the performance of the system under critical test conditions and find the performance breaking points under which the requirements get violated. In this direction, in addition to the conventional software systems, we also broaden the application domain of our case studies to ML-driven systems. ML-based systems involve a big paradigm shift compared to conventional software systems. Accordingly, the robustness testing of these systems requires approaches different from the common methods for conventional software systems and is considered an emerging challenge in the domains using AI systems. Therefore, in this direction, we aim at generating failure-revealing test scenarios leading to the emergence of the performance requirements violations—e.g., violations of response time or resource utilization requirements in conventional software systems or robustness requirements in ML-driven systems.

Scope of the thesis. In this thesis, we focus on the directions of performance testing and runtime performance control within the performance

assurance. We mainly investigate the efficacy and efficiency of the intelligence-driven approaches, based on RL and bio-inspired search-based techniques, in these directions. In this regard, in the performance testing, we leverage search-based and machine learning techniques to generate failure-revealing test scenarios that lead to violations of the requirements in both conventional and AI systems. For the robustness testing of AI systems, we choose autonomous driving systems as one of the pioneering examples of AI systems in the industry and focus on the simulation-based robustness testing approaches for these systems. Regarding the runtime performance control, we use model-free RL techniques to provide adaptive runtime solutions to optimize the performance and keep the performance requirement satisfied despite the varying conditions, for two case studies in different application contexts.

1.2 Background

Performance models and common methods. Performance modeling is a common approach and performance models are widely used tools in different schools of performance assurance in particular for conventional software systems. Various modeling notations such as queueing networks, Markov processes, and Petri nets [27, 28, 29] together with different analytic techniques are commonly used for performance modeling [30, 31, 32]. Although models provide helpful insight into the performance behavior of the system, there are still many details of the implementation and the execution environment that might be ignored in the modeling [33]. Furthermore, building a detailed model might be difficult and costly, and in some cases, the domain knowledge for deriving performance behavior models might be limited. In addition to system models, user behavior models [34, 35], declarative behavior specifications [36, 37, 38, 39], and also source code [40] might be also used in performance assurance techniques, particularly in performance testing of conventional software systems. However, in this regard, one of the main concerns is that those artifacts might not be always available or accessible.

Intelligence-driven Techniques. Machine learning and search-based techniques have always attracted attention from both academia and industry to be used for addressing the challenges in different aspects of software

performance assurance, including performance testing and runtime performance control. For instance, in performance testing, using ML techniques like reinforcement learning (RL) and predictive models together with symbolic execution to find the worst-case execution paths and test inputs [41, 42], are some examples of the application of ML algorithms in the generation of performance test scenarios. Moreover, ML techniques have been frequently used for analyzing the test results from the performance testing, e.g., to identify performance signature from performance metrics [43], and generally detect performance anomalies [44, 45]. Additionally, ML techniques in particular model-free RL has been widely used to address adaptive runtime performance control of software systems in different contexts. For instance, leveraging RL for adaptive service performance control in cloud environments according to the quality of service (QoS) requirements [46, 17] and using a long short-term memory (LSTM) neural network to predict the future workload and adjust the configuration in advance in container-based applications [47] are some examples of the application of different ML techniques for adaptive performance control of software systems.

Evolutionary search-based techniques have been also considered to provide practical and feasible approaches, e.g., particularly in testing domains like performance testing. In [48] a systematic review on the application of search-based testing techniques for different software quality characteristics including performance is presented. Using evolutionary search-based techniques to perform stress testing [49] or generate test inputs leading to the emergence of performance bottlenecks and violations of the performance requirements [50] has been considered a quite common approach in this field. Search-based techniques have been also used frequently together with system models such as UML models [51, 52] and control flow of the software [53] to generate performance test cases. Evolutionary search-based techniques have also shown a great potential to generate failure-revealing test scenarios leading to the violations of robustness (safety) requirements for testing of AI systems [54, 55, 56]. One of the key approaches for generating test input data is based on the manipulation and augmentation of the test scenarios. A big set of works in this category use search-based techniques to go through the search space of the scenarios to find the failure-revealing test scenarios. These techniques mainly formulate test input selection as a search problem, where

optimization algorithms attempt to systematically identify the test input that meets goals of interest [57].

1.3 Thesis Outline

This thesis is divided into two parts. The first part is a summary of the thesis and is organized into five chapters, which are as follows: Chapter 1 gives an overview of the scope of the thesis and the background of the research topic. Chapter 2 discusses the identified research challenges, the research goals followed in the thesis together with the research questions answered within each research goal, and the research process directing our research path. In Chapter 3, we present the contributions of the thesis towards the realization of the research goals. Chapter 4 presents an overview of the related work and preliminary concepts in ML—mainly model-free RL—and computational intelligence techniques, i.e., swarm and evolutionary algorithms, utilized in the solutions. Finally, in Chapter 5, we conclude the first part of the thesis with a discussion on the results, the quality of the research, and threats to validity as well as possible directions for the future work. The second part of the thesis is given as a collection of the included publications presenting the technical contributions of the thesis in detail.

Chapter 2

Research Overview

2.1 Research Challenges

This thesis is motivated by the following existing research challenges within the directions of performance assurance:

Performance Testing. Generating failure-revealing test scenarios with the aim of testing the robustness of the system is a challenging task. The purpose of the performance testing in this direction is to generate critical test scenarios in which the required functionality fails or the requirements of the system are violated.

In *Conventional software systems* performance anomalies or violations of performance requirements are mainly consequences of the emergence of performance bottlenecks in the system [58, 59]. A performance bottleneck is a software, system, or resource component that limits the performance of the system and causes the system to fail to act as well as required [60]. The behavior of the bottleneck is due to some limitations associated with the component such as saturation or contention. A system or resource component saturation happens because of the full utilization of its capacity or when the utilization exceeds a certain usage threshold [60]. The primary causes of performance bottleneck emergence can be categorized into three groups, application-based, platform-based, and workload-based ones. Application-based causes are the issues such as faults in the source code or

system architecture and the platform-based causes are mainly regarded as the issues related to hardware resources, operating system, and execution environment. Workload-based causes also represent the issues such as deviations from the expected workload intensity. Therefore, one perspective to address the challenge of robustness testing—to generate the critical test scenarios—is to provide critical execution conditions which make the performance bottlenecks emerge [61].

While, in *AI Systems*—ML-driven software systems—the research problem might need a different perspective to be addressed. One big differentiating aspect of these systems is that no longer the human engineer explicitly expresses the logic of the system. Instead, these systems are trained using data and experience. For example, in deep neural network (DNN)-based systems, DNNs are trained based on a big amount of annotated data and make the actions based on the learned patterns in the data. Moreover, due to the stochastic nature of the learning techniques, which means different behaviors might result from repeating the training phase, defining deterministic oracles is challenging. Therefore, all in all, the generation of failure-revealing test scenarios for robustness testing of AI systems is also a challenge; furthermore, novel testing techniques are required [62, 63]. In this thesis, we use DNN-based systems in the Automotive AI context as our case studies and focus on the robustness system-level testing in the simulation environment. Hardware-In-the-Loop (HIL), simulation-based, and field testing are common approaches for system-level verification and testing of AI systems [14]. System-level testing mainly targets defining a set of operational scenarios that could lead to failures. In this regard, in the ISO/PAS 21448 Safety of the Intended Function (SOTIF) standard [64] simulation-based testing has been considered a proficient approach and a proper complementary solution to the field testing. The use of simulations, as virtual prototyping platforms, has been growing fast in recent years [65]. It enables testing at the early stages of the development, allows to create the critical test scenarios that are dangerous to be run in the field, and offers the possibility of efficient, inexpensive, and cost-effective testing [66, 55].

Runtime Performance Control. Tuning and optimizing the performance adaptively in the runtime, i.e., through adjusting tunable parameters in the application or execution environment, w.r.t varying conditions is considered a challenging task. Changeable runtime conditions could be, for example,

varying workloads for a software service or the occurrence of unexpected runtime events that might cause deviations in the performance of the system (e.g., response time and resource utilization). Adaptive runtime control approaches to preserve and furthermore optimize the performance has been an interesting challenge in various application domains. For example, effective performance control of cloud-based web services, i.e., response time or resource utilization, to meet the performance requirements with respect to changing workload is a real challenge in cloud infrastructures [17]. Likewise, for other types of applications such as non-cloud web applications and load balancers [21] and real-time systems [22, 23] runtime performance adaptation is a challenge and of interest for the systems in those domains as well. In this thesis, in one part, we do a study on an industrial type of applications, i.e., PLC-based software programs to provide an adaptive performance preservation approach to keep the performance requirement of the system satisfied w.r.t various runtime conditions. Then, in another part, we conduct a study on a data grid application to propose an adaptive runtime resource allocation approach for executing heterogeneous tasks in dynamic workloads, which leads to optimizing the performance of the system.

2.2 Research Goals

The main research goal in this thesis is accomplishing performance assurance from the perspectives of performance testing, i.e., generation of failure-revealing test scenarios and runtime performance control. We mainly study the approaches that do not rely on performance/system models or source code but treat the system as a black-box or work based on the data resulting from running the test scenarios on the system. In this regard, we investigate the use of bio-inspired search-based and model-free reinforcement learning techniques. Model-free RL algorithms are intended to learn the optimal way (policy) of meeting an objective without access to a model of the environment and provide the opportunity to transfer the gained knowledge between similar situations. Bio-inspired search-based techniques such as evolutionary computation and swarm intelligence algorithms are mainly intended for addressing optimization problems, i.e., finding optimal solutions among all possible ones (See Chapter 4 for an overview of how RL algorithms—in particular model-free RL—and also swarm and evolutionary computation

algorithms work).

The general theme of the research goals in this thesis is solution-focused [67], i.e., it focuses on creating better (more effective and efficient) solutions to address the intended challenges. In order to clarify the involved details on how we are going to address the aforementioned research challenges, we shape the research direction of this thesis in the light of the following research goals and answering the following research questions for each goal:

- **RG1:** Effective and efficient **generation of failure-revealing test scenarios**—in which the performance breaking point emerges, i.e., the performance requirements are violated—for conventional software systems, without dependency on the models, source code and other system artifacts.
  - RQ1.1: How could the *platform-based robustness test scenarios* resulting in the emergence of performance breaking points be generated adaptively and efficiently for different software programs? (Paper A and B)
  - RQ1.2: How could the robustness *test workload scenarios* resulting in the emergence of the performance breaking point be generated efficiently for a system under test? (Paper C)
- **RG2:** Effective and efficient **generation of failure-revealing test scenarios**—in which the safety requirements are violated—for automotive AI systems.
  - RQ2.1: How could *diverse failure-revealing* test scenarios resulting in breaking safety requirements for an Automotive AI system be efficiently generated? (Paper D and E)
- **RG3:** To develop adaptive **runtime performance control** to provide runtime performance preservation and optimization w.r.t varying execution conditions.
  - RQ3.1: How could an adaptive response time control for PLC-based industrial applications be developed to keep the performance continually compliant with the requirements w.r.t varying conditions? (Paper F)

- RQ3.2: How could an adaptive resource allocation for dynamic workloads in a data grid application be developed to optimize the performance in the runtime? (Paper G)

During this research, we have used different application domains to accomplish the research goals, since the research has been conducted over different European research projects and in collaboration with the industrial partners. Note that we have mostly used simulation-based methods to carry out the experimental evaluations in our contributions. Our simulation-based experimental evaluation allowed us to show the efficacy and applicability of the proposed techniques in different aspects of performance assurance for both conventional and AI software systems.

2.3 Research Process

Adopting a proper research methodology is an essential part of conducting research on any topic. Dodig-Crnkovic [68] presents a scientific framework widely used, as a logical scheme, by researchers and scientists to address research questions in general sciences. She also discusses scientific differentiating aspects of computer science from other sciences and describes how the general scientific methodology can be customized for computer science fields. Holz et al. [69] also provide an overview of different computing research methods and a general framework for organizing the computing research process. Their framework involves four main steps driving the research process collectively. Identifying research challenges, formulating research goals/questions, proposing solutions, and evaluating the proposed solutions are those main steps.

Meanwhile, it is strongly encouraged to ground the software engineering research on realistic application contexts to utilize its potential to contribute to addressing real-world challenges. In practice, the solutions which do not match the real needs and could not scale are the challenges that hinder the accomplishment of this objective. [70]. In this regard, it is worth noting that, for the most part, our research has been also carried out in collaboration with different industrial partners mainly within two European ITEA research projects. Therefore, in order to customize the original four-step research framework w.r.t the industrial context of our research, we adopted the

technology transfer model proposed by Gorscheck et al. [71]; we also considered an industrial validation step, succeeding the evaluation step and also a feedback from the industry's point of view in each step of the research process, as shown in Figure 2.1. Thus, the involved steps in our research process are described as follows [1]:

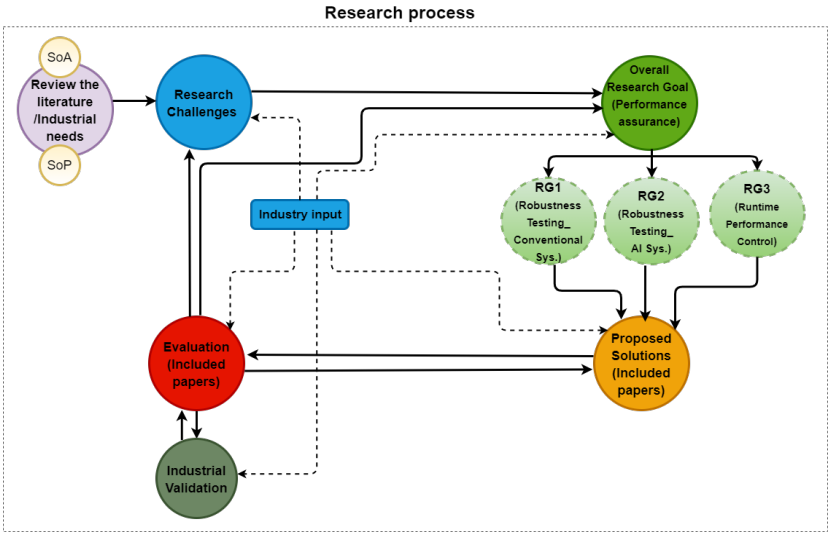


Figure 2.1: Research process

Research challenge: We identify the research challenges based on reviewing both state of the art and practice, and considering the needs of the industrial partners in our projects. We review the literature and collect the related works, for example, through the search methods used in systematic literature reviews (SLR) such as snowballing search [72, 73].

Research goal/question: Over the research journey the main research goals are formulated based on the identified research challenges. As discussed in Section 2.2, we define three research goals related to the performance testing and runtime performance control to drive the research path in this thesis. Meanwhile, in order to explicate the path to reaching the goals, the

[1]This section has been partially adapted from M. Helali Moghadam, Licentiate Thesis, Mälardalen University, 2020 [61].

corresponding research questions are formulated for each research goal.

Proposed solution: After identifying the research challenges, defining the research goals, and formulating the corresponding research questions, we consolidate our ideas for addressing the research questions and reaching the goals in terms of the solutions presented in the included papers (See Chapter 3).

Evaluation: In the evaluation step, we conduct experimentation as an empirical research method w.r.t the guidelines provided by Robson and McCartan [74] for designing experiments in different research projects. We evaluate the efficacy and efficiency of our solutions through conducting a set of controlled experiments in accordance with the existing guidelines for empirical software engineering research [75]. Depending on the evaluation results, the research problems, the goals, and the proposed solutions could be refined. This process is conducted iteratively until reaching the desired results and intended goals. Moreover, an industrial validation of the developed research prototype is conducted, in case the solution has been planned to be evaluated on an industrial case study.

Chapter 3

Research Contribution

This thesis focuses on meeting the research goals discussed in Sec. 2.2 through answering the corresponding research questions for each goal. The research goals in this thesis are to accomplish performance assurance from the perspectives of robustness testing, i.e., generation of failure-revealing test scenarios for both conventional and AI-enabled systems, and runtime performance control. To support meeting the research goals, five RQs have been posed that are targeted by seven studies. This chapter revisits the research goals and summarizes the contributions answering the RQs along with the mapping of the papers to the targeted questions [1]:

.

3.1 RG1: Robustness Testing for Conventional Software Systems

The contributions to meeting RG1 consist of two parts, C1.1 and C1.2, which target RQ1.1 and RQ1.2 respectively. The details of these contributions are discussed in the following sections.

[1] Some parts of this chapter have been partially adapted from M. Helali Moghadam, Licentiate Thesis, Mälardalen University, 2020 [61]

3.1.1 C1.1: SaFReL.

This contribution is a self-adaptive fuzzy reinforcement learning performance test agent, called SaFReL [2]. It generates the platform-based performance test scenarios resulting in finding the intended performance breaking point, for software programs with different performance sensitivity to resources (e.g., CPU-, memory-, and disk-intensive programs) without access to source code or system models (paper B) [76]. The research resulting in this contribution started by investigating the capabilities of RL algorithms to address the challenge of robustness testing with the aim of generating failure-revealing test scenarios resulting in the emergence of performance breaking points—the violations of the performance requirements. We propose an initial architecture for a learning-based performance test agent and present a general overview of the main parts of the architecture and how each step of the learning is formulated (paper A) [77]. Q-learning [78] is used as the core learning mechanism for the proposed agent. The RL-driven smart test agent basically learns the optimal policy for generating test scenarios to accomplish the objective (i.e., finding the performance breaking point) through episodes of interaction with the environment, i.e., SUT and the execution platform. This interaction generally involves observing the state of the environment, taking an action affecting the environment, and receiving a reward signal which shows the effectiveness of the applied action. The smart test agent assumes two phases of learning:

- *Initial learning,* during which the agent learns an optimal policy for the first time.
- *Transfer learning,* during which the agent replays the learned policy in similar cases while keeping the learning running and updating the policy in the long term.

The initial idea (paper A) uses Q-learning together with multiple experience (knowledge) bases. It stores the learned optimal policy—for adjusting the platform-related parameters and generating the test scenarios—to achieve the test objective for different types of SUT, i.e., CPU-intensive, memory-intensive, and disk-intensive programs, in separate knowledge bases. Then, it reuses the learned policies accordingly in the transfer learning. In SaFReL (paper B), We augment the learning by adopting

fuzzy logic to model the state space of the environment (SUT and the execution platform) and fuzzy classification for the state detection in the learning process. It helps tackle the issue of uncertainty in defining discrete classes and improves the accuracy of the learning. Meanwhile, we propose an adaptive action selection strategy adjusting the parameters related to the action selection based on the detected similarity between the performance sensitivity of the SUTs.

We perform a two-fold experimental evaluation, i.e., performance (efficiency and adaptivity) and sensitivity analysis of SaFReL. The evaluation is carried out based on simulating the performance behavior of various SUTs. According to the experimental results, SaFReL meets the test objective, i.e., finds the intended performance breaking point, more efficiently in comparison to a typical performance testing technique which mainly generates the performance test cases based on changing the parameters, by certain steps randomly. SaFReL is able to adapt the action selection strategy of the learning for various SUTs with different performance sensitivity. It leads to cost saving in terms of computation time for performance test scenario generation by reusing the learned policy upon the SUTs with similar performance sensitivity [76].

3.1.2 C1.2: RELOAD.

This contribution introduces an intelligent RL-assisted load testing agent, called RELOAD [2], which learns how to generate an effective test workload, to meet a test objective—finding an intended performance breaking point—efficiently, and is able to reuse the learned policy in further testing objectives (paper C) [79]. The proposed RL-driven load testing agent identifies the effects of different transactions involved in the workload and learns how to adjust the transactions to meet the test objective. It also assumes two learning phases: initial and transfer learning phases. It learns the optimal policy to generate an effective workload for a certain objective in the initial learning. Then, in the transfer learning it reuses adaptively the learned policy for meeting further different test objectives. The test agent has been developed and evaluated based on both Q-learning and Deep Q-learning (DQN), as the

[2] Available at https://github.com/mahshidhelali/RL-Assisted-Performance-Testing.

core learning procedures. It uses an adaptive action selection strategy to reuse the learned policy in the transfer learning. RELOAD uses a well-known load test actuator, namely Apache JMeter [80], to execute the designed workload on the SUT.

We perform an experimental evaluation—efficiency and sensitivity analysis—of the proposed test agent on a functional e-commerce web application as SUT. We compare the efficiency of RELOAD, in terms of test cost saving, based on four configurations of the proposed learning with a random (exploratory) and a standard baseline load testing approaches. The results show that RELOAD learns how to meet the test objective with a more accurate and fine-tuned workload and subsequently leads to test cost saving in comparison to the random and baseline approaches, i.e., accomplishes the test objective using fewer virtual users. The test agent after the initial learning is able to reuse the learned policy for further test objectives on the SUT, fairly keep its efficiency across them, and results in test cost saving in the transfer learning as well.

The pay-as-you-go cost for many of the load generation tools on the market is proportional to the number of generated virtual users. Therefore, the efficient generation of an effective workload by the proposed test agent, RELOAD, could lead to saving cost and time in the testing process. RELOAD along with SaFReL, as two smart test agents forming an RL-driven performance testing framework [81], which is well-suited to the testing contexts where the source code, system models, and behavior specifications are not available. These smart test agents have the capability of reusing the learned policy, while keeping the learning running to adapt the learned policy to changes in the environment. This feature is particularly beneficial to DevOps continuous testing activities such as performance regression testing where performance testing must be repeated for the SUT in a continuous integration process.

3.2 RG2: Robustness Testing of AI Systems

This section of the contributions targets the system-level robustness testing of AI systems (RG2). It involves two parts, C2.1 and C2.2, which target RQ2.1, and present two test generation tools that generate failure-revealing test

scenarios for two AI systems in the domain of autonomous driving. The details of these contributions are discussed in the following sections.

3.2.1 C2.1: Deeper

Deeper (paper D) [82] is a simulation-integrated bio-inspired search-based test generator that generates failure-revealing test scenarios to test a Deep Neural Network (DNN)-based lane keeping system in the BeamNG driving simulator. It uses different types of bio-inspired search algorithms such as genetic algorithm (GA) [83], $(\mu + \lambda)$ and (μ, λ) evolution strategies (ES) [84], and particle swarm optimization (PSO) [85] to search through the space of possible test scenarios and find those leading to the emergence of failures. The test subject for Deeper is BeamNG.AI, the built-in ML-driven driving agent in the BeamNG simulator. In Deeper, a failure is defined in terms of episodes in which the ego car—driven by the BeamNG.AI agent—drives partially outside the lane w.r.t a certain tolerance threshold. The tolerance threshold determines the percentage of the car's bounding box needs to be outside the lane to be regarded as a failure.

The first version of Deeper [3] [56] was our contribution to the cyber-physical testing competition 2021 at the IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST). It uses Non-dominated Sorting Genetic Algorithm-II (NSGA-II) [86], a multi-objective optimization algorithm, to search the input space and generate critical test scenarios leading the car to get out of the lane. The extended version of Deeper (paper D) benefits from four new GA-, $(\mu + \lambda)$ and (μ, λ) ES-, and PSO-driven test generators to generate failure-revealing test scenarios. The problem is basically regarded as an optimization problem, and in order to generate the test scenarios that are of interest, we evaluate the quality of the test scenarios using a fitness (objective) function that guides the search process to maximize the detected distance of the car from the center of the lane during driving of the car on the lane. We leverage an initial quality population seed to boost the search process regarding the fact that the search is done at a fixed test budget. The quality population seed used in the search algorithms is a mix of valid random solutions and a set of solutions generated by the first version of

[3] Available at https://github.com/mahshidhelali/Deeper_ADAS_Test_Generator.

Deeper—based on NSGA-II. We develop a domain-specific *one-point crossover* operation and a polynomial bounded mutation operation for the presentation model used for modeling the test scenarios.

The target in Deeper is to generate the highest number of diverse test scenarios leading to the failures, i.e., causing the car to get out of the lane. We design two groups of experiments—with different experimental configurations—as implemented in the SBST 2021 CPS testing competition. In order to provide a comparative analysis, we compare the results of the proposed test generators in Deeper with the test generators in the tool competition, i.e., Frenetic [87], GABExploit and GABExplore [88], Swat [89], and also the earlier version of Deeper [56]. The generated test scenarios resulting from different test generators are evaluated w.r.t a couple of quality criteria, i.e., number of detected failures, failure diversity, and test generation effectiveness and efficiency. According to the experimental results, the new bio-inspired search-based test generation techniques in Deeper prove to be effective and efficient in provoking a considerable number of diverse failure-revealing test scenarios w.r.t different target failure severity (i.e., tolerance threshold), test budget, and driving style constraints (e.g, speed limits). For Example, in terms of the number of triggered failures within a given test time budget and with less strict driving constraints, the $(\mu + \lambda)$ ES-driven test generator in Deeper considerably outperforms other tools while keeping the level of promoted failure diversity quite close to the counterpart tool with the highest number of detected failures in the competition. Moreover, as a distinctive feature, none of the newly proposed test generators leaves the experiment without triggering any failures, and they act as more reliable test generators than most of the other tools for provoking diverse failures under a limited test budget and strict constraints. With respect to the test effectiveness and efficiency, Deeper $(\mu + \lambda)$ ES-, PSO-, and GA-driven result in high effectiveness in terms of the ratio of the number of detected failures to the generated valid test scenarios.

3.2.2 C2.2: GA-Driven ScenarioGenerator

GA-Driven *ScenarioGenerator* (paper E) [57] is another evolutionary automated test generator to generate failure-revealing test scenarios for testing a pedestrian detection and emergency braking system in Baidu Apollo

[90]—a state-of-the-art autonomous driving platform—in SVL simulation environment [91]. GA-driven ScenarioGenerator uses a generalizable and flexible data structure, called *noise vector*, to model the parameters involved in creating a test scenario (test input space). Meanwhile, it benefits a GA-based technique using a multi-criteria safety heuristic—measuring the quality of the test scenarios in terms of their potential to cause safety violations—for formulating the objective function targeted for the optimization. The evolutionary test scenario generation technique searches through the input space and optimizes the objective function. In this test generator, the occurrence of any crashes between an ego car and pedestrians are considered a failure. To evaluate the efficiency and effectiveness of the approach, the results are compared to a random test scenario generation. Here the target is also to generate the highest number of diverse test scenarios leading to failures. According to the quality criteria considered—detected failures and failure diversity— the proposed GA-driven ScenarioGenerator results in twice as many failures as the random testing technique on the same test configuration and budget. Moreover, it promotes considerably more diversity between the failure-revealing test scenarios. This collaborative work was ranked as one of the qualifying finalists in the 2021 IEEE Autonomous Driving AI Test Challenge at IEEE International Conference on AI Testing 2021.

3.3 RG3: Runtime Performance Control

This section discusses the contributions towards the runtime performance control (RG3) and includes two parts, C3.1 and C3.2, which describe two research works targeting RQ3.1 and RQ3.2.

3.3.1 C3.1: An Adaptive Learning-Driven Runtime Performance Preservation Technique

An RL-Driven Response Time Control for PLC-Based Industrial Applications: The study in paper F proposes an RL-driven runtime performance preservation approach for an industrial type of applications, i.e., PLC-based programs. It presents how the performance of the software system can be controlled to keep it compliant with the requirements, through adaptive

adjusting of tunable parameters in the runtime. In paper F [16], we formulate an adaptive runtime response time control for PLC-based programs using model-free reinforcement learning, and conduct an experimental evaluation based on simulating the performance behavior of the programs. The proposed approach, based on regarding the response time control problem as a Markov decision process (MDP), uses Q-learning, to provide an adaptive control of the response time w.r.t the timing requirements. We evaluate the efficacy of the approach through multiple experiments. The experimental results show that our approach effectively keeps the programs adhering to the response time requirements despite the occurrence of runtime events resulting in response time deviations.

3.3.2 C3.2: An Adaptive Learning-Assisted Runtime Performance Optimization Technique

A RL-Assisted Resource Allocation for Dynamic Workloads in a Data Grid Application: The research study in paper G proposes a runtime learning-driven resource allocation for executing tasks submitted to a cluster-based data grid application. The proposed approach consists of a hierarchical multi-agent system that involves smart agents—inside the clusters—that learn the optimal way to allocate processing resources to the submitted tasks w.r.t the type of the tasks and the current status of the grid in order to optimize the makespan (completion time) of the tasks. The proposed multi-agent system consists of one central broker agent and several RL-driven (based on Q-learning) local agents within the clusters. The local agents learn how to decide about the resource allocation inside the clusters. Upon receiving a task, the central broker agent selects the cluster with the minimum data cost based on the data communication cost measurement, then the RL-driven local agent of the selected cluster schedules the task to a proper node in the cluster. The proposed approach has been developed and integrated into an open source data grid simulator, and the efficacy of the approach has been evaluated in comparison to four common runtime resource allocation strategies under various workloads with different task patterns. The experimental results show that the proposed learning-driven approach learns how to decide about the resource allocation adaptively and leads to optimizing the performance effectively, i.e., reduces the completion time of the tasks

compared to other common techniques.

3.4 Overview of the Included Papers

The main contributions of the thesis are organized and presented as a set of papers included in the thesis. Other non-included papers listed at the beginning of the thesis also strengthen the contributions of the thesis. Every RQ within the research goals has been targeted by at least one included paper. A summary of the included papers and the mapping of the papers to the research goals are as follows:

Table 3.1: Mapping of the papers to the research goals

	RG1		***RG2***	***RG3***	
	RQ1.1	***RQ1.2***	***RQ2.1***	***RQ3.1***	***RQ3.2***
Paper A	✓				
Paper B	✓				
Paper C		✓			
Paper D			✓		
Paper E			✓		
Paper F				✓	
Paper G					✓

Paper A: Machine Learning to Guide Performance Testing: An Autonomous Test Framework [77]
Authors: *Mahshid Helali Moghadam*, Mehrdad Saadatmand, Markus Borg, Markus Bohlin, Björn Lisper
Abstract: Satisfying performance requirements is of great importance for performance-critical software systems. Performance analysis to provide an estimation of performance indices and ascertain whether the requirements are met is essential for achieving this target. Model-based analysis as a common approach might provide useful information but inferring a precise performance model is challenging, especially for complex systems. Performance testing is considered as a dynamic approach for doing

performance analysis. In this work-in-progress paper, we propose a self-adaptive learning-based test framework which learns how to apply stress testing as one aspect of performance testing on various software systems to find the performance breaking point. It learns the optimal policy of generating stress test cases for different types of software systems, then replays the learned policy to generate the test cases with less required effort. Our study indicates that the proposed learning-based framework could be applied to different types of software systems and guides towards autonomous performance testing.

Paper B: An Autonomous Performance Testing Framework using Self-Adaptive Fuzzy Reinforcement Learning [76]
Authors: *Mahshid Helali Moghadam*, Mehrdad Saadatmand, Markus Borg, Markus Bohlin, Björn Lisper
Abstract: Test automation brings the potential to reduce costs and human effort, but several aspects of software testing remain challenging to automate. One such example is automated performance testing to find performance breaking points. Current approaches to tackle automated generation of performance test cases mainly involve using source code or system model analysis or use-case based techniques. However, source code and system models might not always be available at testing time. On the other hand, if the optimal performance testing policy for the intended objective in a testing process instead could be learned by the testing system, then test automation without advanced performance models could be possible. Furthermore, the learned policy could later be reused for similar software systems under test, thus leading to higher test efficiency. We propose SaFReL, a self-adaptive fuzzy reinforcement learning-based performance testing framework. SaFReL learns the optimal policy to generate performance test cases through an initial learning phase, then reuses it during a transfer learning phase, while keeping the learning running and updating the policy in the long term. Through multiple experiments on a simulated environment, we demonstrate that our approach generates the target performance test cases for different programs more efficiently than a typical testing process, and performs adaptively without access to source code and performance models.

Paper C: Performance Testing Using a Smart Reinforcement Learning-Driven Test Agent [79]
Authors: *Mahshid Helali Moghadam*, Golrokh Hamidi, Markus Borg, Mehrdad Saadatmand, Markus Bohlin, Björn Lisper, Pasqualina Potena
Abstract: Performance testing with the aim of generating an efficient and effective workload to identify performance issues is challenging. Many of the automated approaches mainly rely on analyzing system models, source code, or extracting the usage pattern of the system during the execution. However, such information and artifacts are not always available. Moreover, all the transactions within a generated workload do not impact the performance of the system the same way, a finely tuned workload could accomplish the test objective in an efficient way. Model-free reinforcement learning is widely used for finding the optimal behavior to accomplish an objective in many decision-making problems without relying on a model of the system. This paper proposes that if the optimal policy (way) for generating test workload to meet a test objective can be learned by a test agent, then efficient test automation would be possible without relying on system models or source code. We present a self-adaptive reinforcement learning-driven load testing agent, RELOAD, that learns the optimal policy for test workload generation and generates an effective workload efficiently to meet the test objective. Once the agent learns the optimal policy, it can reuse the learned policy in subsequent testing activities. Our experiments show that the proposed intelligent load test agent can accomplish the test objective with lower test cost compared to common load testing procedures, and results in higher test efficiency.

Paper D: Machine Learning Testing in an ADAS Case Study Using Simulation-Integrated Bio-Inspired Search-Based Testing [82]
Authors: *Mahshid Helali Moghadam*, Markus Borg, Mehrdad Saadatmand, Seyed Jalaleddin Mousavirad, Markus Bohlin, Björn Lisper
Abstract: This paper presents an extended version of Deeper, a search-based simulation-integrated test solution that generates failure-revealing test scenarios for testing a deep neural network-based lane-keeping system. In the newly proposed version, we utilize a new set of bio-inspired search algorithms, genetic algorithm (GA), $(\mu + \lambda)$ and (μ, λ) evolution strategies (ES), and particle swarm optimization (PSO), that leverage a quality

population seed and domain-specific crossover and mutation operations tailored for the presentation model used for modeling the test scenarios. In order to demonstrate the capabilities of the new test generators within Deeper, we carry out an empirical evaluation and comparison with regard to the results of five participating tools in the cyber-physical systems testing competition at SBST 2021. Our evaluation shows the newly proposed test generators in Deeper not only represent a considerable improvement on the previous version but also prove to be effective and efficient in provoking a considerable number of diverse failure-revealing test scenarios for testing an ML-driven lane-keeping system. They can trigger several failures while promoting test scenario diversity, under a limited test time budget, high target failure severity, and strict speed limit constraints.

Paper E: Efficient and Effective Generation of Test Cases for Pedestrian Detection – Search-based Software Testing of Baidu Apollo in SVL [57]
Authors: Hamid Ebadi, *Mahshid Helali Moghadam*, Markus Borg, Gregory Gay, Afonso Fontes, Kasper Socha
Abstract: With the growing capabilities of autonomous vehicles, there is a higher demand for sophisticated and pragmatic quality assurance approaches for machine learning-enabled systems in the automotive AI context. The use of simulation-based prototyping platforms provides the possibility for early-stage testing, enabling inexpensive testing and the ability to capture critical corner-case test scenarios. Simulation-based testing properly complements conventional on-road testing. However, due to the large space of test input parameters in these systems, the efficient generation of effective test scenarios leading to the unveiling of failures is a challenge.

This paper presents a study on testing pedestrian detection and emergency braking system of the Baidu Apollo autonomous driving platform within the SVL simulator. We propose an evolutionary automated test generation technique that generates failure-revealing scenarios for Apollo in the SVL environment. Our approach models the input space using a generic and flexible data structure and benefits a multi-criteria safety-based heuristic for the objective function targeted for optimization. This paper presents the results of our proposed test generation technique in the 2021 IEEE Autonomous Driving AI Test Challenge. In order to demonstrate the efficiency and effectiveness of our approach, we also report the results from a

baseline random generation technique. Our evaluation shows that the proposed evolutionary test case generator is more effective at generating failure-revealing test cases and provides higher diversity between the generated failures than the random baseline.

Paper F: Adaptive Runtime Response Time Control in PLC-based Real-Time Systems using Reinforcement Learning [16]
Authors: *Mahshid Helali Moghadam*, Mehrdad Saadatmand, Markus Borg, Markus Bohlin, Björn Lisper
Abstract: Timing requirements such as constraints on response time are key characteristics of real-time systems and violations of these requirements might cause a total failure, particularly in hard real-time systems. Runtime monitoring of the system properties is of great importance to check the system status and mitigate such failures. Thus, a runtime control to preserve the system properties could improve the robustness of the system with respect to timing violations. Common control approaches may require a precise analytical model of the system which is difficult to be provided at design time. Reinforcement learning is a promising technique to provide adaptive model-free control when the environment is stochastic, and the control problem could be formulated as a Markov Decision Process. In this paper, we propose an adaptive runtime control using reinforcement learning for real-time programs based on Programmable Logic Controllers (PLCs), to meet the response time requirements. We demonstrate through multiple experiments that our approach could control the response time efficiently to satisfy the timing requirements.

Paper G: Makespan Reduction for Dynamic Workloads in Cluster-based Data Grids Using Reinforcement Learning-Based Scheduling [18]
Authors: *Mahshid Helali Moghadam*, Seyed Morteza Babamir
Abstract: Scheduling is one of the important problems within the scope of control and management in grid and cloud-based systems. Data grid still as a primary solution to process data-intensive tasks, deals with managing large amounts of distributed data in multiple nodes. In this paper, a two-phase learning-based scheduling is proposed for data-intensive tasks scheduling in cluster-based data grids. In the proposed approach, a hierarchical multi agent system, consisting of one global broker agent and several local agents, is

applied to scheduling procedure in the cluster-based data grids. At the first step of the proposed approach, the global broker agent selects the cluster with the minimum data cost based on the data communication cost measure, then an adaptive policy based on Q-learning is used by the local agent of the selected cluster to schedule the task to the proper node of the cluster. The impacts of three action selection strategies have been investigated in the proposed approach, and the performance of different versions of the approach regarding different action selection strategies, has been evaluated under three types of workloads with heterogeneous tasks. Experimental results show that for dynamic workloads with varying task submission patterns, the proposed learning-based scheduling gives better performance compared to four common scheduling strategies, Queue Length (Shortest Queue), Access Cost, Queue Access Cost (QAC) and HCS, which use regular combinations of primary parameters such as, data communication cost and queue length. Applying a learning-based strategy provides the scheduling with more adaptability to the changing conditions in the environment.

Individual Contributions. I have been the principle driving researcher and main author of papers A, B, C, D, F, and G. It is noted that in paper C, one part of the experiments was carried out by the second author. In Paper E, I have contributed with proposing and developing the GA-driven test case generator, analyzing the test results, and compiling the sections of the paper. The experimental environment was owned and set up by Infotiv AB and the experiments were conducted by the first author. The rest of the co-authors contributed with providing valuable feedback and reviewing the drafts of the papers.

Chapter 4

Background and Related Work

This chapter presents an overview of the primary concepts of reinforcement learning, specifically model-free RL, and bio-inspired search-based algorithms used in the proposed solutions in the thesis. A summary of the related work relevant to the contributions to meeting each research goal is also presented in later sections [1].

4.1 Reinforcement Learning

Machine learning techniques are often categorized into three parts: supervised, unsupervised, and reinforcement learning. Building a model and extracting useful patterns from a training data set with known input and output is the main focus of the supervised learning. The extracted model is commonly used to make predictions—either for classification or regression. Regression models are used to produce/predict continuous output, while classification models work based on discrete data. Support vector machine (SVM), neural network, Naive Bayes, K-nearest neighbors (KNN), and decision trees [92, 93] are some of the most common primary classification

[1] This chapter has been primarily organized based on the reworked chapter 3 from M. Helali Moghadam, Licentiate Thesis, Mälardalen University, 2020 [61].

algorithms. SVM regression, regression trees, Gaussian process regression (GPR), and generalized linear models are also some of the common regression techniques. Unsupervised learning combs through data to find hidden patterns and structures. The most common algorithms in the class of unsupervised learning are clustering techniques. Hierarchical clustering, K-means, K-medois, Fuzzy C-means, Density-based spatial clustering of applications with noise (DBSCAN), and Gaussian mixture models [94, 95, 92] are some of the most common clustering algorithms.

Reinforcement learning (RL) [78] has been widely used for addressing decision-making problems in various fields of science. In fact, RL is a fundamental category of machine learning algorithms intended to find the optimal way to make decisions. Reinforcement learning (RL) differs from the previous learning paradigms in that it works based on interaction with the environment (system[2]) of the problem. The agent observes (senses) the environment at each step of the interaction, takes a possible action, and receives a reward signal from the environment indicating the effectiveness of the action in achieving the agent's intended goal (See Figure 4.1). The following are some of the major distinctions between RL and other learning paradigms [78, 96]:

- In RL, there is no supervisor at work; instead, the agent receives a reward signal.
- The sequential decision-making process is the base of RL. The learning is not carried out based on a batch of training data. Instead, the agent walks through the environment, makes decisions at each step, and learns the optimal way to make decisions by optimizing the reward.
- The agent's actions have an effect on the system, which in turn also influences the data that the agent receives at each step.

4.1.1 Principles

The following are the underlying key concepts in RL [78, 96]:

State. In RL, the agent acts based on what it observes from the environment (system). The agent decides what actions to take w.r.t the history at each step.

[2]System and environment are used interchangeably hereafter in the thesis.

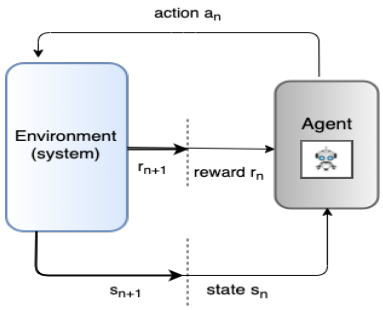


Figure 4.1: Interaction between agent and environment in RL

The sequence of observations, actions, and rewards that occurred during the previous steps is referred to as the history. Because considering the entire history is inefficient, *state* is used to determine what should be done next. It is a concise summary of the history that includes all of the required information.

In this regard, *Markov state* is a concept related to the (summary) function of the history. A state S_t is Markov if and only if

$$P[S_{t+1} \mid S_t] = P[S_{t+1} \mid S_t, S_{t-1}, ..., S_1] \tag{4.1}$$

By definition, the states of the environment are Markov. If the environment is fully observable to the agent, the states for the agent, which are used to make decisions, are also Markov. This representation depicts the Markov Decision Process (MDP), which is the main formalism for RL. The agent's states are not equivalent to the environment's states, in case the environment is partially observable to the agent—which means the agent does not observe the entire environment. In this case, a different formalism, such as *partially observable Markov Decision Process*, is required, and certain heuristics such as keeping the entire history or using a recurrent neural network, could be used to construct the agent's states. The environments (systems) are assumed to be fully observable by the agent in this thesis.

Action and Reward. The agent chooses actions with the aim of maximizing the cumulative long-term reward. The reward is often a scalar feedback signal that indicates how well the agent performs at each step.

Agent's Properties. An RL-based agent may have one or more of the following learning elements:

- Policy: It is a function that describes the agent's behavior, i.e., the actions that the agent chooses given a particular state.
- Value function: It describes the quality of each state and/or action, i.e, how much reward we anticipate from performing a specific action in a specific situation (See Figure 4.2 , which depicts the state value and policy for an RL agent in a Maze).
- Model: It is the agent's view of the environment. The agent may have (or build) a model of the environment. To demonstrate the behavior of the environment, two types of models are mainly used: transition-based and reward-based models. A transition-based model predicts the next state given a certain state and executing a specific action, whereas a reward-based model provides the next instant reward upon taking a specific action in a certain state.

In this regard, *model-free* and *model-based* RL is a fundamental categorization of RL algorithms. Model-free RL is referred to the RL algorithms that are not intended to build or learn a model of the environment in order to understand how it works. The goal of these algorithms is to learn the optimal behavior, i.e., how to behave in order to get as much reward as possible from a series of interaction experiences with the environment. Monte Carlo learning and Temporal-Difference (TD) learning, which include Q-learning algorithms, are common general types of model-free RL. Model-based RL algorithms often begin by creating a model of the environment, which the agent then uses to plan and look ahead to choose the best course of actions, i.e., to find the optimal behavior.

Value-based, policy-based, and actor-critic algorithms are other categorizations of RL algorithms depending on the key parts of the learning. A value-based algorithm mainly employs the value function, while the policy can be read and extracted implicitly. Instead of using value functions, a policy-based algorithm directly saves and utilizes the policy. In an actor-critic algorithm, the value-function and policy are both stored and used jointly.

4.1.2 Model-Free RL Algorithms

For many real-world complex systems, having access to a model of the environment may be an impractical assumption. There is no assumption of

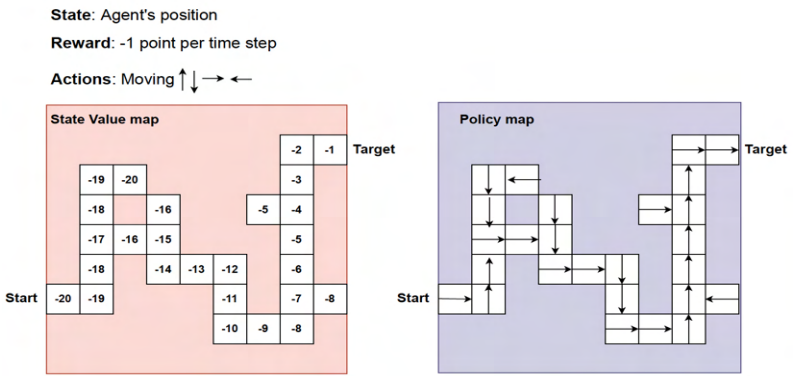


Figure 4.2: State value and policy map for an RL agent in a Maze

access to a model in model-free RL algorithms, i.e., no known MDP of the environment. Instead, the agent utilizes its experience of interactions with the system to estimate the value function and accordingly identify the optimal policy to achieve the goal. Model-free RL algorithms are used in two ways: estimating the value function of a given policy and optimizing the value function—which leads to the optimal policy [78].

Monte-Carlo (MC). This class of model-free RL algorithms, while not the most efficient, is quite successful and widely used in practice. To estimate the value function of states given a certain policy, MC methods employ the entire (terminated) episodes of the interaction with the environment. The value function of state s under policy π, $V_\pi(s)$, is basically the expected return (G_t) from state s under policy π, which is indicated as follows:

$$G_t = r_{t+1} + \gamma r_{t+2} + ... + \gamma^{T-1} r_T \tag{4.2}$$

$$V_\pi(s) = \mathbb{E}_\pi[G_t | s_t = s] \tag{4.3}$$

where r_{t+1} is the reward received after the first transition from state s, γ is the discount factor, and G_t, as return, is the discounted reward till the end of the episode. MC takes the mean of the returns of the experienced episodes originating from state s to estimate its value function. The mean value of returns is computed incrementally episode by episode, without the need to

keep the sum of the episodes. The incremental updates of $V(s)$—as the mean of the returns—in the Monte-Carlo technique is as follows:

$$\begin{aligned} N(s) &= N(s) + 1 \\ V(s) &= V(s) + \frac{1}{N(s)}(G_s - V(s)) \end{aligned} \tag{4.4}$$

where $N(s)$ denotes the number of visits to state s, G_s indicates the current episode's return from state s, and $V(s)$ represents the previous estimate of $V(s)$. The *First time* or *Every time* that state s is visited in the episode are two ways to count the number of visits to state s in an episode and update the value. In the end, to avoid saving all of the episode information, the number of state visits might be substituted with a fixed step size, called $alpha$ (α), and the incremental updating could be formulated as follows:

$$V(s) = V(s) + \alpha(G_s - V(s)) \tag{4.5}$$

Temporal-Difference (TD) learning. In a similar way, this group of algorithms learns from the experienced episodes of the interaction. The main distinction between MC and TD techniques is that TD learns from *incomplete* episodes (trajectories) by conducting *bootstrapping*, or incremental updating using an estimate instead of the actual reward. It is an online learning approach because it learns the state value under policy π online, i.e., at every step of the interaction with the environment. The following are some of the primary advantages of TD over MC:

- TD can learn online after every step without having to wait until the end of the episodes and can learn from incomplete episodes, whereas MC must wait until the episodes are completed.
- TD can work in continuing environments additionally, whereas MC can only work in terminating episodic environments. This characteristic qualifies TD for non-terminating, changing situations. In this regard, because of the bootstrapping, TD is more efficient than MC.

One of the primary characteristics of TD is that it converges to a solution that is associated with an MDP with maximum likelihood. It utilizes the Markov property, identifies an MDP fitting the observations, and attempts to

solve it. As a result of this feature, TD can even converge on a number of repeated sample episodes, but MC can not. MC does not take use of the Markov property, therefore it may be more effective for non-Markov environments.

The basic type of TD learning, TD(0), utilizes a one-step ahead estimate of the return and performs the incremental updating of the state value as follows:

$$V(s_t) = V(s_t) + \alpha(r_{t+1} + \gamma V(s_{t+1}) - V(s_t)) \tag{4.6}$$

where r_{t+1} is the reward received upon the transition to s_{t+1}, $r_{t+1} + \gamma V(s_{t+1})$ represents the estimated target return, *TD target*, and accordingly $r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$ is called *TD error*. TD(0) estimates the state value based on one step of reality, but it is feasible to have TD look into additional steps of reality and then make a better estimate. For incremental updates, TD(n) is intended to use an n-step return (See Equation 4.7), which is as follows:

$$G_{s_t}^{(n)} = r_{t+1} + \gamma r_{t+2} + ... + \gamma^{n-1} r_{t+n} + \gamma^n V(s_{t+n}) \tag{4.7}$$

$$V(s_t) = V(s_t) + \alpha(G_{s_t}^{(n)} - V(s_t)) \tag{4.8}$$

Then, TD(λ) is a technique for effectively using returns from all time-steps. It utilizes a geometrically weighted average of all n-step returns using a constant weight, λ, $0 \leq \lambda \leq 1$ (See Equation 4.9). In this regard, the updating equation presents a *Forward-view* $TD(\lambda)$, which is as follows:

$$G_{s_t}^{(\lambda)} = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_{s_t}^{(n)} \tag{4.9}$$

$$V(s_t) = V(s_t) + \alpha(G_{s_t}^{(\lambda)} - V(s_t)) \tag{4.10}$$

Nonetheless, forward-view $TD(\lambda)$ still suffers from the same drawbacks as MC, since it needs to be computed from completed episodes. In this regard, there is another option called *Backward-view* $TD(\lambda)$, which offers a method for online, step-by-step updates from incomplete episodes to $TD(\lambda)$. It uses a one-step estimate, meanwhile keeps an eligibility trail for each state visited using a simple function called $E_t(s)$). To credit the states, it uses both

frequency and recency heuristics (See Equation 4.11).

$$\begin{aligned} E_0(s) &= 0 \\ E_t(s) &= \gamma\lambda E_{t-1}(s) + 1 \end{aligned} \tag{4.11}$$

Thus, backward-view $TD(\lambda)$ updates the state value based on a one-step TD-error and eligibility trace, which is as follows:

$$\begin{aligned} \delta_t &= r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \\ V(s_t) &= V(s_t) + \alpha\delta_t E_t(s) \end{aligned} \tag{4.12}$$

4.1.3 Model-Free RL for Optimal Behavior

The key motivations for utilizing model-free RL to determine the optimal policy to behave could be generally discussed as follows [78, 96]:
First, the environment's MDP model may be unknown, though experience sampling is conceivable. Second, while the MDP model might be known, it is too difficult and computationally demanding to utilize, so it is more efficient to use samples of the environment.

In Section 4.1.2, we discussed how the main types of model-free RL algorithms, Monte Carlo and TD learning, evaluate a given policy, i.e., estimate the value function in an unknown MDP. Then, in this section, We will look at how the aforementioned estimating approaches, together with an improvement strategy, could be utilized to optimize the value function in an unknown MDP, and accordingly to find out the optimal policy for reaching a target without a priori knowledge about the environment.

There are generally two models for learning the optimal behavior to achieve a target: off-policy and on-policy learning. On-policy learning finds the optimal policy by optimizing the policy from which the experience samples are derived, whereas off-policy learning learns the optimal policy from the experience samples produced based on other policies, such as the behavior of others. These learning paradigms are based on the use of a policy iteration process that includes policy evaluation and policy improvement. The agent alternates between policy evaluation and improvement at each step, as it evaluates the policy first (e.g., estimates the value function) and then attempts to improve the policy using a greedy method. This procedure leads to convergence on the optimal policy (See Figure 4.3).

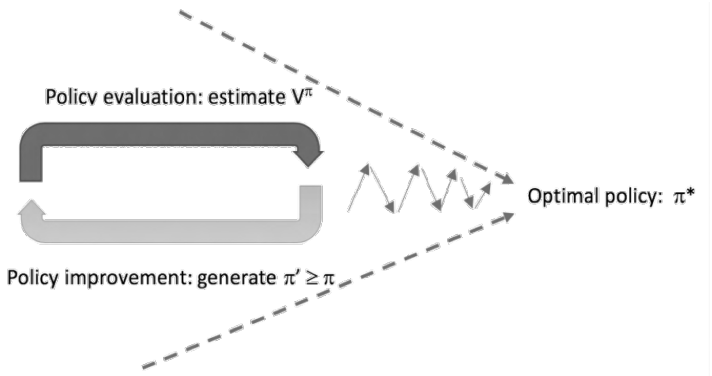


Figure 4.3: Iterative policy evaluation and improvement towards optimal policy

Nonetheless, in policy iteration in order to apply a greedy method to improve the state value function, we require an MDP model, which we do not have access to in model-free RL. Hence, instead of using $V(s)$, an action-value function, $Q(s,a)$, could be used to solve the issue and enable greedy policy improvement (See Equation 4.13).

$$\pi'(s) = \underset{a}{\operatorname{argmax}}\, Q(s,a) \tag{4.13}$$

Furthermore, to ensure the possibility of exploration, ε-greedy, $0 < \varepsilon < 1$, is used as a policy improvement technique in which a greedy action, $\underset{a \in A}{\operatorname{argmax}}\, Q(s,a)$, is selected with probability $1-\varepsilon$, otherwise a random action is chosen.

Both Monte-Carlo and TD algorithms can be used in the policy iteration. However, the strengths of TD such as the capability of online updating, working in continual environments, and learning from incomplete episodes, make it a more common and efficient alternative for the policy iteration process. Then, the main idea is to use TD to evaluate $Q(s,a)$, utilize ε-greedy policy improvement and update at each time-step.

On-policy learning. In this paradigm of model-free RL for learning the optimal behavior, updating at each step is performed based on *Sarsa* rule. It considers an estimate of the policy in one step ahead and updates the Q-value

of the current state in the direction of that estimate, which is as follows:

$$Q(s,a) = Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)] \tag{4.14}$$

where s' is the next state and a' is the action taken at the next state. Sarsa (See Algorithm 1) is the principle on-policy model-free RL based on the use of TD to find the optimal policy to achieve a goal.

Algorithm 1 Sarsa Algorithm

Initialize Q-values, $\forall s \in \mathbb{S},\ \forall a \in \mathbb{A}$;
while *Not (end of learning)* **do**
    Initialize s;
    Choose action a based on the policy derived from Q (e.g., using ε-greedy);
    repeat
        Take action a;
        Observe s', r;
        Choose action a' based on the policy derived from Q (e.g., using ε-greedy);
        $Q(s,a) = Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)]$;
        $s \leftarrow s'\ a \leftarrow a'$;
    until *meeting the terminal (target) state*;
end

In addition, Sarsa can also benefit from the concept of TD(n), which considers n-step returns and updates $Q(s,a)$ in the direction of n-step Q-returns. Thus, similar to TD(λ), backward-view Sarsa(λ) can use an eligibility trace for each pair of state and action, which is as follows:

$$\begin{aligned} E_0(s,a) &= 0 \\ E_t(s,a) &= \gamma\lambda E_{t-1}(s,a) + 1(s_t = s, a_t = a) \end{aligned} \tag{4.15}$$

After visiting a state-action pair, its eligibility increases by one and the eligibility of others decays. Furthermore, in addition to the $Q(s,a)$ of the state-action pair that has been visited, Sarsa (λ) updates all the Q-values of all other state-action pairs in proportion to TD-error and eligibility trace (See Algorithm 2).

Algorithm 2 Sarsa(λ) Algorithm

Initialize Q-values, $\forall s \in \mathbb{S},\ \forall a \in \mathbb{A}$;
while *Not (end of learning)* **do**
  Initialize S , A
  Choose action A based on the policy derived from Q-values (e.g., using ε-greedy)
  repeat
    Take action A
    Observe S', r
    Choose action A' based on the derived policy (e.g., using ε-greedy)
    $\delta = r + \gamma Q(S', A') - Q(S, A)$
    $E(S, A) = E(S, A) + 1$
    For all $s \in \mathbb{S},\ a \in \mathbb{A}$
    $\{\ Q(s, a) = Q(s, a) + \alpha\delta E(s, a)$
    $E(s, a) = \gamma\lambda E(s, a)\ \}$
    $S \leftarrow S'$; $A \leftarrow A'$
  until *meeting the terminal (target) state*;
end

Off-policy learning. Aside from the on-policy learning, off-policy learning is a similar learning paradigm that provides the capability of learning from observing others' behavior, which is a significant benefit over the on-policy learning. It can reuse the previous experiences guided by old policies or even follow an exploratory policy to learn the optimal policy.

Off-policy learning utilizes the concept of importance sampling to learn the optimal policy while following other policies. It evaluates the target policy, π—to compute $Q_\pi(s, a)$—while following a behavior policy, μ. It means that off-policy TD uses the TD targets derived from μ to evaluate the other policy, π. Thus, the primary updating is re-formulated based on using one-step importance sampling correction, which is as follows:

$$V(s_t) = V(s_t) + \alpha[\frac{\pi(a_t|s_t)}{\mu(a_t|s_t)}(r_{t+1} + \gamma V(s_{t+1}) - V(s_t))] \tag{4.16}$$

Q-learning, which applies off-policy learning to Q-values, is one of the primary, though quite successful off-policy learning algorithms. It is built on

TD(0) and works in a specific way without the need for doing explicitly importance sampling. In Q-learning, the next action, A_{t+1}, is chosen based on the behavior policy, μ, while an alternative successor action based on π, A', is also considered. It means that although the next action is chosen based on the behavior policy, the bootstrapping is done towards the Q-value of the alternative successor action. Thus, updating the Q-value is done as follows:

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha[r_{t+1} + \gamma Q(S_{t+1}, A') - Q(S_t, A_t)] \quad (4.17)$$

The behavior policy in the most common form of Q-learning (See Algorithm 3) is ε-greedy with regard to Q-values, and the target policy that we want to improve is *greedy* with regard to Q-values. Q-learning, in fact, lets both behavior and target policies improve at the same time. The target and the Q-value updates in the well-known type of Q-learning are simplified as follows:

$$\begin{aligned} r_{t+1} + \gamma Q(S_{t+1}, A') &= r_{t+1} + \gamma Q(S_{t+1}, \underset{a'}{\operatorname{argmax}}\, Q(S_{t+1}, a')) \\ &= r_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a') \end{aligned} \quad (4.18)$$

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha[r_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a') - Q(S_t, A_t)] \quad (4.19)$$

Algorithm 3 Q-Learning Algorithm

Initialize Q-values, $\forall s \in \mathbb{S},\ \forall a \in \mathbb{A}$;
while *Not (end of learning)* **do**
  Initialize S;
  repeat
    Choose action A based on the behavior policy (e.g., using ε-greedy);
    Take action A;
    Observe S', r;
    $Q(S, A) = Q(S, A) + \alpha[r + \gamma \max_{a'} Q(S', a') - Q(S, A)]$
    $S \leftarrow S'$
  until *meeting the terminal (target) state*;
end

4.2 Swarm and Evolutionary Computation

Evolutionary computing methods and swarm intelligence algorithms are two classes of bio-inspired algorithms mainly intended for solving optimization problems—finding optimal (best) solutions among all feasible ones. Genetic algorithms (GA) and evolution strategies (ES) are among the primary categories within the family of evolutionary algorithms (EAs). Particle swarm optimization (PSO) is one of the key representatives of swarm intelligence algorithms [97].

Genetic algorithm is one of the most common nature-inspired optimization techniques. It begins with a random population of individuals each of which is referred to as a chromosome, representing a potential solution for the problem. The problem's objectives are defined in an *objective (fitness) function* and the quality of the solutions is assessed using this function. It basically demonstrates how effectively (well) each solution "satisfies" the objective of the problem. Each generation produces a new population depending on the individuals chosen in the previous generation. The following are the three main operations involved in forming the new generation:

1. Selection: It identifies highly-valued individuals from the preceding generation.
2. Crossover: It breeds "child" individuals by swapping parts of the "parent" individuals.
3. Mutation: By applying mutation, individuals are subjected to minor random changes (adjustments).

Crossover and mutation operations are applied with respect to user-set probabilities, and they can be utilized alone or together to generate new individuals. The resulting individuals are added to the new population and the fitness values—based on the objective function—are calculated and stored for each individual in the new population. This process iterates through each generation until meeting stopping requirements, such as a user-specified number of generations or an allotted time limit, are fulfilled (See Algorithm 4) [57, 97].

Evolution strategy, as a common class of EAs, is also widely used in most of the optimization problems including those with discrete and

Algorithm 4 GA Algorithm

1. Initialize a population with random individuals
2. Evaluate the individuals in the population

repeat

3. Select highly-valued individuals using selection operation
4. Create offspring by applying crossover and mutation operation
5. Evaluate the offspring based on the objective function

until *meeting the termination criteria*;

6. Return the best individuals in the population

continuous input spaces. ES mainly entails applying recombination, mutation, and selection to a population of individuals across several generations. In ES, individuals, in addition to the set of parameters related to the problem, can also encompass a set of evolvable strategy parameters, which are used for tuning and controlling the statistical properties of the evolution operations (e.g., mutation). The $(\mu/\rho + \lambda)$ and $(\mu/\rho, \lambda)$ evolution strategies are two canonical forms of ES. If $\rho = 1$ the recombination is simply creating a replica of the parent. λ and μ represent the size of the offspring and population respectively. One of the key distinctions between GA and ES is connected to the selection process. In GA, a new generation is produced at each iteration by choosing highly-valued individuals while keeping the population size constant [97]. In ES, a temporary population of size λ is created and the individuals of this temporary population experience mutation at user-specified probabilities irrespective of their fitness values. In $(\mu + \lambda)$ ES, both parents and the resulting offspring from the temporary population are copied to a selection pool of size $(\mu + \lambda)$, then a new population with size μ is created by selecting the best individuals from the pool. On the contrary, in (μ, λ) ES, the individuals of the new generation—with size μ—are chosen only from the offspring (with size λ). Thus, a convergence condition as $\mu < \lambda$ is needed to ensure an optimal solution (See Algorithm 5, which presents a simple $(\mu + \lambda)$ ES) [84].

Particle swarm optimization is one of the primary representatives of the swarm intelligence (SI) algorithms, which comprise a large class of nature-inspired optimization techniques. The SI algorithms present the notion of collective intelligence, which is primarily defined as collective behavior

Algorithm 5 $(\mu + \lambda)$ ES

1. Initialize a population P with μ random individuals
2. Evaluate the individuals in the population P

repeat

3. Create a temporary population P_T with size λ by reproduction of individuals from the population P
4. Mutate the strategy parameters
5. Apply mutation to the individuals in the population P_T
6. Evaluate the offspring based on the objective function
7. Select μ highly-valued individuals using selection operation from the original population P and the offspring

until *meeting the termination criteria*;

8. Return the best individuals in the population

among a group of individuals. SI algorithms were inspired by collective behavior and self-organizing interactions between living agents in the nature, such as honey bees and ant colonies [98].

cooperation is a crucial component of PSO since each individual alters their searching pattern depending on their own and others' experiences. PSO begins with a swarm of randomly generated particles. Each particle has position and velocity vectors that are updated at each iteration based on the local and global best values. In PSO, each particle (individual) represents a potential solution and is often modeled as a vector containing n elements each of which represents a variable of the problem. PSO seeks the optimal solution through updating solutions and creating subsequent generations, but without utilizing evolution operators [99]. The position (elements) and velocity of each particle are updated as follows:

$$P^{t+1} = P^t + V^{t+1} \tag{4.20}$$

$$V^{t+1} = wV^t + c_1 r_1 (P^t_{best} - P^t) + c_2 r_2 (G^t_{best} - P^t) \tag{4.21}$$

where P^t and V^t are the position and velocity of the particle at iteration t. P^t_{best} and G^t_{best} are the local best position of the particle and the global best

one up to the iteration t, respectively. The first part of Equation 9.2 is regarded as *inertia*, which denotes the tendency of the particle to continue moving in the same direction, while the second part reflects a cognitive behavior and presents the tendency towards the local best position found by the particle. The last part is the social knowledge and reflects the tendency to follow the best position discovered so far by other particles.

At each iteration, the position of each particle is updated based on its velocity, and the velocity is governed by inertia and accelerated stochastically towards the local and global best values by r_1 and r_2, $0 < r_1, r_2 \leq 2$—random weights used to adjust the cognitive and social acceleration (See Algorithm 6). In Equation 9.2, w is inertia weight, which adjusts the ability of the swarm to change the direction and provides a balance between the level of exploration and exploitation in the search process. A lower w encourages more exploitation of the best solutions discovered, whereas a higher w promotes more exploration around the found solutions. The acceleration hyperparameters, c_1 and c_2, specify how much the solutions are influenced by the local best and global best solutions. These hyperparameters and the inertia weight could be set as static parameters or dynamically adjusted over iterations [100].

Algorithm 6 PSO Algorithm

1. Initialize a swarm with random particles
2. Evaluate the particles of the swarm based on the fitness function
3. Select the global best particle w.r.t the fitness value (G_{best})

repeat
  for *each particle P in the swarm* **do**
    4. Calculate particle's velocity according to Eq. 9.2
    5. Update particle's position according to Eq. 9.1
    6. Evaluate particle based on the fitness function
    if *fitness value of P is better than the local best of P, (P_{best})* **then**
      7. Update P_{best} with P
    end
    if *fitness value of P is better than the global best, (G_{best})* **then**
      8. Update G_{best} with P
    end
  end
until *meeting the termination criteria*;
9. Return the best particles in the swarm

4.3 Related Work

4.3.1 Performance Testing: Conventional Software Systems

Measuring performance metrics under various execution conditions, such as different workload and platform configurations [101, 102], and detecting various performance-related issues, such as functional problems or violations of performance requirements [103, 104] are common goals of various types of performance testing. Although there are some definitions/interpretations that distinguish between performance, load, and stress (robustness) testing, they are used interchangeably in many cases [25]. Stress testing is a sort of performance testing used to determine the robustness of a system under stress conditions such as high workload and/or restricted resource availability. Performance testing goals are often linked to the verification of performance requirements, and where performance criteria are unavailable, the notion of "no worse than before" is frequently applied [105, 106].

The following is a summarized overview of the approaches used to generate workload-based and platform-based performance test scenarios:

Analyzing system models. Using evolutionary search-based algorithms such as GA to generate test workload from the control flow graph of SUT [53], analyzing a performance model of SUT in Petri nets using constraint solving techniques [107], applying genetic algorithms to other types of system models such as UML to generate stress test workload [52, 108, 109, 110] are samples of the techniques used for generating performance test scenarios based on analyzing different types of system models of SUT.

Analyzing source code. In [111, 103], an analysis of SUT's data-flow together with symbolic execution is used to generate test load scenarios and find performance-related issues.

Modeling real usage. In [34, 35], form-oriented models are used to present users' behavior and identify the usage pattern of real users, which are then used to generate realistic test workload scenarios. In this regard, finding workload characteristics and modeling the user behavior based on Extended Finite State Machines [112] and Markov chains [113] via monitoring requests submitted to SUT are also other examples of modeling real usage and using usage patterns to generate test workload scenarios. In [114] workload characterization is done through clustering users based on the business-level attributes extracted from usage data, which is also another example of this class of techniques.

Declarative specification-based techniques. In [37, 39], a declarative Domain Specific Language (DSL) is used together with a model-driven test execution framework to specify and execute the performance testing scenarios. Using a specific behavior-driven language, to specify the load testing process in combination with a declarative performance testing framework like BenchFlow [36] is another example of using declarative techniques for generating and executing performance test scenarios.

Machine learning-enabled techniques. Machine learning techniques such as supervised and unsupervised algorithms are mainly used to create models and extract knowledge patterns from data, whereas reinforcement learning algorithms are used to train an intelligent agent how to accomplish a goal through interaction with the environment. Machine learning techniques have been widely used for analyzing the performance test results for different purposes, e.g., performance anomaly detection based on performance metrics data—resource usage—using clustering algorithms [44], reliability prediction

based on load testing data using Bayesian Network [115], performance signature identification based on performance metrics data [43, 116], and performance anomaly detection of microservices through analysis of latency distribution data [117].

Machine learning techniques were also used to generate performance test scenarios. For instance, in [41], RL together with symbolic execution is used to find the worst-case execution path within a SUT and in [118] RL is utilized to find a sequence of input values resulting in performance degradation. A feedback-driven learning technique [119] that extracts some rules from the execution traces to find the performance bottlenecks, i.e., the method calls which their execution highly affects the performance, and using Generative Adversarial Networks (GANs) for automated efficient performance test generations [120, 121] are also some other examples in this regard.

Additionally, some other adaptive techniques, search-based performance profilers, and fuzzers have been also used to generate performance test scenarios in some studies; for instance, a feedback-based approach based on search algorithms to benchmark an NFS server w.r.t changing the test workload is presented in [122] and an adaptive generation of test workload based on using some pre-defined tuning policies is proposed in [104]. Finally, a performance fuzzing tool for C programs to generate test inputs leading to the worst-case performance behavior in [123], and a GA-driven performance profiler that uses a GA search together with an execution trace analyzer to explore the SUT input parameter space for detecting performance bottlenecks in [124] are some other related examples.

4.3.2 Robustness Testing: AI Systems

Robustness is also one of the main focuses of performance requirements of AI systems [14, 15]. The robustness of ML-driven systems mainly indicates how well the system can perform when it is exposed to the inputs different but similar to those ones in the training data, and it can cover both environmental and system uncertainty [14].

Different test levels for ML systems mainly involve input data assurance, ML model, and integration testing [14, 63]. In a sense, ML model testing could be considered unit testing, whereas integration testing might be regarded as system testing since it focuses on issues arising after the integration of the ML

model into the system. Similar to traditional conventional software systems, black-box and white-box testing practices could be also valid for testing AI systems. Then, from a similar perspective, only the ML inputs and outputs are accessible for a black-box testing approach, whereas white-box testing includes access to the architecture, code, hyperparameters, and training/test data of the ML test subject. In this regard, Riccio et al. [63] introduced an extra type of ML testing approach, called data-box, that involves access to data as well as what has been required by black-box testing. Depending on the test level and the test subject, the test inputs could be, for instance, images as used in DeepTest [125], or test scenario configurations as used in [57]. The following is a quick summary of the most prevalent approaches for generating failure-revealing test data for testing AI systems:

Input data mutation is a term used to describe the process of changing data. This sort of mutation entails creating new inputs based on existing input data using various transformations. The input data mutation is basically done to find the inputs triggering failures or behaviors different from what is expected. For example, in DeepXplore [126], input transformation is used to identify inputs that cause distinct behaviors in comparable autonomous driving DNN models. Those transformations are often done to support metamorphic testing on the ML test subject. DeepTest [125] applies different metamorphic transformations to a set of seed images to reveal the erroneous behaviors of different Udacity DNN models for self-driving cars, while also striving to increase the level of neuron coverage. For the same purpose, DeepRoad [127] uses a GAN-based metamorphic testing approach to generate failure-revealing input images to test autonomous driving Udacity DNN models.

Manipulation of test scenarios is another significant category of approaches for generating test input data. The majority of the works in this category employ search-based strategies to explore the scenarios' search space for failure-revealing or collision-provoking test scenarios. In this respect, simulators and accordingly simulation-based testing have played a crucial role in generating and capturing critical failure-revealing test scenarios. Regarding the impacts of the simulators, Haq et al. [128] compare the outcomes of testing DNN-based ADAS using online and offline testing. Their findings clearly support a strong emphasis on online testing, as it can discover failures that could otherwise stay undetected in offline testing.

Various simulation-based testing approaches utilizing search-based techniques to address the generation of failure-revealing test scenarios have been presented in the literature. For example, Abdessalem et al. [129] use NSGA-II—as a multi-objective search algorithm—together with surrogate models to find critical test scenarios for a pedestrian detection system with fewer simulations and less computing time. Then, in another study, they use MOSA [130] (a many-objective optimization search algorithm) with objectives based on the branch coverage and some failure-based heuristics to identify failure-revealing feature interaction scenarios for integration testing of an autonomous car [131].

For a face key-points detection system in the automotive domain, Haq et al. [132] also utilize many-objective search algorithms to generate test data resulting in significant mispredictions. In order to test the pedestrian detection and emergency braking system of the Baidu Apollo—an autonomous driving platform—within the SVL simulator, Ebadi et al. [57] use GA along with a specific data structure to model the test scenario space and a safety-based heuristic to define the objective function.

4.3.3 Runtime Performance Control

In this thesis, runtime performance control refers to methods for keeping performance requirements satisfied and/or optimizing performance in the runtime, by modifying the application's tunable parameters or resource configuration in a way adaptive to changing runtime conditions.

Performance Preservation. Keeping performance continually compliant with the requirements for the systems subject to changing runtime conditions is a challenging task, and meanwhile an objective of interest for a variety of software systems running on various platforms. In this regard, ML techniques such as model-free RL have been frequently applied to runtime performance control of software systems running on different platforms. For example, in several studies RL-driven techniques have been used for adaptive service performance control in cloud environments according to quality of service (QoS) requirements [17, 20, 133, 46]. Imdoukh et al. [47], for container-based applications, utilize a long short-term memory (LSTM) neural network model to predict the future workload and adjust the configuration (e.g., number of needed containers) proactively to handle the

submitted workload ahead of time. Incerto et al. [21] use a model predictive control technique in combination with queuing networks (QN) to provide a performance adaptation mechanism for a load balancer system. This performance adaptation is to continually tune and configure QN's properties such as service rates, routing probabilities, and concurrency level, w.r.t the desired performance requirements.

The related performance preservation techniques, particularly applied to the context of real-time systems, are mainly connected to the timing properties of these systems and performed based on runtime monitoring of the properties. In this regard, Mezzetti et al. [134] present a timing properties preservation strategy using the Ada Ravenscar Profile. It does the timing properties preservation in three steps: enforcing the timing properties that must remain constant, monitoring the fundamentally variable timing properties, and handling the occurred violations. In [135, 136], on top of a real-time operating system, an auxiliary scheduler is presented that considers the timing properties of the tasks such as period, execution time, and deadline, creates real-time tasks with well-defined specifications and schedules them using the operating system's underlying scheduler. The goal of this auxiliary scheduler is to as far as possible keep the timing requirements of the tasks satisfied.

Givel et al. [22] present a runtime enforcement policy that pushes the system to attain an expected specific state—by adding delay—to control the behavior of the system as desired. The enforcement approach works based on an offline model-based analysis and is primarily meant for evaluating the embedded fault tolerance-related mechanisms. Damschen et al. [23] also introduce a command-based reconfiguration queue (CoRQ) as a runtime reconfiguration controller for real-time systems. It supports worst-case execution time (WCET) guarantee and can provide guaranteed latencies for the operations. Tracealyzer [137] is also another prevalent industrial tool for tracing, visualizing, and measuring different performance properties of real-time embedded systems.

Performance Optimization. Regarding runtime performance optimization—in our application context, i.e., data-intensive tasks in data grid—machine learning techniques, in particular RL-based approaches, have been widely used for resource allocation and task scheduling. A quick overview of a number of relevant studies in this regard is as follows:

Galstyan et. al [138] present a multi-agent RL approach for resource allocation in a gird environment. In the proposed solution, the agents do not communicate with each other and each agent assigns scores to the resources based on the resource efficiency in executing the tasks. To execute a new task, the agent chooses the resource with the highest score, gets a reward signal, and updates the selected resource's score. Weighted policy learner (WPL) [139] is also a gradient ascent multi-agent learning technique for distributed task allocation problem in domains like grids. Vengerov [140] introduces an RL-driven dynamic resource allocation solution augmented with a fuzzy rule base, called DRA-FRL. The proposed solution utilizes RL to maximize the utility function for resource allocation decisions in large-scale computing facilities with a big pool of shared resources.

Fair Action Learning (FAL) [141] is a multi-agent learning solution for distributed sequential resource allocation in a cluster-based network, in which each agent refers to a cluster and has a limited view of the whole system. The decisions of each agent are associated with two learning problems: local resource allocation problem, i.e., what resources tasks are to be allocated, and task routing, i.e., where the task should be forwarded to. FAL is mainly a direct policy search algorithm that uses Q-value to approximate the policy gradients. In [142], Wu et al. present a multi-agent RL-based solution, called Ordinal Sharing Learning (OSL), for large-scale grids that mainly focuses on balancing the load between the nodes. In OSL, the agents share the utility tables for decision making and make their decisions in an ordinal way.

In [143], Qureshi et al. in an extensive study survey various resource allocation approaches for different grid architectures w.r.t computing- and data-intensive tasks. Orhean et al. [144] use BURLAP library [145] to implement an RL-based task scheduler—based on Q-learning and SARSA—integrated into WorkflowSim toolkit. In [146], Cui et al. implement and evaluate a multi-agent Q-learning task scheduler for grid and IaaS cloud—integrated into both CloudSim toolkit and a private real computing platform. They consider a hierarchical cluster-based architecture and the Q-learning agents are augmented with a rule bank containing user data, Q-table and also knowledge transferred from other agents.

Dakkak et al. [147] present a resource allocation approach, called Swift Gap (SG), which focuses on improving slowdown, waiting time, and response time of the tasks. SG employs the backfilling technique and comprises two

steps as first, placing the job in the earliest available gap in the resources' schedules, and second, optimizing the job placement using Tabu search. In a further step, they improve SG with a new heuristic, called completion time scheme, that additionally reduces the start time, minimizes the processing time by choosing the fastest available resource, and outperforms SG w.r.t the intended performance metrics.

Chapter 5

Discussion, Conclusion and Future Work

In this chapter, we conclude with presenting a summary of our results, discussing the threats to validity, as well as outlining some potential directions for future research.

5.1 Discussion and Conclusion

Performance quality assurance is an important and challenging aspect of software quality assurance. In this thesis, we investigated the directions of robustness testing and runtime performance control within the body of performance quality assurance, and tried to consider both domains of conventional and ML-driven software systems. We identified some challenges and formulated three main research goals based on them. The research goals are effective and efficient generation of robustness test scenarios leading to the emergence of performance breaking points and requirements violations; and the development of runtime performance preservation and optimization techniques. We focused on five research questions shaping our path towards reaching those research goals; they were answered by seven studies included in this thesis.

In this thesis, we identified room for intelligence-driven techniques, i.e.,

model-free RL and bio-inspired search-based optimization algorithms to help tackle the targeted challenges and meet the research goals. In this regard, a summary of the achievements with respect to the research goals is given as follows:

RG1: Effective and efficient **generation of failure-revealing test scenarios**—in which the performance breaking point emerges, i.e., the performance requirements are violated— for conventional software systems, without dependency on the models and source code artifacts.

In this part of the thesis, we proposed and developed two smart performance test agents, SaFReL and RELOAD, that were able to effectively and efficiently generate critical performance test scenarios leading to the emergence of performance breaking points. These smart agents are intended to learn the optimal policy to meet the test objective. Therefore, via these test agents, first, test automation would be realized without the need for models or source code. Second, the learned policy could be reused in potential further situations, e.g., for testing other similar test subjects (SUTs) or to meet further test objectives on the SUT. The capability of knowledge formation and reusing knowledge leads to test efficiency improvement. The proposed test agents benefit from model-free RL—Q-learning and DQN—to learn the optimal policy and experience two phases of learning in the operation: initial and transfer learning.

SaFReL is a self-adaptive fuzzy RL-driven agent that generates the platform-based performance test scenarios efficiently. In other words, it learns the optimal policy to tune the resource availability in the test setup to find the performance breaking point for different types of SUT (in terms of their resource sensitivity) and replays the learned policy on similar test subjects (SUTs). To assess the idea of SaFReL, we conducted an empirical evaluation consisting of 50 software programs with different resource sensitivity (CPU-intensive, memory-intensive and disk-intensive types) and different response time requirements. Our results show that after the initial learning, due to the embedded adaptation mechanism—adaptive action selection strategy—SaFReL is able to adapt to different test subjects, i.e, strive for more policy reusing for test subjects with similar resource sensitivity and retune/update the learned policy once observing test subjects with different resource sensitivity. So, it can meet the test objective—find the performance breaking point—more efficiently than random testing, which is a common

Table 5.1: Average test cost saving of RELOAD in the initial learning

	RELOAD Learning Configurations			
Test Cost Saving	$\varepsilon = 0.5$	$\varepsilon = 0.2$	decaying ε	DQN setup
w.r.t Standard Baseline	30%	30%	34%	34%
w.r.t Random Testing	17%	17%	20%	20%

technique for this mission. It results in 42% and 31% test efficiency improvement when it does testing for a homogeneous and heterogeneous set of SUTs, respectively. A homogeneous set of SUTs refers to a set of software programs which are similar in terms of resource sensitivity.

RELOAD is the other smart test agent that generates a cost-efficient test workload to meet the test objective—find the performance breaking point. It identifies the effects of different constituent transactions of the workload and learns how to tune the workload optimally to meet the test objective. It is also able to reuse the learned policy for further similar test objectives on the SUT that has been trained on. RELOAD has been developed and evaluated based on both Q-learning and DQN, and uses Apache JMeter as the load test actuator to execute the designed test workload on the SUT. We evaluated RELOAD in terms of resulting test cost saving—w.r.t to four learning configurations—in comparison to random and a baseline load testing techniques. The results of the empirical evaluation show that RELOAD learns how to meet the objective with a more accurate and fine-tuned workload and subsequently leads to test cost saving with regard to the random and baseline approaches (See Table 5.1). Furthermore, after the initial learning, it is able to reuse the learned policy for testing w.r.t other objectives on the SUT and fairly still keep its efficiency. It can lead to considerable test cost saving in the transfer learning as well. Table 5.2 shows the resulting test cost reduction of RELOAD in the transfer learning for 10 testing episodes with new test objectives after the initial learning.

Overall, the capabilities of these smart test agents such as learning, policy reusing, and ”adaptiveness” (i.e., continual learning to adapt the learned policy to changes in the environment) make them well-suited for particularly continuous testing activities in CI/CD practices in software development life cycle. For instance, RELOAD could be beneficial for continual performance

Table 5.2: Efficiency and average test cost saving in the transfer learning

	RELOAD (Q-learning)	**Standard Testing**	**Random Testing**
Range of the number of generated virtual users	48-62	55-99	55-68
RELOAD test cost saving		25%	13%

testing where performance tests must be repeated for the SUT w.r.t various performance requirements or SaFReL could be used for continuous performance testing and quality assurance of product deltas (incremented software products) in an incremental software development process.

RG2: Effective and efficient **generation of failure-revealing test scenarios**—in which the safety requirements are violated—for automotive AI systems.

With regard to RG2, we proposed and developed two simulation-integrated bio-inspired search-based test solutions, Deeper and GA-driven ScenarioGenerator, to generate failure-revealing test scenarios leading to the violations of safety requirements in two AI systems.

Deeper is a test generator for effective and efficient generation of failure-revealing test scenarios to test a DNN-based lane-keeping system in the BeamNG driving simulator. The test subject is an ML-driven driving agent in the simulator—which utilizes a DNN-based steering angle predictor model for keeping the car inside the lane. To generate failure-revealing test scenarios, Deeper benefits from genetic algorithm (GA), $(\mu + \lambda)$ and (μ, λ) evolution strategies (ES), and particle swarm optimization (PSO) together with the heuristics of using an initial quality population seed and domain-specific crossover and mutation operations developed for the presentation model used for modeling the test scenarios. The comparative analysis on the performance of the proposed bio-inspired test generators in Deeper and five counterpart tools, shows that Deeper techniques perform as effective and efficient test generators that can provoke a considerable number of diverse failure-revealing test scenarios w.r.t different target failure severity (i.e., tolerance threshold), available test budget, and driving style constraints

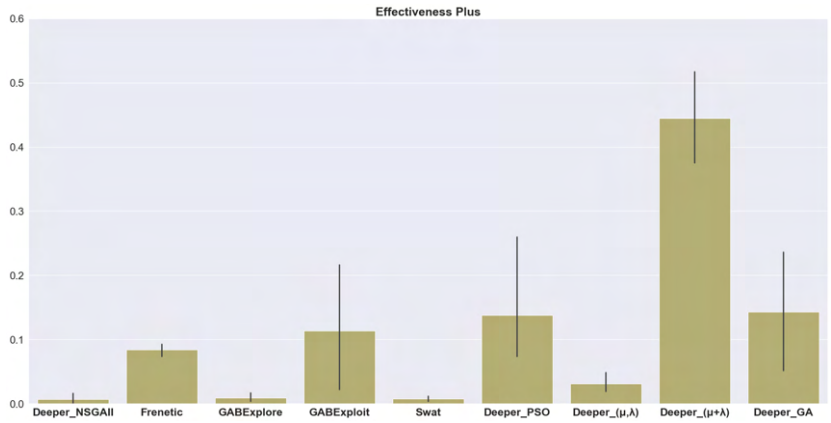


Figure 5.1: Test generation effectiveness

(e.g, speed limits). As some distinctive features, none of the test generators leaves the test without triggering any failures, and in particular, they prove to be more reliable test generators than most of the other tools for provoking diverse failures under a limited test budget and strict constraints. In this regard, Deeper $(\mu + \lambda)$ ES-, PSO-, and GA-driven result in high effectiveness in terms of the ratio of the number of detected failures to the generated valid test scenarios (See Figure 5.1—the effectiveness of Deeper test generators compared to counterpart tools in one of the experimental configurations in the empirical evaluation).

GA-Driven ScenarioGenerator is the other evolutionary search-based test generator that generates failure-revealing test scenarios, which lead to the safety requirements violations, for pedestrian detection and emergency braking system in Baidu Apollo within SVL simulator. GA-Driven ScenarioGenerator utilizes a generic data structure to model the parameters involved in creating test scenarios along with a GA-based technique to generate the test scenarios. It uses a safety heuristic to measure the quality of the test scenarios in terms of their potential to cause safety violations. The empirical evaluation of the approach shows that within a certain given budget, it triggers twice as many failures as a random testing technique (See Figure 5.2); meanwhile, it promotes a higher level of diversity between the triggered failures, as shown in Table 5.3.

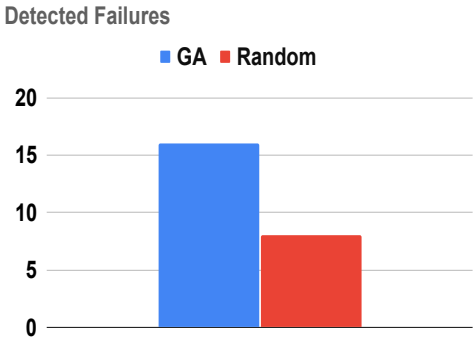


Figure 5.2: Test generation effectiveness

Table 5.3: Failure diversity, shown as the range of the average pairwise Euclidean distance between failure-revealing test scenarios

	Genetic Algorithm	Random
Range of Euclidean Distances	$4.1 - 4.7$	$3.2 - 4.2$

RG3: To develop adaptive **runtime performance control** to provide runtime performance preservation and optimization w.r.t varying execution conditions.

In order to meet RG3, we developed an adaptive learning-driven runtime performance preservation technique, which is an RL-based response time control approach for PLC-based industrial applications. The proposed approach aims to keep the performance of the system, i.e., response time, compliant with the timing requirements, through adaptive adjusting of tunable parameters in the runtime. The simulation-based empirical evaluation shows that the proposed approach—with different learning configurations—can effectively lead the program to stay compliant with the response time requirements despite the occurrence of runtime events resulting in response time deviations. For instance, in one of the learning configurations (i.e., ε-greedy, ε= 0.5), the proposed adaptive response time control can provide a high adaptation to the varying conditions and keep the response time very close to the requirement threshold (See Figure 5.3). It can be quite desired in

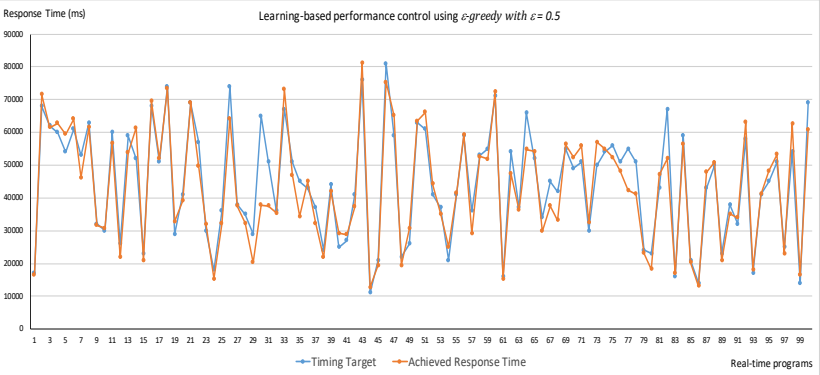


Figure 5.3: Response time control in the RL-driven performance preservation approach for PLC-based industrial applications

the cases where a sharp satisfaction of the requirements is needed, e.g., airbag control systems in the automotive domain. While, using another learning configuration, i.e., ε-greedy with decaying ε, the control approach can even keep the response time below the requirement threshold.

Regarding RG3, in the last study of the thesis, we also developed an adaptive learning-assisted runtime performance optimization technique, which was an RL-driven resource allocation approach for dynamic workloads in a data grid application. It involves smart RL-driven agents that learn the optimal way to allocate resources to the submitted tasks with respect to the type of the tasks and the current status of the grid in order to optimize the makespan (completion time) of the tasks. The empirical evaluation of the approach in a simulation environment shows that it can lead to optimizing the performance effectively, i.e., reduces the completion time of the tasks, compared to four common techniques used for resource allocation in this application. As shown in Figure 5.4—which presents the resulting performance from different techniques given using a Gaussian distribution pattern for task submission—the proposed learning-assisted approach adapts to the workload pattern and status of the environment, and accordingly assigns the tasks to the proper processing nodes. It keeps the makespan below a certain low threshold over the experiments compared to other algorithms. In this regard, the proposed approach with the learning configuration based on

ε-greedy, $\varepsilon = 0.2$ leads to the highest performance improvement in the experimental evaluation. The amount of performance improvement is also more considerable for bigger task sets, since in those experiments the learned policy results from more observations and then gets closer to the optimal one.

5.1.1 Threats to Validity

A number of sources of threat to the validity of the experimental results are as follows:

Construct validity. Regarding the proposed RL-driven approaches in the thesis, a main source of threat is the formulation of the reinforcement learning to address the problem. Modeling of the state space and formulations of actions and reward functions are primary factors to create a smart agent able to learn how to meet an intended objective in the problem. Regarding the bio-inspired search-based solutions in the thesis, the choices of the fitness functions, the proposed domain-specific evolutionary operations, i.e., crossover and mutation, and also the metrics used for calculating the diversity of the failure-revealing test scenarios—weighted Levenshtein and Euclidean distance—are domain-specific and sources of threat to construct validity.

Internal validity. The randomized nature of the action selection strategy in RL algorithms could be a source of threat to internal validity of the results in RL-driven solutions. Then, in order to mitigate the effects of this threat, we report the average values for the measured related metrics. Meanwhile, similar to other ML algorithms, RL techniques are also affected by the hyperparameters such as learning rate and discount factor. In this regard, we kept the hyperparameters fixed during the experiments conducted for the efficiency analysis of the RL-driven approaches, and also conducted a separate set of controlled experiments to examine the effects of the learning hyperparameters on the efficiency of the approaches.

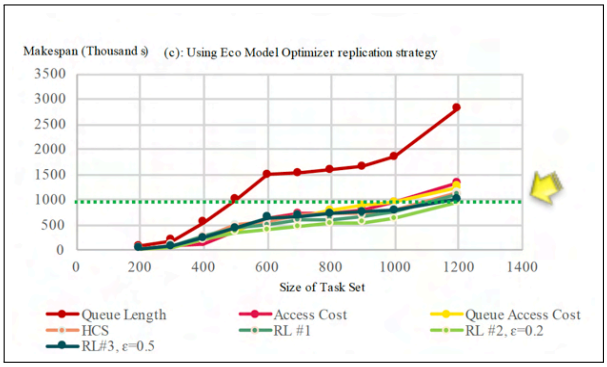
Regarding the search-based solutions, the randomness involved in the bio-inspired search algorithms could be similarly a source of threat to the internal validity of the results of this category of solutions. Therefore, for the empirical evaluation of these solutions we follow the guidelines given by Arcuri and Briand [148] and alleviate the effects of this threat by running the experiments multiple times, reporting the distribution of the results, and using the same settings, i.e., population size, crossover and mutation probabilities,

for the algorithms over the experiments.

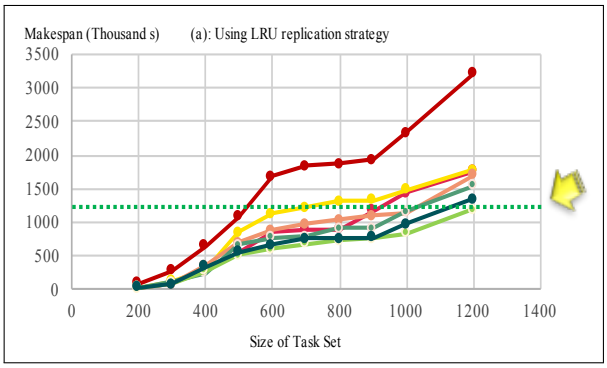
External validity. The choice of the SUT, e.g., an e-commerce web application in RELOAD study, a lane-keeping and a pedestrian detection and emergency braking systems as AI-enabled test subjects in other studies; and considering SUTs with certain performance sensitivity, i.e., CPU-intensive, memory-intensive and disk-intensive software programs in SaFReL study, are samples of potential threats to external validity of the results. However, for example, regarding AI-enabled test subjects, self-driving cars are one of the prevailing examples of safety-critical AI systems and the test subjects selected in the studies are among the commonly used ML-driven systems in automotive AI. Additionally, in those studies different ML models with various quality levels (i.e., different accuracy levels) could be deployed within the simulation environments that were used. Nonetheless, those studies still offer solutions for certain types of ML-driven systems in self-driving cars, and further studies are required to address the testing of other ML-driven systems.

Concerning RELOAD study, the proposed approach has been formulated based on a common e-commerce web application as SUT. However, in order to apply the approach to other web-based systems, the set of transactions for load testing of the new system should be extracted and included in the set of actions in the body of RELOAD agent. Regarding SaFReL study, in order to consider SUTs with other types of performance sensitivity such as network-intensive programs, then the approach needs to be slightly reformulated to support new types of performance sensitivities.

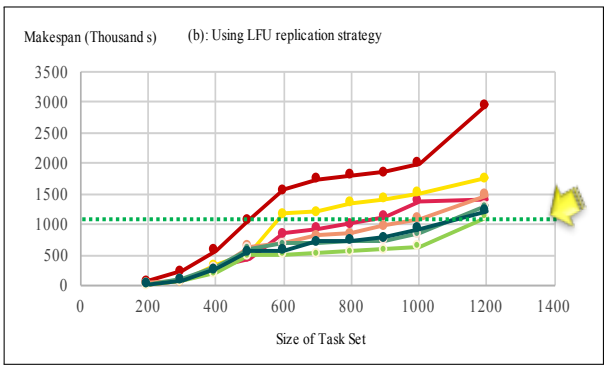
Similarly, the proposed runtime performance control approaches—learning-driven performance preservation and optimization techniques—have been formulated based on certain types of applications, and further studies are required to use the proposed approaches for other types of software systems.



(a)



(b)



(c)

Figure 5.4: Optimization of tasks completion time

5.2 Future Work

Regarding robustness testing, as one of the directions of software performance assurance, we mostly focused on tackling the challenge of effective and efficient generation of critical test scenarios leading to performance failures—violation of performance or safety requirements—using different computational intelligence-driven techniques. We tried to answer the question ”How could these test scenarios be generated more effectively and efficiently?”. However, in the context of performance testing, generation and selection of test scenarios is one step of the testing process and after executing the test scenarios, analyzing the test results is another essential step, in particular to augment the testing processes in CI/CD practices within DevOps context. In the real-life software development cycle, executing various types of performance testing might be planned upon changes (or increments) committed to the code repository, so regression performance testing could be a fundamental running flow in continuous testing. In this regard, addressing the research question ”What useful information could be discovered from the analysis of performance test results?” could be an interesting direction for future work. Analyzing the test results could lead to detection of some certain issues that might not be readily detectable during the test, e.g., potential performance bottlenecks that could lead to performance degradation further in certain conditions. In this regard, for instance, different statistical and ML techniques—such as clustering, Isolation forest, and correlation detection methods—could help with performance anomaly detection. Moreover, this analysis could be further used by possible adaptive performance control approaches included in the performance assurance activity later.

Within the scope of robustness testing, in one part of the thesis, we investigated effective and efficient generation of failure-revealing test scenarios in automotive AI systems through using bio-inspired search-based optimization algorithms. In this regard, there will be also interesting room to extend the proposed approaches by applying ML techniques such as RL or Generative Adversarial Networks (GANs) for empowering the discovery of failure-revealing test scenarios. Meanwhile, exploring evolutionary search-based or ML-based testing approaches for generating failure-revealing test scenarios for other types of ML-driven systems—within the domain of self-driving cars or other domains—would be also another potential direction

for future studies.

One more detailed view on the directions of future work would be related to improving or scaling up the proposed approaches to facilitate their use in real-world more complex problems. In some studies of the thesis, simulation-based empirical evaluations have been used to show the applicability of the proposed approaches and how the involved principles of the approaches work. In this regard, for example, there are some heuristics and customized techniques to facilitate the use of RL techniques in scaled-up environments with more complexity. Therefore, using scaling up techniques such as function approximations in place of Q-tables to deal with the problems with large MDPs or using multi-agent learning techniques to speed up the exploration of state space could be some future extensions with respect to the learning techniques.

Bibliography

[1] Christof Ebert, Gorka Gallardo, Josune Hernantes, and Nicolas Serrano. Devops. *IEEE Software*, 33(3):94–100, 2016.

[2] James Roche. Adopting DevOps practices in quality assurance. *Communications of the ACM*, 56(11):38–43, 2013.

[3] Nayan B Ruparelia. Software development lifecycle models. *ACM SIGSOFT Software Engineering Notes*, 35(3):8–13, 2010.

[4] ISO 25000. ISO/IEC 25010 - System and software quality models, 2019. Available at https://iso25000.com/index.php/en/iso-25000-standards/iso-25010, Retrieved April, 2022.

[5] Martin Glinz. On non-functional requirements. In *15th IEEE International Requirements Engineering Conference (RE 2007)*, pages 21–26. IEEE, 2007.

[6] Lawrence Chung, Brian A Nixon, Eric Yu, and John Mylopoulos. *Non-functional requirements in software engineering*, volume 5. Springer Science & Business Media, 2012.

[7] Leonid Grinshpan. *Solving enterprise applications performance puzzles: queuing models to the rescue*. John Wiley & Sons, 2012.

[8] Andreas Brunnert, André van Hoorn, Felix Willnecker, Alexandru Danciu, Wilhelm Hasselbring, Christoph Heger, Nikolas Herbst, Pooyan Jamshidi, Reiner Jung, Joakim von Kistowski, et al. Performance-oriented devops: A research agenda. *arXiv preprint arXiv:1508.04752*, 2015.

[9] Bold Eye Media. Your slow loading website is costing you money, 2020. Available at https://boldeyemedia.com/blog/your-slow-loading-website-is-costing-you-money/, Retrieved April, 2022.

[10] Elaine J Weyuker and Filippos I Vokolos. Experience with performance testing of software systems: issues, an approach, and case study. *IEEE transactions on software engineering*, 26(12):1147–1156, 2000.

[11] A definition of artificial intelligence: Main capabilities and scientific disciplines. Technical report, High-Level Expert Group on Artificial Intelligence, Brussels, Belgium, 2018.

[12] European Commission. Ethics guidelines for trustworthy AI. Available at https://digital-strategy.ec.europa.eu/en/library/ethics-guidelines-trustworthy-ai, Retrieved April, 2022.

[13] Andrej Karpathy. Software 2.0. Available at https://karpathy.medium.com/software-2-0-a64152b37c35, Retrieved April, 2022.

[14] Richard Hawkins, Colin Paterson, Chiara Picardi, Yan Jia, Radu Calinescu, and Ibrahim Habli. Guidance on the assurance of machine learning in autonomous systems (AMLAS). *arXiv preprint arXiv:2102.01564*, 2021.

[15] Jens Henriksson, Christian Berger, Markus Borg, Lars Tornberg, Sankar Raman Sathyamoorthy, and Cristofer Englund. Performance analysis of out-of-distribution detection on trained neural networks. *Information and Software Technology*, 130:106409, 2021.

[16] Mahshid Helali Moghadam, Mehrdad Saadatmand, Markus Borg, Markus Bohlin, and Björn Lisper. Adaptive runtime response time control in PLC-based real-time systems using reinforcement learning. In *2018 IEEE/ACM 13th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 217–223. IEEE, 2018.

[17] Olumuyiwa Ibidunmoye, Mahshid Helali Moghadam, Ewnetu Bayuh Lakew, and Erik Elmroth. Adaptive service performance control using cooperative fuzzy reinforcement learning in virtualized environments. In *Proceedings of the10th International Conference on Utility and Cloud Computing*, pages 19–28. ACM, 2017.

[18] Mahshid Helali Moghadam and Seyed Morteza Babamir. Makespan reduction for dynamic workloads in cluster-based data grids using reinforcement-learning based scheduling. *Journal of computational science*, 24:402–412, 2018.

[19] László Toka, Gergely Dobreff, Balázs Fodor, and Balázs Sonkoly. Machine learning-based scaling management for kubernetes edge clusters. *IEEE Transactions on Network and Service Management*, 18(1):958–972, 2021.

[20] T Veni and S Mary Saira Bhanu. Auto-scale: automatic scaling of virtualised resources using neuro-fuzzy reinforcement learning approach. *International Journal of Big Data Intelligence*, 3(3):145–153, 2016.

[21] Emilio Incerto, Mirco Tribastone, and Catia Trubiani. Software performance self-adaptation through efficient model predictive control. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 485–496, 2017.

[22] Louis-Marie Givel, Jean-Luc Béchennec, Matthias Brun, Sébastien Faucou, and Olivier H Roux. Testing real-time embedded software using runtime enforcement. In *2016 11th IEEE Symposium on Industrial Embedded Systems (SIES)*, pages 1–6. IEEE, 2016.

[23] Marvin Damschen, Lars Bauer, and Jörg Henkel. CoRQ: Enabling runtime reconfiguration under WCET guarantees for real-time systems. *IEEE Embedded Systems Letters*, 9(3):77–80, 2017.

[24] Geeks for Geeks. Difference between System Testing and Integration Testing. Available at `https://www.geeksforgeeks.org/difference-between-system-testing-and-integration-testing/`, Retrieved March, 2022.

[25] Zhen Ming Jiang and Ahmed E Hassan. A survey on load testing of large-scale software systems. *IEEE Transactions on Software Engineering*, 41(11):1091–1118, 2015.

[26] Zoltán Micskei, Henrique Madeira, Alberto Avritzer, István Majzik, Marco Vieira, and Nuno Antunes. Robustness testing techniques and tools. In *Resilience Assessment and Evaluation of Computing Systems*, pages 323–339. Springer, 2012.

[27] Dorina Petriu, Christiane Shousha, and Anant Jalnapurkar. Architecture-based performance analysis applied to a telecommunication system. *IEEE Transactions on Software Engineering*, 26(11):1049–1065, 2000.

[28] Simona Bernardi, Susanna Donatelli, and José Merseguer. From UML sequence diagrams and statecharts to analysable petri net models. In *Proceedings of the 3rd international workshop on Software and performance*, pages 35–45. ACM, 2002.

[29] Edward D Lazowska, John Zahorjan, G Scott Graham, and Kenneth C Sevcik. *Quantitative system performance: computer system analysis using queueing network models*. Prentice-Hall, Inc., 1984.

[30] Vittorio Cortellessa, Antinisca Di Marco, and Paola Inverardi. *Model-based software performance analysis*. Springer Science & Business Media, 2011.

[31] Mor Harchol-Balter. *Performance modeling and design of computer systems: queueing theory in action*. Cambridge University Press, 2013.

[32] Krishna Kant and MM Srinivasan. *Introduction to computer system performance evaluation*. McGraw-Hill College, 1992.

[33] Giovanni Denaro, Andrea Polini, and Wolfgang Emmerich. Early performance testing of distributed software applications. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 94–103. ACM, 2004.

[34] Dirk Draheim, John Grundy, John Hosking, Christof Lutteroth, and Gerald Weber. Realistic load testing of web applications. In *Conference*

on Software Maintenance and Reengineering (CSMR'06), pages 11–pp. IEEE, 2006.

[35] Christof Lutteroth and Gerald Weber. Modeling a realistic workload for performance testing. In *2008 12th International IEEE Enterprise Distributed Object Computing Conference*, pages 149–158. IEEE, 2008.

[36] Henning Schulz, Dušan Okanović, André van Hoorn, Vincenzo Ferme, and Cesare Pautasso. Behavior-driven load testing using contextual knowledge-approach and experiences. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, pages 265–272. ACM, 2019.

[37] Vincenzo Ferme and Cesare Pautasso. A declarative approach for performance tests execution in continuous software development environments. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 261–272. ACM, 2018.

[38] Vincenzo Ferme and Cesare Pautasso. Towards holistic continuous software performance assessment. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, pages 159–164. ACM, 2017.

[39] Jürgen Walter, Andre van Hoorn, Heiko Koziolek, Dusan Okanovic, and Samuel Kounev. Asking what?, automating the how?: The vision of declarative performance engineering. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, pages 91–94. ACM, 2016.

[40] Pingyu Zhang, Sebastian Elbaum, and Matthew B Dwyer. Compositional load test generation for software pipelines. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 89–99. ACM, 2012.

[41] Jinkyu Koo, Charitha Saumya, Milind Kulkarni, and Saurabh Bagchi. PySE: Automatic worst-case test generation by reinforcement learning. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 136–147. IEEE, 2019.

[42] Charitha Saumya, Jinkyu Koo, Milind Kulkarni, and Saurabh Bagchi. Xstressor: Automatic generation of large-scale worst-case test inputs by inferring path conditions. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 1–12. IEEE, 2019.

[43] Haroon Malik, Hadi Hemmati, and Ahmed E Hassan. Automatic detection of performance deviations in the load testing of large scale systems. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 1012–1021. IEEE Press, 2013.

[44] Mark D Syer, Bram Adams, and Ahmed E Hassan. Identifying performance deviations in thread pools. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 83–92. IEEE, 2011.

[45] Jacopo Soldani and Antonio Brogi. Anomaly detection and failure root cause analysis in (micro) service-based cloud applications: A survey. *ACM Computing Surveys (CSUR)*, 55(3):1–39, 2022.

[46] Alexandros Kontarinis, Verena Kantere, and Nectarios Koziris. Cloud resource allocation from the user perspective: A bare-bones reinforcement learning approach. In *International Conference on Web Information Systems Engineering*, pages 457–469. Springer, 2016.

[47] Mahmoud Imdoukh, Imtiaz Ahmad, and Mohammad Gh Alfailakawi. Machine learning-based auto-scaling for containerized applications. *Neural Computing and Applications*, 32(13):9745–9760, 2020.

[48] Wasif Afzal, Richard Torkar, and Robert Feldt. A systematic review of search-based testing for non-functional system properties. *Information and Software Technology*, 51(6):957–976, 2009.

[49] Abdullah Alourani, Md Abu Naser Bikas, and Mark Grechanik. Search-based stress testing the elastic resource provisioning for cloud-based applications. In *International symposium on search based software engineering*, pages 149–165. Springer, 2018.

[50] Du Shen, Qi Luo, Denys Poshyvanyk, and Mark Grechanik. Automating performance bottleneck detection using search-based application

profiling. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 270–281, 2015.

[51] Vahid Garousi. Empirical analysis of a genetic algorithm-based stress test technique. In *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 1743–1750. ACM, 2008.

[52] Vahid Garousi. A genetic algorithm-based stress test requirements generator tool and its empirical evaluation. *IEEE Transactions on Software Engineering*, 36(6):778–797, 2010.

[53] Yuanyan Gu and Yujia Ge. Search-based performance testing of applications with composite services. In *2009 International Conference on Web Information Systems and Mining*, pages 320–324. IEEE, 2009.

[54] Raja Ben Abdessalem, Shiva Nejati, Lionel C. Briand, and Thomas Stifter. Testing vision-based control systems using learnable evolutionary algorithms. In *Proc. of the 40th International Conference on Software Engineering*, pages 1016–1026, 2018.

[55] Markus Borg, Raja Ben Abdessalem, Shiva Nejati, François-Xavier Jegeden, and Donghwan Shin. Digital twins are not monozygotic–cross-replicating ADAS testing in two industry-grade automotive simulators. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 383–393. IEEE, 2021.

[56] Mahshid Helali Moghadam, Markus Borg, and Seyed Jalaleddin Mousavirad. Deeper at the SBST 2021 tool competition: ADAS testing using multi-objective search. In *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*, pages 40–41. IEEE, 2021.

[57] Hamid Ebadi, Mahshid Helali Moghadam, Markus Borg, Gregory Gay, Afonso Fontes, and Kasper Socha. Efficient and effective generation of test cases for pedestrian detection-search-based software testing of Baidu Apollo in SVL. In *2021 IEEE International Conference on Artificial Intelligence Testing (AITest)*, pages 103–110. IEEE, 2021.

[58] Olumuyiwa Ibidunmoye, Francisco Hernández-Rodriguez, and Erik Elmroth. Performance anomaly detection and bottleneck identification. *ACM Computing Surveys (CSUR)*, 48(1):4, 2015.

[59] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM computing surveys (CSUR)*, 41(3):15, 2009.

[60] Brendan Gregg. *Systems performance: enterprise and the cloud*. Pearson Education, 2013.

[61] Mahshid Helali Moghadam. Machine learning-assisted performance assurance. *Licentiate Thesis, Mälardalen University*, 2020.

[62] Jie M Zhang, Mark Harman, Lei Ma, and Yang Liu. Machine learning testing: Survey, landscapes and horizons. *IEEE Transactions on Software Engineering*, 2020.

[63] Vincenzo Riccio, Gunel Jahangirova, Andrea Stocco, Nargiz Humbatova, Michael Weiss, and Paolo Tonella. Testing machine learning based systems: a systematic mapping. *Empirical Software Engineering*, 25(6):5193–5254, 2020.

[64] Road Vehicles - Safety of the Intended Functionality. International Organization for Standardization, Tech. Rep. ISO/PAS 21448:2019, 2019.

[65] Florian Bock, Christoph Sippl, Sebastian Siegl, and Reinhard German. Status report on automotive software development. In *Automotive Systems and Software Engineering*, pages 29–57. Springer, 2019.

[66] Philip Koopman and Michael Wagner. Challenges in autonomous vehicle testing and validation. *SAE International Journal of Transportation Safety*, 4(1):15–24, 2016.

[67] Robert Feldt. Guide to research questions, 2010. Available at `http://www.robertfeldt.net/advice/index.html`, Retrieved March, 2022.

[68] Gordana Dodig-Crnkovic. Scientific methods in computer science. In *Proceedings of the Conference for the Promotion of Research in IT at*

New Universities and at University Colleges in Sweden, Skövde, Suecia, pages 126–130, 2002.

[69] Hilary J Holz, Anne Applin, Bruria Haberman, Donald Joyce, Helen Purchase, and Catherine Reed. Research methods in computing: what are they, and how should we teach them? *ACM SIGCSE Bulletin*, 38(4):96–114, 2006.

[70] Victor Basili, Lionel Briand, Domenico Bianculli, Shiva Nejati, Fabrizio Pastore, and Mehrdad Sabetzadeh. Software engineering research and industry: a symbiotic relationship to foster impact. *IEEE Software*, 35(5):44–49, 2018.

[71] Tony Gorschek, Per Garre, Stig Larsson, and Claes Wohlin. A model for technology transfer in practice. *IEEE software*, 23(6):88–95, 2006.

[72] Claes Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th international conference on evaluation and assessment in software engineering*, pages 1–10, 2014.

[73] Samireh Jalali and Claes Wohlin. Systematic literature studies: database searches vs. backward snowballing. In *Proceedings of the 2012 ACM-IEEE international symposium on empirical software engineering and measurement*, pages 29–38. IEEE, 2012.

[74] Colin Robson and Kieran McCartan. *Real world research*. John Wiley & Sons, 2016.

[75] Forrest Shull, Janice Singer, and Dag IK Sjøberg. *Guide to advanced empirical software engineering*. Springer, 2007.

[76] Mahshid Helali Moghadam, Mehrdad Saadatmand, Markus Borg, Markus Bohlin, and Björn Lisper. An autonomous performance testing framework using self-adaptive fuzzy reinforcement learning. *Software quality journal*, pages 1–33, 2021.

[77] Mahshid Helali Moghadam, Mehrdad Saadatmand, Markus Borg, Markus Bohlin, and Björn Lisper. Machine learning to guide performance testing: An autonomous test framework. In *2019*

IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pages 164–167. IEEE, 2019.

[78] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[79] Mahshid Helali Moghadam, Golrokh Hamidi, Markus Borg, Mehrdad Saadatmand, Markus Bohlin, Björn Lisper, and Pasqualina Potena. Performance testing using a smart reinforcement learning-driven test agent. In *2021 IEEE Congress on Evolutionary Computation (CEC)*, pages 2385–2394, 2021.

[80] Apache. Jmeter. Available at `https://https://jmeter.apache.org/`, Retrieved March, 2022.

[81] Mahshid Helali Moghadam, Mehrdad Saadatmand, Markus Borg, Markus Bohlin, and Björn Lisper. Poster: Performance testing driven by reinforcement learning. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 402–405. IEEE, 2020.

[82] Mahshid Helali Moghadam, Markus Borg, Mehrdad Saadatmand, Seyed Jalaleddin Mousavirad, Markus Bohlin, and Björn Lisper. Machine learning testing in an ADAS case study using simulation-integrated bio-inspired search-based testing. Technical report, Mälardalen University, arXiv preprint arXiv:2203.12026, March 2022.

[83] Thomas Back. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford university press, 1996.

[84] Hans-Georg Beyer and Hans-Paul Schwefel. Evolution strategies–a comprehensive introduction. *Natural computing*, 1(1):3–52, 2002.

[85] Yuhui Shi. Particle swarm optimization. *IEEE connections*, 2(1):8–13, 2004.

[86] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.

[87] Ezequiel Castellano, Ahmet Cetinkaya, Cédric Ho Thanh, Stefan Klikovits, Xiaoyi Zhang, and Paolo Arcaini. Frenetic at the SBST 2021 tool competition. In *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*, pages 36–37, 2021.

[88] Florian Klück, Lorenz Klampfl, and Franz Wotawa. GABezier at the SBST 2021 tool competition. In *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*, pages 38–39, 2021.

[89] Dmytro Humeniuk, Giuliano Antoniol, and Foutse Khomh. SWAT tool at the SBST 2021 tool competition. In *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*, pages 42–43, 2021.

[90] Baidu. Apollo. Available at `https://apollo.auto/devcenter/devcenter.html`, Retrieved April, 2022.

[91] LG Electronics America Research Center. SVL Simulator: An Autonomous Vehicle Simulator. Available at `https://www.svlsimulator.com/docs/getting-started/getting-started/`, Retrieved April, 2022.

[92] Taiwo Oladipupo Ayodele. Types of machine learning algorithms. *New advances in machine learning*, pages 19–48, 2010.

[93] Sotiris B Kotsiantis, I Zaharakis, and P Pintelas. Supervised machine learning: A review of classification techniques. *Emerging artificial intelligence applications in computer engineering*, 160:3–24, 2007.

[94] Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.

[95] Carl Edward Rasmussen. Gaussian processes in machine learning. In *Summer School on Machine Learning*, pages 63–71. Springer, 2003.

[96] Csaba Szepesvári. Algorithms for reinforcement learning. *Synthesis lectures on artificial intelligence and machine learning*, 4(1):1–103, 2010.

[97] Adam Slowik and Halina Kwasnicka. Evolutionary algorithms and their applications to engineering problems. *Neural Computing and Applications*, pages 1–17, 2020.

[98] Michalis Mavrovouniotis, Changhe Li, and Shengxiang Yang. A survey of swarm intelligence for dynamic optimization: Algorithms and applications. *Swarm and Evolutionary Computation*, 33:1–17, 2017.

[99] Adam Slowik and Halina Kwasnicka. Nature inspired methods and their industry applications—swarm intelligence algorithms. *IEEE Transactions on Industrial Informatics*, 14(3):1004–1015, 2017.

[100] Maurice Clerc and James Kennedy. The particle swarm-explosion, stability, and convergence in a multidimensional complex space. *IEEE transactions on Evolutionary Computation*, 6(1):58–73, 2002.

[101] Varsha Apte, TVS Viswanath, Devidas Gawali, Akhilesh Kommireddy, and Anshul Gupta. AutoPerf: Automated load testing and resource usage profiling of multi-tier internet applications. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, pages 115–126. ACM, 2017.

[102] Anshul Jindal, Vladimir Podolskiy, and Michael Gerndt. Performance modeling for cloud microservice applications. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, pages 25–32. ACM, 2019.

[103] Pingyu Zhang, Sebastian Elbaum, and Matthew B Dwyer. Automatic generation of load tests. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 43–52. IEEE Computer Society, 2011.

[104] Vanessa Ayala-Rivera, Maciej Kaczmarski, John Murphy, Amarendra Darisa, and A Omar Portillo-Dominguez. One size does not fit all: In-test workload adaptation for performance testing of enterprise applications. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 211–222. ACM, 2018.

[105] Frank Huebner, Kathleen Meier-Hellstern, and Paul Reeser. Performance testing for IP services and systems. In *Performance Engineering*, pages 283–299. Springer, 2000.

[106] Ivo Jimenez, Noah Watkins, Michael Sevilla, Jay Lofstead, and Carlos Maltzahn. Quiho: Automated performance regression testing using inferred resource utilization profiles. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 273–284. ACM, 2018.

[107] Jian Zhang and Shing Chi Cheung. Automated test case generation for the stress testing of multimedia systems. *Software: Practice and Experience*, 32(15):1411–1435, 2002.

[108] Vahid Garousi, Lionel C Briand, and Yvan Labiche. Traffic-aware stress testing of distributed real-time systems based on UML models using genetic algorithms. *Journal of Systems and Software*, 81(2):161–185, 2008.

[109] Leandro T Costa, Ricardo M Czekster, Flávio Moreira de Oliveira, Elder de M Rodrigues, Maicon Bernardino da Silveira, and Avelino F Zorzo. Generating performance test scripts and scenarios based on abstract intermediate models. In *SEKE*, pages 112–117, 2012.

[110] Maicon Bernardino da Silveira, Elder de M Rodrigues, Avelino F Zorzo, Leandro T Costa, Hugo V Vieira, and Flávio Moreira de Oliveira. Generation of scripts for performance testing based on UML models. In *SEKE*, pages 258–263, 2011.

[111] Cheer-Sun D Yang and Lori L Pollock. Towards a structural load testing tool. In *ACM SIGSOFT Software Engineering Notes*, volume 21, pages 201–208. ACM, 1996.

[112] Mahnaz Shams, Diwakar Krishnamurthy, and Behrouz Far. A model-based approach for testing the performance of web applications. In *Proceedings of the 3rd international workshop on Software quality assurance*, pages 54–61. ACM, 2006.

[113] Christian Vögele, André van Hoorn, Eike Schulz, Wilhelm Hasselbring, and Helmut Krcmar. WESSBAS: extraction of probabilistic workload

specifications for load testing and performance prediction—a model-driven approach for session-based application systems. *Software & Systems Modeling*, 17(2):443–477, 2018.

[114] Gururaj Maddodi, Slinger Jansen, and Rolf de Jong. Generating workload for ERP applications through end-user organization categorization using high level business operation data. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 200–210. ACM, 2018.

[115] Alberto Avritzer, Flávio P Duarte, Rosa Maria Meri Leao, Edmundo de Souza e Silva, Michal Cohen, and David Costello. Reliability estimation for large distributed software systems. In *Cascon*, page 12. Citeseer, 2008.

[116] Haroon Malik, Zhen Ming Jiang, Bram Adams, Ahmed E Hassan, Parminder Flora, and Gilbert Hamann. Automatic comparison of load tests to support the performance analysis of large enterprise systems. In *2010 14th European conference on software maintenance and reengineering*, pages 222–231. IEEE, 2010.

[117] Richard Li, Min Du, Zheng Wang, Hyunseok Chang, Sarit Mukherjee, and Eric Eide. LongTale: Toward automatic performance anomaly explanation in microservices. In *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*, pages 5–16, 2022.

[118] Tanwir Ahmad, Adnan Ashraf, Dragos Truscan, and Ivan Porres. Exploratory performance testing using reinforcement learning. In *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 156–163. IEEE, 2019.

[119] Mark Grechanik, Chen Fu, and Qing Xie. Automatically finding performance problems with feedback-directed learning software testing. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 156–166. IEEE, 2012.

[120] Ali Sedaghatbaf, Mahshid Helali Moghadam, and Mehrdad Saadatmand. Automated performance testing based on active deep

learning. In *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*, pages 11–19. IEEE, 2021.

[121] Ivan Porres, Hergys Rexha, and Sébastien Lafond. Online GANs for automatic performance testing. In *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 95–100. IEEE, 2021.

[122] Piyush Shivam, Varun Marupadi, Jeffrey S Chase, Thileepan Subramaniam, and Shivnath Babu. Cutting corners: Workbench automation for server benchmarking. In *USENIX Annual Technical Conference*, pages 241–254, 2008.

[123] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. Perffuzz: Automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 254–265, 2018.

[124] Du Shen, Qi Luo, Denys Poshyvanyk, and Mark Grechanik. Automating performance bottleneck detection using search-based application profiling. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 270–281, 2015.

[125] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th international conference on software engineering*, pages 303–314. IEEE/ACM, 2018.

[126] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *proceedings of the 26th Symposium on Operating Systems Principles*, pages 1–18. ACM, 2017.

[127] Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. DeepRoad: GAN-based metamorphic testing and input validation framework for autonomous driving systems. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 132–142. IEEE, 2018.

[128] Fitash Ul Haq, Donghwan Shin, and Lionel Briand. Efficient online testing for DNN-enabled systems using surrogate-assisted and many-objective optimization. In *Proceedings of the 44th International Conference on Software Engineering (ICSE'22)*. ACM, 2022.

[129] Raja Ben Abdessalem, Shiva Nejati, Lionel C Briand, and Thomas Stifter. Testing advanced driver assistance systems using multi-objective search and neural networks. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 63–74, 2016.

[130] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Reformulating branch coverage as a many-objective optimization problem. In *2015 IEEE 8th international conference on software testing, verification and validation (ICST)*, pages 1–10. IEEE, 2015.

[131] Raja Ben Abdessalem, Annibale Panichella, Shiva Nejati, Lionel C Briand, and Thomas Stifter. Testing autonomous cars for feature interaction failures using many-objective search. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 143–154. IEEE, 2018.

[132] Fitash Ul Haq, Donghwan Shin, Lionel C Briand, Thomas Stifter, and Jun Wang. Automatic test suite generation for key-points detection DNNs using many-objective search (experience paper). In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 91–102, 2021.

[133] Pooyan Jamshidi, Amir Sharifloo, Claus Pahl, Hamid Arabnejad, Andreas Metzger, and Giovani Estrada. Fuzzy self-learning controllers for elasticity management in dynamic cloud architectures. In *2016 12th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA)*, pages 70–79. IEEE, 2016.

[134] Enrico Mezzetti, Marco Panunzio, and Tullio Vardanega. Preservation of timing properties with the ada ravenscar profile. In *International Conference on Reliable Software Technologies*, pages 153–166. Springer, 2010.

[135] Mehrdad Saadatmand, Mikael Sjödin, and Naveed Ul Mustafa. Monitoring capabilities of schedulers in model-driven development of real-time systems. In *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation (ETFA 2012)*, pages 1–10. IEEE, 2012.

[136] Nima Asadi, Mehrdad Saadatmand, and Mikael Sjödin. Run-time monitoring of timing constraints: A survey of methods and tools. In *the Eighth International Conference on Software Engineering Advances*, 2013.

[137] Johan Kraft, Anders Wall, and Holger Kienle. Trace recording for embedded systems: Lessons learned from five industrial projects. In *International Conference on Runtime Verification*, pages 315–329. Springer, 2010.

[138] Aram Galstyan, Karl Czajkowski, and Kristina Lerman. Resource allocation in the grid with learning agents. *Journal of Grid Computing*, 3(1-2):91–100, 2005.

[139] Sherief Abdallah and Victor Lesser. Learning the task allocation game. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 850–857. ACM, 2006.

[140] David Vengerov. A reinforcement learning approach to dynamic resource allocation. *Engineering Applications of Artificial Intelligence*, 20(3):383–390, 2007.

[141] Chongjie Zhang, Victor Lesser, and Prashant Shenoy. A multi-agent learning approach to online distributed resource allocation. In *Twenty-First International Joint Conference on Artificial Intelligence*, 2009.

[142] Jun Wu, Xin Xu, Pengcheng Zhang, and Chunming Liu. A novel multi-agent reinforcement learning approach for job scheduling in grid computing. *Future Generation Computer Systems*, 27(5):430–439, 2011.

[143] Muhammad Bilal Qureshi, Maryam Mehri Dehnavi, Nasro Min-Allah, Muhammad Shuaib Qureshi, Hameed Hussain, Ilias Rentifis, Nikos

Tziritas, Thanasis Loukopoulos, Samee U Khan, Cheng-Zhong Xu, et al. Survey on grid resource allocation mechanisms. *Journal of Grid Computing*, 12(2):399–441, 2014.

[144] Alexandru Iulian Orhean, Florin Pop, and Ioan Raicu. New scheduling approach using reinforcement learning for heterogeneous distributed systems. *Journal of Parallel and Distributed Computing*, 117:292–302, 2018.

[145] James MacGlashan. Brown-UMBC reinforcement learning and planning (BURLAP) java code library.

[146] Delong Cui, Zhiping Peng, Jianbin Xiong, Bo Xu, and Weiwei Lin. A reinforcement learning-based mixed job scheduler scheme for grid or iaas cloud. *IEEE Transactions on Cloud Computing*, 8(4):1030–1039, 2017.

[147] Omar Dakkak, Yousef Fazea, Shahrudin Awang Nor, and Suki Arif. Towards accommodating deadline driven jobs on high performance computing platforms in grid computing environment. *Journal of Computational Science*, 54:101439, 2021.

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

II

Included Papers

Chapter 6

Paper A: Machine Learning to Guide Performance Testing: An Autonomous Test Framework

Mahshid Helali Moghadam, Mehrdad Saadatmand, Markus Borg, Markus Bohlin, and Björn Lisper
In the Proceedings of the 12th IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), IEEE 2019.

Abstract

Satisfying performance requirements is of great importance for performance-critical software systems. Performance analysis to provide an estimation of performance indices and ascertain whether the requirements are met is essential for achieving this target. Model-based analysis as a common approach might provide useful information but inferring a precise performance model is challenging, especially for complex systems. Performance testing is considered as a dynamic approach for doing performance analysis. In this work-in-progress paper, we propose a self-adaptive learning-based test framework which learns how to apply stress testing as one aspect of performance testing on various software systems to find the performance breaking point. It learns the optimal policy of generating stress test cases for different types of software systems, then replays the learned policy to generate the test cases with less required effort. Our study indicates that the proposed learning-based framework could be applied to different types of software systems and guides towards autonomous performance testing.

6.1 Introduction

Nowadays, on one hand we face the increasing number of software-based services, on the other hand, the expectations on the quality of these services are considerably rising. In general, the properties of a software system could be described in terms of functional and non-functional aspects. Non-functional properties describe the quality of functional aspects of the system and represent quality attributes like *performance* [1]. Requirements are the main means against which the functional and non-functional aspects of a system are assessed. The non-functional requirements are often described based on the software metrics quantifying the non-functional properties of the system. Assessing the satisfaction of non-functional requirements plays a crucial role in assuring the user's expected quality and even the behavioral correctness in many systems, particularly resource-constrained, and safety critical systems.

Performance is a non-functional property indicating the operational efficiency of a software system with respect to different execution conditions like various types of workload and allocation of available resources [2]. It is measured and quantified using multiple indices such as response time, throughput, and resource utilization. Performance analysis could be done through both performance modeling and performance testing. Performance modeling generally involves identifying the proper performance indices, building a performance model expressing the relevant indices. Consequently, different model-driven engineering techniques like model verification, model refactoring, and performance tuning could be performed based on the performance model.

Performance testing is intended to ascertain whether the software system performs well under the actual execution conditions (i.e., internal and external factors affecting the performance) and meets the performance requirements. Various methods have been proposed for building software performance models [3, 4, 5, 6]. Performance models might provide helpful hints of the performance of the system and even probable bottlenecks in the architecture; however, they cannot represent the whole details of the system. For example, many of the details of the deployment environment may be ignored in the models [7], although they might still have significant effects on the performance of the system. Testing the software system under stress, which is

called stress testing, is one of directions involved in performance testing. The main objective is to find the performance breaking point, at which the systems breaks, or performance requirements are not met anymore. Two general views as internal and external could be assigned to performance analysis. Analysis with internal view considers internal conditions causing a performance bottleneck and consequently affecting the performance of the system. Performance testing with external view is evaluating/examining how the system will perform under different external conditions like heavier workload and limited resource availability.

In this paper, we present a learning-based self-adaptive framework for providing autonomous performance testing. The proposed smart framework is able to learn how to apply stress testing efficiently to different types of software systems, including CPU-intensive, memory-intensive and disk-intensive programs, to find the performance breaking point. It basically uses model-free reinforcement learning (RL), i.e., Q-learning with multiple experience (knowledge) bases to learn the policy for finding performance breaking point of different types of software under test (SUT) without having performance models.

The rest of this paper is organized as follows; Section 6.2 discusses the background concepts of RL and the motivation for proposing learning-based testing, Section 6.3 presents the details of the proposed smart performance testing framework, with a short discussion on its applicability and operational performance. Section 6.4 provides a review on the background relevant approaches. The paper concludes with a conclusion and future directions of this work-in-progress research in Section 6.5.

6.2 Motivation and Background

Performance analysis is an essential step towards performance assurance to keep the performance requirements satisfied. Performance testing and performance modeling are dynamic and static approaches for realizing performance analysis. Regarding complex systems, providing a precise model of the system and execution environment is challenging. In the context of performance testing, the complexity of SUTs and the dynamism of the performance affecting factors in execution environments are the major barriers motivating application of learning-based performance testing.

Reinforcement learning [8] has been frequently used as one of the key approaches for building self-adaptive smart systems. RL is a semi-supervised learning involving interaction with the environment. In RL, an agent (the learner) continuously detects the status (state) of the environment (the system under control). Then, it selects an action to be applied on the environment and in return it receives a reinforcement signal (reward signal) showing the effectiveness of the action. The final objective during the learning, is to find a policy maximizing the total long-term received reward. The agent mainly uses a strategy based on a combination of trying out actions (exploration) and selecting highly valued actions (exploitation). Q-learning [8] is a well-known model-free algorithm in the context of RL, in which the agent learns the utility value of the long-term reward associated to pairs of states and actions. Q-learning is off-policy, since the agent learns the optimal policy independently of the selected strategy for the action selection step.

6.3 Self-Adaptive Learning-Based Performance Testing

This section presents the architecture and operating procedure of a smart framework providing autonomous performance testing. It focuses on stress testing as one of the main target fields in the scope of performance testing. It supports automated performance test case generation for different software systems without having performance models. The proposed framework as a smart agent uses reinforcement learning as its core learning mechanism. It aims at learning how to find the performance breaking point of various software systems depending on their performance sensitivity nature. The learning mechanism includes *initial convergence* and *transfer learning phases*.

Initial convergence. An initial experience (knowledge) convergence is achieved upon the first learning episodes in interaction with the first SUT instance of each type. The smart agent stores the achieved experience under three experience (knowledge) bases, i.e., experience for CPU-intensive, memory-intensive and disk-intensive SUTs. Therefore, the experience bases initially converge upon interaction with the first CPU-intensive, memory-intensive and disk-intensive SUT respectively.

Transfer learning. After the initial convergence of experience bases, the smart agent keeps the learning running to update the knowledge bases upon observing new SUTs. It is supposed that during the transfer learning, the smart agent mostly relies on the achieved experience, while also partly explores the environment to keep the gained knowledge updated. Using the learnt policy during the interaction with SUT instances, causes the agent to generate the stress test cases/test conditions to find the performance breaking point with less effort (in terms of learning trials) and leads to a better efficiency. Experience exploitation is the key concept of this phase which results in more efficiency in test case generation. The policies learnt for CPU-, memory- and disk-intensive programs are quite different. Then, this is where separating the experience bases of the agent is beneficial. Upon observing a CPU-, memory- or disk-intensive SUT, the agent activates the corresponding experience base for taking actions on the observed SUT instance. Figure 6.1 depicts an overview of the architecture of our smart tester agent. The details including the components, and main steps of the learning part is as follows:

I. State Detection. Detecting the current state of the system is one of the main steps of an RL-based algorithm. In our smart framework, four measurements of the SUT and execution environment including CPU, memory and disk utilization, and also SUT response time are used to specify the state of the system. The state detector component receives a tuple consisting of (CPU_U, Mem_U, $Disk\_U$, Rt) as input to specify the state of the system, where CPU_U, Mem_U, $Desk\_U$, Rt present the CPU, memory, disk utilization and response time respectively. These continuous parameters form the state space of the system, then the next step is dividing the state space into multiple discrete states. The values of these parameters are classified into multiple classes to specify the discrete states of the system, as shown in Figure 6.2.

II. Apply Action. After state detection, the agent applies one possible action to the system. Actions are operations which change (reduce) the available resources including CPU cores, memory and disk, and also change the factors affecting the performance like increasing the workload. In the first prototype of our smart framework, actions include the operations modifying the available resources by a decreasing factor:

$$DecFac\_CPU = \left\{\tfrac{1}{4}, \tfrac{2}{4}, \tfrac{3}{4}, 1\right\} \tag{6.1}$$

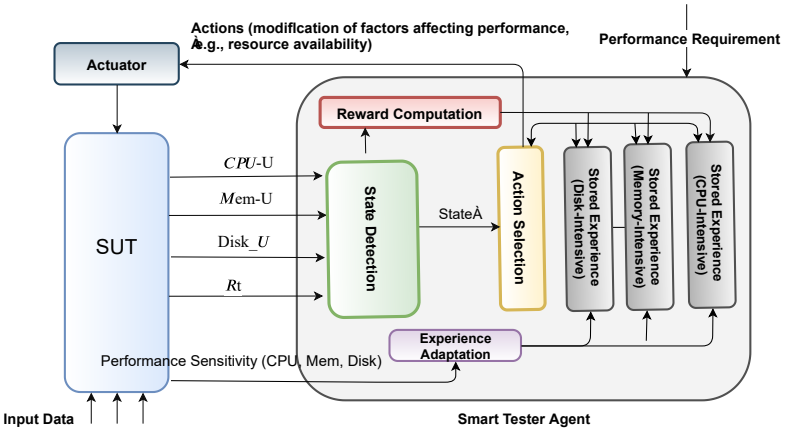


Figure 6.1: Architecture of the smart framework

$$DecFac\_MemDisk = \left\{ d.\frac{memory(disk)}{4} \mid d \in \left\{ \frac{1}{4}, \frac{2}{4}, \frac{3}{4}, 1 \right\} \right\} \tag{6.2}$$

where memory ($disk$) represent the current amount of available memory (disk). Then, the set of actions have been specified as shown in Table 6.1.

III. Compute Reward. After applying each action, the agent computes a reward signal showing the effectiveness of the applied action. The reward is calculated using the following utility function:

$$U(n) = kU_r(n) + (1-k)U_E \tag{6.3}$$

where $U_r(n)$ indicates to what extent the response time of the system deviates from the acceptable region, U_E represents the efficiency of the resource usage, and $k, 0 \leq k \leq 1$ is a weighting parameter to allow the agent to prioritize different aspects of the stress conditions.

IV. Experience Adaptation. This component receives a performance sensitivity indication expressing the type of sensitivity of SUT, i.e., being CPU-, memory- or disk-intensive. Then, it selects the corresponding

experience (knowledge) base for the stress test case generation on the observed SUT.

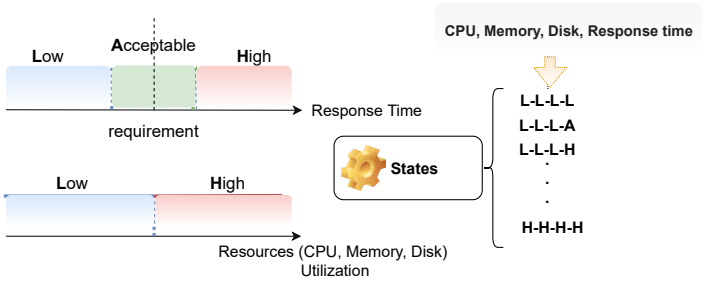


Figure 6.2: States of the system

Table 6.1: set of actions

Actions	
Reducing the available CPU cores	by a factor in $DecFac\_CPU$
Reducing the available memory	by a factor in $DecFac\_MemDisk$
Reducing the available disk	
No action	

How the learning works. The concept of the achieved experience in RL is defined in terms of policy. A policy is defined as a mapping between the states and actions and specifies the action which should be taken in each state. A utility value, $Q^{\pi}(s,a)$, is assigned to taking action, a, in given state s, according to the policy π. $Q^{\pi}(s,a)$ as the expected long-term reward of the pair (s,a) is defined as follows [8]:

$$Q^{\pi}(s,a) = E^{\pi}[R_n | S_n = s, A_n = a] \tag{6.4}$$

$$R_n = \sum_{k=0}^{\infty} \gamma^k r_{n+k+1} \tag{6.5}$$

Where S_n ,A_n and $r_{(n+k+1)}$ are current state, action and future rewards respectively. $\gamma \in [0,1]$ is a discount factor specifying to what extent the agent gives more weight to the future reward compared to the immediately achieved

reward. Q-values are stored in a look-up table (Q-table) and considered as the experience of the agent. The Q-values are used for deciding between actions when the agent relies on using its experience (exploitation). The Q-values are also updated incrementally (via temporal differencing) during the learning using Eq. 6.6. The final objective of Q-learning is finding a policy maximizing the expected long-term reward of pairs of states and actions.

$$Q(s_n, a_n) = Q(s_n, a_n) + \alpha[r_{n+1} + \gamma \max_{a'} Q(s_{n+1}, a') - Q(s_n, a_n)] \quad (6.6)$$

Learning performance. Different methods like ε-greedy with various ε-values and Softmax can be used for action selection. They provide different trade-offs between exploration and exploitation of the state-action space which could impact the efficiency of the learning e.g., in terms of convergence speed. Setting different values for learning parameters such as discount factor γ, and learning rate α could also affect the learning performance.

How the performance test is done. In our smart framework, the agent learns how to provide efficiently stress conditions for different types of SUTs to find the performance breaking point, from which the performance requirement of the SUT is not met anymore. The agent stops applying actions upon reaching the breaking point. Algorithm 7 presents the operating procedure of our learning-based performance testing framework.

Applicability. Performance-critical programs are sensitive parts in many software-intensive systems like industrial control systems. The proposed framework provides a model-free autonomous performance testing which could be easily applied to different types of software-intensive systems. Virtualized test environments would be a perfect infrastructure for executing this approach. Moreover, the proposed framework would be also integrated into the simulation environments which could be highly useful for testing purposes on industrial software systems.

6.4 Related Work

Model-based software performance engineering is mainly based on building a performance model of the system. Some of the specific modeling notations used for performance modeling are Queueing Networks, Markov Process, Petri Nets, Process Algebras and also simulation models [3, 4, 5, 6]. Pushing

towards automation in generating the performance model is essential to eliminate the manual model generation and bring the performance analysis to early stages in the software life cycle. There has been a substantial literature published in the area of software performance modeling [2, 9, 10]. In the context of testing, although performance testing tightly overlaps with performance modeling in some cases, it is intended to satisfy some partially different objectives. Performance testing, load testing and stress testing are three terms which are used in many cases interchangeably [11].

Algorithm 7 Learning-based Performance Testing

Initialize Q-values, $Q(s, a) = 0 \, \forall \, s \in S, \forall \, a \in A$;
Observe and identify the type of the SUT instance;
if SUT is the first instance of CPU-intensive, memory-intensive or disk-intensive instances {
repeat
1. Detect the state of the SUT;
2. Select an action based on the action selection strategy;
3. Apply the selected action, re-execute the SUT;
4. Detect the new state of the system regarding the selected action;
5. Receive the reward signal, r_{n+1};
6. Update the Q-value in the corresponding experience base (Q-table);

until *the initial convergence* `/* Initial Convergence */`;
} **Else** {
7. Select the proper experience (knowledge) base;
repeat
8. Detect the state of the SUT;
9. Select an action,
$a_n = argmax_{a \in A} Q(s_n, a)$ from the experience base with probability (1- ε) or a random action with probability $\varepsilon, \varepsilon \leq 0.2$);
10. Apply the selected action, re-execute the SUT;
11. Detect the new state of the system;
12. Receive the reward signal, r_{n+1};
13. Update the Q-value;
until *finding performance breaking point* `/* Transfer learning */`;
}

In general, load testing has been considered as behavior assessment of a SUT under load expected in a real-world execution context, from two perspectives of functional problems and violation of non-functional requirements caused under load. Stress testing has been defined as the behavioral assessment of a SUT under extreme conditions including heavy load and limited available resources [11]. Performance testing is often considered as a more general term which often includes both load testing and stress testing. There are many commonalities between these types of testing. Regarding common and different performance test scenarios, the behavioral assessment of a SUT from the perspective of performance-related issues and aspects generally aim at the following objectives:
I. Measurement of performance under load and/or different resource configuration. This process might overlap with performance modeling in many cases. It can be done under expected load or stress conditions [12, 13, 14, 15].
II. Detection of functional problems under load and/or different resource configuration. It can be also done under expected load or stress conditions [16, 17].
III. Detection of performance requirements violation such as violating reliability, and robustness requirements. This process can also be done under typical expected load or stress conditions [18].

This work-in-progress paper proposes a learning-based framework for performance testing, in particular stress testing.

6.5 Conclusion

Performance analysis to provide an estimation of performance indices in different execution conditions is a challenge for complex software systems. In addition to static model-driven techniques, performance testing is considered as a dynamic approach for performance analysis. Efficient automated test case/test condition generation is a challenging activity in software testing. In this paper, we present a self-adaptive learning-based framework to conduct stress testing on various software systems without having the performance models of the systems. We used Q-learning as a model-free RL algorithm in our smart test framework. It learns the optimal policy of generating stress test cases for various software systems. After the initial learning convergence, it

uses the learnt policy for further SUT instances and generates test cases efficiently with less required effort. Detailed efficacy analysis of the proposed framework on different software systems and deploying it on a virtualized infrastructure will be our next steps in this research.

Acknowledgements

This research has been funded partially by Vinnova through the ITEA3 TESTOMAT (www.testomatproject.eu) and XIVT.

Bibliography

[1] Martin Glinz. On non-functional requirements. In *15th IEEE International Requirements Engineering Conference (RE 2007)*, pages 21–26. IEEE, 2007.

[2] Vittorio Cortellessa, Antinisca Di Marco, and Paola Inverardi. *Model-based software performance analysis*. Springer Science & Business Media, 2011.

[3] Dorina Petriu, Christiane Shousha, and Anant Jalnapurkar. Architecture-based performance analysis applied to a telecommunication system. *IEEE Transactions on Software Engineering*, 26(11):1049–1065, 2000.

[4] Rob Pooley. Using UML to derive stochastic process algebra models. 1999.

[5] Simona Bernardi, Susanna Donatelli, and José Merseguer. From UML sequence diagrams and statecharts to analysable petri net models. In *Proceedings of the 3rd international workshop on Software and performance*, pages 35–45, 2002.

[6] Edward D Lazowska, John Zahorjan, G Scott Graham, and Kenneth C Sevcik. *Quantitative system performance: computer system analysis using queueing network models*. Prentice-Hall, Inc., 1984.

[7] Giovanni Denaro, Andrea Polini, and Wolfgang Emmerich. Early performance testing of distributed software applications. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 94–103. ACM, 2004.

[8] Richard S Sutton, Andrew G Barto, et al. *Introduction to reinforcement learning*, volume 135. MIT press Cambridge, 1998.

[9] Mike Kosta Loukides. *System performance tuning*. O'Reilly & Associates, Inc., 1996.

[10] Mor Harchol-Balter. *Performance modeling and design of computer systems: queueing theory in action*. Cambridge University Press, 2013.

[11] Zhen Ming Jiang and Ahmed E Hassan. A survey on load testing of large-scale software systems. *IEEE Transactions on Software Engineering*, 41(11):1091–1118, 2015.

[12] Daniel A Menascé. Load testing, benchmarking, and application performance management for the web. In *Int. CMG Conference*, pages 271–282, 2002.

[13] Mitashree Kalita and Tulshi Bezboruah. Investigation on performance testing and evaluation of prewebd: A. net technique for implementing web application. *IET software*, 5(4):357–365, 2011.

[14] Mahshid Helali Moghadam, Mehrdad Saadatmand, Markus Borg, Markus Bohlin, and Björn Lisper. Learning-based response time analysis in real-time embedded systems: a simulation-based approach. In *Proceedings of the 1st International Workshop on Software Qualities and Their Dependencies*, pages 21–24. ACM, 2018.

[15] Olumuyiwa Ibidunmoye, Mahshid Helali Moghadam, Ewnetu Bayuh Lakew, and Erik Elmroth. Adaptive service performance control using cooperative fuzzy reinforcement learning in virtualized environments. In *Proceedings of the10th International Conference on Utility and Cloud Computing*, pages 19–28. ACM, 2017.

[16] Bruno Dillenseger. Clif, a framework based on fractal for flexible, distributed load testing. *annals of telecommunications-annales des télécommunications*, 64(1-2):101–120, 2009.

[17] Frank Huebner, Kathleen Meier-Hellstern, and Paul Reeser. Performance testing for IP services and systems. In *Performance Engineering*, pages 283–299. Springer, 2000.

[18] Saeed Abu-Nimeh, Suku Nair, and Marco Marchetti. Avoiding denial of service via stress testing. In *IEEE International Conference on Computer Systems and Applications, 2006.*, pages 300–307. IEEE, 2006.

Chapter 7

Paper B: An Autonomous Performance Testing Framework Using Self-Adaptive Fuzzy Reinforcement Learning

Mahshid Helali Moghadam, Mehrdad Saadatmand, Markus Borg, Markus Bohlin, and Björn Lisper
Software Quality Journal, 1-33, Springer 2021.

Abstract

Test automation brings the potential to reduce costs and human effort, but several aspects of software testing remain challenging to automate. One such example is automated performance testing to find performance breaking points. Current approaches to tackle automated generation of performance test cases mainly involve using source code or system model analysis or use-case based techniques. However, source code and system models might not always be available at testing time. On the other hand, if the optimal performance testing policy for the intended objective in a testing process instead could be learned by the testing system, then test automation without advanced performance models could be possible. Furthermore, the learned policy could later be reused for similar software systems under test, thus leading to higher test efficiency. We propose SaFReL, a self-adaptive fuzzy reinforcement learning-based performance testing framework. SaFReL learns the optimal policy to generate performance test cases through an initial learning phase, then reuses it during a transfer learning phase, while keeping the learning running and updating the policy in the long term. Through multiple experiments in a simulated performance testing setup, we demonstrate that our approach generates the target performance test cases for different programs more efficiently than a typical testing process, and performs adaptively without access to source code and performance models.

7.1 Introduction

Quality assurance with respect to both functional and non-functional quality characteristics of software becomes crucial to the success of software products. For example, an extra one-second delay in load time of a storefront page can cause 11% reduction in page views, and 16% less customer satisfaction [1]. Moreover, banking, retailing, and airline reservation systems as samples of mission-critical systems are all required to be resilient against varying conditions affecting their functional performance [2, 3, 4].

Performance, which has been also called "efficiency" in the classification schemes of quality characteristics [5, 6, 7], is generally referred to as how well a software system (service) accomplishes the expected functionalities. Performance requirements mainly describe time and resource bound constraints on the behavior of software, which are often expressed in terms of performance metrics such as response time, throughput, and resource utilization.

Performance evaluation. Performance modeling and testing are common evaluation approaches to accomplish the associated objectives such as measurement of performance metrics, detection of functional problems emerging under certain performance conditions, and also violations of performance requirements [8]. Performance modeling mainly involves building a model of the software system's behavior using modeling notations such as queueing networks, Markov processes, Petri nets, and simulation models [9, 10, 11]. Although models provide helpful insights into the performance behavior of the system, there are also many details of implementation and execution platform that might be ignored in the modeling [12]. Moreover, drawing a precise model expressing the performance behavior of the software under different conditions is often difficult. Performance testing as another family of techniques is intended to achieve the aforementioned objectives by executing the software under the actual conditions.

Verifying robustness of the system in terms of finding performance breaking point is one of the primary purposes of performance testing. A performance breaking point refers to the status of software at which the system becomes unresponsive or certain performance requirements get violated.

Research challenge. Performance testing to find performance breaking points remains a challenge for complex software and execution platforms. Testing approaches mainly raise issues of automated and efficient generation of test cases (test conditions) resulting in accomplishing the intended objective. Common approaches for generating the performance test cases such as using source code analysis [13], linear programs and evolutionary algorithms on performance models [14, 15, 16] and UML models [17, 18, 19, 20, 21], using use case-based [22, 23], and behavior-driven techniques [24, 25, 26, 27] mainly rely on source code or other artifacts, which might not always be available during the testing.

Regarding the aforementioned issues, we propose that machine learning techniques could tackle them. One category of machine learning algorithms is reinforcement learning (RL), which is mainly intended to train an agent (learner) on how to solve a problem in an environment through being rewarded or punished in a trial and error interaction with the environment. Model-free RL is a subset of RL enabling the learner to explore the environment (the behavior of the software under test (SUT) in an execution environment in our case) and learn the optimal policy, to accomplish the objective (generating performance test cases resulting in an intended performance breaking point in our case) without access to source code and a model of the system. The learner can store the learned policy and is able to replay the learned policy in future situations, which can lead to efficiency improvements.

Goal of the paper. Our research goal is represented by the following question:

How can we adaptively and efficiently generate the performance test cases resulting in the performance breaking points for different software programs without access to the underlying source code and performance models?

Finding performance breaking point is a key purpose in robustness analysis, which is of great importance for many types of software systems, particularly in mission- and safety-critical domains [28]. Moreover, the question above is worth exploring also in applications specifically, such as resource management (scaling, provisioning and scheduling) for cloud services [29], performance prediction [30, 31], and performance analysis of software services in other areas [32, 33].

Contribution. In this paper, we present the design and experimental

evaluation of a self-adaptive fuzzy reinforcement learning-based (SaFReL) performance testing framework. It is intended to efficiently and adaptively generate the (platform-based) performance test conditions leading to the performance breaking point for different software programs with different performance sensitivity to resources (e.g., CPU-, memory-, and disk-intensive programs) without access to source code and performance models. An early-stage general formulation of the idea of using RL particularly in performance testing was introduced in our prior work [34]. The initial formulation introduces a single smart tester agent that uses RL (simple Q-learning) in a two-phase learning together with an initial architecture in the abstract. This paper extends the initial abstract formulation of the RL-assisted performance testing [34]. It uses an elaborate learning technique originally inspired by the conference paper by [35], which presents an adaptive performance (response time) control approach for cloud services using cooperative fuzzy multi-agent reinforcement learning. However, regarding the distinguishing learning details, the proposed RL-assisted performance testing framework is based on a single smart agent, involves two distinct phases of learning, and benefits a particular adaptive learning strategy which plays an important role in the functionality of the agent. The proposed smart performance testing framework is intended to conduct performance testing to meet a testing objective that is finding an intended performance breaking point. The proposed framework, SaFReL, is a two-phase RL-assisted performance testing agent that is able to learn the efficient generation of performance test cases to meet the testing objective and more importantly replay the learned policy in further similar testing situations.

SaFReL assumes two phases of learning: initial and transfer learning. In the initial learning phase, it learns the optimal policy to generate the target performance test cases initially upon observing the behavior of the first SUT. Afterward in the transfer learning, it reuses the learned policy for the SUTs with a performance sensitivity analogous to already observed ones while still keeping the learning running in the long term. The learning mechanism uses Q-learning augmented by fuzzy logic in one part of the learning to deal with the issue of uncertainty in defining discrete categories over continuous values as used by [35]. The single light-weight RL tester agent has the capability of transfer learning and reusing knowledge in similar situations. It benefits an adaptive action selection strategy that adapts the learning to various testing

situations and subsequently makes the agent able to act efficiently on various SUTs.

We demonstrate that SaFReL works adaptively and efficiently on different sets of SUTs, which are either homogeneous or heterogeneous in terms of their performance sensitivity. Our experiments are based on simulating the performance behavior of 50 instances of 12 well-known programs as the SUTs. Those instances are characterized by various initial amounts of granted resources and different values of response time requirements. We use two evaluation criteria, namely efficiency and adaptivity, to evaluate our approach. We investigate the efficiency of the approach in generating the test cases that result in reaching the intended performance breaking point and also the behavioral sensitivity of the approach to the learning parameters. In particular, SaFReL reaches the intended objective more efficiently compared to a typical stress testing technique, which generates the performance test cases based on changing the conditions, e.g., decreasing the availability of resources, by certain steps in an exploratory way. SaFReL leads to reduced cost (in terms of computation time) for performance test case generation by reusing the learned policy upon the SUTs with similar performance sensitivity. Moreover, it adapts its operational strategy to various SUTs with different performance sensitivity effectively while preserving efficiency. To summarize, our contributions in this paper are:

- A smart performance testing framework (agent) that learns the optimal policy (way) to generate the performance test cases meeting the testing objective without access to source code and models, and reuses the learned policy in further testing cases. It uses fuzzy RL and an adaptive action selection strategy for the generation of test cases, and implements two phases of learning:
  - Initial learning during which the agent learns the optimal policy for the first time,
  - Transfer learning during which the agent replays the learned policy in similar cases while keeping the learning running in the long term.
- A two-fold experimental evaluation involving performance (efficiency and adaptivity) and sensitivity analysis of the approach.

The evaluation is carried out based on simulating the performance behavior of various SUTs. We use a performance simulation module instead of actually executing SUTs. The main function of the performance simulation module is estimating the performance behavior of SUTs in terms of their response time.

Structure of the paper. The rest of the paper is organized as follows: Section 7.2 discusses the background concepts and motivations for the proposed self-adaptive learning-based approach. Section 7.3 presents an overview of the architecture of the proposed testing framework, while the technical details of the constituent parts are described in Sections 7.4 and 7.5. In Section 7.6, we explain the functions of the learning phases. Section 7.7 reports on the experimental evaluation involving the experiments setup, and the results of the experimentation. Section 7.8 discusses the results, the lessons learned during the experimentation, and also the threats to the validity of the results. Section 7.9 provides a review on the related work, and finally Section 7.10 concludes the paper and discusses some future directions.

7.2 Motivation and Background

Performance analysis, realized through modeling or testing, is important for performance-critical software systems in various domains. Anomalies in performance behavior of a software system or violations of performance requirements are generally consequences of the emergence of performance bottlenecks at the system or platform levels [36, 37]. A performance bottleneck is a system or resource component limiting the performance of the system and hinders the system from acting as required [38]. The behavior of a bottleneck component is due to some limitations associated with the component such as saturation and contention. A system or resource component saturation happens upon full utilization of its capacity or when the utilization exceeds a usage threshold [38]. Capacity expresses the maximum available processing power, service (giving) rate, or storage size. Contention occurs when multiple processes contend for accessing a limited number of shared components including resource components like CPU cycles, memory, and disk or software (application) components.

There are various application-, platform- and workload-based causes for

the emergence of performance bottlenecks [36]. Application-based causes represent issues such as defects in the source code or system architecture faults. Platform-based causes characterize the issues related to hardware resources, operating system, and execution platform. High deviations from the expected workload intensity and similar issues such as workload burstiness are denoted by workload-based causes.

On the other hand, detecting violations of performance requirements and finding performance breaking points are challenging, particularly for complex software systems. To address these challenges, we need to find how to provide critical execution conditions that make the performance bottlenecks emerge. The focus of performance testing in our case is to assess the robustness of the system and find the performance breaking point.

The effects of the internal causes (application/architecture-based ones) could vary, e.g., due to continuous changes and updates of the software during Continuous Integration/Continuous Delivery (CI/CD), and even vary upon different execution platforms and under different workload conditions. Therefore, the complexity of SUT and a variety of affecting factors make it hard to build a precise performance model expressing the effects of all types of factors at play. This is a major barrier motivating the use of model-free learning-based approaches like model-free RL in which the optimal policy for accomplishing the objective could be learned indirectly through interaction with the environment (SUT and the execution platform). In this problem statement, the testing system learns the optimal policy to achieve the target that is finding an intended performance breaking point, for different types of software without access to a model of the environment. The testing system explores the behavior of the SUT through varying the platform-based (and workload-based in future work) test conditions, stores the learned policy and is able to later reuse the learned policy in similar situations, i.e., other SUTs with similar performance sensitivity to resource restriction. This is the feature of the proposed learning approach that is supposed to lead to a considerable reduction in the testing system's effort, and subsequently saving computation time.

Regarding the aforementioned challenges and strong points of the model-free learning-based approach, we hypothesize that in a CI/CD process based on agile software development, performance engineers and testers can save time and resources by using SaFReL for performance (stress) testing of various

releases or variants. SaFReL provides an agile efficient performance test case generation technique (See Section 7.7 and Section 7.8 for efficiency evaluation) while eliminating the need for source code or system model analysis.

7.2.1 Reinforcement Learning

Reinforcement learning (RL) [39] is a fundamental category of machine learning algorithms generally intended to find the optimal behavior (way) in decision-making problems. RL is an interactive learning paradigm that is different from the common supervised and unsupervised machine learning algorithms and has been frequently applied to building many self-adaptive smart systems. It involves continuous interaction between the agent (learner) and the environment that is controlled. At each step of the interaction, the agent observes (senses) the *state* of the environment, takes a possible *action* and receives a reinforcement signal as a scalar *reward* from the environment that shows the effectiveness of the applied action to guide the agent towards accomplishing the intended objective. There is no supervisor in RL, and the agent just receives a reward signal. RL basically involves a sequential decision-making process. The RL agent goes through the environment, decides how to behave at each step, and based on optimizing the long-term received reward, learns the optimal way of decision making.

The agent actually decides between actions based on the history of its observations. However, considering the whole history of observations is not efficient, therefore, *state* should be formulated as a concise summary of the history including all the required information. Keeping in mind this issue, a related helpful concept to formulate the state as a summary function is the *Markov state*. The states of the environment are Markov by definition. Then, when the environment is fully observable to the agent, the states that the agent observes and uses for making decisions, are Markov too. The environment in our case is the SUT and the execution platform. The state is modeled in terms of response time and resource utilization improvement. The actions are some operations for modifying/adjusting the available capacity of resources and the objective of the agent is finding an intended performance breaking point. Figure 7.1 shows the interaction between the agent and the environment that is the composition of SUT and execution platform in our case.

There are three main elements in an RL agent: policy, value function, and

model. The Policy is the behavior function describing what actions the agent takes in a certain state. Value function indicates how good each state and/or action is, in terms of the amount of reward expected upon taking a particular action given a particular state. Finally, the model is the agent's view of the environment and describes what the environment does next, e.g., shows the state transitions of the environment.

Model-free RL algorithms are special types of RL that are not intended to build or learn a model of the environment. Instead, they learn the optimal behavior to achieve the intended objective through multiple experiences of interaction with the environment. Temporal Difference (TD) [39] is one of the main types of model-free RL, which is able to learn from the incomplete episodes of the interaction with the environment. Q-learning as a model-free TD learns the optimal policy through learning the optimal value function, i.e., Q-values. It uses an action selection strategy based on a combination of trying out the available actions, namely exploration, and relying on the previously achieved experience to select the highly-valued actions, namely exploitation. It is off-policy, which means that the agent learns the optimal policy regardless of how the agent explores the environment. After learning the optimal policy, in the transfer learning phase, the agent is able to replay the learned policy while keeping the learning running, which implies occasionally exploring the action space and trying out different actions.

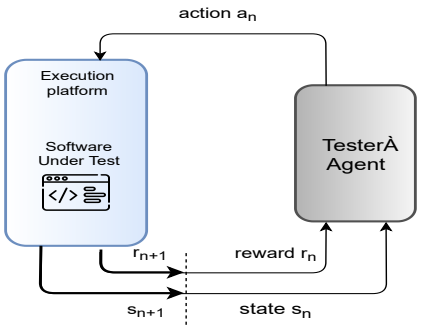


Figure 7.1: Interaction between agent and SUT in RL

7.3 Architecture

This section provides an overview of the architecture of the proposed smart performance testing framework, SaFReL (see Figure 7.2). The entire interaction of the smart framework with each SUT, as a learning episode, consists of a number of learning trials. The steps of learning in each trial and the components involved in each step are described as follows:

1. Fuzzy State Detection. The fuzzification, fuzzy inference, and rule base components in Figure 7.2 are involved in the state detection. The agent uses the values of four quality metrics, 1) response time, and utilization improvements of 2) CPU, 3) memory, and 4) disk, to identify the state of the environment. In other words, the *state* expresses the status of the environment relative to the testing target. In our case, these quality metrics are used to model (represent) the state space of the environment. An ordinary approach for state modeling in RL problems is dividing the state space into multiple mutually exclusive discrete sets. Each set represents a discrete state. At each time, the environment must be at one distinct state. The relevant challenges of such crisp categorization or defining discrete states, include knowing how much a value is suitable to be a threshold for categories of a metric, and how we can treat the boundary values between categories. Instead of crisp discrete states, using fuzzy logic and defining fuzzy states can help address these challenges. We use fuzzy classification as a soft labeling technique for presenting the values of the metrics used for modeling the state of the environment. Then, using a fuzzy inference engine and fuzzy rule base, the agent detects the fuzzy state of the environment. More details about the fuzzy state detection of the agent are presented in Section 7.4.

2. Action Selection and Strategy Adaptation. After detecting the fuzzy state of the SUT, the agent takes an action. The actions are operations modifying the factors affecting the performance, i.e., the available resource capacity, in the current prototype. The agent selects the action according to an *action selection strategy* that it follows. The action selection strategy determines to what extent the agent should explore and try out the available actions, and to what extent it should rely on the learned policy and select a high-value action that has been tried and assessed before. The role of this strategy is guiding the action

selection of the agent throughout the learning and is of importance for the efficiency of the learning. In order to obtain the desired efficiency, a proper trade-off between the exploration of the state action space and exploitation of the previously learned policy is critical.

In our proposed framework, the smart agent is augmented by a *strategy adaptation* characteristic, as a meta-learning feature responsible for dynamically adapting the degree of exploration and exploitation in various situations. This feature makes SaFReL able to detect where it should rely on the previously learned policy and where it should make a change in the strategy to update its policy and adapt to new situations. New situations mean acting on new SUTs that are different from the previously observed ones in terms of performance sensitivity to resources.

Software programs have different levels of sensitivity to resources. SUTs with different performance sensitivity to resources, e.g., CPU-intensive, memory-intensive, or disk-intensive SUTs, will react to changes in resource availability differently. Therefore, when the agent observes a SUT that is different from the previously observed ones in terms of performance sensitivity, the strategy adaptation tries to guide the agent towards doing more exploration than exploitation. A performance sensitivity indicator showing the sensitivity of SUT to the resources (i.e., being CPU-intensive, memory-intensive or disk-intensive) is an input to the strategy adaptation mechanism (see Figure 7.2).

The components corresponding to the action selection, the stored experience (learned policy), and the strategy adaptation are shown as yellow components in Figure 7.2. More details about the set of actions and the mechanism of strategy adaptation are described in Section 7.5.

3. Reward Computation. After taking the selected action, the agent receives a reward signal indicating the effectiveness of the applied action to approach the intended performance breaking point. The reward computation component (red block) in Figure 7.2 calculates the received reward (see Section 7.5) for the taken actions.

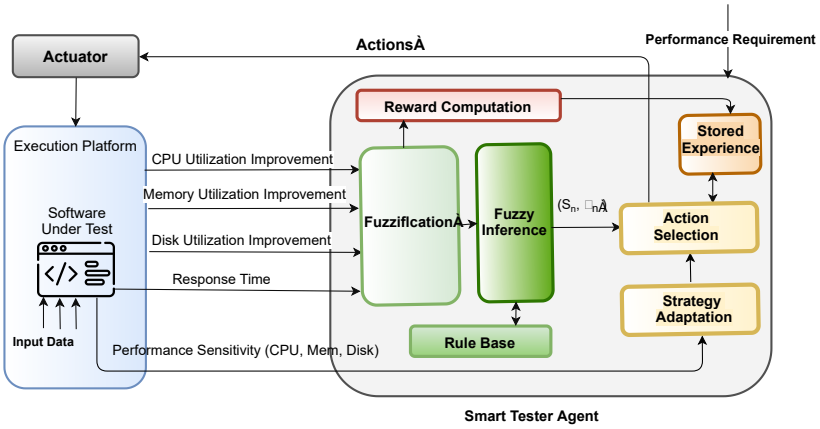


Figure 7.2: SaFReL architecture

7.4 Fuzzy State Detection

The state space of the environment in our learning problem is modeled by the quality measurements, CPU, memory, and disk resource utilization improvement and response time of the SUT, which is shown in Figure 7.3. The learning approach works based on detecting (discrete) states of the system. These states could be typically defined based on classifying the continuous values of the quality measurements that were mentioned above. On the other hand, defining such crisp boundaries on a number of continuous domains is an issue that might involve many uncertainties. In order to address this issue and preserve the desired precision of the model, fuzzy classification and reasoning is used to specify the states of the system. Therefore, the states of the environment are defined in terms of some fuzzy states and the environment can be in one or more fuzzy states at the same time with different degrees of certainty. The agent detects the state of the system using a fuzzy inference engine and a rule base [40, 41] (Figure 7.2). In summary, the step of state detection is done based on making fuzzy inference about the state of the system. The fuzzy state detection consists of three main parts: normalization of the input values (quality measurements), fuzzification of the measurements, and the fuzzy inference to identify the state of the environment. The details of

these parts together with the fuzzy rules, fuzzy operators, and the implication method that are used, are described in Section 7.4.1.

7.4.1 State Modeling and Fuzzy Inference

Normalization. As described in the previous section, a set of quality measurements, CPU, memory, and disk utilization improvements and response time of the SUT, represent the state of the environment. The values of these measurements are not bounded, then for simplifying the inference and also the exploration of the state space, we normalize the values of these parameters to the interval [0, 1] using the following functions:

$$RT_n = \frac{2}{\pi}\tan^{-1}(\frac{RT'_n}{RT^q}) \tag{7.1}$$

$$CUI_n = \frac{1}{CUI'_n} \qquad MUI_n = \frac{1}{MUI'_n} \qquad DUI_n = \frac{1}{DUI'_n} \tag{7.2}$$

where RT'_n, CUI'_n, MUI'_n, and DUI'_n are the measured values of the response time, CPU, memory and disk utilization improvements at time step n respectively and RT^q is the response time requirement. CUI'_n as the CPU utilization improvement is the ratio between the CPU utilization at time step n and its initial value (at the start of learning), that is, $CUI'_n = \frac{CU_n}{CU^i}$. Likewise, those are, $MUI'_n = \frac{MU_n}{MU^i}$ and $DUI'_n = \frac{DU_n}{DU^i}$. Using the normalization function in Eq. 7.1, when $RT'_n = RT^q$ the normalized value of the response time, RT_n is 0.5, and for $RT'_n > RT^q$ the normalized values will be toward 1 and for $RT'_n < RT^q$ the normalized values will be toward 0. A tuple as $(CUI_n, MUI_n, DUI_n, RT_n)$ consisting of the normalized values of quality measurements is the input to the fuzzy state detection.

Fuzzification. Input fuzzification involves defining fuzzy sets and corresponding membership functions over the values of the quality measurements. A membership function is characterized by a linguistic term. A fuzzy set L is defined as $L = \{(x, \mu_L(x))|\ 0 < x, \quad x \in \mathbb{R}\}$ where a membership function $\mu_L(x)$ defines membership degrees of the values as $\mu_L : x \rightarrow [0, 1]$. Figure 7.3 shows the membership functions defined over the value domains of quality measurements. As shown in Figure 7.3, trapezoidal membership functions are used for *High* and *Low* fuzzy sets and a triangular

counterpart for the *Normal* fuzzy set on the response time. In Figure 7.3, where RT^q is the requirement, a normal (medium) fuzzy set over the values of response time implies a small range around the requirement value as normal response time values. Moreover, in this case the ranges of membership functions were selected empirically and could be updated based on the requirements.

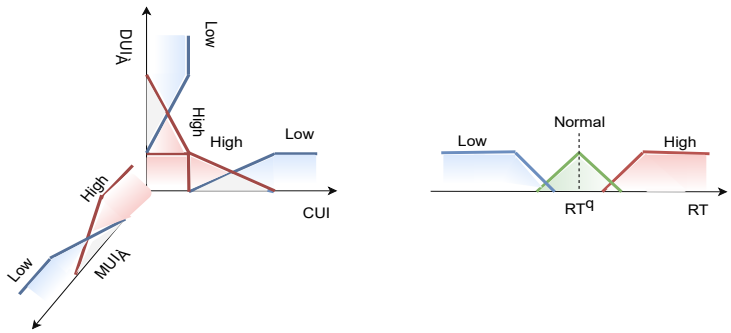


Figure 7.3: Fuzzy representation of quality measurements

Fuzzy Inference. After input fuzzification, inferring the possible states that the environment assumes is directed by the fuzzy rules that have formed based on the domain knowledge.

Fuzzy Rules. A fuzzy rule, as shown in Eq. 7.3, consists of two parts: antecedent and consequent. The former is a combination of linguistic terms of the input normalized quality measurements and the consequent is a fuzzy set with a membership function showing to what extent the environment is in the associated state.

$$\begin{aligned}&\text{Rule 1: If CUI is High AND MUI is High AND DUI is Low AND}\\&\qquad\text{RT is Normal, then State is HHLN.}\end{aligned} \tag{7.3}$$

Rule 1 is a sample of the fuzzy rules in the rule base. The rest of the rules are defined similarly based on the fuzzy sets defined over the values of the quality measurements and the combinations of them. Based on the number of fuzzy sets, namely two fuzzy sets, *High* and *Low*, over the value range of each resource utilization improvement and three sets, *High*, *Normal*, and *Low*, over

the value range of the response time, we define 24 rules in our rule base to define the fuzzy states of the environment.

Fuzzy Operators. When the antecedents of the rules are made of multiple linguistic terms, which are associated to fuzzy sets, e.g., "High, High, Low and Normal", then fuzzy operators are applied to the antecedent to obtain one number showing the support or activation degree of the rule. Two well-known methods for the fuzzy AND operator are $minimum(min)$ and $product(prod)$. In our case, we use method min for the fuzzy AND operation. It shows that given a set of input parameters A, the degree of support for rule Ri is given as $\tau_{Ri} = \min_j \mu_L(a_j)$ where a_j is an input parameter in A and L is its associated fuzzy set in the rule Ri.

Implication Method. After obtaining the membership degree for the antecedent, the membership function of the consequent is reshaped using an implication method. There are also two well-known methods for implication process, $minimum(min)$ and $product(prod)$, which truncate and scale the membership function of the output fuzzy set respectively. The membership degree of the antecedent is given as input to the implication method. We use method min as the implication method in our case.

Finally, the most effective rule, the one with the maximum support degree, is selected to determine the final fuzzy state of the environment (S_n, μ_n). In summary, the fuzzy state with the highest likelihood is considered as the state of the system. Figure 7.4 shows a representation of the fuzzy states. Each of them represents one state based on the fuzzy values (linguistic terms) assigned to quality measurements (CPU, memory and disk utilization improvement, and response time). Regarding the presentation of fuzzy states, L, H, and N stand for Low, High, and Normal terms respectively.

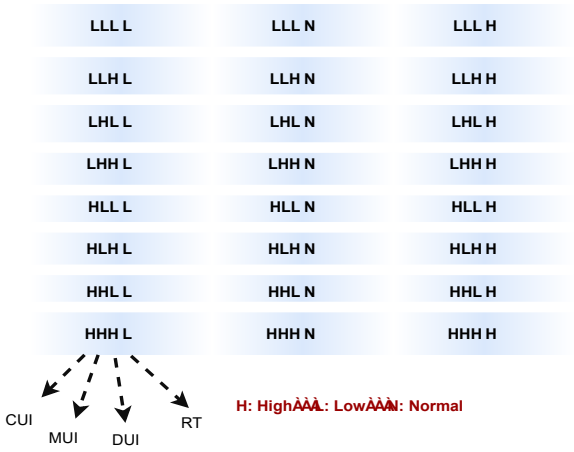


Figure 7.4: Fuzzy states of the environment

7.5 Adaptive Action Selection and Reward Computation

Actions. In SaFReL, the actions are the operations changing the platform-based factors affecting the performance, i.e., the available resources such as computation (CPU), memory and disk capacity. In the current prototype, the set of actions contains operations reducing the available resource capacity with finely tuned steps, which are as follows:

$$\begin{aligned} AC_n = &\{\text{no action}\} \cup \{(CPU_n - y) \mid y \in CDF\} \cup \{(Mem_n - k) \mid k \in MDF_n\} \\ &\cup \{(Disk_n - k) \mid k \in MDF_n\} \end{aligned} \tag{7.4}$$

$$CDF = \{\frac{1}{4}, \frac{2}{4}, \frac{3}{4}, 1\} \tag{7.5}$$

$$MDF_n = \{(x \times \frac{Mem(Disk)_n}{4}) \mid x \in \{\frac{1}{4}, \frac{2}{4}, \frac{3}{4}, 1\}\} \tag{7.6}$$

where AC_n, CPU_n, Mem_n and $Disk_n$ represent the set of actions, the current available computation (CPU), memory and disk capacity at time step n

respectively. The list of actions is as shown in Table 7.1.

Strategy Adaptation. The agent can use different strategies for selecting the actions. ε-greedy with different ε-values and Softmax are well-known methods for action selection in RL algorithms. They are intended to provide a right trade-off between exploration of the state action space and exploitation of the learned policy. In SaFReL, we use ε-greedy as the action selection strategy and the proposed strategy adaptation feature acts as a simple meta-learning algorithm intended to make changes to the ε value dynamically to make the action selection strategy well-adapted to new situations (new SUTs). Upon observing a SUT instance with a performance sensitivity different from the already observed ones, it adjusts the value of the parameter ε to direct the agent toward more exploration (setting ε to higher values). On the other hand, upon interaction with SUT instances that are similar to the previous ones, the parameter ε is adjusted to increase exploitation (setting ε to lower values). SaFReL detects the similarity between SUT instances by calculating *cosine similarity* between the performance sensitivity vectors of SUT instances, as shown in Eq. 7.7.

$$\begin{aligned} \text{similarity}(k, k-1) &= \frac{SV^k \; SV^{k-1}}{\|SV^k\| \|SV^{k-1}\|} \\ &= \frac{\sum_{i=1}^{3} SV_i^k SV_i^{k-1}}{\sqrt{\sum_{i=1}^{3} \left(SV_i^k\right)^2} \sqrt{\sum_{i=1}^{3} \left(SV_i^{k-1}\right)^2}} \end{aligned} \tag{7.7}$$

where SV^k represents the sensitivity vector of the k^{th} SUT instance and SV_i^k represents the i^{th} element of vector SV^k. The sensitivity vector contains the values of the sensitivity indicators of the SUT instance, Sen^C, Sen^M and

Table 7.1: Actions in SaFReL

Actions	
Operation	***Decrease***
Reducing memory / disk capacity	by a factor in MDF_n
Reducing computation (CPU) capacity	by a factor in CDF
No action	-

Sen^D. The performance sensitivity indicators assume values in the range $[0, 1]$ and represent the sensitivity degree of the SUT to CPU, memory and disk respectively. Their values could be set empirically or even intuitively, and SaFReL uses the approximate estimated similarity to tune the ε value adaptively (See Section 7.7.2).

Reward Signal. The agent receives a reward signal indicating the effectiveness of the applied action in each learning step to guide the agent toward reaching the intended performance breaking point. We derive a utility function as a weighted linear combination of two functions indicating the response time deviation and resource usage, which is as follows:

$$R_n = \beta U_n^r + (1 - \beta) U_n^E \tag{7.8}$$

where U_n^r represents the deviation of response time from the response time requirement, U_n^E indicates the resource usage, and $\beta, 0 \leq \beta \leq 1$ is a parameter intended to prioritize different aspects of stress conditions, i.e., response time deviation or limited resource availability. U_n^r is defined as follows:

$$U_n^r = \begin{cases} 0, & RT'_n \leq RT^q \\ \frac{(RT'_n - RT^q)}{(RT^b - RT^q)}, & RT'_n > RT^q \end{cases} \tag{7.9}$$

where RT'_n is the measured response time, RT^q is the response time requirement and RT^b is the threshold defining the performance breaking point. U_n^E represents the resource utilization in the reward signal, and is a weighted combination of the resource utilization values. It is defined using the following equation:

$$U_n^E = Sen^C CUI'_n + Sen^M MUI'_n + Sen^D DUI'_n \tag{7.10}$$

where CUI'_n, MUI'_n, and DUI'_n represent CPU, memory and disk utilization improvements respectively, and Sen^C, Sen^M and Sen^D are the performance sensitivity indicators of the SUT, and assume values in the range $[0, 1]$.

7.6 Performance Testing using Self-Adaptive Fuzzy Reinforcement Learning

In this section, we describe details of the procedure of SaFReL to generate the performance test cases resulting in reaching the performance breaking points for various types of SUTs. The tester agent learns how to generate the target test cases for different types of software without access to source code or system models. The procedure of SaFReL, which includes initial and transfer learning phases, is as follows:

The agent measures the quality parameters and identifies the state-membership degree pair (S_n, μ_n) through the fuzzy state detection, where S_n is the fuzzy state of the environment and μ_n indicates the membership degree, which means to what extent the environment has assumed that state. Then, according to the action selection strategy, the agent selects one action, $a_n \in A_n$ based on the previously learned policy or through exploring the state action space. The agent takes the selected action and executes the SUT. In the next step the agent detects the new state of the SUT, (S_{n+1}, μ_{n+1}) and receives a reward signal, $r_{n+1} \in \mathbb{R}$, indicating effectiveness of the applied action. After detecting the new state and receiving the reward, it updates the stored experience (learned policy). The whole procedure is repeated until meeting the stopping criterion that is reaching the performance breaking point, (RT^b). The experience of the agent is defined in terms of the policy that the agent learns. A policy is a mapping between each state and action and specifies the probability of taking action a in a given state s. The purpose of the agent in the learning is to find a policy that maximizes the expected long-term reward achieved over the further learning trials, which is formulated as follows: [39]:

$$R_n = r_{n+1} + \gamma r_{n+2} + ... + \gamma^k r_{n+k+1} = \sum_{k=0}^{\infty} \gamma^k r_{n+k+1} \tag{7.11}$$

where γ is a discount factor specifying to what extent the agent prioritize future rewards compared to the immediate one. We use Q-learning as a model-free RL algorithm in our framework. In Q-Learning, a utility value $Q^\pi(s, a)$ is assigned to each pair of state and action, which is defined as follows: [39]:

$$Q^\pi(s, a) = E^\pi[R_n | s_n = s, a_n = a] \tag{7.12}$$

The q-values, $Q^{\pi}(s,a)$, form the experience base of the agent, on which the agent relies for the action selection. The q-values are updated incrementally during the learning. According to using fuzzy state modeling, we include the membership degree of the detected state of the environment, μ_n^s, in the typical updating equation of q-values to take into account the impact of the uncertainty associated with the fuzzy state, which is as follows:

$$Q(s_n,a_n) = \mu_n^s[(1-\alpha)Q(s_n,a_n) + \alpha(r_{n+1} + \gamma \max_{a'} Q(s_{n+1},a'))] \quad (7.13)$$

where α, $0 \leq \alpha \leq 1$ is the learning rate, which adjusts to what extent the new utility values affect (overwrite) the previous q-values. Finally, the agent finds the optimal policy to reach the target, which suggests the action maximizing the utility value for a given state s:

$$a(s) = \operatorname*{argmax}_{a'} Q(s,a') \quad (7.14)$$

The agent selects the action based on Eq. 7.14 when it is supposed to exploit the learned policy. SaFReL implements two learning phases: initial and transfer learning.

Initial learning. Initial learning occurs during the interaction with the first SUT instance. The initial convergence of the policy takes place upon the initial learning. The agent stores the learned policy (in terms of a table containing q-values, Q-table). It repeats the learning episode multiple times on the first SUT instance to achieve the initial convergence of the policy.

Transfer learning. SaFReL goes through the transfer learning phase, after the initial convergence. During this phase, the agent uses the learned policy upon observing SUT instances with similar performance sensitivity to the previously observed ones, while keeping the learning running, i.e., updating the policy upon detecting new SUT instances with different performance sensitivity. Strategy adaptation is used in the transfer learning phase and makes the agent adapt to various SUT instances. Algorithms 8 and 9 present the procedure of SaFReL in both initial learning and transfer learning phases.

Algorithm 8 SaFReL: Self-adaptive Fuzzy Reinforcement Learning-based Performance Testing

Required: S, A, α, γ;
Initialize q-values, $Q(s, a) = 0\ \forall s \in \mathbb{S},\ \forall a \in \mathbb{A}$ and $\epsilon = \upsilon\ , 0 < \upsilon < 1$;
Observe the first SUT instance;
repeat
    Fuzzy Q-Learning (with initial action selection strategy, e.g. ϵ-greedy, initialized ϵ);
until *initial convergence*;
Store the learnt policy;
Start the transfer learning phase;
while *true* **do**
    Observe a new SUT instance;
    Measure the similarity;
    Apply strategy adaptation to adjust the degree of exploration and exploitation (e.g. tuning parameter ϵ in ϵ-greedy);
    Fuzzy Q-Learning with adapted strategy (e.g. new value of ϵ);
end

7.7 Evaluation

In this section, we present the experimental evaluation of the proposed self-adaptive fuzzy RL-based performance testing framework, SaFReL. We assess the performance of SaFReL, in terms of efficiency in generating the performance test cases and adaptivity to various types of SUT programs, i.e., how well it can adapt its functionality to new cases while preserving its efficiency. Therefore, we examine the efficiency of SaFReL (in the transfer learning phase) compared to a typical testing process for this target, which involves generating the performance test cases through changing the availability of the resources based on the defined actions in an exploratory (random) way, which is called *typical stress testing* hereafter. We also evaluate the sensitivity of SaFReL to the learning parameters. The goal of the experimental evaluation is to answer the following research questions:

Algorithm 9 Fuzzy Q-Learning

repeat

1. Detect the fuzzy state-degree pair (S_n, μ_n) of the SUT;
2. Select an action using the action selection strategy (e.g. ϵ-greedy: select $a_n = \text{argmax}_{a \in \mathbb{A}} Q(s_n, a)$ with probability (1-ϵ) or a random a_k, $a_k \in \mathbb{A}$ with probability ϵ);
3. Take the selected action, execute the SUT;
4. Detect the new fuzzy state-degree (S_{n+1}, μ_{n+1}) of the environment;
5. Receive the reward signal, R_{n+1};
6. Update the q-value of the pair of previous state and applied action $Q(s_n, a_n) = \mu_n^s[(1-\alpha)Q(s_n, a_n) + \alpha(r_{n+1} + \gamma \max_{a'} Q(s_{n+1}, a'))]$

until *meeting the stopping criteria (reaching performance breaking point)*;

- RQ1. How efficiently can SaFReL generate the test cases leading to the performance breaking points for different software programs compared to a typical testing procedure?
- RQ2. How adaptively can SaFReL act on various software programs with different performance sensitivity?
- RQ3. How is the efficiency of SaFReL affected by changing the learning parameters?

The following sub-sections describe the proposed setup for conducting the experiments, the evaluation metrics, and the analysis scenarios designed for answering the above research questions.

7.7.1 Experiments Setup

In this study, we implement the proposed smart testing framework (agent) along with a performance simulation module simulating the performance behavior of SUT programs under different execution conditions. The simulation module receives the resource sensitivity values and based on the amounts of resources demanded initially and the amounts of them granted after taking each action, estimates the program throughput using the following

equation proposed by [42]:

$$Thr_j = \frac{\frac{CPU_j^g}{CPU_j^i} Sen_j^C + \frac{Mem_j^g}{Mem_j^i} Sen_j^M + \frac{Disk_j^g}{Disk_j^i} Sen_j^D}{Sen_j^C + Sen_j^M + Sen_j^D} \times Thr_j^N \qquad (7.15)$$

where CPU_j^i, Mem_j^i and $Disk_j^i$ the amounts of CPU, memory and disk resources demanded by program j at the initial state and CPU_j^g, Mem_j^g and $Disk_j^g$ are the amounts of resources granted to program j after taking an action, which modifies the resource availability. Sen_j^C, Sen_j^M and Sen_j^D represent the CPU, memory and disk sensitivity values of program j, and Thr_j^N represents the nominal throughput of program j in an isolated, contention free environment. The response time of the program is calculated as $RT_j = \frac{1}{Thr_j}$ in the simulation module. Figure 7.5 presents the implementation structure including SaFReL along with the implemented performance simulation module. In our implementation, the performance simulation module simulates the performance behavior of the SUT program and the testing agent interacts with the simulation module to capture the quality measures used for state detection.

Table 7.2 shows the list of programs and the corresponding resource sensitivity values used in the experimentation, the table data obtained from [42]. The collection listed in Table 7.2 includes various CPU-intensive, memory-intensive and disk-intensive types of programs and also the programs with combined types of resource sensitivity. The SUTs are instances of the programs listed in Table 7.2 and are characterized with various initial amounts of resources and also different values of response time requirements. Two analysis scenarios are designed to answer the evaluation research questions. The first one focuses on efficiency and adaptivity evaluation of the framework on various SUTs. In the second analysis scenario, the sensitivity of the approach to changes of the learning parameters are studied. The efficiency and adaptivity are measured (evaluated) according to following specification:

- *Efficiency* is measured in terms of number of learning trials required by the tester agent to achieve the testing target, which is reaching the intended performance breaking point. Number of learning trials is an indicator of the required computation time to generate the proper test case leading to the performance breaking point.

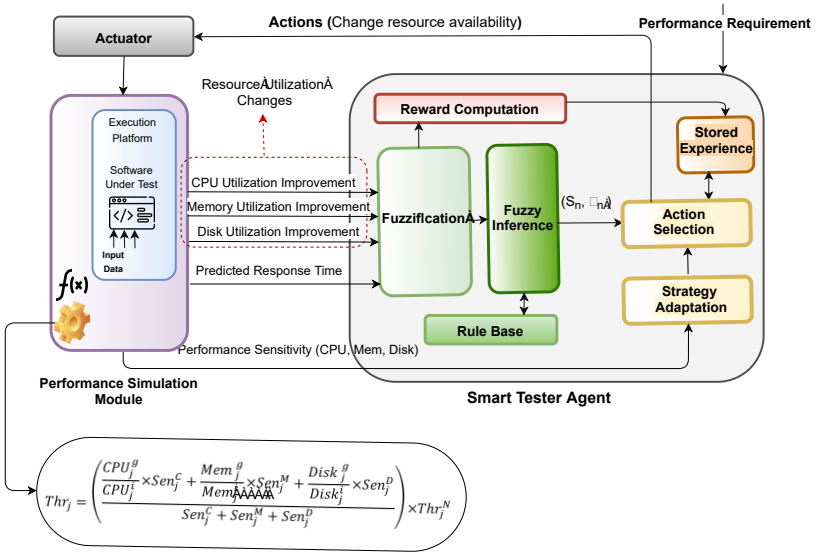


Figure 7.5: Implementation structure

- *Adaptivity* is evaluated in terms of number of additional learning trials (computation time) required to re-adapt the learned policy to new observations for achieving the target.

7.7.2 Experiments and Results

Efficiency and Adaptivity Analysis

To answer RQ1 and RQ2, the performance of SaFReL is evaluated based on its efficiency in generating the performance test cases leading to the performance breaking points of different SUTs and its adaptation capability to new SUTs with performance sensitivity different from previously observed ones. We select two sets of SUT instances: i) one including SUTs similar in the aspect of performance sensitivity to resources, i.e., similar with regard to the primarily demanded resource (homogenous SUTs); and ii) the other set contains SUT instances different in performance sensitivity (heterogeneous SUTs). The SUT instances assume different initial amounts of CPU, memory

Table 7.2: Programs and the corresponding sensitivity values used for experimental evaluation [42]

Programs	**Resource Sensitivity Values (Sen^C, Sen^M and Sen^D)**
Build-apache	(0.96, 0.04, 0.00)
n-queens	(0.97, 0.00, 0.00)
John-the-ripper	(0.96, 0.00, 0.00)
Apache	(0.97, 0.03, 0.00)
Dcraw	(0.48, 0.04, 0.00)
X264	(0.41, 0.02, 0.00)
Unpack-linux	(0.18, 0.09, 0.35)
Build-php	(0.97, 0.07, 0.00)
Blogbench	(0.11, 0.81, 0.18)
Bork	(0.00, 0.53, 0.20)
Compress-gzip	(0.00, 0.00, 0.47)
Aio-stress	(0.00, 0.30, 0.80)

and disk resources, and response time requirements. The amounts of resources, CPU, memory and disk capacity, were initialized with different values in the range [1, 10] cores, [1, 50] GB, [100, 1000] GB respectively. The response time requirements range from 500 to 3000 ms. The intended performance breaking point for the SUT instances is defined as the point in which the response time exceeds 1.5 times the response time requirement.

In the efficiency analysis, we set the learning parameters, learning rate and discount factor, to 0.1 and 0.5, respectively. We study the impacts of different variants of ε-greedy algorithm as the action selection strategy on the efficiency and adaptivity of the approach during the analysis. We investigate three variants of ε-greedy with $\varepsilon = 0.2$, $\varepsilon = 0.5$, and decaying ε, and also the proposed adaptive ε selection method.

Learning setup. First, we need to set up the initial learning. For choosing a proper configuration for the action selection strategy in the initial learning, we evaluate the performance of different variants of ε-greedy algorithm, in terms of the number of required learning trials for initial convergence (Figure 7.6). For the initial convergence, we run the initial learning on the first SUT

100 times, namely 100 learning episodes. Table 7.3 presents a quick summarized view of the average learning trials during the last 10 episodes that are considered as the achieved values upon the convergence of the initial learning. As shown in Figure 7.6 and Table 7.3, using ε-greedy with $\varepsilon = 0.2$ results in the fastest initial convergence, which has also led to the lowest number of trials compared to the other variants of ε-greedy. The number of learning trials after about 10 episodes starts converging and during the last 10 episodes it converges to approximately 7 trials.

Figure 7.6: Initial convergence of SaFReL in 100 learning episodes during the initial learning

Once the initial convergence occurs, SaFReL is ready to act on various SUTs and is expected to be able to reuse the learned policy to meet the intended performance breaking points on further SUT instances, while still keeping the learning running. The optimal policy learned in the initial

Table 7.3: Initial convergence of SaFReL in the initial learning regarding using different variants of action selection strategy

	SaFReL - Initial Learning			
Action Selection Strategy: ϵ-greedy	$\epsilon = 0.85$	$\epsilon = 0.5$	$\epsilon = 0.2$	decaying ϵ
Number of learning trials (after convergence)	22	21	7	9

learning is not influenced by the used action selection strategy, since Q-learning is an off-policy learning algorithm [39]. It implies that the learner finds the optimal policy independently of how the actions have been selected (action selection strategy). For the sake of efficiency, we choose the one that resulted in the fastest convergence.

In the following sections, first, we investigate the efficiency of SaFReL compared to a typical stress testing procedure, when acting on homogeneous and heterogeneous sets of SUTs, then its capability to adapt to new SUTs with different performance sensitivity.

I. Homogeneous set of SUTs. We select CPU-intensive programs and make a homogeneous set of SUT instances during our analysis in this step. We simulate the performance behavior of 50 instances of the CPU-intensive programs, Build-apache, n-queens, John-the-ripper, Apache, Dcraw, Build-php, X264, and vary both the initial amounts of resources granted and the response time requirements. Figure 7.7 shows the efficiency of SaFReL on a homogeneous set of CPU-intensive SUTs compared to a typical stress testing procedure regarding using ε-greedy with different values of ε. Table 7.4 presents the average number of trials/steps for generating the target performance test case in the proposed approach and the typical testing procedure. As shown in Figure 7.7, it keeps the number of required trials for $\approx 94\%$ of the SUTs below the average number of required steps in the typical stress testing. Table 7.5 shows the resulting improvement in the average number of required trials/steps for meeting the target, which implies reduction in the required computation time, compared to the typical stress testing process.

In the transfer learning, the agent reuses the learned policy based on the allowed degree of policy reusing according to its action selection strategy in the transfer learning. As shown in Table 7.4, it implies that in the transfer learning the agent does fewer trials (based on the degree of allowed policy reusing) to meet the target on new cases, which leads to a higher efficiency. According to Table 7.5, on a homogeneous set of SUTs, more policy reusing leads to higher efficiency (more computation time improvement).

Figure 7.7: Efficiency of SaFReL on a homogeneous set of SUTs in the transfer learning

II. Heterogeneous set of SUTs. In this part of the analysis, to complete the answer to RQ1 and and also answer RQ2, we examine the efficiency and adaptivity of SaFReL during the transfer learning on a heterogeneous set of SUTs including various CPU-intensive, memory-intensive and disk-intensive ones. We simulate the performance behavior of 50 SUT instances from the list

Table 7.4: Average number of trials/steps for generating the target performance test case on the homogeneous set of SUTs

	SaFReL with ϵ-greedy			
Approach	$\epsilon = 0.5$	decaying ϵ	$\epsilon = 0.2$	Typical stress testing
Average number of trials/steps	10	10	7	12

Table 7.5: Computation time improvement on the homogeneous set of SUTs

	SaFReL		
Action Selection Strategy: ϵ-greedy	$\epsilon = 0.5$	decaying ϵ	$\epsilon = 0.2$
Improvement in the number of trials	16%	16%	42%

of the programs in Table 7.2. We evaluate the efficiency of SaFReL on the heterogeneous set of SUTs compared to the typical stress testing procedure regarding using ε-greedy with $\varepsilon = 0.2, 0.5$, and decaying ε (Figure 7.8). As shown in Figure 7.8 the transfer learning algorithm with a typical configuration of the action selection strategy, such as $\varepsilon = 0.2, 0.5$ and decaying ε, which imposes a certain degree of policy reusing based on the value of ε does not work well. It does not outperform the typical stress testing, but also slightly degrades in some cases of ε. When the smart agent acts on a heterogeneous set of SUTs, blind replaying of the learned policy (i.e., just based on the value of ε) is not effective, and the tester agent needs to know where it should do policy reusing and where it requires more exploration to update the policy.

As described in Section 7.5, to solve this issue and improve the performance of SaFReL when it acts on a heterogeneous set of SUTs, it is augmented with a simple meta-learning feature enabling it to detect the heterogeneity of the SUT instances and adjust the value of parameter ε, adaptively. In general, it implies that when the smart tester agent observes a SUT instance different from the previously observed ones wrt the performance sensitivity, it changes the action selection strategy to doing more

Figure 7.8: Efficiency of SaFReL on a heterogeneous set of SUTs regarding the use of typical configurations of ϵ-greedy

exploration and upon detecting a SUT instance with the same performance sensitivity as the previous ones, it makes the action selection strategy strive for more exploitation. As illustrated in Section 7.5, the strategy adaptation module, which fulfills this function, measures the similarity between SUTs at two levels of observations, then based on the measured values, adjusts the value of parameter ε. The threshold values of similarity measures and the adjustments for parameter ε in the experimental analysis are described in Algorithm 10.

Figure 7.9 shows the efficiency of SaFReL regarding the use of similarity detection and the adaptive ε-greedy action selection strategy on a heterogeneous set of SUTs. Regarding the use of adaptive ε selection, SaFReL makes a considerable improvement and is able to keep the number of

Algorithm 10 Adaptive ϵ selection

if $similarity_{k,k-1} \geq 0.8$ **then**
  if $similarity_{k,k-2} \geq 0.8$ **then**
    $\epsilon \leftarrow 0.2$
  else
    $\epsilon \leftarrow 0.5$
  end if
else if $similarity_{k,k-1} < 0.8$ **then**
  $\epsilon \leftarrow 0.5$
end if

required trials for reaching the target on approximately $\approx 82\%$ of SUTs below the corresponding average value in the typical stress testing. Meanwhile, the average number of learning trials is totally lower than the typical stress testing procedure. Table 7.6 presents the average number of trials/steps for generating the target performance test case in SaFReL and the typical stress testing when they act on a heterogeneous set of SUTs. Table 7.7 shows the corresponding resulting improvement in the computation time respectively.

Table 7.6: Average number of trials/steps for generating the target performance test case on the heterogeneous set of SUTs

	SaFReL with ϵ-greedy				
Approach	$\epsilon = 0.5$	decaying ϵ	$\epsilon = 0.2$	adaptive ϵ	Typical stress testing
Average number of trials/steps	18	17	18	11	16

To answer RQ2, we investigate the adaptivity of SaFReL on the heterogeneous set of SUTs regarding the use of different variants of action selection strategy including adaptive ε selection (Figure 7.10). As shown in Figure 7.10, the number of required learning trials versus detected similarity is used to depict how adaptive SaFReL can act on a heterogeneous set of

Table 7.7: Computation time improvement on the heterogeneous set of SUTs

	SaFReL			
Action Selection Strategy: ϵ-greedy	$\epsilon = 0.5$	decaying ϵ	$\epsilon = 0.2$	adaptive ϵ
Improvement in the number of trials	No	No	No	31%

SUTs regarding the use of different configurations of ε. It shows that SaFReL with adaptive ε is able to adapt to changing situations, e.g., a mixed heterogeneous set of SUTs. In other words, on around $\approx 75\%$ of SUTs that are completely different from the previous ones (i.e., with $similarity_{k,k-1} < 0.8$) it still keeps the number of required trials to meet the target below the average value of the typical stress testing. It implies that it can act adaptively, which means it reuses the policy wherever it is useful and does more exploration wherever required.

Figure 7.9: Efficiency of SaFReL on a heterogeneous set of SUTs regarding the use of adaptive ϵ-greedy action selection strategy

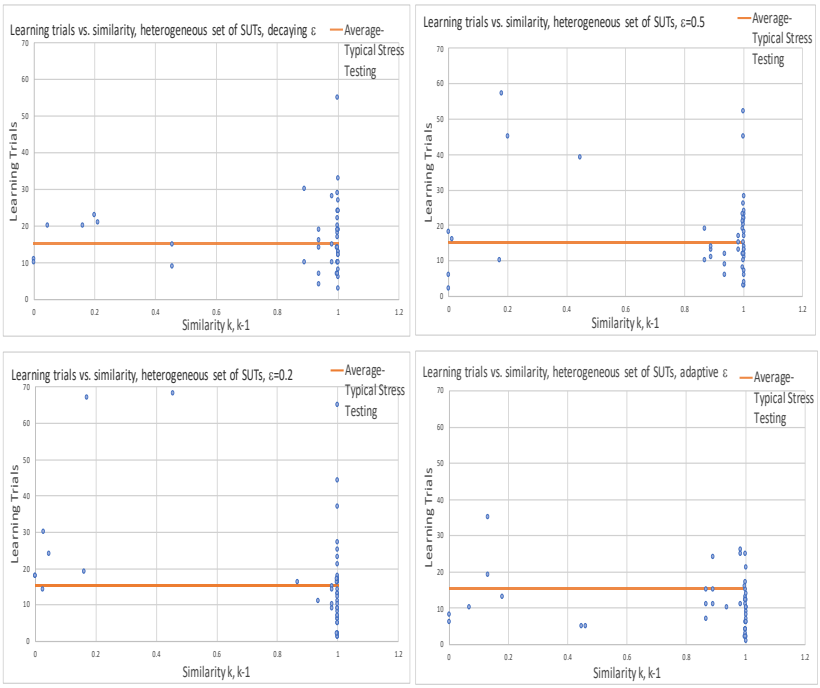
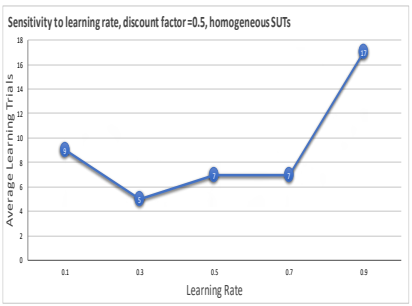


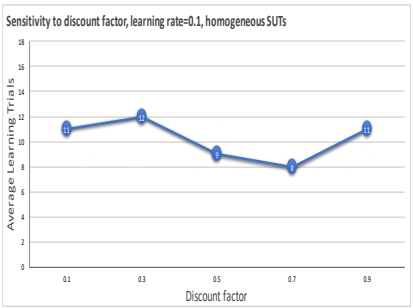
Figure 7.10: Adaptivity of SaFReL on a heterogeneous set of SUTs regarding the use of different variants of action selection strategy

Sensitivity Analysis

To answer RQ3, we study the impacts of the learning parameters including learning rate (α) and discount factor (γ), on the efficiency of SaFReL on both homogeneous and heterogeneous sets of SUTs. For conducting sensitivity analysis, we implement two sets of experiments that involve changing one learning parameter while keeping the other one constant. For the experiments running on a homogeneous set of SUTs, we use ε-greedy with $\varepsilon = 0.2$ as the well-suited variant of action selection strategy with respect to the results of efficiency analysis (See Figure 7.7) and on the heterogeneous set of SUTs, we use adaptive ε selection (See Figure 7.9). During the sensitivity analysis experiments, to study the impact of the learning rate changes, we set the discount factor to 0.5. While examining the impact of the discount factor

changes, we keep the learning rate fixed to 0.1. Figure 7.11 shows the sensitivity of SaFReL to changing learning rate and discount factor parameters when it acts on a homogeneous set of SUTs (CPU-intensive). Figure 7.12 depicts the results of the sensitivity analysis of SaFReL on a heterogeneous set of SUTs.

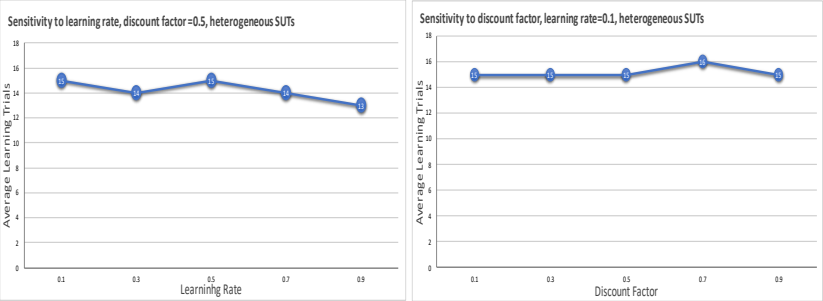




Average Efficiency of SaFReL with $\epsilon = 0.2$, Discount factor $\gamma = 0.5$					
	$\alpha = 0.1$	$\alpha = 0.3$	$\alpha = 0.5$	$\alpha = 0.7$	$\alpha = 0.9$
Average number of learning trials	9	5	7	7	17

Average Efficiency of SaFReL with $\epsilon = 0.2$, Learning rate $\alpha = 0.1$					
	$\gamma = 0.1$	$\gamma = 0.3$	$\gamma = 0.5$	$\gamma = 0.7$	$\gamma = 0.9$
Average number of learning trials	11	12	9	8	11

Figure 7.11: Sensitivity of SaFReL to learning rate and discount factor on the homogeneous set of SUTs



Average Efficiency of SaFReL with adaptive ϵ, Discount factor $\gamma = 0.5$					
	$\alpha = 0.1$	$\alpha = 0.3$	$\alpha = 0.5$	$\alpha = 0.7$	$\alpha = 0.9$
Average number of learning trials	15	14	15	14	13

Average Efficiency of SaFReL with adaptive ϵ, Learning rate $\alpha = 0.1$					
	$\gamma = 0.1$	$\gamma = 0.3$	$\gamma = 0.5$	$\gamma = 0.7$	$\gamma = 0.9$
Average number of learning trials	15	15	15	16	15

Figure 7.12: Sensitivity of SaFReL to learning rate and discount factor on the heterogeneous set of SUTs

7.8 Discussion

7.8.1 Efficiency, Adaptivity and Sensitivity Analysis

RQ1: Using multiple experiments, we studied the efficiency of SaFReL compared to a typical stress testing procedure, on both a set of homogeneous and heterogeneous SUTs regarding the use of different action selection strategies. The results of the experiments running on a set of 50 CPU-intensive SUT instances as a homogeneous set of SUTs, Figure 7.7 and Tables 7.4 and 7.5, show that using ε-greedy, $\varepsilon = 0.2$ as action selection strategy in the transfer learning leads to desired efficiency and an

improvement in the computation time (around 42%) compared to the typical stress testing. It causes SaFReL to rely more on reusing the learned policy and results in computation time saving. The existing similarity between the performance sensitivity of SUTs in a homogeneous set of SUTs makes the strategy of policy reusing successful in this type of testing situations.

Furthermore, we studied the efficiency of SaFReL on a heterogeneous set of 50 SUTs containing different CPU-intensive, memory-intensive and disk-intensive ones. The results of the analysis illustrate that choosing an action selection strategy without considering the heterogeneity among the SUTs (e.g., using the typical variants of ε-greedy) does not lead to desirable efficiency compared to the typical stress testing (See Figure 7.8, Table 7.6 and 7.7). Then, we augmented our fuzzy RL-based approach with an adaptive action selection strategy that is a heterogeneity-aware strategy for adjusting the value of ε. It measures the similarity between the performance sensitivity of the SUTs and adjusts the ε parameter. As shown in Figure 7.9, using the adaptive ε-greedy, addressed the issue and led to an efficient generation of the target performance test case and a computation time improvement (around 31%). It makes the agent able to reuse the learned policy according to the conditions, which means it uses the learned policy wherever it is useful and does more exploration wherever it is required.

RQ2: At the last part of the efficiency and adaptivity analysis, we extended our analysis by measuring the adaptivity of SaFReL when it performs on a heterogeneous set of SUTs. As shown in Figure 7.10, with the use of the adaptive ε-greedy, SaFReL is able to adapt to changing testing situations while preserving the efficiency.

RQ3: The results of the sensitivity analysis experiments on the homogeneous set of SUTs show that adjusting the learning rate to lower values such as 0.1 leads to better efficiency. Furthermore, regarding the sensitivity analysis of SaFReL to the discount factor on a homogeneous set of SUTs, the experimental results depict that lower values of the discount factor are suitable choices for the desired operation that we expect. However, the results of the sensitivity analysis on the heterogeneous set of SUTs do not show a considerable effects on the average efficiency of SaFReL when it acts on a heterogeneous set of SUTs regarding the use of adaptive ε-greedy.

7.8.2 Lessons Learned

The experimental evaluation of SaFReL shows how machine learning can guide performance testing towards being automated and taking one step further towards being autonomous. Common approaches for generating performance test cases mostly rely on source code or system models, but such development artifacts might not always be available. Moreover, drawing a precise model of a complex system predicting the state of the system upon given performance-related conditions requires a solid endeavor. This makes room for machine learning, particularly model-free learning techniques. Model-free RL is a machine learning technique enabling the learner to explore the environment (the behavior of the SUT on the execution platform in this case) and learn the optimal policy to accomplish the objective (finding the intended performance breaking point in this case) without having a model of the system. The learner stores the learned policy and is able to replay the learned policy in further suitable situations. This important characteristic of RL leads to a reduction in the effort of the learner to accomplish the objective in further cases and consequently leads to improved efficiency. Therefore, the main features that lead SaFRel to outperform an exploratory (search-based) technique are the capability of storing knowledge during the exploration and reusing the knowledge in suitable situations, and the possibility of selective and adaptive control on exploration and exploitation.

In general, automation, reduction of computation time and cost, and less dependency on source code and models are profound strengths of the proposed RL-assisted performance testing. Regarding applicability, according to the aforementioned strengths and the results of the experimental evaluation, the proposed approach could be beneficial to performance testing of software variants in software product lines, evolving software in continuous Integration/Delivery process and performance regression testing.

Changes in Future Trends. With the emergence of serverless architecture, which incorporates third-party backend services (BaaS) and/or runs the server-side logic in state-less containers that are fully-managed by providers (FaaS), a slight shift in the objectives of performance evaluation, particularly performance testing on cloud-native applications is expected. Within the serverless architecture, the backend code is run without the need to manage and provision the resources on servers. For example in FaaS, scaling,

including the resource provisioning and allocation, is automatically done by the provider whenever it is needed, to preserve the response time requirement of the application. In general, regarding the capabilities of new execution platforms and deployment architectures, the objectives of performance testing might be slightly influenced. Nevertheless, it is still crucial for a wide range of software systems.

7.8.3 Threats to Validity

Some of the main sources of threat to the validity of our experimental evaluation results are as follows:

Construct. One of the main sources of threat is the formulation of the RL technique to address the problem, which is very important for successful learning. Modeling the state space, actions, and also the reward function are major players to guide the agent throughout the learning and make it learn the optimal policy. For example, boundaries defined in discrete states modeling are a threat to internal validity. To mitigate this threat, we used a fuzzy labeling technique to deal with the issue of uncertainty in defining sharp values for boundaries. Regarding the actions, the formulation of actions affects the granularity of the exploration steps, thus we tried to define actions in a way to provide reasonable granularity for the exploration steps.

Internal. There are a number of threats to the internal validity of the results. RL techniques like many other machine learning algorithms are influenced by their hyperparameters such as learning rate and discount factor. During our efficiency and adaptivity analysis experiments, we did not change the learning parameters, we also conducted a set of controlled experiments to study the influence of learning parameters on the efficiency of our approach.

The insufficient number of learning episodes/iterations could also act as a source of threat in the initial learning. To alleviate this threat, we iterated the initial learning sufficiently to ensure the convergence. Moreover, using a performance simulation module instead of executing SUTs actually is considered as a source of threat to the validity of results.

Finally, model-free RL is mainly intended to solve a decision-making problem (to find an optimal policy to behave) without access to a model of the environment. Therefore, not considering the structure of the environment might be a source of threat in case of improper formulation of the RL

technique.

External. Model-free RL learns the optimal policy to achieve the target through interaction with the environment. Our approach was formulated based on the SUTs with three types of performance sensitivity that are CPU-intensive, memory-intensive, and disk-intensive, and our results are derived from the experimental evaluation of our approach on these types of SUTs. If the experiment contains SUTs with other types of performance sensitivity such as network-intensive programs, then the approach needs to be reformulated slightly to support new types of performance sensitivities.

Moreover, the dependency of the performance simulation module on the performance sensitivity values could raise a threat to validity in case of deploying the smart tester agent with the performance simulation module. The performance simulation module requires the performance sensitivity values for the SUTs as we described in our experiments. However, given a real deployment of the approach, e.g., in a cloud-based testing setup without the performance simulation module, the dependency on the performance sensitivity values are lighter and their exact values are not necessary. Nonetheless, it is still considered as a source of threat.

7.9 Related Work

Measurement of performance metrics under typical or stress test execution conditions, which involve both workload and platform configuration aspects [43, 44, 45, 46, 47], detection of performance-related issues such as functional problems or violations of performance requirements emerging under certain workload or resource configuration conditions [48, 49, 50, 24] are common objectives of different types of performance testing.

Different approaches have been proposed to design the target performance test cases for accomplishing performance-related objectives such as finding intended performance breaking points. Performance test conditions involve both workload and resource configuration status. A general high-level categorization of main techniques for generating the performance test cases is as follows:

Source code analysis. Deriving workload-based performance test conditions using data-flow analysis and symbolic execution are examples of techniques for designing fault-inducing performance test cases based on

source code analysis to detect performance-related issues such as functional problems (like memory leaks) and performance requirement violations [51, 49].

System model analysis. Modeling the system behavior in terms of performance models like Petri nets and using constraint solving techniques [14], using the control flow graph of the system and applying search-based techniques [15, 16], and using other types of system models like UML models and using genetic algorithms [17, 18, 19, 20, 21] to generate the performance test cases are examples of the techniques based on system model analysis for generating performance test cases.

Behavior-driven declarative techniques. Using a Domain Specific Language (DSL) to provide declarative goal-oriented specifications of performance tests and model-driven execution frameworks for automated execution of the tests [25, 26, 27], and using a high-level behavior-driven language inspired from Behavior-Driven Development (BDD) techniques to define test conditions [24] in combination with a declarative performance testing framework like BenchFlow [26] are examples of behavior-driven techniques for performance testing.

Modeling the realistic conditions. Modeling the real user behavior through stochastic form-oriented models [22, 23], extracting workload characteristics from the recorded requests and modeling the user behavior using, e.g., extended finite state machines (EFSMs) [52] or Markov chains [53], sandboxing services and deriving a regression model of the deployment environment based on the data resulting from sandboxing to estimate the service capacity [47], end-user clustering based on the business-level attributes extracted from usage data [54], and using automated GUI testing tools with capture and replay techniques to generate realistic interactive usage sequences [55] are examples of techniques based on modeling the realistic conditions to generate the performance test cases.

Machine learning-enabled techniques. Machine learning techniques such as supervised and unsupervised algorithms mainly work based on building models and extracting patterns (knowledge) from the data. While, some other techniques such as RL algorithms are intended to train the learner agent to solve the problems (tasks). The agent learns an optimal way to achieve an objective through interacting with the system. Machine learning has been widely used for analysis of data resulting from the performance testing and

also for performance preservation. For example, anomaly detection through analysis of performance data, e.g., resource usage, using clustering techniques [56], predicting reliability from the testing data using Bayesian Networks [57], performance signature identification based on performance data analysis using supervised and unsupervised learning techniques [58, 59], and also adaptive RL-driven performance in particular response time control for cloud services [35, 60, 61] and also software on other execution platforms, e.g., PLC-based real-time systems [62]. Machine learning has been also applied to the generation of performance test cases in some studies. For example, using symbolic execution in combination with an RL algorithm to find the worst-case execution path within a SUT [63], using RL to find a sequence of input workload leading to performance degradation [64], and a feedback-driven learning to identify the performance bottlenecks through extracting rules from execution traces [65]. There are also some adaptive techniques slightly analogous to the concept of RL for generating performance test cases. For example, an adaptive workload generation that adapts the workload dynamically based on some pre-defined adjustment policies [50], and a feedback-driven approach that uses search algorithms to benchmark an NFS server based on varying workload parameters to find the workload peak rate reaching the target response time confidence level.

7.10 Conclusion

Performance testing is a family of techniques commonly used as part of performance analysis, e.g., estimating performance metrics or detecting performance violations. One important goal of performance testing, particularly in mission-critical domains, is to verify the robustness of the SUT in terms of finding performance breaking point. Model-driven techniques might be used for this purpose in some cases, but drawing a precise model of the performance behavior of a complex software system under different application-, platform- and workload-based affecting factors is difficult. Furthermore, such modeling might disregard important implementation and deployment details. In software testing, source code analysis, system model analysis, use-case based design, and behavior-driven techniques are some common approaches for generating performance test cases. However, source code or other artifacts might not be available during the testing.

In this paper, we proposed a fuzzy reinforcement learning-based performance testing framework (SaFReL) that adaptively and efficiently generates the target performance test cases resulting in the intended performance breaking points for different software programs, without access to source code and system models. We used Q-learning augmented by fuzzy state modeling and an action selection strategy adaptation that resulted in a self-adaptive autonomous tester agent. The agent can learn the optimal policy to achieve the target (reaching the intended performance breaking point), reuse its learned policy when deployed to test similar software and adapt its strategy when targeting software with different characteristics.

We evaluated the efficiency and adaptivity of SaFReL through a set of experiments based on simulating the performance behavior of various SUT programs. During the experimental evaluation, we tried to answer how efficiently and adaptively SaFReL can perform testing of different SUT programs compared to a typical stress testing approach. We also performed a sensitivity analysis to explore how the efficiency of SaFReL is affected by changing the learning parameters.

We believe that the main strengths of using the intelligent automation offered by SaFReL are 1) efficient generation of test cases and reduction of computation time, and 2) less dependency on source code and models. Regarding applicability, we believe that SaFReL could be beneficial to the testing of software variants, evolving software during the (CI/CD) process, and regression performance testing. Applying some heuristics and techniques to speed up the exploration of the state space like using multiple cooperating agents, and also extending the proposed approach to support workload-based performance test cases are further steps to continue this research.

Acknowledgment

This work has been supported by and received funding partially from the TESTOMAT, XIVT, IVVES and MegaM@Rt2 European projects.

Bibliography

[1] NS8. Did You Know A Slow Webpage Can Cost You 7% of Your Sales, 2018. Available at https://www.ns8.com/en/ns8u/did-you-know/a-slow-webpage-can-cost-you-7-percent-of-your-sales, Retrieved July, 2019.

[2] Elaine J Weyuker and Filippos I Vokolos. Experience with performance testing of software systems: issues, an approach, and case study. *IEEE transactions on software engineering*, 26(12):1147–1156, 2000.

[3] Andreas Brunnert, André van Hoorn, Felix Willnecker, Alexandru Danciu, Wilhelm Hasselbring, Christoph Heger, Nikolas Herbst, Pooyan Jamshidi, Reiner Jung, Joakim von Kistowski, et al. Performance-oriented devops: A research agenda. *arXiv preprint arXiv:1508.04752*, 2015.

[4] Leonid Grinshpan. *Solving enterprise applications performance puzzles: queuing models to the rescue*. John Wiley & Sons, 2012.

[5] ISO 25000. ISO/IEC 25010 - System and software quality models, 2019. Available at https://iso25000.com/index.php/en/iso-25000-standards/iso-25010, Retrieved July, 2019.

[6] Martin Glinz. On non-functional requirements. In *15th IEEE International Requirements Engineering Conference (RE 2007)*, pages 21–26. IEEE, 2007.

[7] Lawrence Chung, Brian A Nixon, Eric Yu, and John Mylopoulos. *Non-functional requirements in software engineering*, volume 5. Springer Science & Business Media, 2012.

[8] Zhen Ming Jiang and Ahmed E Hassan. A survey on load testing of large-scale software systems. *IEEE Transactions on Software Engineering*, 41(11):1091–1118, 2015.

[9] Vittorio Cortellessa, Antinisca Di Marco, and Paola Inverardi. *Model-based software performance analysis*. Springer Science & Business Media, 2011.

[10] Mor Harchol-Balter. *Performance modeling and design of computer systems: queueing theory in action*. Cambridge University Press, 2013.

[11] Krishna Kant and MM Srinivasan. *Introduction to computer system performance evaluation*. McGraw-Hill College, 1992.

[12] Giovanni Denaro, Andrea Polini, and Wolfgang Emmerich. Early performance testing of distributed software applications. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 94–103. ACM, 2004.

[13] Pingyu Zhang, Sebastian Elbaum, and Matthew B Dwyer. Compositional load test generation for software pipelines. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 89–99. ACM, 2012.

[14] Jian Zhang and Shing Chi Cheung. Automated test case generation for the stress testing of multimedia systems. *Software: Practice and Experience*, 32(15):1411–1435, 2002.

[15] Yuanyan Gu and Yujia Ge. Search-based performance testing of applications with composite services. In *2009 International Conference on Web Information Systems and Mining*, pages 320–324. IEEE, 2009.

[16] Massimiliano Di Penta, Gerardo Canfora, Gianpiero Esposito, Valentina Mazza, and Marcello Bruno. Search-based testing of service level agreements. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1090–1097. ACM, 2007.

[17] Vahid Garousi. A genetic algorithm-based stress test requirements generator tool and its empirical evaluation. *IEEE Transactions on Software Engineering*, 36(6):778–797, 2010.

[18] Vahid Garousi. Empirical analysis of a genetic algorithm-based stress test technique. In *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 1743–1750. ACM, 2008.

[19] Vahid Garousi, Lionel C Briand, and Yvan Labiche. Traffic-aware stress testing of distributed real-time systems based on UML models using genetic algorithms. *Journal of Systems and Software*, 81(2):161–185, 2008.

[20] Leandro T Costa, Ricardo M Czekster, Flávio Moreira de Oliveira, Elder de M Rodrigues, Maicon Bernardino da Silveira, and Avelino F Zorzo. Generating performance test scripts and scenarios based on abstract intermediate models. In *SEKE*, pages 112–117, 2012.

[21] Maicon Bernardino da Silveira, Elder de M Rodrigues, Avelino F Zorzo, Leandro T Costa, Hugo V Vieira, and Flávio Moreira de Oliveira. Generation of scripts for performance testing based on UML models. In *SEKE*, pages 258–263, 2011.

[22] Dirk Draheim, John Grundy, John Hosking, Christof Lutteroth, and Gerald Weber. Realistic load testing of web applications. In *Conference on Software Maintenance and Reengineering (CSMR'06)*, pages 11–pp. IEEE, 2006.

[23] Christof Lutteroth and Gerald Weber. Modeling a realistic workload for performance testing. In *2008 12th International IEEE Enterprise Distributed Object Computing Conference*, pages 149–158. IEEE, 2008.

[24] Henning Schulz, Dušan Okanović, André van Hoorn, Vincenzo Ferme, and Cesare Pautasso. Behavior-driven load testing using contextual knowledge-approach and experiences. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, pages 265–272. ACM, 2019.

[25] Vincenzo Ferme and Cesare Pautasso. A declarative approach for performance tests execution in continuous software development

environments. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 261–272. ACM, 2018.

[26] Vincenzo Ferme and Cesare Pautasso. Towards holistic continuous software performance assessment. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, pages 159–164. ACM, 2017.

[27] Jürgen Walter, Andre van Hoorn, Heiko Koziolek, Dusan Okanovic, and Samuel Kounev. Asking what?, automating the how?: The vision of declarative performance engineering. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, pages 91–94. ACM, 2016.

[28] Kim Fowler. *Mission-critical and safety-critical systems handbook: Design and development for embedded applications*. Newnes, 2009.

[29] Brendan Jennings and Rolf Stadler. Resource management in clouds: Survey and research challenges. *Journal of Network and Systems Management*, 23(3):567–619, 2015.

[30] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: efficient performance prediction for large-scale advanced analytics. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, pages 363–378, 2016.

[31] Sergiy Kolesnikov, Norbert Siegmund, Christian Kästner, Alexander Grebhahn, and Sven Apel. Tradeoffs in modeling performance of highly configurable software systems. *Software & Systems Modeling*, 18(3):2265–2283, 2019.

[32] Roberto Morabito. Virtualization on internet of things edge devices with container technologies: a performance evaluation. *IEEE Access*, 5:8835–8850, 2017.

[33] Zoran B Babovic, Jelica Protic, and Veljko Milutinovic. Web performance evaluation for internet of things applications. *IEEE Access*, 4:6974–6992, 2016.

[34] Mahshid Helali Moghadam, Mehrdad Saadatmand, Markus Borg, Markus Bohlin, and Björn Lisper. Machine learning to guide performance testing: An autonomous test framework. In *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 164–167. IEEE, 2019.

[35] Olumuyiwa Ibidunmoye, Mahshid Helali Moghadam, Ewnetu Bayuh Lakew, and Erik Elmroth. Adaptive service performance control using cooperative fuzzy reinforcement learning in virtualized environments. In *Proceedings of the10th International Conference on Utility and Cloud Computing*, pages 19–28. ACM, 2017.

[36] Olumuyiwa Ibidunmoye, Francisco Hernández-Rodriguez, and Erik Elmroth. Performance anomaly detection and bottleneck identification. *ACM Computing Surveys (CSUR)*, 48(1):4, 2015.

[37] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM computing surveys (CSUR)*, 41(3):15, 2009.

[38] Brendan Gregg. *Systems performance: enterprise and the cloud*. Pearson Education, 2013.

[39] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[40] Ludmila I Kuncheva. Fuzzy classifiers. *Scholarpedia*, 3(1):2925, 2008.

[41] MathWorks. Fuzzy Inference Process, 2019. Available at `https://www.mathworks.com/help/fuzzy/fuzzy-inference-process.html`, Retrieved July, 2019.

[42] Javid Taheri, Albert Y Zomaya, and Andreas Kassler. vmbbthrpred: A black-box throughput predictor for virtual machines in cloud environments. In *European Conference on Service-Oriented and Cloud Computing*, pages 18–33. Springer, 2016.

[43] Daniel A Menascé. Load testing, benchmarking, and application performance management for the web. In *Int. CMG Conference*, pages 271–282, 2002.

[44] James Hill, D Schmidt, James Edmondson, and Aniruddha Gokhale. Tools for continuously evaluating distributed system qualities. *IEEE software*, 27(4):65–71, 2009.

[45] Varsha Apte, TVS Viswanath, Devidas Gawali, Akhilesh Kommireddy, and Anshul Gupta. AutoPerf: Automated load testing and resource usage profiling of multi-tier internet applications. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, pages 115–126. ACM, 2017.

[46] Nicolas Michael, Nitin Ramannavar, Yixiao Shen, Sheetal Patil, and Jan-Lung Sung. Cloudperf: A performance test framework for distributed and dynamic multi-tenant environments. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, pages 189–200. ACM, 2017.

[47] Anshul Jindal, Vladimir Podolskiy, and Michael Gerndt. Performance modeling for cloud microservice applications. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, pages 25–32. ACM, 2019.

[48] Lionel C Briand, Yvan Labiche, and Marwa Shousha. Stress testing real-time systems with genetic algorithms. In *Proceedings of the 7th annual conference on Genetic and evolutionary computation*, pages 1021–1028. ACM, 2005.

[49] Pingyu Zhang, Sebastian Elbaum, and Matthew B Dwyer. Automatic generation of load tests. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 43–52. IEEE Computer Society, 2011.

[50] Vanessa Ayala-Rivera, Maciej Kaczmarski, John Murphy, Amarendra Darisa, and A Omar Portillo-Dominguez. One size does not fit all: In-test workload adaptation for performance testing of enterprise applications. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 211–222. ACM, 2018.

[51] Cheer-Sun D Yang and Lori L Pollock. Towards a structural load testing tool. In *ACM SIGSOFT Software Engineering Notes*, volume 21, pages 201–208. ACM, 1996.

[52] Mahnaz Shams, Diwakar Krishnamurthy, and Behrouz Far. A model-based approach for testing the performance of web applications. In *Proceedings of the 3rd international workshop on Software quality assurance*, pages 54–61. ACM, 2006.

[53] Christian Vögele, André van Hoorn, Eike Schulz, Wilhelm Hasselbring, and Helmut Krcmar. WESSBAS: extraction of probabilistic workload specifications for load testing and performance prediction—a model-driven approach for session-based application systems. *Software & Systems Modeling*, 17(2):443–477, 2018.

[54] Gururaj Maddodi, Slinger Jansen, and Rolf de Jong. Generating workload for ERP applications through end-user organization categorization using high level business operation data. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 200–210. ACM, 2018.

[55] Andrea Adamoli, Dmitrijs Zaparanuks, Milan Jovic, and Matthias Hauswirth. Automated gui performance testing. *Software Quality Journal*, 19(4):801–839, 2011.

[56] Mark D Syer, Bram Adams, and Ahmed E Hassan. Identifying performance deviations in thread pools. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 83–92. IEEE, 2011.

[57] Alberto Avritzer, Flávio P Duarte, Rosa Maria Meri Leao, Edmundo de Souza e Silva, Michal Cohen, and David Costello. Reliability estimation for large distributed software systems. In *Cascon*, page 12. Citeseer, 2008.

[58] Haroon Malik, Hadi Hemmati, and Ahmed E Hassan. Automatic detection of performance deviations in the load testing of large scale systems. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 1012–1021. IEEE Press, 2013.

[59] Haroon Malik, Zhen Ming Jiang, Bram Adams, Ahmed E Hassan, Parminder Flora, and Gilbert Hamann. Automatic comparison of load tests to support the performance analysis of large enterprise systems. In *2010 14th European conference on software maintenance and reengineering*, pages 222–231. IEEE, 2010.

[60] T Veni and S Mary Saira Bhanu. Auto-scale: automatic scaling of virtualised resources using neuro-fuzzy reinforcement learning approach. *International Journal of Big Data Intelligence*, 3(3):145–153, 2016.

[61] Pooyan Jamshidi, Amir Sharifloo, Claus Pahl, Hamid Arabnejad, Andreas Metzger, and Giovani Estrada. Fuzzy self-learning controllers for elasticity management in dynamic cloud architectures. In *2016 12th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA)*, pages 70–79. IEEE, 2016.

[62] Mahshid Helali Moghadam, Mehrdad Saadatmand, Markus Borg, Markus Bohlin, and Björn Lisper. Adaptive runtime response time control in PLC-based real-time systems using reinforcement learning. In *2018 IEEE/ACM 13th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 217–223. IEEE, 2018.

[63] Jinkyu Koo, Charitha Saumya, Milind Kulkarni, and Saurabh Bagchi. PySE: Automatic worst-case test generation by reinforcement learning. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 136–147. IEEE, 2019.

[64] Tanwir Ahmad, Adnan Ashraf, Dragos Truscan, and Ivan Porres. Exploratory performance testing using reinforcement learning. In *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 156–163. IEEE, 2019.

[65] Mark Grechanik, Chen Fu, and Qing Xie. Automatically finding performance problems with feedback-directed learning software testing. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 156–166. IEEE, 2012.

Chapter 8

Paper C: Performance Testing Using a Smart Reinforcement Learning-Driven Test Agent

Mahshid Helali Moghadam, Golrokh Hamidi, Markus Borg, Mehrdad Saadatmand, Markus Bohlin, Björn Lisper, and Pasqualina Potena
In the Proceedings of IEEE Congress on Evolutionary Computation (CEC), IEEE 2021.

Abstract

Performance testing with the aim of generating an efficient and effective workload to identify performance issues is challenging. Many of the automated approaches mainly rely on analyzing system models, source code, or extracting the usage pattern of the system during the execution. However, such information and artifacts are not always available. Moreover, all the transactions within a generated workload do not impact the performance of the system the same way, a finely tuned workload could accomplish the test objective in an efficient way. Model-free reinforcement learning is widely used for finding the optimal behavior to accomplish an objective in many decision-making problems without relying on a model of the system. This paper proposes that if the optimal policy (way) for generating test workload to meet a test objective can be learned by a test agent, then efficient test automation would be possible without relying on system models or source code. We present a self-adaptive reinforcement learning-driven load testing agent, RELOAD, that learns the optimal policy for test workload generation and generates an effective workload efficiently to meet the test objective. Once the agent learns the optimal policy, it can reuse the learned policy in subsequent testing activities. Our experiments show that the proposed intelligent load test agent can accomplish the test objective with lower test cost compared to common load testing procedures, and results in higher test efficiency.

8.1 Introduction

Performance as an important quality characteristic plays a key role in the success of software products. Performance assurance is of great importance particularly in the domains where quality assurance of both functional and non-functional aspects of system's behavior is essential. For example, enterprise applications (EAs) [1] with Internet-based user interfaces such as e-commerce websites are examples of systems whose success is subject to performance assurance. EAs are often the core parts of the business organizations and their performance is a prerequisite for acceptable execution of business functions [2].

Performance, which is also called efficiency in classifications of quality attributes [3, 4, 5], generally describes how well the system accomplishes its functionality. It presents time and resource bound aspects of a system's behavior, which are indicated by some common performance metrics such as throughput, response time, and resource utilization. Performance analysis is conducted to meet the primary objectives as I) evaluating (measuring) performance metrics, II) detecting the functional problems emerging under specific execution conditions such as heavy workload, and III) detecting violations of performance requirements [6].

Performance modeling and testing are considered common approaches to accomplish the mentioned objectives at different stages of performance analysis. Although performance models [7, 8, 9] provide helpful insight into the behavior of a system, there are still many details of the implementation and the execution environment that might be ignored in the modeling [10]. Moreover, building a precise detailed model of the system behavior with regard to all the factors at play is often costly and sometimes impossible.

Performance testing as another family of techniques is supposed to meet the objectives of the performance analysis by executing the software under various realistic conditions.

Research Challenge. Load testing is a type of performance testing that focuses on analyzing the performance of the system when subjected to workloads. Workload is often configured as a set of concurrent (virtual) users doing different transactions on the software under test (SUT), which often mimic the behavior of the real users of the system [11]. Different transactions do not have the same impact on the performance, and generating an effective

test workload in an optimal way is challenging. Common load testing approaches such as the techniques using source code [12] and system model analysis (e.g., performance and UML models) [13, 14, 15, 16, 17], and also use case-based [18, 19] and behavior-driven [20, 21, 22] techniques all mostly rely on the artifacts that are not always available during the testing. Meanwhile, in the black-box testing, in order to efficiently generate an effective workload identifying the performance effects of the transactions involved in the workload is important and still challenging. Therefore, this paper is organized based on addressing the following question:

Research Goal. *How can we efficiently and adaptively generate test workload resulting in reaching the performance test objective for a SUT without relying on the source code and performance/system models?*

Contribution. In this paper, we present a self-adaptive model-free reinforcement learning load testing agent (RELOAD), which learns how to generate an effective test workload efficiently without relying on the system model or source code, and is able to reuse the learned policy in further testing scenarios. The test objective is defined as reaching a status under which a certain performance requirement gets violated.

Solution proposal. The proposed reinforcement learning-driven load testing agent identifies the effects of different transactions involved in the workload and learns how to adjust the transactions to meet the test objective. It assumes two learning phases: initial and transfer learning phases. It learns the optimal policy (way) to generate an effective workload in the initial learning. Then, in the transfer learning it is able to reuse adaptively the learned policy in further testing scenarios, i.e., with different test objectives. It uses Q-learning, a model-free reinforcement learning (RL) algorithm, as the core learning with an adaptive action selection strategy to be able to reuse the learned policy in the transfer learning. RELOAD uses a well-known load test actuator, i.e., Apache JMeter [23], to execute the designed workload on the SUT.

Experimental evaluation. We present a two-fold experimental evaluation, i.e., efficiency and sensitivity analysis, of the proposed approach on a functional e-commerce web application as SUT. In the experimental evaluation we address two main research questions which are as follows:

RQ1: How efficiently can RELOAD generate an effective test workload to meet the test objective?

RQ2: How is the efficiency of RELOAD affected by changing the learning parameters?

We consider test cost saving (reduction) and compare the efficiency of RELOAD based on four configurations of the proposed learning with a random (exploratory) and a standard baseline load testing approaches. According to the results of the efficiency analysis, after the initial learning RELOAD generates a more accurate and finely-tuned workload to meet the test objective with around 32% and 17% test cost saving compared to baseline and random approaches respectively. Moreover, once it learns how to tune the transactions to reach the objective, it reuses the learned policy and keep the efficiency, i.e., preserve around 25% and 13% test cost saving compared to baseline and random approaches respectively, in further testing scenarios without a need to redo the learning. Lastly, we also study the behavioral sensitivity of RELOAD to the learning parameters influencing the learning mechanism.

The rest of this paper is organized as follows: Section 8.2 discusses the motivation for applying model-free reinforcement learning to the problem and the primary concepts of RL. Section 8.3 presents the architecture and technical details of the proposed RL-assisted load testing agent. Section 8.4 presents the research method and experiments' setup. Section 8.5 discusses the experimental results, answers to RQs, and the threats to validity. Section 8.6 gives an overview of the related work. The conclusion and future research directions are presented in Section 8.7.

8.2 Motivation and Background

Any anomalies in the performance behavior of the system (e.g., performance requirement violation) could be mainly a consequence of emerging bottlenecks at the level of platform or application [24, 25]. A bottleneck can make the system fail or not perform as required, and can happen due to the full utilization of the component capacity, exceeding a usage threshold or occurrence of contention [26].

Possible defects in source code or architecture and some issues related to platform resources could be often the root causes of the emergence of bottlenecks. Moreover, all transactions do not have the same effect on the performance and some of them are more critical to lead to the emergence of

performance bottlenecks. Therefore, due to the existing interplay between the involved factors, drawing a detailed model expressing the performance behavior of the system, is not easily possible. This issue makes room for model-free machine learning techniques, such as model-free reinforcement learning (RL) [27] to play an interesting role in addressing the related challenges, in particular from testing perspective. RL algorithms are mainly intended to address decision-making problems and have been widely used to build self-adaptive intelligent systems.

In model-free RL the intelligent agent can learn an optimal behavior to achieve an intended objective based on the interaction with the environment (i.e., the system under test in this problem) without access to the source code or a model of SUT. Furthermore, the agent is able to store the gained knowledge and reuse the learned behavior in further potential testing situations such as regression testing or testing of SUT with regard to different test objectives. Model-free RL algorithms are not intended to build or learn a model of the environment. Instead, they learn optimal behavior to accomplish the objective through various episodes of interaction with the environment. They are apt for the problems where the model (i.e., the dynamics) of the environment is unknown or costly to be built, but the experience of interaction with the environment can be sampled and used.

8.2.1 Reinforcement Learning

Using RL, the agent learns the optimal behavior to meet the objective through being rewarded or punished in the interaction with the environment. At each step of the interaction, the agent observes the state of the system. It takes one possible action. The system undergoes changes upon actions. Then, the agent receives a reward signal showing how good the action was to direct the agent towards accomplishing the objective. The overall goal of the agent is formulated in terms of maximizing the cumulative long-term reward. The agent decides how to behave at each step of the interaction and based on optimizing the long-term received reward, learns the optimal behavior function which is called *optimal policy*. The agent uses an action selection strategy to interact with and apply actions to the system. The action selection is often based on trying the available actions, i.e., exploration of the action space, or relying on the learned policy which leads to selecting highly valued

actions, i.e., exploitation of the gained knowledge.

8.3 RELOAD Test Agent for Optimal Test Workload Generation

In this section, we present an overview of the architecture of our proposed RL-driven load testing agent, RELOAD, and describe the technical details of the learning procedure.

How it learns. It assumes two phases of learning, i.e., initial and transfer learning. During the initial learning, the test agent learns the optimal policy to generate an effective workload to accomplish the test objective. During the transfer learning, the learned policy is reused in further potential testing scenarios, e.g., regression testing scenarios or testing with regard to different test objectives. In the transfer learning phase, the agent also still continues with the learning to keep the policy updated.

We use Q-learning [27], a model-free RL algorithm, as the core learning technique. Figure 8.1 shows the architecture of RELOAD. The main constituent parts of each learning step in RELOAD are *detecting state*, *taking actions* and *computing reward* (See Section 8.2.1). We have formulated these parts in RELOAD as follows:

State Detection. Average response time and error rate, as two performance metrics, are used to indicate the performance state of the SUT. The values of these performance metrics are classified under a number of discrete classes, which are described as *Low*, *Normal* and *High* for response time and *Low* and *High* for error rate. The threshold (boundary) values for defining these classes are selected empirically and could be updated based on the requirements. The combinations of these classes form the discrete classes for the state of the system, as shown in Figure 8.2. Actually, different transactions do not have the same impact on the performance of the SUT, and test workloads with different configurations, i.e., in terms of constituent transactions, might lead the SUT to different performance states. The agent fetches these metrics from the test actuator at each learning step and identifies the state of the SUT.

Actions. At each learning step, the test agent takes one action after

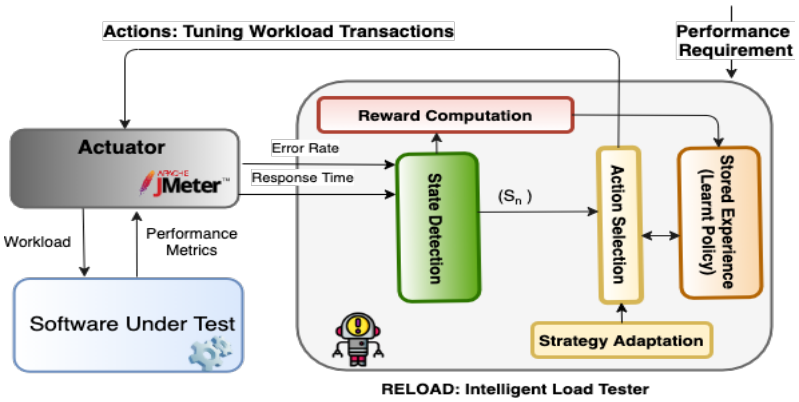


Figure 8.1: RELOAD, an RL-driven load testing agent

detecting the state of the SUT. We define the actions as adjusting the load of constituent transactions in the workload, in terms of numbers of virtual users running each transaction. Table 8.1 presents the list of transactions for the SUT in our case study, which is an e-commerce web application.

Each transaction involves a certain function together with its functional dependencies. For example, transaction *Add to cart* involves performing *login, accessing the search page, and selecting the product* as well, since all those functions are prerequisites for function *Add to cart*. Therefore, the function in each transaction of the workload is considered together with its functional dependencies. Then, the set of actions for the test agent is defined as follows:

$$ActionList = \{\cup\, action_k,\ 1 \leq k \leq |\textit{List of Transactions}|\} \tag{8.1}$$

$$\begin{aligned} action_k : \{&W_n^{T_j} = W_{n-1}^{T_j},\ \ for\ j \neq\ k, \\ &W_n^{T_j} = W_{n-1}^{T_j} + \frac{W_{n-1}^{T_j}}{3},\ for\ j = k, \\ &T_j \in \textit{List of transactions}, \\ &1 \leq j \leq |\textit{List of Transactions}|\} \end{aligned} \tag{8.2}$$

where T_j indicates a transaction of the SUT. $W_n^{T_j}$ indicates the load of transaction T_j at time step n, i.e., the number of users running this transaction.

After the agent decides on an action, a test plan is generated by the agent, and then is executed on the SUT by the test actuator, i.e., Apache JMeter.

Table 8.1: List of transactions for the SUT

Operation	***Description***
Home	Access to home page
Sign up page	Access to Sign up page
Sign up	Register and add a new user
Login page	Access to login page
Login	Sign in at the system
Search page	Access to search page
Select product	See the details of the selected product
Add to cart	Add the selected product to the cart
Payment	Access to payment page
Confirm	Confirm the order (payment)
Log out	Log out

Reward Signal. After taking the selected action and running the tuned workload, the test agent receives a reward signal which shows how effective the applied action was in leading the test agent to reaching the test objective. We define a function to represent the reward signal as follows:

$$R_n = (\frac{RT_n}{RT_{threshold}})^2 + (\frac{ER_n}{ER_{threshold}})^2 \tag{8.3}$$

where R_n, RT_n, and ER_n indicate the reward, the average response time, and the average error rate respectively, in step n. Also $RT_{threshold}$ and $ER_{threshold}$ are the response time and error rate thresholds related to the test objective.

Learning Procedure. In RL, the agent is intended to learn the optimal policy to accomplish the objective of the problem. The policy determines which action to be taken by the agent, given a certain state. The key idea for finding the optimal policy is the use of an iterative policy iteration process at

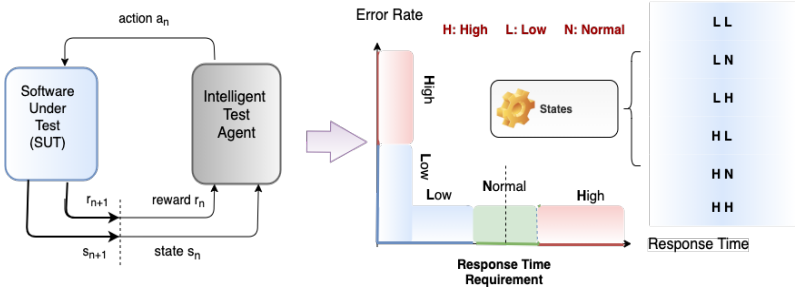


Figure 8.2: States of SUT in the proposed RL-based load testing

each step of the learning, which consists of policy evaluation and policy improvement. At each step of the interaction, the agent performs both evaluation and improvement. First, it evaluates the policy which it follows, then it tries to improve it through a greedy approach (e.g., ε-greedy). Finally, this process will converge on the optimal policy. In model-free RL, there are generally two approaches to realize this: learning the policy directly and indirectly. In the Q-learning algorithm, the agent learns an optimal value function, i.e., an action-value function $Q^*(s,a)$, from which the optimal policy can be obtained. The optimal action-value function, $Q^*(s,a)$, gives the expected long-term return, given state s, taking an arbitrary action a, and then following the optimal policy. It is presented as follows:

$$Q^*(s,a) = \underset{\pi}{\operatorname{argmax}}\, E^{\pi}[q_n | s_n = s, a_n = a] \tag{8.4}$$

$$q_n = \sum_{k=0}^{\infty} \gamma^k R_{n+k+1} \tag{8.5}$$

where γ is a discount factor for future rewards and q_n is the long-term return in terms of cumulative discounted reward. In general, the optimal policy selects the action maximizing the expected return given starting from state s. Moreover, according to the definition of $Q^*(s,a)$, given Q^*, the optimal action for state s, $a^*(s)$, is obtained as:

$$a^*(s) = \underset{a'}{\operatorname{argmax}}\, Q^*(s,a') \tag{8.6}$$

In order to obtain the optimal policy, Q-values are stored (e.g., in a Q-table or a neural network) and considered the experience of the agent. During the learning, the Q-values are updated incrementally according to Eq. 8.7:

$$Q(s_n, a_n) = (1 - \alpha)Q(s_n, a_n) + \alpha[R_{n+1} + \gamma \max_{a'} Q(s_{n+1}, a^{'})] \quad (8.7)$$

where α, $0 \leq \alpha \leq 1$ adjusts the rate of learning which controls the impact of new Q-values on the previous ones.

In this study the research problem, i.e., generating an effective workload to meet an intended test objective, is regarded as a sequential decision-making problem. Model-free RL is proposed as a beneficial learning solution to this problem since the SUT (environment) and execution platform are supposed to be initially unknown to the test agent. Then, in the proposed model-free RL-driven solution, the agent finds (learns) the optimal policy to generate an effective workload to accomplish the test objective through a built-in iterative policy evaluation-improvement process. Algorithms 11 and 12 present the procedure of the learning in the proposed RL-driven load testing agent.

In model-free RL, ε-greedy is a well-known method for action selection, when RL is used to find the optimal policy in a decision-making problem. It guarantees the sufficient continual exploration required for finding the optimal policy, and meanwhile provides a proper trade-off between exploration of the state-action space and exploitation of the learned value function. In ε-greedy, the value of ε adjusts the degree of exploration versus exploitation, as it leads the agent to select a high-value action based on the learned value function with probability (1-ε) or a random possible action with probability ε, given a certain state. In addition to Q-learning, we also implemented RELOAD with DQN [28], which is a combination of Q-learning and deep neural networks and suits the large scale problems where due to the big number of states and actions using tabular methods (i.e., Q-table) is not practical.

Algorithm 11 Adaptive Reinforcement Learning-Driven load Testing

Required: $\mathbb{S}, \mathbb{A}, \alpha, \gamma$;
Initialize Q-values, $Q(s,a) = 0\ \forall s \in \mathbb{S},\ \forall a \in \mathbb{A}$ and $\varepsilon = \upsilon\ , 0 < \upsilon < 1$;
while *Not (initial convergence reached)* **do**
    Learning_Episode (with initial action selection strategy, e.g., ε-greedy, initialized ε);
end
Store the learned policy;
Adapt the action selection strategy to transfer learning, i.e., tune parameter ε in ε-greedy;
while *true* **do**
    Learning_Episode with adapted strategy (e.g., new value of ε);
end

8.4 Method

We perform empirical evaluations of RELOAD [1] by running experiments on a mature open-source software, an e-commerce web application. Our target SUT is based on the widely-used WooCommerce platform and deployed using XAMPP on an Apache web server with PHP 7.4.13 and MariaDB 10.4.17. The experiments' environment consists of two virtual machines (VMs), as one of them hosts the SUT and the other one runs the load testing agent together with the test actuator. Each VM has 2 CPUs at 3.1GHz, 8GB of RAM, and Linux Ubuntu 16.04. We use Apache JMeter 5.2.1 as an actuator to execute the test workload on the SUT.

We design a series of experiments to assess the efficiency and sensitivity of RELOAD. The experiments investigate how different learning configurations (setups) affect the outcome of RELOAD. For comparative purposes, we also report results from random (exploratory) testing and a standard (naive) testing baseline. For all experimental runs, we translate differences in the number of generated concurrent virtual users to reduced testing costs.

Figure 8.3 shows an overview of the experimental setup. The Dependent Variable (DV) in all experimental runs is the number of generated virtual users. The Independent Variable (IV) defining different experimental runs is the test

[1] https://github.com/mahshidhelali/RL-Assisted-Performance-Testing

Algorithm 12 Learning_Episode

repeat

1. Detect the state (S_n) of the SUT;
2. Select an action (See Eq. 8.1) according to the action selection strategy, e.g., ε-greedy: select $a_n = \text{argmax}_{a \in \mathbb{A}} Q(s_n, a)$ with probability (1-ε) or a random a_k, $a_k \in \mathbb{A}$ with probability ε;
3. Take the selected action: Tune the workload and run the modified workload on the SUT;
4. Detect the new state (S_{n+1}) of the SUT;
5. Compute the reward, R_{n+1};
6. Update the Q-value of the pair of previous state and taken action $Q(s_n, a_n) = (1 - \alpha)Q(s_n, a_n) + \alpha[R_{n+1} + \gamma \max_{a'} Q(s_{n+1}, a')]$

until *meeting the stopping criteria (reaching the test objective)*;

generation technique. We explore six discrete levels of the IV: A1) RELOAD with $\varepsilon = 0.2$, A2) RELOAD with $\varepsilon = 0.5$, A3) RELOAD with decaying ε, A4) RELOAD with DQN; B) Standard Baseline; C) Random Testing. In A1)-A3) RELOAD is based on Q-learning together with ε-greedy with different values for ε.

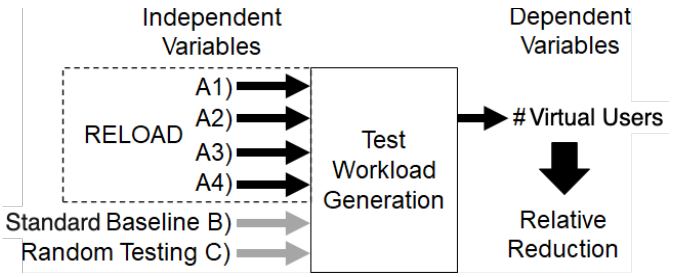


Figure 8.3: Overview of the experimental setup.

Section 8.3 describes the details of the RELOAD configurations in A1)-A4). The Standard Baseline (B) applies an initial workload that contains all the transactions with the *same* number of users per each transaction, then increases the number of users in fixed steps by 33% until accomplishing the

test objective. In Random Testing (C), a transaction is chosen randomly at each step, and then the number of virtual users allocated for the selected transaction is increased by 33%. The process is repeated until the test objective has been met.

The experimental runs corresponding to the six test generation techniques (i.e., values of the IV) are executed the same number of times, i.e., the same number of *episodes*. In RL each learning *episode* constitutes one complete sequence of states and actions in RL till reaching the objective (i.e., equivalent to one epoch in supervised learning). The agents' properties including the value function and policy are updated gradually over the learning episodes. Despite the lack of learning in the Baseline and Random testing, we refer to one complete execution for those techniques as an episode too.

In the **efficiency analysis**, we report results corresponding to the two learning phases of RELOAD. First, we analyze the initial learning. Second, we study how efficiently RELOAD performs during the transfer learning, i.e., when the agent reuses learned policies in new similar testing scenarios.

In the **sensitivity analysis**, we investigate the performance sensitivity of RELOAD to two learning hyperparameters, i.e., the *learning rate* α and the *discount factor* γ. We explore the two hyperparameters by changing one parameter while keeping the other one constant. As the sensitivity analysis followed the efficiency study, we based the design on our empirical observations at that point.

In the efficiency experiments, we use baseline values of $\alpha = 0.5$ and $\gamma = 0.5$. In the sensitivity experiments, we conduct four experimental runs to analyze the sensitivity of RELOAD. First, we set α to 0.1 and $decaying$ values while keeping the value of γ fixed at 0.5. Second, we set the γ to 0.1 and 0.9, while fixing α at 0.5.

8.5 Results and Discussion

This section presents our experimental results, answers the RQs, and discusses the main threats to validity.

8.5.1 Experimental Results

Efficiency Analysis. *Initial Learning*. To see how it works during the initial learning, we compare the efficiency of RELOAD for the learning configurations A1)-A4) (i.e., $\varepsilon = 0.2$, 0.5, decaying ε, and DQN) with the Standard Baseline and Random Testing. In particular, we are interested in studying the behavior of RELOAD after the initial convergence in comparison with other approaches. The convergence happens after around 30 episodes in Q-learning with ε-greedy (A1-A3) and in some episodes later for the DQN configuration (A4), i.e., after roughly 37 episodes. We consider the performance of the learning-based approach during the last 10 episodes after the convergence. We also run the Standard Baseline (B) and the Random Testing (C) 40 episodes. The test objective is reaching a performance status under which 1) the response time of the SUT exceeds $1,500ms$ or 2) the error rate in the received responses exceeds 20%.

Figure 8.4 shows the number of generated virtual users in all approaches to produce an effective workload accomplishing the test objective. Table 8.2 presents the resulting test cost saving at the last 10 episodes in RELOAD, i.e., the last 10 episodes show the behavior of the RL approach when it has almost achieved an initial convergence. We proceed by discussing the performance of RELOAD using the four configurations A1)-A4).

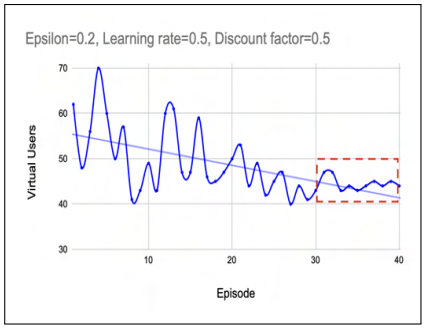
Q-learning with ε-greedy. Using $\varepsilon = 0.2$ (A1) makes the agent mainly rely on the stored experience rather than exploring new actions. It might slow down the learning convergence in a varying environment in which more exploration is needed. This issue is observable in terms of high spikes in Figure 8.4a. The configuration $\varepsilon = 0.5$. (A2) provides an equal likelihood for the exploitation of the learned policy and the exploration of new actions. The *decaying* ε setting (A3) decreases ε gradually over the learning episodes. It makes the agent explore new actions mainly during the early episodes of the learning and do more exploitation of the learned policy in the later episodes. The efficiency of the three configurations A1)-A3) are comparable, i.e., they converge roughly on the same number of virtual users needed to meet the test objective. *DQN setup.* DQN (A4) is an extension of Q-learning that uses a deep neural network as a function approximator instead of a Q-table to approximate the Q-values. In this experiment, the A4 obtain roughly the same efficiency as Q-learning with ε-greedy (A1-A3). This is also in line with

previous works on the use of Q-learning for performance assurance purposes [29], i.e., for problems that are not high-dimensional and satisfy the required conditions for Q-learning convergence, it is possible to obtain desired results using Q-learning with a carefully selected configuration. For the transfer learning part, we proceed with the A3 configuration.

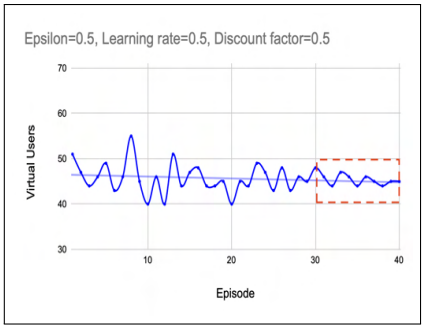
Table 8.2: Average test cost saving of RELOAD in the initial learning

	RELOAD			
Test Cost Saving	$\varepsilon = 0.5$	$\varepsilon = 0.2$	decaying ε	DQN setup
w.r.t Standard Baseline	30%	30%	34%	34%
w.r.t Random Testing	17%	17%	20%	20%

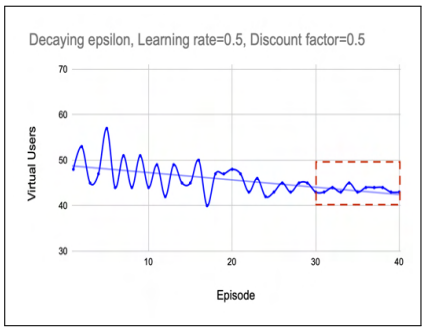
Transfer Learning. After the initial convergence, we study the efficiency of RELOAD in reusing the learned policy in further similar testing situations (scenarios) during the transfer learning. In this part of the experimentation, after an initial learning of 40 episodes with RELOAD configuration A3, we continue with 10 additional episodes (i.e., episodes 41-50 in Figure 8.5a). For these 10 episodes, we change the test objective and keep the ε low to guide the agent towards relying on the learned policy. Over the episodes of transfer learning, we alter the threshold of the target performance status (i.e., test objective). We change the target error rate threshold from 0.2 to 0.3 gradually by an increase of 0.01 at each episode and also change the target threshold for response time from $1,500ms$ to $2,500ms$ by an increase of $100ms$ at each episode. Figure 8.5 shows the efficiency of RELOAD in accomplishing the test objective in the further similar testing scenarios (i.e., represented by the 10 episodes, 41-50, after the initial learning) compared to the Standard Baseline and Random Testing. It indicates that the smart test agent is able to properly reuse the learned policy in the similar testing scenarios, i.e., the episodes with new values of test objectives, and still accomplish the test objective more efficiently. Table 8.3 presents the resulting test cost reduction of RELOAD in the transfer learning.



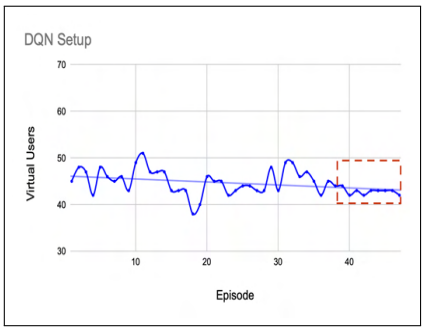
(a) RELOAD with Q-learning, ε-greedy, $\varepsilon = 0.2$



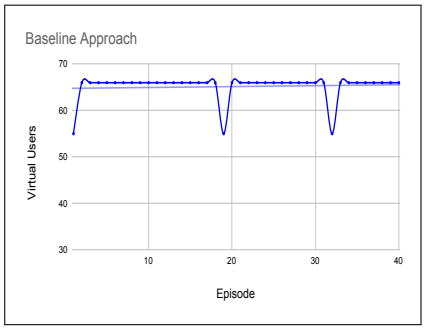
(b) RELOAD with Q-learning, ε-greedy, $\varepsilon = 0.5$



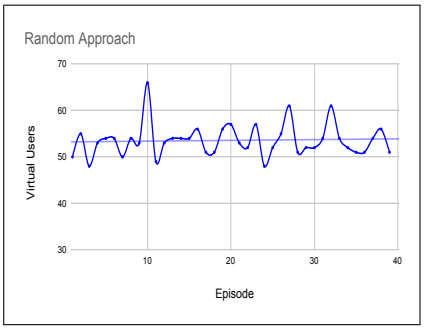
(c) RELOAD with Q-learning, ε-greedy, decaying ε



(d) RELOAD with DQN setup



(e) Baseline Approach

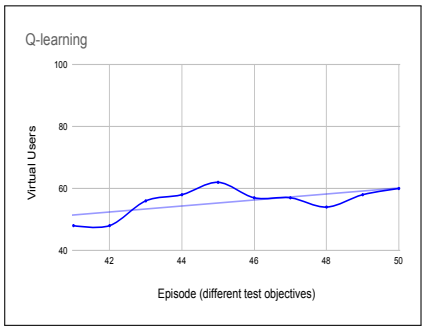


(f) Random Approach

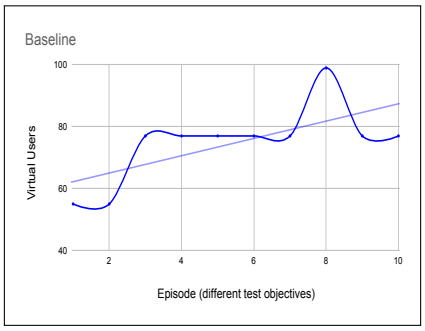
Figure 8.4: Test efficiency of RELOAD (initial learning for configurations A1-A4), Standard Baseline, and Random Testing

Table 8.3: Efficiency and average test cost saving in the transfer learning

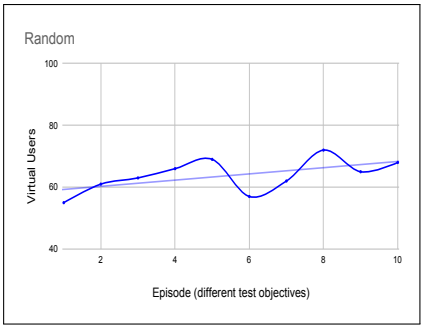
	RELOAD (with Q-learning)	**Standard Testing**	**Random Testing**
Range of the number of generated virtual users	48-62	55-99	55-68
RELOAD test cost saving		25%	13%



(a) RELOAD in the transfer learning



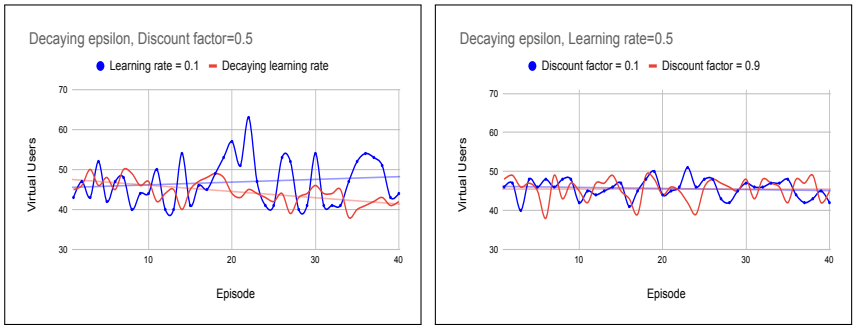
(b) Standard Baseline



(c) Random Testing

Figure 8.5: Efficiency of RELOAD (in the transfer learning) vs. the baseline and random approaches in new similar testing scenarios

Sensitivity Analysis. We select ε-greedy with decaying ε (A3) as the learning configuration in the sensitivity analysis. Figure 8.6 shows the behavioral performance of RELOAD regarding changing the values of hyperparameters as described in Section 8.4. It presents how different values for the learning hyperparameters influence the learning behavior, e.g., convergence, and the learning trend, in the proposed RL-driven test agent. We observe that RELOAD does not converge using a low learning rate, i.e., $\alpha = 0.1$. Furthermore, we find slower convergence using both lower and higher discount rates, i.e,. $\gamma = 0.1$ and $\gamma = 0.9$.



(a) (b)

Figure 8.6: Behavioral sensitivity of RELOAD to the learning hyperparameters

8.5.2 Revisiting the Research Questions

RQ1. As shown in Figure 8.4 and Table 8.2, on average RELOAD leads to accomplishing the test objective using fewer virtual users than the Standard Baseline and Random Testing. RELOAD learns how to meet the objective with a more accurate and fine-tuned workload and subsequently leads to a considerable test cost saving. In particular, RELOAD based on ε-greedy, decaying ε and DQN setup, offers a smoother learning trend and after the convergence results in a slightly higher cost saving, i.e., 34% and 20%, compared to other learning configurations. Based on our experiments, we conclude that RELOAD results in 10-30% increased test efficiency.

The test agent learns the optimal policy to meet the test objective efficiently

over the learning episodes. The optimal policy is learned through the value function. The agent stores the learned value function and is able to exploit it in further testing scenarios. The results of the efficiency analysis in the transfer learning (see Figure 8.5 and Table 8.3) confirm that the test agent after the initial learning is able to reuse the gained knowledge in subsequent testing scenarios, in which the SUT displays similar characteristics, and maintain its efficiency across scenarios. As shown in Table 8.3, RELOAD leads to 25% and 13% test cost saving in the transfer learning compared to the Standard Baseline and Random Testing.

RQ2. As shown in Figure 8.6, in the sensitivity analysis experiments, fixing the learning rate at a low value such as 0.1 did not lead to a learning convergence. Whereas a higher value such as 0.5 (as used in efficiency analysis) or using a decaying learning rate results in faster updates in the stored Q-table of the agent and works better in this case study. Moreover, changing the values of the discount factor, e.g., setting it to 0.1 or 0.9, appears to slow down the learning convergence.

Applicability. RELOAD generates an effective workload efficiently without relying on source code or a system model. It is well-suited to operational contexts where the source code, system models, and behavior specifications are not available. Meanwhile, the pay-as-you-go cost for many of the load generation tools on the market is proportional to the number of generated virtual users. Therefore, the efficient generation of an effective workload by the proposed test agent could lead to considerable cost and time savings in the testing process. Moreover, the proposed smart test agent has the capability of reusing the learned policy in further similar testing scenarios. RELOAD keeps the learning running to adapt the learned policy to changes in the environment. This feature is particularly beneficial to DevOps continuous testing activities such as performance regression testing where performance testing scenarios must be repeated for the SUT in a continuous integration process.

8.5.3 Threats to Validity

Some of the potential sources of threats to validity of the experimental results are described as follows:

Construct validity. One of the main sources of threat is the formulation of

the RL technique to address the problem. Formulating the states, actions, and also the reward function is a major step in building an RL-driven smart agent.

Internal validity. Dependency on the resource availability in the execution environment of the SUT is another common source of threats to the validity of the results in performance testing. To tackle this potential threat, we perform the experiments on dedicated virtual machines, i.e., two separate VMs were used for running the SUT and the test agent.

External validity. In our case, the approach has been formulated based on a particular e-commerce web-application as SUT, which supports a certain set of transactions. Therefore, in order to apply the approach to other types of applications, e.g., other web-based systems, the involved transactions of the new system should be extracted and included in the set of actions.

8.6 Related Work

Measuring performance metrics under different execution conditions including various workload and platform configurations [30, 31], detecting different performance-related issues such as functional problems or violations of performance requirements [32, 33] are common objectives of different types of performance testing. An overview of the techniques used for generating test workload is presented as follows:

Analyzing system models. Analysis of a performance model of SUT in Petri nets using constraint solving techniques [34], using genetic algorithms to generate test load based on the control flow graph of SUT [13], applying genetic algorithms to other types of system models such as UML models to generate stress test load [14, 15, 16, 17] are samples of the techniques in this category.

Analyzing source code. Generating test load using the analysis of SUT's data-flow and symbolic execution [35, 32] are examples of using source code analysis to generate test load and find performance-related issues.

Modeling real usage. Extracting the usage pattern of real users and modeling their behavior using form-oriented models [18, 19], extracting workload characteristics and modeling the user behavior based on Extended Finite State Machines [36] and Markov chains [37] through monitoring submitted requests to SUT, and workload characterization through users clustering based on the business-level attributes extracted from usage data [38] are examples of the

techniques used for modeling the realistic workload.

Declarative specification-based methods. Using a declarative Domain Specific Language (DSL) to specify the performance testing process together with a model-driven test execution framework [21, 22] and also using a specific behavior-driven language, to specify load testing process in combination with a declarative performance testing framework like BenchFlow [20] are examples of declarative techniques for performance and load testing.

Machine learning-assisted methods. Machine learning techniques such as supervised and unsupervised algorithms are often intended to build models and knowledge patterns from the data, while in other techniques like reinforcement learning algorithms, the intelligent agent learns the way to accomplish an objective through interaction with the environment. Machine learning techniques have been frequently used for analyzing the resulted data, e.g., for anomaly detection [39] and reliability prediction [40].

Machine learning techniques have been also applied to the generation of performance test conditions in some studies. For example, using RL together with symbolic execution to find the worst-case execution path within an SUT in [41], a feedback-driven learning technique which extracts some rules from the execution traces to find the performance bottlenecks, [42], and using RL to build a smart performance testing framework which mainly generates the platform-based test conditions [43, 44, 45]. Regarding generating performance test conditions, a few studies have also used some other adaptive techniques to generate the test workload. A feedback-based approach using search algorithms to benchmark an NFS server based on changing the test workload in [46], and an adaptive generation of test workload based on using some pre-defined tuning policies in [33] are some other examples of using adaptive approaches for the generation of performance test conditions.

8.7 Conclusion

System models, source code, and user behavior patterns are common sources of information in load testing techniques for generating test workload to find performance issues. Nonetheless, those artifacts might not be available all the time during the testing. Moreover, in black-box testing approaches, it is important to consider that not all transactions have the same effect on the

performance, i.e., tuning the workload optimally is crucial for test efficiency. We proposed RELOAD, a self-adaptive model-free RL-driven load testing agent that learns how to tune transactions in the workload to accomplish the test objective. It learns an optimal policy to generate an effective workload efficiently and is able to reuse the learned policy in further similar testing scenarios, e.g., in performance regression testing. Furthermore, RELOAD adapts the learned policy to continuous changes in the SUT and the execution environment, thus we believe the smart test agent to be particularly well-suited to the continuous performance testing context within DevOps. The smart test agent assumes two phases of initial and transfer learning and uses Q-learning as the core learning algorithm. It performs more efficiently than random and baseline load testing approaches, which enables reduced testing costs.

We conclude that RELOAD provides three main strengths. First, RELOAD provides efficient generation of effective test workloads. Second, the RL approach reduces source code and model dependencies, e.g., system models and user behavior models. Third, RELOAD enables generalizable knowledge representation, i.e., previously learned policies can be reused for other testing scenarios on the SUT. We posit that RELOAD can reduce costs in performance testing. Furthermore, the continuous testing context that permeates contemporary DevOps processes would further amplify the benefits. In future work, we plan to conduct empirical studies to validate our claims.

Acknowledgment

This work has been supported by and received funding from the TESTOMAT and IVVES European projects.

Bibliography

[1] Leonid Grinshpan. *Solving enterprise applications performance puzzles: queuing models to the rescue*. John Wiley & Sons, 2012.

[2] Andreas Brunnert, André van Hoorn, Felix Willnecker, Alexandru Danciu, Wilhelm Hasselbring, Christoph Heger, Nikolas Herbst, Pooyan Jamshidi, Reiner Jung, Joakim von Kistowski, et al. Performance-oriented devops: A research agenda. *arXiv preprint arXiv:1508.04752*, 2015.

[3] ISO 25000. ISO/IEC 25010 - System and software quality models, 2019. Available at https://iso25000.com/index.php/en/iso-25000-standards/iso-25010, Retrieved July, 2019.

[4] Martin Glinz. On non-functional requirements. In *15th IEEE International Requirements Engineering Conference (RE 2007)*, pages 21–26. IEEE, 2007.

[5] Lawrence Chung, Brian A Nixon, Eric Yu, and John Mylopoulos. *Non-functional requirements in software engineering*, volume 5. Springer Science & Business Media, 2012.

[6] Zhen Ming Jiang and Ahmed E Hassan. A survey on load testing of large-scale software systems. *IEEE Transactions on Software Engineering*, 41(11):1091–1118, 2015.

[7] Vittorio Cortellessa, Antinisca Di Marco, and Paola Inverardi. *Model-based software performance analysis*. Springer Science & Business Media, 2011.

[8] Mor Harchol-Balter. *Performance modeling and design of computer systems: queueing theory in action*. Cambridge University Press, 2013.

[9] Krishna Kant and MM Srinivasan. *Introduction to computer system performance evaluation*. McGraw-Hill College, 1992.

[10] Giovanni Denaro, Andrea Polini, and Wolfgang Emmerich. Early performance testing of distributed software applications. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 94–103. ACM, 2004.

[11] patterns & practices: Performance Testing Guidance for Web Applications, 2010. Available at `https://archive.codeplex.com/?p=perftestingguide`, Retrieved January, 2020.

[12] Pingyu Zhang, Sebastian Elbaum, and Matthew B Dwyer. Compositional load test generation for software pipelines. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 89–99. ACM, 2012.

[13] Yuanyan Gu and Yujia Ge. Search-based performance testing of applications with composite services. In *2009 International Conference on Web Information Systems and Mining*, pages 320–324. IEEE, 2009.

[14] Vahid Garousi. A genetic algorithm-based stress test requirements generator tool and its empirical evaluation. *IEEE Transactions on Software Engineering*, 36(6):778–797, 2010.

[15] Vahid Garousi, Lionel C Briand, and Yvan Labiche. Traffic-aware stress testing of distributed real-time systems based on UML models using genetic algorithms. *Journal of Systems and Software*, 81(2):161–185, 2008.

[16] Leandro T Costa, Ricardo M Czekster, Flávio Moreira de Oliveira, Elder de M Rodrigues, Maicon Bernardino da Silveira, and Avelino F Zorzo. Generating performance test scripts and scenarios based on abstract intermediate models. In *SEKE*, pages 112–117, 2012.

[17] Maicon Bernardino da Silveira, Elder de M Rodrigues, Avelino F Zorzo, Leandro T Costa, Hugo V Vieira, and Flávio Moreira de Oliveira.

Generation of scripts for performance testing based on UML models. In *SEKE*, pages 258–263, 2011.

[18] Dirk Draheim, John Grundy, John Hosking, Christof Lutteroth, and Gerald Weber. Realistic load testing of web applications. In *Conference on Software Maintenance and Reengineering (CSMR'06)*, pages 11–pp. IEEE, 2006.

[19] Christof Lutteroth and Gerald Weber. Modeling a realistic workload for performance testing. In *2008 12th International IEEE Enterprise Distributed Object Computing Conference*, pages 149–158. IEEE, 2008.

[20] Henning Schulz, Dušan Okanović, André van Hoorn, Vincenzo Ferme, and Cesare Pautasso. Behavior-driven load testing using contextual knowledge-approach and experiences. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, pages 265–272. ACM, 2019.

[21] Vincenzo Ferme and Cesare Pautasso. A declarative approach for performance tests execution in continuous software development environments. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 261–272. ACM, 2018.

[22] Jürgen Walter, Andre van Hoorn, Heiko Koziolek, Dusan Okanovic, and Samuel Kounev. Asking what?, automating the how?: The vision of declarative performance engineering. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, pages 91–94. ACM, 2016.

[23] Apache. JMeter. Available at `https://https://jmeter.apache.org/`, Retrieved October, 2019.

[24] Olumuyiwa Ibidunmoye, Francisco Hernández-Rodriguez, and Erik Elmroth. Performance anomaly detection and bottleneck identification. *ACM Computing Surveys (CSUR)*, 48(1):4, 2015.

[25] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM computing surveys (CSUR)*, 41(3):15, 2009.

[26] Brendan Gregg. *Systems performance: enterprise and the cloud*. Pearson Education, 2013.

[27] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[28] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[29] Olumuyiwa Ibidunmoye, Mahshid Helali Moghadam, Ewnetu Bayuh Lakew, and Erik Elmroth. Adaptive service performance control using cooperative fuzzy reinforcement learning in virtualized environments. In *Proceedings of the10th International Conference on Utility and Cloud Computing*, pages 19–28. ACM, 2017.

[30] Varsha Apte, TVS Viswanath, Devidas Gawali, Akhilesh Kommireddy, and Anshul Gupta. AutoPerf: Automated load testing and resource usage profiling of multi-tier internet applications. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, pages 115–126. ACM, 2017.

[31] Anshul Jindal, Vladimir Podolskiy, and Michael Gerndt. Performance modeling for cloud microservice applications. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, pages 25–32. ACM, 2019.

[32] Pingyu Zhang, Sebastian Elbaum, and Matthew B Dwyer. Automatic generation of load tests. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 43–52. IEEE Computer Society, 2011.

[33] Vanessa Ayala-Rivera, Maciej Kaczmarski, John Murphy, Amarendra Darisa, and A Omar Portillo-Dominguez. One size does not fit all: In-test workload adaptation for performance testing of enterprise applications. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 211–222. ACM, 2018.

[34] Jian Zhang and Shing Chi Cheung. Automated test case generation for the stress testing of multimedia systems. *Software: Practice and Experience*, 32(15):1411–1435, 2002.

[35] Cheer-Sun D Yang and Lori L Pollock. Towards a structural load testing tool. In *ACM SIGSOFT Software Engineering Notes*, volume 21, pages 201–208. ACM, 1996.

[36] Mahnaz Shams, Diwakar Krishnamurthy, and Behrouz Far. A model-based approach for testing the performance of web applications. In *Proceedings of the 3rd international workshop on Software quality assurance*, pages 54–61. ACM, 2006.

[37] Christian Vögele, André van Hoorn, Eike Schulz, Wilhelm Hasselbring, and Helmut Krcmar. WESSBAS: extraction of probabilistic workload specifications for load testing and performance prediction—a model-driven approach for session-based application systems. *Software & Systems Modeling*, 17(2):443–477, 2018.

[38] Gururaj Maddodi, Slinger Jansen, and Rolf de Jong. Generating workload for ERP applications through end-user organization categorization using high level business operation data. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 200–210. ACM, 2018.

[39] Mark D Syer, Bram Adams, and Ahmed E Hassan. Identifying performance deviations in thread pools. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 83–92. IEEE, 2011.

[40] Alberto Avritzer, Flávio P Duarte, Rosa Maria Meri Leao, Edmundo de Souza e Silva, Michal Cohen, and David Costello. Reliability estimation for large distributed software systems. In *Cascon*, page 12. Citeseer, 2008.

[41] Jinkyu Koo, Charitha Saumya, Milind Kulkarni, and Saurabh Bagchi. PySE: Automatic worst-case test generation by reinforcement learning. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 136–147. IEEE, 2019.

[42] Mark Grechanik, Chen Fu, and Qing Xie. Automatically finding performance problems with feedback-directed learning software testing. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 156–166. IEEE, 2012.

[43] Mahshid Helali Moghadam, Mehrdad Saadatmand, Markus Borg, Markus Bohlin, and Björn Lisper. Machine learning to guide performance testing: An autonomous test framework. In *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 164–167. IEEE, 2019.

[44] Mahshid Helali Moghadam. Machine learning-assisted performance testing. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1187–1189, 2019.

[45] Mahshid Helali Moghadam, Mehrdad Saadatmand, Markus Borg, Markus Bohlin, and Bjorn Lisper. Poster: Performance testing driven by reinforcement learning. In *Proceedings of the 2020 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2020.

[46] Piyush Shivam, Varun Marupadi, Jeffrey S Chase, Thileepan Subramaniam, and Shivnath Babu. Cutting corners: Workbench automation for server benchmarking. In *USENIX Annual Technical Conference*, pages 241–254, 2008.

Chapter 9

Paper D: Machine Learning Testing in an ADAS Case Study Using Simulation-Integrated Bio-Inspired Search-Based Testing

Mahshid Helali Moghadam, Markus Borg, Mehrdad Saadatmand, Seyed Jalaleddin Mousavirad, Markus Bohlin, and Björn Lisper
Technical report, Mälardalen University, 2022 (submitted for journal publication).

Abstract

This paper presents an extended version of Deeper, a search-based simulation-integrated test solution that generates failure-revealing test scenarios for testing a deep neural network-based lane-keeping system. In the newly proposed version, we utilize a new set of bio-inspired search algorithms, genetic algorithm (GA), $(\mu + \lambda)$ and (μ, λ) evolution strategies (ES), and particle swarm optimization (PSO), that leverage a quality population seed and domain-specific cross-over and mutation operations tailored for the presentation model used for modeling the test scenarios. In order to demonstrate the capabilities of the new test generators within Deeper, we carry out an empirical evaluation and comparison with regard to the results of five participating tools in the cyber-physical systems testing competition at SBST 2021. Our evaluation shows the newly proposed test generators in Deeper not only represent a considerable improvement on the previous version but also prove to be effective and efficient in provoking a considerable number of diverse failure-revealing test scenarios for testing an ML-driven lane-keeping system. They can trigger several failures while promoting test scenario diversity, under a limited test time budget, high target failure severity, and strict speed limit constraints.

9.1 Introduction

Machine Learning (ML) nowadays is used in a wide range of application areas such as automotive [1, 2], health care [3] and manufacturing [4]. Many of the ML-driven systems in these domains present a high level of autonomy [5] and meanwhile are subject to rigorous safety requirements [6]. In 2018, the European Commission (EC) published a strategy for trustworthy Artificial Intelligence (AI) systems [7]. In this strategy, AI systems are defined as "systems that display intelligent behavior by analyzing their environment and taking actions–with some degree of autonomy–to achieve specific goals". The EC states that a trustworthy AI system must be lawful, ethical, and robust. Self-driving cars are examples of safety-critical AI systems, which leverage various ML techniques such as Deep Neural Networks (DNN), machine vision, and sensor data fusion. Meanwhile, in the context of automotive software engineering, there is always a set of strict safety requirements to meet.

The quality assurance methodology for AI systems [8] is quite different from the conventional software systems, since the included ML components in those systems are not explicitly programmed, they are intended to learn from data and experience instead—called *Software 2.0* [9]. In addition, in AI systems, a part of the requirements is mainly seen as encoded implicitly in the data and the challenge of under-specificity is common in requirements definitions. However, it is still highly expected to assure the ability of the AI system to control the risk of hazardous events in particular in safety-critical domains. In this regard, Hawkins et al. [8] introduced a methodology for assurance of ML in autonomous systems, called AMLAS. It presents a systematic process for integrating safety assurance into the development of ML systems and introduces verification procedures at different stages, e.g., learning model verification and system-level (integration) verification that happens after integrating the ML model into the system.

There is also a vigorous need for integration verification and validation (V&V) of ML models deployed in self-driving cars to make sure that they are safe and dependable. Many of the failures basically emerge in the interplay between software containing ML components, hardware, and remote sensing devices, e.g., sensors, cameras, RADAR, and LiDAR technologies. Hardware-In-the-Loop (HIL), simulation-based and field testing are common

approaches for system-level verification of deployed ML models [8]. System-level testing mainly targets defining a set of operational scenarios that could lead to failures. In this regard, in the ISO/PAS 21448 Safety of the Intended Function (SOTIF) standard [10]—which addresses complementary aspects of functional safety in ISO 26262 [11]—simulation-based testing has been considered a proficient approach and a proper complementary solution to the on-road testing. Testing on real-world roads is costly, does not scale to cover all the needed scenarios, and in addition, it is dangerous to create and execute critical scenarios. The use of virtual prototyping allows testing and verification at the early stages of the development and offers the possibility of efficient and effective testing. It can capture the whole of the operational environment to a great extent using thoroughgoing physics-based simulators [12]. Recently, a growing number of commercial and open-source simulators have been developed to support the need for realistic simulation of self-driving cars [13, 14, 15]—we refer interested readers to a review by Rosique et al. [16].

Research Challenge. In this study, we target an Advanced Driver-Assistance System (ADAS) that provides lane-keeping assistance. Effective and efficient system-level testing in simulation environments requires sophisticated approaches to generate critical test scenarios. The critical test scenarios are those that break or are close to break the safety requirements of the ADAS under test, which hence result in *safety violations*. Generating effective test scenarios involves sampling from a large and complex set of test inputs. Several authors have shown the potential of search-based software test generation techniques to address this challenge. Various system-level testing techniques using different search-based testing approaches [17, 18, 19, 20, 21, 22] for different types of ADAS, relying on simulators, have been proposed in recent years.

Research Contribution. In this paper, we present a bio-inspired computation-driven test generator, called Deeper, for effective and efficient generation of failure-revealing test scenarios to test a Deep Neural Network (DNN)-based lane-keeping system in the BeamNG driving simulator. The test subject is BeamNG.AI, the built-in ML-driven driving agent in the BeamNG simulator. In this study, a failure is defined in terms of episodes in which the ego car—driven by the BeamNG.AI agent—drives partially outside the lane w.r.t a certain tolerance threshold. The tolerance threshold determines the

percentage of the car's bounding box needed to be outside the lane to be regarded as a failure.

Deeper in its current version in this paper benefits from the genetic algorithm (GA), $(\mu + \lambda)$ and (μ, λ) evolution strategies (ES), and the particle swarm optimization (PSO) to generate failure-revealing test scenarios, which are test roads in our study. The problem is basically regarded as an optimization problem, and in order to generate the test scenarios that are of interest, we evaluate the quality of the test scenarios using a *fitness (objective) function* that guides the search process to maximize the detected distance of the car from the center of the lane during driving of the car on the lane. The initial version of Deeper [20] contained a test generator based on NSGA-II. In this paper, we extend Deeper with four additional test generators based on GA, $(\mu + \lambda)$ and (μ, λ) ESs, and PSO. In the newly proposed test generators, we leverage an initial quality population seed to boost the search process, and also propose and develop domain-specific cross-over and mutation operations tailored for the presentation model used for modeling the test scenarios in the search algorithms. We rely on the presentation model used by DeepJanus [23] based on Catmull-Rom cubic splines [24].

Empirical evaluation. In order to carry out an empirical evaluation, we use the setup provided by the cyber-physical systems (CPS) testing competition[1] at the IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST). Our experiments are designed to answer three main research questions which are as follows:

RQ1: How capable are these test generators to trigger failures?

RQ2: How diverse are the generated failure-revealing test scenarios?

RQ3: How effectively and efficiently do the test generators perform? I.e., given a certain test budget, how many test scenarios are generated, what proportion of the scenarios is valid, and finally what proportion of the valid test scenarios leads to triggering failures?

We provide a comparative analysis on the performance of the proposed bio-inspired test generators in Deeper and five counterpart tools all integrated into the BeamNG simulator. In this regard, we compare the results of the proposed test generators in Deeper with five other test generator tools, Frenetic [25], GABExploit and GABExplore [22], Swat [26], and also the earlier version of the Deeper (based on NSGA-II) [20]—all participating tools

[1] Available at https://github.com/se2p/tool-competition-av.

in the CPS testing competition at SBST 2021. In order to do a fair comparison, we consider the same experimental evaluation procedures as the original CPS tool competition. Our experimental results show that first, the newly proposed test generators in Deeper present a considerable improvement on the previous version, second, they perform as effective and efficient test generators that can provoke a considerable number of diverse failure-revealing test scenarios w.r.t different target failure severity (i.e., in terms of tolerance threshold), available test budget, and driving style constraints (e.g, setting speed limits). For instance, in terms of the number of triggered failures within a given test time budget and with less strict driving constraints, the $(\mu + \lambda)$ ES-driven test generator in Deeper considerably outperforms other tools while keeping the level of promoted failure diversity quite close to the counterpart tool with the highest number of detected failures in the competition. Meanwhile, as a distinctive feature, none of the newly proposed test generators leaves the experiment without triggering any failures, and in particular, they act as more reliable test generators than most of the other tools for provoking diverse failures under a limited test budget and strict constraints. With respect to the test effectiveness and efficiency, Deeper $(\mu + \lambda)$ ES-, PSO-, and GA-driven result in high effectiveness in terms of the ratio of the number of detected failures to the generated valid test scenarios.

The rest of this paper is organized as follows: Section 9.2 presents background information on bio-inspired search techniques including evolutionary and swarm intelligence techniques. Section 9.3 presents the problem formulation and the technical details of our proposed test generators in Deeper. Section 9.4 elaborates on the empirical evaluation, including the research method and experiments setup. Section 9.5 discusses the results, answers to the RQs, and the threats to the validity of the results. Section 9.6 provides an overview of the related work, and finally, Section 9.7 concludes the paper with our findings and the potential research directions for future work.

9.2 Background

Evolutionary and swarm intelligence algorithms are two main classes of random search techniques, which are widely used in many different optimization problems. Genetic algorithms (GA), genetic programming (GP),

differential evolution (DE), and evolution strategies (ES) are the main categories inside the family of evolutionary algorithms (EAs). Particle swarm optimization (PSO) is one of the primary representatives of swarm intelligence algorithms [27].

Genetic algorithms is one of the most common nature-inspired optimization techniques. It starts with a random population of individuals—each called a chromosome—representing a potential solution for the problem. The objectives to be optimized in the problem are defined in an *objective function* and the quality of the solutions is measured via this function. It shows how "well" each solution satisfies the objective. The quality of each individual, which is also referred to as "fitness", is a main factor during the evolution process. At each generation, a new population is formed based on the selected individuals from the previous generation. Three operations are involved in forming the new generation, which are as follows:

1. Selection, which mainly identifies highly-valued individuals from the previous generation.

2. Crossover, which breeds "child" individuals by exchanging parts of the "parent" individuals. The child individuals (offspring) are formed by selecting genes from each parent individual.

3. Mutation, which applies small random adjustments to the individuals.

Crossover and mutation operations are applied w.r.t user-set probabilities, and these two operations might be used, either independently or jointly, to create new individuals to form a new population. The resulting individuals are added to the new population. The fitness values are calculated and stored for each individual in this population. This process iterates each generation until stopping criteria are met, e.g., a user-set number of generations or an allowed time budget is exhausted [21, 27].

Evolution strategy is another common class of EAs. It is commonly used in almost all fields of optimization problems including discrete and continuous input spaces. ES also involves applying selection, recombination, and mutation to a population of individuals over various generations to get iteratively evolved solutions. Individuals in ES can also include a collection of evolvable strategy parameters, which are utilized for adjusting and managing the statistical features of the evolution operations. Two canonical versions of

ES are $(\mu/\rho + \lambda)$ and $(\mu/\rho, \lambda)$ evolution strategies. If $\rho = 1$, we have ES cases without recombination, which are denoted by $(\mu + \lambda)$ and (μ, λ) ESs. In the case of $\rho = 1$, the recombination is simply making a copy of the parent. In these notations, λ and μ indicate the size of the offspring and population respectively. One of the main differences between GA and ES is related to the selection step. In GA, at each iteration, the next generation is formed by selecting highly-valued individuals, while the size of the population is kept fixed [27]. In ES, a temporary population with the size of λ is created and the individuals in this temporary population undergo mostly mutation at user-set probabilities regardless of their fitness values. In $(\mu + \lambda)$ ES, then, both parents and the generated offspring resulting from the temporary population are copied to a selection pool—with size $(\mu + \lambda)$—and a new population with size μ is formed by selecting the best individuals. While, in (μ, λ) ES, the new generation with size μ is selected only from the offspring (with size λ). Therefore, a convergence condition as $\mu < \lambda$ is required to guarantee an optimal solution [28].

Particle swarm optimization is one of the most common representatives of the swarm intelligence (SI) algorithms, which form a big class of nature-inspired optimization methods alongside the evolutionary algorithms. The SI algorithms present the concept of collective intelligence, which is mainly defined as a collective behavior in a group of individuals that seem intelligent. SI algorithms have been inspired from collective behavior and self-organizing interactions between living agents in the nature, e.g., ant colonies and honey bees [29].

PSO is an optimization method simulating the collective behavior of certain types of living species. In PSO, *cooperation* is an important feature of the system as each of the individuals changes its searching pattern based on its own and others' experiences. PSO starts with a swarm of random particles. Each particle has position and velocity vectors, which are updated w.r.t the local and global best values. The best values get updated at each iteration. In the application of PSO, each particle (individual) represents a potential solution and is often modeled as a vector containing n elements, in which each element represents a variable of the problem that is being optimized. Like GA, PSO searches for the optimal solution through updating solutions and creating subsequent generations, though without using evolution operators [30]. The position (the elements) and velocity of each particle are

updated as follows:

$$P^{t+1} = P^t + V^{t+1} \tag{9.1}$$

$$V^{t+1} = wV^t + c_1 r_1 (P^t_{best} - P^t) + c_2 r_2 (G^t_{best} - P^t) \tag{9.2}$$

where P^t and V^t are the position and velocity of the particle at iteration t, respectively; P^t_{best} and G^t_{best} are the local best position of the particle and the global best one up to the iteration t. The first part of (9.2) is perceived as *inertia*, which indicates the tendency of the particle to keep moving in the same direction, while the second part—which reflects a cognitive behavior—indicates the tendency towards the local best position discovered by the particle and the last part—which is the social knowledge—reflects the tendency to follow the best position found so far by other particles.

Therefore, at each iteration, the position of each particle is updated based on its velocity, and the velocity is controlled by the inertia and accelerated stochastically towards the local and global best values. r_1 and r_2 are random weights from range $(0, 2]$ which adjust the cognitive and social acceleration. In (9.2), w is inertia weight, which adjusts the ability of the swarm to change the direction and makes a balance between the level of exploration and exploitation in the search process. A lower w leads to more exploitation of the best solutions found, while a higher value of w facilitates more exploration around the found solutions. c_1 and c_2 are the acceleration hyperparameters defining to what extent the solutions are influenced by the local best solutions and global best solution. These hyperparameters and the inertia weight could be static or changed dynamically over the iterations. For instance, $w = 0.72984$ and $c1 + c2 > 4$ is a common setup for a static configuration of these parameters [31].

9.3 Deeper: A Bio-Inspired Simulation-Integrated Testing Framework

This section presents the technical details of Deeper and shows how it challenges a DNN-based lane-keeping system in a self-driving car trained and tested in the BeamNG simulator environment [13]. The subject system is a

Figure 9.1: A test scenario executed in the BeamNG simulator.

built-in AI driving agent encompassing a steering angle predictor (ML model) which receives images captured by an onboard camera in the simulation environment. Then, the test inputs (test cases) generated by Deeper are defined as scenarios in which the car drives. Our target is to generate diverse test scenarios triggering the misbehavior of the subject system. In this regard, we benefit from bio-inspired search-based techniques to explore the input space and generate highly-valued failure-revealing test scenarios.

9.3.1 Test Scenario and Failure Specification

Test scenarios are defined as combinations of roads, the environment including e.g., the weather and illumination, and the driving path, i.e., starting and end points and the lane to keep. Hereafter, we consider scenarios involving a single asphalt road surrounded by green grass where the car is to drive on the right lane, and the environment is set to a clear day with the overhead sun (See Figure 9.1). Therefore, the focus of Deeper is to generate diverse roads which trigger failures in the system under test. In this system, failure is defined in terms of an episode, in which the car drives partially outside the lane meaning that $X\%$ of the car's bounding box gets outside the lane. X is a configurable tolerance threshold for Deeper.

In the test scenarios, the road composes two fixed-width lanes with a yellow center line and two white lines separating the lanes from the non-drivable area. In BeamNG, each road is mainly described by a set of points that are used by the simulation engine to render the road. The simulation engine accomplishes

the rendering by interpolating the points and creating a sequence of polygons on the points—provided by the SBST competition setup [2]. Notably, not every sequence of road points results in valid roads, so each sequence of points is also validated against some initial geometrical constraints related to the road polygons and some other domain-specific constraints. The main constraints are: 1) the start and end points of the road shall be different, 2) the road shall be completely contained in the map used in the simulation, 3) the road shall not self-intersect, and 4) the road shall not contain too sharp turns that force the vehicle to invade the opposite lane. To assure the satisfaction of these constraints, Deeper validates the generated roads before getting executed and consequently, the invalid roads are not counted as failed test scenarios.

Road Representation Model: In order to convert the abstract road model into a proper set of points that can be rendered by the simulation engine—as candidate solutions in the test generator—we rely on the representation model used by DeepJanus [23] based on Catmull-Rom cubic splines [24]. Therefore, each road is represented by two sets of points, *control points* and *sample points*. First, control points are provided as an input specification for a candidate road. Second, sample points are calculated using the Catmull-Rom calculation algorithm. Third, the simulation engine uses the sample points, if they are valid, to render the road. Figure 9.2 shows the representation model of a road in terms of control and sample points (9.2a and 9.2b) and the corresponding rendered road in the simulation (9.2c).

$$CP = \langle C_1, C_2, \cdots, C_m \rangle \,, \quad R_{imin} \leq C_i \leq R_{imax} \tag{9.3}$$

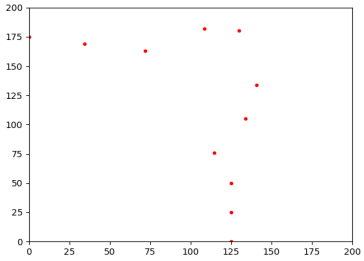
$$R_{imin},\ R_{imax} \in R \tag{9.4}$$

$$SP = \langle S_1, S_2, \cdots, S_n \rangle \,, \quad SP = Catmull\_Rom\_Spline(CP) \tag{9.5}$$

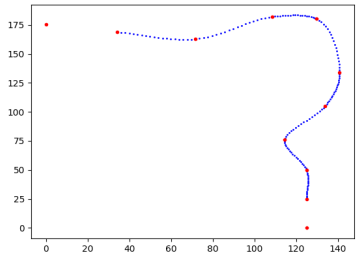
$$R_{imin} \leq S_i \leq R_{imax} \tag{9.6}$$

$$R_{imin},\ R_{imax} \in R \tag{9.7}$$

[2] https://github.com/se2p/tool-competition-av.git



(a) Control Points.



(b) Calculated Sample Points.

(c) The corresponding road in the simulation.

Figure 9.2: The representation model of a road

9.3.2 Fitness Function

As indicated above, the focus is the generation of diverse failure-revealing test scenarios w.r.t the intended tolerance threshold. The competition setup elaborates the positional data and detects the episodes in which the car breaks out of the lane bounds, i.e., out of bound episodes (OBE). It computes the distance of the car from the center of the lane in those OBE episodes and reports an OBE failure [32] each time that the car drives outside the lane if the percentage of the area of the car that is outside the lane is bigger than the intended threshold (referred to as X in Section 9.3.1).

The problem is regarded as an optimization problem, and in order to

generate valuable test scenarios leading us to meet the target, we evaluate the quality of the test scenarios using a *fitness (objective) function*. In this regard, for each test scenario, the main objective of interest to be optimized is the maximum detected distance of the car from the center of the lane during driving of the car on the lane. So, more accurately the fitness function that we want to minimize is as follows:

$$Fitness\_Function = (lane\_width)/2 - d(center\_polyline, car\_position) \tag{9.8}$$

where $lane\_width$ is the width of the lane and $d(center\_polyline, car\_position)$ indicates the distance of the car position from the central polyline (center) of the lane.

9.3.3 Bio-Inspired Search Algorithms

We are interested in sampling from the space of possible test scenarios in an effective way to generate those that lead to the emergence of failures. This can be achieved by using an optimization algorithm to guide the search by the fitness function. Therefore, in order to find the solutions of interest, we use bio-inspired search-based algorithms, i.e., GA, $(\mu + \lambda)$ and (μ, λ) ESs, and PSO, guided by the fitness function introduced in Section 9.3.2. These search algorithms are mainly modeled on the basis of population evolution over time and they usually get started with the creation of a random population of solutions. In Deeper, we leverage an initial quality population seed to boost the search process regarding the fact that the search is done at a fixed test budget. Throughout the development, the impact of different initial population seeds was investigated. For instance, starting from an initial random population seed was not quite effective to lead the search to find the failure-revealing solutions w.r.t high tolerance thresholds within a reasonable test budget. The quality population seed used in the current search algorithms of Deeper is a mix of some valid random solutions and some extracted randomly from the solutions generated by a 5-hour execution of the first version of Deeper (based on NSGA-II [33]) that start from an initial *random* population seed. For the latter part, those solutions are extracted randomly from the the generated solutions by Deeper NSGA-II which could cause OBEs w.r.t a tolerance threshold $\tau \geq 0.5$.

Genetic Algorithm

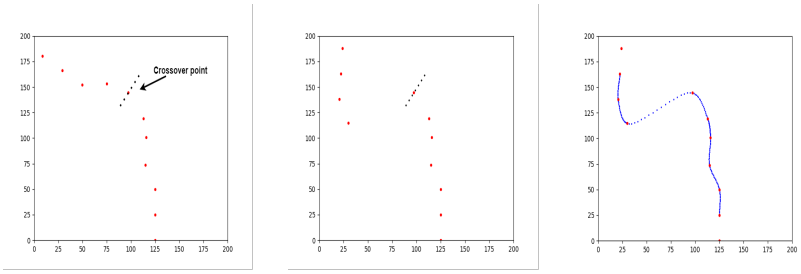
The GA-driven test generator in Deeper starts with forming an initial population by sampling from the quality population seed. Over various generations, new populations of test scenarios are formed through applying crossover and mutation operations to the best ones selected from the previous generations.

Selection: We use *tournament selection* for identifying the promising test scenarios, which have a high probability to lead to failures and safety violations. In tournament selection, a subset of the population is selected at random in each tournament and the best test scenario of the subset is picked. The number of individuals participating in each tournament indicates the size of the tournament.

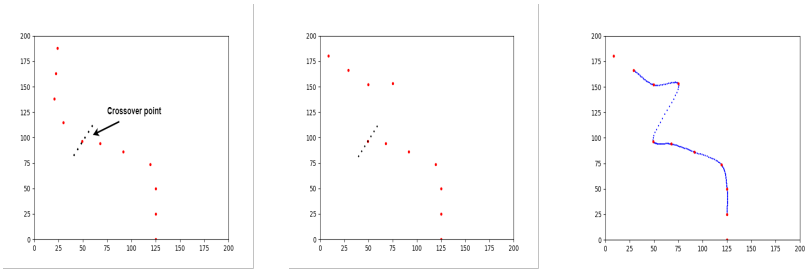
Crossover: We develop a domain-specific *one-point crossover* operation to create new test scenarios, i.e., new test roads, from the ones selected from the previous generation. The proposed crossover operation performs the segment exchange at the sets of *control points*, which means that a random point is selected as the crossover point in the sets of control points in the parent roads, then the parts of the sets beyond the crossover point are swapped between the parents, and accordingly, two new sets of control points for two child roads are formed. The corresponding sample points for the generated child roads are calculated using the Catmull-Rom calculation algorithm (See Figure 9.3).

However, still, these resulting sets of points might not represent valid roads w.r.t the geometrical constraints, so before adding these new test scenarios to the offspring, we also check their validity and let only the resulting valid roads be added to the offspring. If both generated child roads are valid, then both of them will be transferred to the offspring, while if one of them is valid, we keep the valid child and another crossover point is tried to breed the parent roads and generate the second valid child. Likewise, if none of the child roads are identified as valid roads, another crossover point is tried. All in all, in order to generate two valid child roads from a crossover operation, at most five attempts to try with different crossover points are done. If in the end, the attempts do not lead to valid child roads, the whole process will be rolled back and the original parents will be added to the offspring.

Mutation: The mutation operation also targets the coordinate values of the control points. We use *Polynomial Bounded mutation*, a bounded mutation



(a) Parent Road 1: Control Points. (b) Child Road 1: Control Points. (c) Child Road 1: Sample Points.

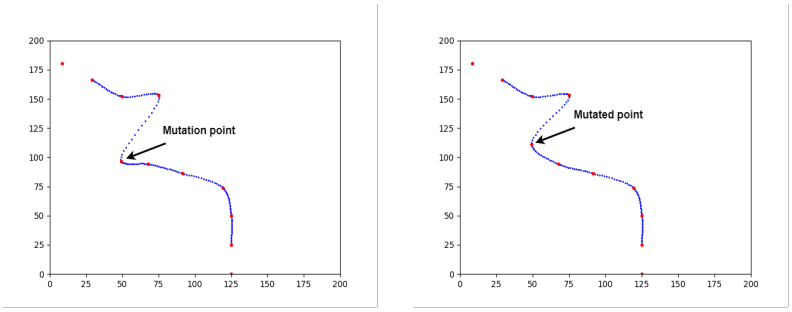


(d) Parent Road 2: Control Points. (e) Child Road 2: Control Points. (f) Child Road 2: Sample Points.

Figure 9.3: Crossover operation on two sample roads

operation for real-valued individuals which was used in NSGA-II [33]. It features using a polynomial function for the probability distribution and a user-set parameter, η, presenting the *crowding degree* of the mutation and adjusting the diversity in the resulting mutant. A high value for η results in a mutant resembling the original solution, while a low η leads to a more divergent mutant from the original. This domain-specific polynomial bounded mutation operation selects randomly a point—mutation point—in the set of control points and mutates randomly the x or y coordinate of the selected control point (See Figure 9.4). Accordingly, the sample points are re-calculated for the mutated set of control points, and their validity w.r.t the geometrical constraints are checked. In case the mutant does not represent a

valid road, another control point is tried. The GA-driven test generator of Deeper is configured as presented in Algorithm 13.



(a) Original road. (b) Mutated road.

Figure 9.4: Mutation operation on the road

Evolution Strategies

The ES-driven test generators in Deeper use two canonical $(\mu + \lambda)$ and (μ, λ) ES algorithms. The ES-driven test generators start with initializing a population with μ test scenarios sampled from the population seed. The fitness value of each test scenario is calculated through rendering them in the simulation. Next, a temporary population with size λ is formed by the reproduction of test scenarios from the original population. This temporary population is used to create offspring by applying the proposed domain-specific crossover or mutation operations (See Section 9.3.3) to its individuals according to user-set crossover and mutation probabilities. In $(\mu + \lambda)$ ES, then both parents and the resulting offspring are copied to a selection pool, and a new population with size μ is created by using tournament selection. While, in (μ, λ) ES-driven test generator, the new population is selected only from the offspring. Algorithm 14 presents the procedure of $(\mu + \lambda)$ ES-driven test generator, where $\mu = 70$ and $\lambda = 30$. The procedure of (μ, λ) ES test generator is almost the same as $(\mu + \lambda)$ ES, and the main difference is in the selection step, where the μ highly-valued individuals are selected only from the offspring in (μ, λ) ES. However, with

Algorithm 13 GA-driven test generator in Deeper

1. Initialize a population of test scenarios (with $size = 70$) from the quality population seed
2. Evaluate the test scenarios through rendering them in the simulation and computing the fitness values

repeat

3. Select highly-valued scenarios using tournament selection ($Tournament\ size = 3$)
4. Create offspring by using crossover and mutation operations
   4.1. Apply the domain-specific crossover operation ($Crossover\ rate = 0.3$)
   4.2. Apply the domain-specific polynomial Bounded mutation operation ($Mutation\ rate = 0.7$)
5. Evaluate the offspring

until *reaching the end of the test budget (e.g., given time)*;

6. Collect the test scenarios revealing OBE failures

regard to the required convergence condition ($\mu < \lambda$) in this case, then those parameters in (μ, λ) ES test generator are configured as $\mu = 70$ and $\lambda = 100$ (the size of the main population is fixed at both ES-driven test generators).

Particle Swarm Optimization

The PSO-driven test generator in Deeper starts with initializing a population of particles from the quality population seed. Each test scenario is modeled as a particle, and the set of control points represents the position vector of the particle that is updated according to Eq. (9.1) and (9.2) over various generations. After each updating, the corresponding sample points for the updated set of control points are calculated and the validity of the new set of road points is checked against the geometrical constraints. The PSO-driven test generator is configured as presented in Algorithm 15 and based on the following setting for the parameters, which is $w = 0.8$, $c1 = 2.0$, and $c2 = 2.0$.

Algorithm 14 $(\mu + \lambda)$ ES-driven test generator in Deeper

1. Initialize a population P with μ test scenarios sampled from the quality population seed
2. Evaluate the test scenarios in the population P through simulation and computing the fitness values

repeat

3. Create a temporary population P_T with size λ by reproduction of test scenarios from the population P
4. Create an offspring by applying the crossover or mutation operation to the test scenarios in the population P_T (with crossover probability $Cx_P = 0.3$ and mutation probability $Mu_P = 0.7$)
   4.1. $choice = Random.random(),\ 0 < choice < 1$
   4.2. If $choice < Cx_P$ then
      Apply the domain-specific crossover operation
   else if $choice < Cx_P + Mu_P$
      Apply the domain-specific polynomial Bounded mutation operation
5. Evaluate the offspring
6. Select μ highly-valued test scenarios using tournament selection ($Tournament\ size = 3$) from the original population P and the offspring

until *reaching the end of the test budget (e.g., given time)*;

7. Collect the test scenarios revealing OBE failures

9.4 Empirical Evaluation

We conduct an empirical evaluation of the proposed simulation-integrated bio-inspired test generators in Deeper, by running experiments on an experimental setup based on a PC with 64-bit Windows 10 Pro, Intel Core i7-8550U CPU @ 1.80GHz, 16GB RAM, Intel UHD Graphics 620, and BeamNG.tech driving simulator together with the software requirements for running Deeper[3].

Test Subject: The system under test is BeamNG's built-in driving agent, BeamNG.AI. It is an autonomous agent utilizing optimization techniques to plan the driving trajectory according to the speed limit while keeping the ego car inside the road lane. It is equipped with a DNN-based lane-keeping

[3]See requirements at https://github.com/mahshidhelali/Deeper_ADAS_Test_Generator.git

Algorithm 15 PSO-driven test generator in Deeper

1. Initialize a swarm of test scenario particles (with $size = 70$) from the quality population seed
2. Evaluate the particles of the swarm through simulation and computing the fitness values
3. Select the global best particle w.r.t the fitness value (G_{best})
repeat
  for *each test scenario particle P in the swarm* **do**
    4. Calculate the particle's velocity according to Eq. 9.2
    5. Update particle's position according to Eq. 9.1
    6. Evaluate the particle based on the fitness function
    if *fitness value of P is better than the local best of P, (P_{best}),* **then**
      7. Update P_{best} with P
    end
    if *fitness value of P is better than the global best, (G_{best}),* **then**
      8. Update G_{best} with P
    end
  end
until *reaching the end of the test budget (e.g., given time)*;
9. Collect the test scenario particles revealing OBE failures

ADAS. The DNN-based lane-keeping system learns a mapping from the input of the onboard camera in the simulated environment to the steering angle. It is based on the DAVE-2 architecture including a normalization layer, five convolutional layers followed by three fully connected layers [34]. This test subject has been used in previous research and also in the SBST 2021 cyber-physical tool competition for evaluating test scenario generators [32, 23, 35], and moreover does not require manual training, which can mitigate the threats to the validity of the results [35].

9.4.1 Research Method

We design and implement a set of experiments to answer the research questions:

1. RQ1: How capable are these test generators to trigger failures?

2. RQ2: How diverse are the generated failure-revealing test scenarios?

3. RQ3: How effectively and efficiently do the test generators perform, i.e., given a certain test budget how many test scenarios are generated, what proportion of the scenarios is valid, and what proportion of the valid test scenarios leads to triggering failures?

The experiments are simulation scenarios generated by a Python test scenario generator and executed by the simulation engine. BeamNG.AI is the autonomous driving agent controlling the ego car in the simulation (Figure 9.5). In order to provide quantitative answers to the RQs, we use the following quality criteria to assess the bio-inspired test scenario generators in Deeper:

- Detected Failures: The number of generated test scenarios that lead to failures, w.r.t the given tolerance threshold.

- Failure Diversity: The dissimilarity between the test scenarios that lead to the failures. Generating diverse failure-revealing test scenarios is of interest, since triggering the same failures multiple times results in wasting the test budget, e.g., computation resources. In order to measure the failure diversity, we rely on a two-step strategy adopted by the SBST 2021 tool competition. It extracts, first, the road segments related to the failures, then computes the *sparseness*, which is considered as the average of the maximum Levenshtein distance [36] between those road segments.
  The failure-related road segments are referred to as the parts of the road 30 meters before the OBE and 30 meters after it, and accordingly, the sparseness is calculated as follows:

$$Sparseness = \frac{\sum_{i \in OBEs} \max_{j \in OBEs} Lev\_dist(i,j)}{|OBEs|} \tag{9.9}$$

  where $Lev\_dist(i,j)$ indicates the weighted Levenshtein distance between the road segments.

- Test generation efficiency and effectiveness: It indicates how the test generator uses the given test budget to generate the test scenarios, in particular how many test scenarios are generated in total, what fraction of them are valid, and what fraction of the valid ones triggers failures.

Figure 9.5: An overview of the experimental setup

Experiments: We design two sets of experiments as implemented in the SBST 2021 CPS testing tool competition. In order to provide a comparative analysis, we compare the results of the proposed test generators in Deeper with the presented test generators in the tool competition, i.e., Frenetic [25], GABExploit and GABExplore [22], Swat [26], and also the earlier version of Deeper [20]. We run the test generators on the test subject based on the same two experiment configurations as in the competition, which are shown in table 9.1. The *SET1* of experiments provides a 5-hour test generation budget, meanwhile sets the failing tolerance threshold to a high value, 0.95, and does not consider any speed limit. This experiment configuration might lead to a more careless style of driving. The *SET2* of the experiments allocates a shorter time budget for the test generation and considers a lower tolerance threshold, 0.85, while imposing a speed limit of 70 km/h—promoting a more careful driving style. To ensure a fair comparison, we run each tool the same number of times on the same dedicated machine. We run each test generator 5 times in experiment configuration SET1 and 10 times in SET2 and report distributions of the results.

Table 9.1: Experiment configurations

Name	**Test Budget (h)**	**Map Size (m^2)**	**Speed Limit (Km/h)**	**Failing Tolerance Threshold** (%)
SET1 (careless driving)	5h	200×200	None	0.95
SET2 (cautious driving)	2h	200×200	70 Km/h	0.85

9.5 Results and Discussion

This sections reports results corresponding to the three RQs.

9.5.1 Detected Failures (RQ1)

Figure 9.6 reports the number of triggered failures by each of the tools in experiment configuration SET1. In this regard, $(\mu + \lambda)$ ES-driven test generator in Deeper could successfully trigger at least 2X more OBEs than the highest record—held by GABExploit—in the competition. At the same time, Deeper $(\mu + \lambda)$ ES showed more consistent performance over the runs, i.e., with lower standard deviation, compared to GABExploit, which revealed highly different behavior across the runs (e.g., returning over 100 OBEs in some runs, but failed to trigger any failure in other runs [35]). PSO- and GA-driven test generators in Deeper, in half of the cases and also on average, triggered higher numbers of failures than the competition's test generators—except for GABExploit, which showed comparable results. Moreover, the PSO- and GA-driven approaches were able to trigger the failures in which the ego car invades the opposite lane of the road—a type of failure that has been typically considered difficult to trigger in most of the test generator tools [35]. Lastly, in SET1, Deeper (μ, λ) ES clearly gave a better performance, in terms of the number of detected failures, than Deeper NSGA-II, GABExplore, and Swat.

Similarly, Figure 9.7 presents the number of triggered failures in experiment configuration SET2. With regard to the limited time budget, the

speed limit of 70 km/h and the tolerance threshold 0.85, in contrast to Deeper NSGA-II, GABExplore, GABExploit, and Swat, none of the newly proposed test generators in Deeper left an experiment without triggering any failure or with a very low number of failures (i.e., less than 3), which means they are able to detect failures even within a limited test budget and strict constraints such as setting a speed limit (See Table 9.2). It is noted that Deeper NSGA-II, GABExplore, and Swat triggered just equal or less than 1 failure in a considerable number of experiments done based on configuration SET2. Therefore, all the new test generators in Deeper outperform Deeper NSGA-II, GABExplore, and Swat w.r.t the number of triggered failures within a limited test time and moreover the PSO-driven test generator results in a very comparable number of detected failures to Frenetic. Additionally, in the experiment configuration SET2, the GA-driven test generator is still able to trigger the failures showing the invasion of the car to the opposite lane of the road.

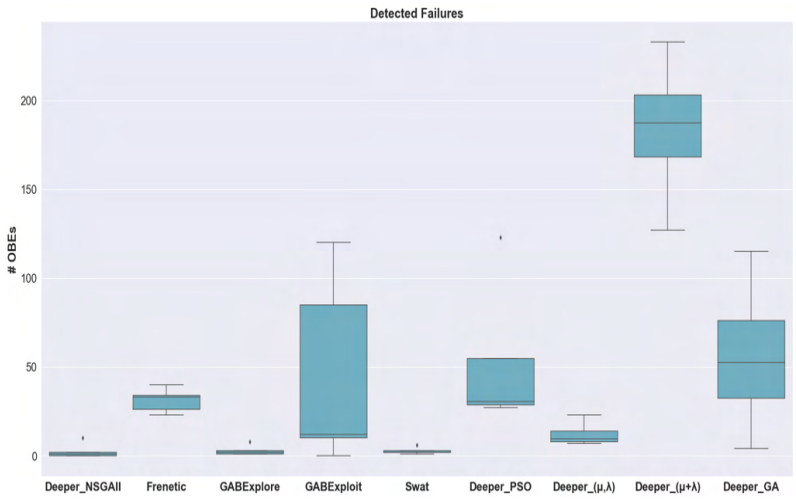


Figure 9.6: Number of detected failures in SET1

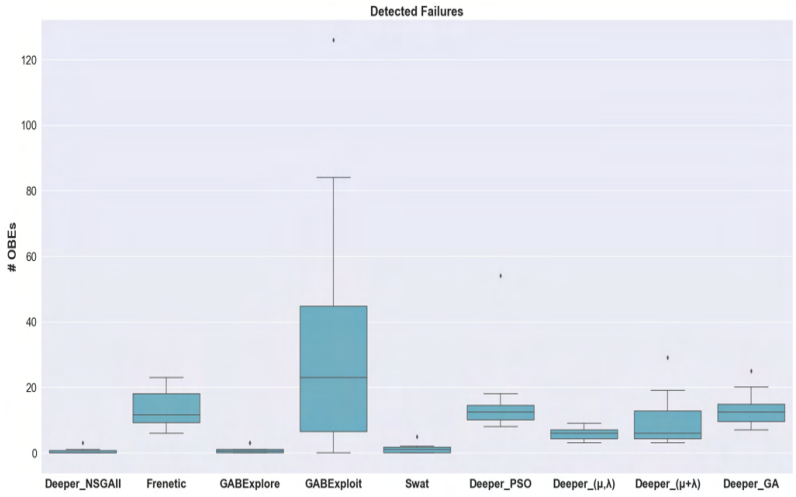


Figure 9.7: Number of detected failures in SET2

9.5.2 Diversity of Failures (RQ2)

Figure 9.8 depicts the diversity of the detected failures in SET1 in terms of the distribution of the failures' sparseness. All the new test generators in Deeper resulted in a considerable improvement on failure sparseness compared to the first version, Deeper NSGA-II in the competition. Meanwhile, among the new test generators of Deeper, the PSO- and GA-driven test generators lead to a higher average sparseness compared to the $(\mu + \lambda)$ and (μ, λ) ES approaches. Deeper PSO, $(\mu + \lambda)$, and GA also show a higher sparseness than GABExploit—both on average and in almost half of the cases. However, still the failure diversity—in terms of the special sparseness metric defined by the competition—promoted by the new test generators in Deeper are not as high as Frenetic, GABExplore, and Swat. Note that GABExplore and Swat in 20% of the experiments did not report any sparseness figure, since they just triggered one failure in each of those experiments.

Figure 9.9 shows the distribution of the failures' sparseness in SET2. In the limited test budget and strict driving constraints, all the newly test generators again show a big improvement on promoting failure sparseness in comparison to the first version, Deeper NSGA-II. In the meantime, Deeper PSO, $(\mu + \lambda)$,

Table 9.2: Triggered failures in SET2

Test Generator	Exp.1	Exp.2	Exp.3	Exp.4	Exp.5	Exp.6	Exp.7	Exp.8	Exp.9	Exp. 10
Deeper NSGA-II	0	0	0	1	1	0	0	0	3	0
Frenetic	12	8	9	11	19	10	15	20	6	23
GABExplore	0	1	1	0	1	3	0	0	3	0
GABExploit	84	18	126	38	0	11	47	0	28	5
Swat	0	0	0	0	5	1	1	2	1	2
Deeper PSO	54	8	13	13	10	9	10	15	18	12
Deeper (μ, λ) ES	8	4	3	5	7	7	3	5	7	9
Deeper $(\mu + \lambda)$ ES	29	5	9	4	4	7	19	3	14	5
Deeper GA	11	7	11	15	14	9	25	9	14	20

and GA promote comparable levels of failure diversity, though more consistent, compared to GABExploit. At the same time, GABExplore and Swat in around 70% of the experiments in SET2 did not provide any sparseness figure, since they triggered just one or zero failure (See Table 9.2).

9.5.3 Test Effectiveness and Efficiency (RQ3)

In SBST competition, test effectiveness and efficiency were indicated by how many test scenarios are generated and what proportion of the scenarios is valid, given a certain test budget. They basically show how well the test generator is able to utilize the test budget. Figures 9.10 and 9.11 report the average number of total test scenarios, as well as the number of valid and invalid scenarios generated by each tool in SET1 and SET2 respectively. Generally, the new test generators in Deeper utilize the test budget more efficiently than the competition tools and generate a higher number of test scenarios within the given test time. In this regard, Deeper PSO results in the

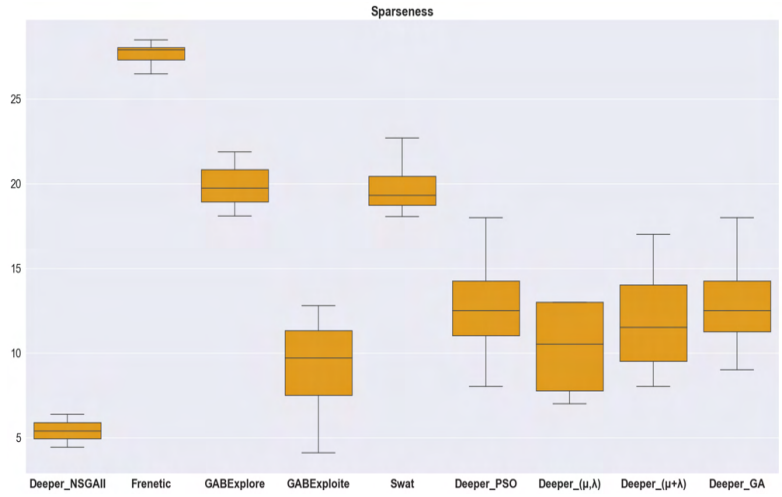


Figure 9.8: Failure diversity in terms of sparseness in SET1

highest efficiency (e.g., generates more than 650 scenarios on average within 5 hours) among all the test generators in both experimental configurations. Regarding the number of valid test scenarios, all the Deeper test generators along with Swat lead to an almost comparable number of valid scenarios. However, with respect to the ratio of the valid test scenarios to the total generated ones—called test effectiveness according to the competition evaluation—Swat, Deeper NSGA-II, and GABExploit are the ones showing the highest result.

To answer RQ3 on test effectiveness and efficiency, in addition to the metrics defined and used by the competition, we defined an extra metric, aggregated test effectiveness called *effectiveness plus*, which indicates what proportion of the valid test scenarios leads to triggering failures. It is defined as the ratio of the triggered failures to the number of valid test scenarios and is intended to present the effectiveness of the test generators w.r.t meeting the target—detecting failures. Figures 9.12 and 9.13 report the test effectiveness plus for the test generators in SET1 and SET2 respectively. With respect to the effectiveness plus, Deeper $(\mu + \lambda)$ ES-driven test generator results in the highest target-based effectiveness in SET1 and then Deeper PSO and GA are

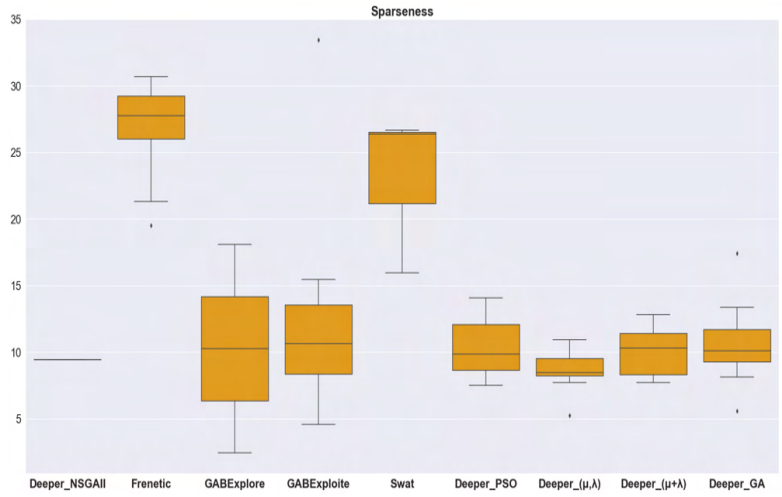


Figure 9.9: Failure diversity in terms of sparseness in SET2

the next most effective tools. In SET2, GABExploit shows the highest effectiveness plus, while Frenetic, Deeper PSO and GA are the next most effective ones. It is worth noting that as shown in Figures 9.12 and 9.13 both Deeper PSO- and GA-driven test generators keep their effectiveness in generating failure-revealing scenarios in both experimental conditions of SET1 and SET2, which means they are effective test generators even within a limited test budget and strict driving constraints.

9.5.4 Threats to Validity

The evaluation of Deeper comes with a set of threats to construct, internal, and external validity of the results.

Construct validity: The choices of the fitness function—the distance of the car position from the center of the lane—and also the metrics used for calculating the sparseness and indicating the diversity of the test scenarios—weighted Levenshtein distance—in this study are domain-specific. However, we have based our choices on the sound metrics adopted by other research works in the literature [35, 32, 23].

Internal validity: The randomized nature of the used bio-inspired algorithms

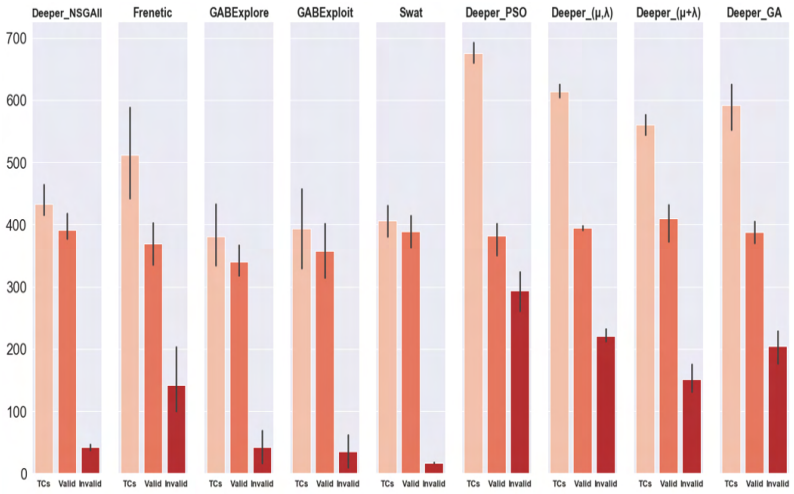


Figure 9.10: Test generation effectiveness and efficiency in SET1

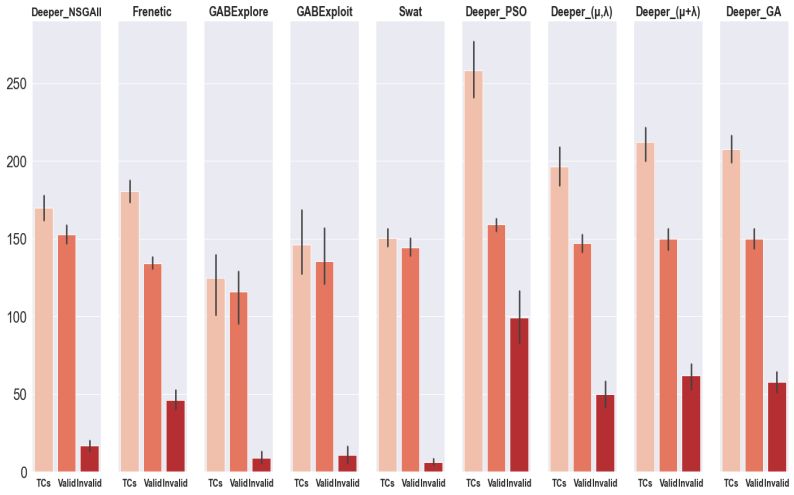


Figure 9.11: Test generation effectiveness and efficiency in SET2

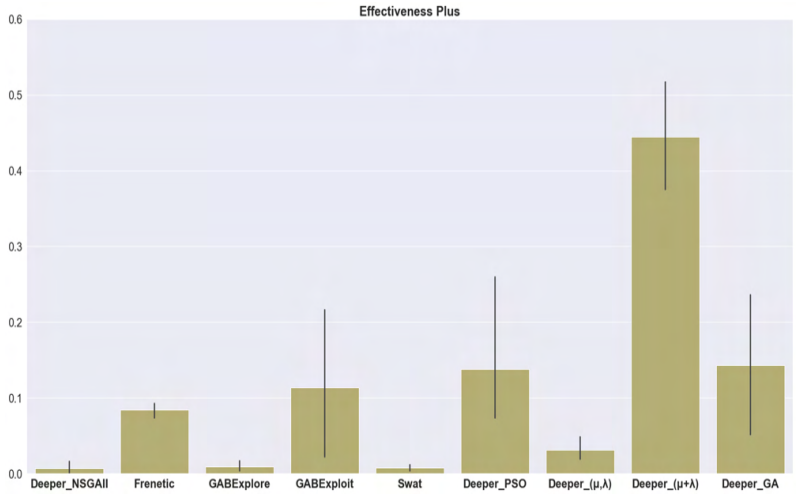


Figure 9.12: Test generation effectiveness plus in SET1

could be a source of threats to the internal validity of the results. In this regard, we follow the guidelines given by Arcuri and Briand [37] for the evaluation and analysis of the results, and mitigate this threat by running the experiments multiple times (e.g., 5 times in SET1 and 10 times in SET2), reporting the distribution and statistics of the results (e.g., using Box plots to show the results), using same algorithm settings, e.g., population size, crossover and mutation probabilities, in the proposed test generators.

External validity: The choice of the test subject system is a potential threat to the external validity of the results. However, the test system in this study is one of the main and commonly used systems in self-driving cars, and furthermore different ML models with various quality levels (i.e., different accuracy levels) could be deployed within the BeamNG.AI agent, and in this regard, the proposed test scenario generation techniques can still be used. Nonetheless, it still offers one type of deep learning(DL)-based systems in self-driving cars, and further studies are required to address the testing of other DL-driven systems, meanwhile, we also keep the tool open for extensions, for example, to support the execution of test scenarios in other simulators.

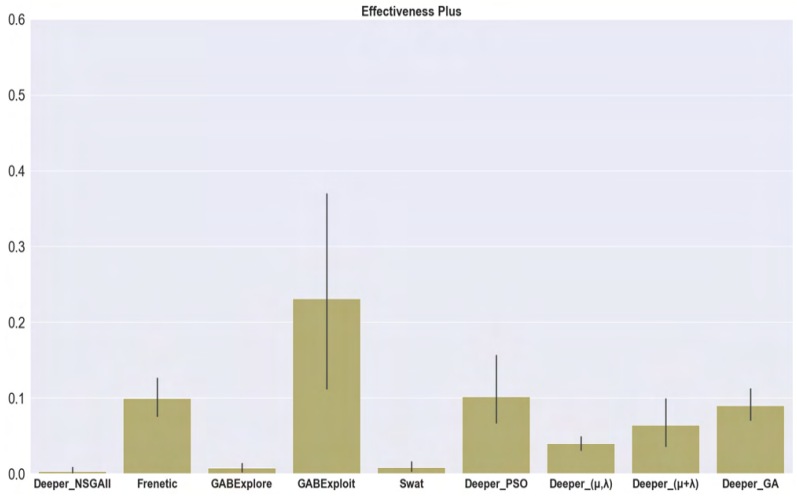


Figure 9.13: Test generation effectiveness plus in SET2

9.6 Related Work

Input data assurance alongside the model and integration testing [8, 38] are considered different test levels investigated by various research works for ML systems. Model testing could be regarded as unit testing for ML components and integration testing focuses on the issues emerging after the integration of the ML model into the system. Regarding access to the test subject, there are black-box and white-box testing approaches analogous to traditional non-ML systems. Black-box testing involves access only to the ML inputs and outputs, while the white-box testing implies access to the internal architecture of the ML test subject, code, hyperparameters, and training/test data. However, Riccio et al. in [38] also introduced another type of ML testing called data-box, which requires access to data plus everything that a black-box test requires.

Test input data that can reveal failures in the test subject is the most common generated test artifact in the literature related to testing of automotive AI systems [38]. Depending on the test level and the test subject, the inputs could be images, for instance, as used in DeepTest [39], or test scenario configurations as used in [21]. A brief overview of the most common techniques used to generate the test data is as follows:

Input data mutation. This type of mutation involves generating new inputs based on the transformation of the existing input data. For instance, DeepXplore [40] uses such input transformations to find the inputs triggering different behaviors between similar autonomous driving DNN models, while also striving to increase the level of neuron coverage. Moreover, in many studies, those transformations are based on metamorphic relations. DeepTest [39] applies different transformations to a set of seed images with the aim of increasing neuron coverage and uses metamorphic relations to find the erroneous behaviors of different Udacity DNN models for self-driving cars. DeepRoad [41] uses a GAN-based metamorphic testing technique to generate input images to test three autonomous driving Udacity DNN models. It defines the metamorphic relations such that the driving behavior in a new synthesized driving scene is expected to be consistent with the one in the corresponding original scene.

Test scenario manipulation. Another major category of the methods to generate test input data is based on the manipulation and augmentation of the test scenarios. Most of the works in this category use search-based techniques to go through the search space of the scenarios to find the failure-revealing or collision-provoking test scenarios. In this regard, simulators as a form of digital twins have played a key role to generate and capture those critical failure-revealing test scenarios. Simulation-based testing can act as an effective complementary solution to field testing, since exhaustive field testing is expensive, meanwhile inefficient, and even dangerous, in some cases. Recently, various high-fidelity simulators such as the ones using physics-based models (e.g., SVL simulator [42], Pro-SiVIC [43], and PreScan [44]) and the ones based on game engines (e.g., BeamNG.tech [13] and CARLA [14]) have considerably contributed to this area by providing the possibility of realistic simulations of functionalities in autonomous driving.

Accordingly, various testing approaches relying on the simulators have been presented in the literature and in this regard, search-based techniques have been frequently used to address the generation of failure-revealing test scenarios. Abdessalem et al. utilize multi-objective search algorithms such as NSGA-II [45] along with surrogate models to find critical test scenarios with fewer simulations and then at less computation time for a pedestrian detection system. In a following study [19], they use MOSA [46]—a many-objective optimization search algorithm— along with objectives based on the branch

coverage and some failure-based heuristics to detect undesired and failure-revealing feature interaction scenarios for integration testing in a self-driving car. Further, in another study [47], they leverage a combination of multi-objective optimization algorithms (NSGA-II) and decision tree classifier models—referred to as a learnable evolutionary algorithm—to guide the search-based process of generating critical test scenarios and also to refine a classification model that can characterize the failure-prone regions of the test input space for a pedestrian detection and emergency braking system. Haq et al. [48] use many-objective search algorithms to generate critical test data resulting in severe mispredictions for a facial key-points detection system in the automotive domain. Ebadi et al. [21] benefit from GA along with a flexible data structure to model the test scenarios and a safety-based heuristic for defining the objective function to test the pedestrian detection and emergency braking system of the Baidu Apollo (an autonomous driving platform) within the SVL simulator.

Regarding the impacts of the simulators in this area, Haq et al. [49] provide a comparison between the results of testing DNN-based ADAS using online and offline testing. Their results clearly motivate an increased focus on online testing as it can identify faults that never would be detected in offline settings—whereas the opposite does not appear to be likely. Our current study responds to this call, and motivates our work on systems testing in simulated environments. With regard to a different perspective, Borg et al. [50] discuss the consistency between the test results obtained from running the same experiments based on two different simulators and investigate the reproducibility of the results in both simulators. When running the same testing campaign in PreScan and ESI Pro-SiVIC, the authors found notable differences in the test outputs related to revealed safety violations and the dynamics of cars and pedestrians.

9.7 Conclusion

Deeper in its extended version utilizes a set of bio-inspired algorithms, genetic algorithm (GA), $(\mu + \lambda)$ and (μ, λ) evolution strategies (ES), and particle swarm optimization (PSO), to generate failure-revealing test scenarios for testing a DL-based lane-keeping system. The test subject is an AI agent in BeamNG.tech's driving simulator. The extended Deeper contains four new bio-inspired test generators that leverage a quality seed population and

domain-specific cross-over and mutation operations tailored for the presentation model used for modeling the test scenarios. Failures are defined as episodes where the ego car drives partially out of the lane w.r.t a certain tolerance threshold. In our empirical evaluation we focused to answer three main questions: first how many failures the test generators can detect, second how much diversity they can promote in the failure-revealing test scenarios, and third how effectively and efficiently they can perform, w.r.t different target failure severity (i.e., tolerance threshold), available test budget, and driving style constraints (e.g., speed limits). Our results show that the newly proposed test generators in Deeper present a considerable improvement on the previous version and they are able to act as effective and efficient test generators that provoke a considerable number of diverse failure-revealing test scenarios for testing an ML-driven lane-keeping system. They show considerable effectiveness in meeting the target, i.e., detecting diverse failures, with respect to different target failures intended and constraints imposed. In particular, they act as more reliable test generators than most of the counterpart tools for provoking diverse failures within a limited test budget and with respect to strict constraints.

As some directions for future work, we plan to apply the proposed approaches to testing further types of ML-based lane-keeping systems, i.e., more industrial ones and also in other state-of-the-art simulation platforms. We also plan to extend the approaches by applying machine learning-based techniques such as reinforcement learning or Generative Adversarial Networks (GANs) for empowering the discovery of failure-revealing test scenarios.

Acknowledgment

This work has been funded by Vinnova through the ITEA3 European IVVES (https://itea3.org/project/ivves.html) and H2020-ECSEL European AIDOaRT (https://www.aidoart.eu/) projects. Furthermore, the project received partially financial support from the SMILE III project financed by Vinnova, FFI, Fordonsstrategisk forskning och innovation under the grant number: 2019-05871.

Bibliography

[1] Markus Borg, Cristofer Englund, Krzysztof Wnuk, Boris Duran, Christoffer Levandowski, Shenjian Gao, Yanwen Tan, Henrik Kaijser, Henrik Lönn, and Jonas Törnqvist. Safely entering the deep: A review of verification and validation for machine learning and a challenge elicitation in the automotive industry. *Journal of Automotive Software Engineering*, 1(1):1–19, 2019.

[2] Markus Borg, Joshua Bronson, Linus Christensson, Fredrik Olsson, Olof Lennartsson, Elias Sonnsjö, Hamid Ebabi, and Martin Karsberg. Exploring the assessment list for trustworthy ai in the context of advanced driver-assistance systems. *arXiv preprint arXiv:2103.09051*, 2021.

[3] Eric J Topol. High-performance medicine: the convergence of human and artificial intelligence. *Nature medicine*, 25(1):44–56, 2019.

[4] Jay Lee, Hossein Davari, Jaskaran Singh, and Vibhor Pandhare. Industrial artificial intelligence for industry 4.0-based manufacturing systems. *Manufacturing letters*, 18:20–23, 2018.

[5] ML Cummings and David Britton. Regulating safety-critical autonomous systems: past, present, and future perspectives. In *Living with robots*, pages 119–140. Elsevier, 2020.

[6] Simon Burton, Ibrahim Habli, Tom Lawton, John McDermid, Phillip Morgan, and Zoe Porter. Mind the gaps: Assuring the safety of autonomous systems from an engineering, ethical, and legal perspective. *Artificial Intelligence*, 279:103201, 2020.

[7] Ethics guidelines for trustworthy AI. European Commission, Available at https://digital-strategy.ec.europa.eu/en/library/ethics-guidelines-trustworthy-ai, Retrieved August, 2021.

[8] Richard Hawkins, Colin Paterson, Chiara Picardi, Yan Jia, Radu Calinescu, and Ibrahim Habli. Guidance on the assurance of machine learning in autonomous systems (AMLAS). *arXiv preprint arXiv:2102.01564*, 2021.

[9] Andrej Karpathy. Software 2.0. Available at https://karpathy.medium.com/software-2-0-a64152b37c35, Retrieved August, 2021.

[10] Road Vehicles - Safety of the Intended Functionality. International Organization for Standardization, Tech. Rep. ISO/PAS 21448:2019, 2019.

[11] Road vehicles — Functional safety. International Organization for Standardization, Tech. Rep. ISO 26262-1:2018, 2018.

[12] Markus Borg, Raja Ben Abdessalem, Shiva Nejati, Francois-Xavier Jegeden, and Donghwan Shin. Digital Twins Are Not Monozygotic–Cross-Replicating ADAS Testing in Two Industry-Grade Automotive Simulators. *arXiv preprint arXiv:2012.06822, accepted in IEEE International Conference on Software Testing, Verification and Validation*, 2021.

[13] BeamNG GmbH. BeamNG.research. https://beamng.gmbh/research/, Retrieved March, 2021.

[14] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. CARLA: An open urban driving simulator. In *Conference on robot learning*, pages 1–16. PMLR, 2017.

[15] Guodong Rong, Byung Hyun Shin, Hadi Tabatabaee, Qiang Lu, Steve Lemke, Mārtiņš Možeiko, Eric Boise, Geehoon Uhm, Mark Gerow, Shalin Mehta, et al. Lgsvl simulator: A high fidelity simulator for autonomous driving. In *2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC)*, pages 1–6. IEEE, 2020.

[16] Francisca Rosique, Pedro J Navarro, Carlos Fernández, and Antonio Padilla. A systematic review of perception system and simulators for autonomous vehicles research. *Sensors*, 19(3):648, 2019.

[17] Raja Ben Abdessalem, Shiva Nejati, Lionel C Briand, and Thomas Stifter. Testing vision-based control systems using learnable evolutionary algorithms. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 1016–1026. IEEE, 2018.

[18] Alessio Gambi, Marc Mueller, and Gordon Fraser. Automatically testing self-driving cars with search-based procedural content generation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 318–328. ACM, 2019.

[19] Raja Ben Abdessalem, Annibale Panichella, Shiva Nejati, Lionel C Briand, and Thomas Stifter. Testing autonomous cars for feature interaction failures using many-objective search. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 143–154. IEEE, 2018.

[20] Mahshid Helali Moghadam, Markus Borg, and Seyed Jalaleddin Mousavirad. Deeper at the SBST 2021 tool competition: ADAS testing using multi-objective search. In *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*, pages 40–41. IEEE, 2021.

[21] Hamid Ebadi, Mahshid Helali Moghadam, Markus Borg, Gregory Gay, Afonso Fontes, and Kasper Socha. Efficient and effective generation of test cases for pedestrian detection-search-based software testing of Baidu Apollo in SVL. In *2021 IEEE International Conference on Artificial Intelligence Testing (AITest)*, pages 103–110. IEEE, 2021.

[22] Florian Klück, Lorenz Klampfl, and Franz Wotawa. GABezier at the SBST 2021 tool competition. In *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*, pages 38–39. IEEE, 2021.

[23] Vincenzo Riccio and Paolo Tonella. Model-based exploration of the frontier of behaviours for deep learning system testing. In *Proceedings*

of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pages 876–888. ACM, 2020.

[24] Edwin Catmull and Raphael Rom. A class of local interpolating splines. In *Computer aided geometric design*, pages 317–326. Elsevier, 1974.

[25] Ezequiel Castellano, Ahmet Cetinkaya, Cédric Ho Thanh, Stefan Klikovits, Xiaoyi Zhang, and Paolo Arcaini. Frenetic at the SBST 2021 tool competition. In *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*, pages 36–37. IEEE, 2021.

[26] Dmytro Humeniuk, Giuliano Antoniol, and Foutse Khomh. SWAT tool at the SBST 2021 tool competition. In *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*, pages 42–43. IEEE, 2021.

[27] Adam Slowik and Halina Kwasnicka. Evolutionary algorithms and their applications to engineering problems. *Neural Computing and Applications*, pages 1–17, 2020.

[28] Hans-Georg Beyer and Hans-Paul Schwefel. Evolution strategies–a comprehensive introduction. *Natural computing*, 1(1):3–52, 2002.

[29] Michalis Mavrovouniotis, Changhe Li, and Shengxiang Yang. A survey of swarm intelligence for dynamic optimization: Algorithms and applications. *Swarm and Evolutionary Computation*, 33:1–17, 2017.

[30] Adam Slowik and Halina Kwasnicka. Nature inspired methods and their industry applications—swarm intelligence algorithms. *IEEE Transactions on Industrial Informatics*, 14(3):1004–1015, 2017.

[31] Maurice Clerc and James Kennedy. The particle swarm-explosion, stability, and convergence in a multidimensional complex space. *IEEE transactions on Evolutionary Computation*, 6(1):58–73, 2002.

[32] Alessio Gambi, Marc Müller, and Gordon Fraser. Asfault: Testing self-driving car software using search-based procedural content generation. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 27–30. IEEE, 2019.

[33] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.

[34] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

[35] Sebastiano Panichella, Alessio Gambi, Fiorella Zampetti, and Vincenzo Riccio. Sbst tool competition 2021. In *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*, pages 20–27. IEEE, 2021.

[36] Vladimir I Levenshtein et al. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710. Soviet Union, 1966.

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

[38] Vincenzo Riccio, Gunel Jahangirova, Andrea Stocco, Nargiz Humbatova, Michael Weiss, and Paolo Tonella. Testing machine learning based systems: a systematic mapping. *Empirical Software Engineering*, 25(6):5193–5254, 2020.

[39] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th international conference on software engineering*, pages 303–314. IEEE/ACM, 2018.

[40] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *proceedings of the 26th Symposium on Operating Systems Principles*, pages 1–18. ACM, 2017.

[41] Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. Deeproad: Gan-based metamorphic testing and

input validation framework for autonomous driving systems. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 132–142. IEEE, 2018.

[42] LG Electronics. SVL Simulator. https://www.svlsimulator.com/, Retrieved July, 2021.

[43] Assia Belbachir, Jean-Christophe Smal, Jean-Marc Blosseville, and Dominique Gruyer. Simulation-driven validation of advanced driving-assistance systems. *Procedia-Social and Behavioral Sciences*, 48:1205–1214, 2012.

[44] TASS International. . PreScan Simulator. https://tass.plm.automation.siemens.com/prescan-overview, Retrieved July, 2021.

[45] Raja Ben Abdessalem, Shiva Nejati, Lionel C Briand, and Thomas Stifter. Testing advanced driver assistance systems using multi-objective search and neural networks. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 63–74. IEEE/ACM, 2016.

[46] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Reformulating branch coverage as a many-objective optimization problem. In *2015 IEEE 8th international conference on software testing, verification and validation (ICST)*, pages 1–10. IEEE, 2015.

[47] Raja Ben Abdessalem, Shiva Nejati, Lionel C Briand, and Thomas Stifter. Testing vision-based control systems using learnable evolutionary algorithms. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 1016–1026. IEEE, 2018.

[48] Fitash Ul Haq, Donghwan Shin, Lionel C Briand, Thomas Stifter, and Jun Wang. Automatic test suite generation for key-points detection dnns using many-objective search. *arXiv preprint arXiv:2012.06511*, 2020.

[49] Fitash Ul Haq, Donghwan Shin, and Lionel Briand. Efficient online testing for dnn-enabled systems using surrogate-assisted and many-objective optimization. In *Proceedings of the 44th International Conference on Software Engineering (ICSE'22)*. ACM, 2022.

[50] Markus Borg, Raja Ben Abdessalem, Shiva Nejati, François-Xavier Jegeden, and Donghwan Shin. Digital twins are not monozygotic–cross-replicating adas testing in two industry-grade automotive simulators. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 383–393. IEEE, 2021.

Chapter 10

Paper E: Efficient and Effective Generation of Test Cases for Pedestrian Detection – Search-based Software Testing of Baidu Apollo in SVL

Hamid Ebadi, Mahshid Helali Moghadam, Markus Borg, Gregory Gay, Afonso Fontes, and Kasper Socha
In the Proceedings of IEEE International Conference on Artificial Intelligence Testing, IEEE 2021.

Abstract

With the growing capabilities of autonomous vehicles, there is a higher demand for sophisticated and pragmatic quality assurance approaches for machine learning-enabled systems in the automotive AI context. The use of simulation-based prototyping platforms provides the possibility for early-stage testing, enabling inexpensive testing and the ability to capture critical corner-case test scenarios. Simulation-based testing properly complements conventional on-road testing. However, due to the large space of test input parameters in these systems, the efficient generation of effective test scenarios leading to the unveiling of failures is a challenge.

This paper presents a study on testing pedestrian detection and emergency braking system of the Baidu Apollo autonomous driving platform within the SVL simulator. We propose an evolutionary automated test generation technique that generates failure-revealing scenarios for Apollo in the SVL environment. Our approach models the input space using a generic and flexible data structure and benefits a multi-criteria safety-based heuristic for the objective function targeted for optimization. This paper presents the results of our proposed test generation technique in the 2021 IEEE Autonomous Driving AI Test Challenge. In order to demonstrate the efficiency and effectiveness of our approach, we also report the results from a baseline random generation technique. Our evaluation shows that the proposed evolutionary test case generator is more effective at generating failure-revealing test cases and provides higher diversity between the generated failures than the random baseline.

10.1 Introduction

The capabilities of autonomous vehicles have increased remarkably in recent years. A self-driving car is arguably the most tangible example of what the European Commission (EC) defines as an Artificial Intelligence (AI) system [1]. From an AI perspective, the automotive industry has successfully harnessed the disruptive potential of machine learning over the last decade. Driven by the availability of big data and computing power, deep neural networks (DNNs) have enabled new levels of vehicular perception. However, performing effective quality assurance of systems that rely on DNNs requires a paradigm shift [2]. No longer do human engineers explicitly express all logic of the system in source code. Instead, DNNs are trained using enormous quantities of manually annotated data and perform actions probabilistically based on patterns observed in that data. The research community has put substantial effort into making DNN-based systems trustworthy in the automotive AI context, spurring major R&D projects and global safety standardization efforts.

The concept of Trustworthy AI receives particular attention in the EC's AI Strategy [3]. EC defines AI systems as "software (and possibly also hardware) systems designed by humans that, given a complex goal, act in the physical or digital dimension by perceiving their environment through data acquisition, interpreting the collected structured or unstructured data, reasoning on the knowledge, or processing the information, derived from this data and deciding the best action(s) to take to achieve the given goal" [1]. Novel ways to test AI systems, including autonomous vehicles, are urgently needed—and the research community has taken up the challenge [4, 5].

The use of virtual prototyping platforms for automotive software engineering has rapidly grown in recent years [6]. The use of virtual methods allows testing and validation at early development stages, which leads to fewer development cycles and faster time-to-market. Simulation-based testing is required to complement conventional on-road testing due to severe drawbacks in the use of on-road testing [7], i.e., system testing on public roads is costly and does not scale to the quantity of scenarios needed—in addition, it can be dangerous to provoke a critical situation on the road. Testing autonomous vehicles in simulators is fundamental to quality assurance in the automotive sector—as indicated in the evolving standard ISO 21448

Safety of the Intended Functionality [8].

Efficient and effective testing in simulated environments requires sophisticated approaches to automatically generating test cases. Several authors have demonstrated that search-based software test generation (SBST) [9] is a feasible approach to generate critical test scenarios in the automotive context [10, 11, 12, 13, 14], i.e., test scenarios that lead to the violation of safety requirements. SBST formulates test input selection as a search problem, where optimization algorithms attempt to systematically identify the test input that meet goals of interest. Given a scoring function denoting *closeness to the attainment of those goals*—called *objective function*—optimization algorithms can sample from a large and complex set of test inputs as guided by a chosen sampling strategy (a *metaheuristic*—in our case, a genetic algorithm) [9].

In the 2021 IEEE Autonomous Driving AI Test Challenge competition, our contribution, `ScenarioGenerator`, uses SBST to generate test scenarios that cause the Baidu Apollo's autonomous driving platform to fail. While different scenarios can be tested using `ScenarioGenerator`, for the purpose of this work, we assume a scenario with a pedestrian crossing a street with the following high-level safety goal: "The ego car shall not crash into pedestrians on collision course." We refer to any crashes between an ego car and pedestrians as a safety violation or failure.

Our work relies on a test strategy involving the following steps of simulation-based automotive testing using SBST. We:

1. Build a scene in the virtual environment.

2. Define the parameters involved in creating a varied set of test cases.

3. Define ranges for each parameter, representing the test input space to explore.

4. Define an objective function that measures the *quality* of a generated test case, in terms of its potential to demonstrate a safety violation. In our case, lower scores indicate more dangerous scenarios.

5. Apply a genetic algorithm to generate test cases that minimize the objective function, leading to safety-critical scenarios.

To accomplish this, we first import a pre-existing map into the SVL Visual Scenario Editor and create an initial movement path for a pedestrian using fixed waypoints—a set of coordinates (points) showing the initial path of the pedestrian's movements. Then, during the simulation, in the designed scene, the ego car moves forward toward a target and a pedestrian crosses the road from the right.

The proposed evolutionary test case generation formulates the search space using a generic *noise vector* data structure and minimizes a multi-criteria objective function that combines (1) distances between the ego car and other road agents, (2) the distance of the journey taken by the ego car towards the target, and (3), the number of accidents detected. Using the noise vector, as a generic and flexible structure for representing the search space of the problem, facilitates the use of a wide variety of search algorithms. This paper presents the results of our proposed test case generation technique in the 2021 IEEE Autonomous Driving AI Test Challenge. To provide the comparative results and demonstrate the efficiency and effectiveness of our evolutionary text case solution, we also compare our results to random generation of test scenarios.

The rest of the paper is organized as follows: Section 10.2 presents the details of the proposed search-based test case generation approach. Section 10.3 elaborates on the empirical evaluation, including the research method, test scenario execution, and experiment setup. Section 10.4 discusses results and threats to the validity of the results. Section 10.5 presents an overview of related work, and Section 10.6 summarizes our findings in light of the importance of simulation-based testing of autonomous vehicles and potential research directions for future work.

10.2 Search-based Test Case Generation

This section presents how we use an evolutionary search-based technique to generate test cases. Since each scenario takes a few seconds to execute, it is not feasible to try all possible test scenarios. Our approach is to adopt a generic data structure, i.e., a data vector called a "noise vector", to represent the test input domain for producing test scenarios. Each element of this vector represents a parameter that defines a test scenario, e.g., waypoints, illumination, and weather. The values of these parameters do not lie within

the same range, so to bind the values within a specific range, the input representation also scales the concrete real values to values within the range $[-1, +1]$. The values in the noise vector are manipulated by the search algorithm to produce test cases. In our approach, we use a genetic algorithm to explore the search space and produce test cases that are judged as more valuable using an objective function based on potential pedestrian collisions.

10.2.1 Scenario Creation and Manipulation

We use SVL Visual Scenario Editor as the first step to create a basic scheme of the test scenarios that are going to be executed by SVL simulator. SVL Visual Scenario Editor is a GUI application that can be used to create basic scenarios specifying where agents (pedestrians, vehicles, ego vehicle, etc.) are positioned in a map and the basic scheme of the path that they should take through the map, which is specified in the form of waypoints.

The basic scenario is created and exported from SVL Visual Scenario Editor to SVL simulator. This scenario is then manipulated by `ScenarioGenerator` to produce new test scenarios. In `ScenarioGenerator`, a derived test scenario is specified by a vector of real numbers, the *noise vector*, with values between -1 and $+1$.

10.2.2 Scenario Specification

A test scenario is defined as a set of parameters used for test scenario generation, i.e., modeling the test inputs, which is shown as follows:

$$TS = \langle S_1, S_2, \cdots, S_m \rangle \, , \, R_{imin} \leq S_i \leq R_{imax} \quad R_{imin}, R_{imax} \in \mathbb{R} \tag{10.1}$$

Where TS indicates a test scenario and S_i denotes a test input parameter. The values of the test input parameters often vary over different ranges. R_{imin} and R_{imax} represent the upper and lower boundaries of the value range for parameter S_i.

For example, the scenario may define a variable S_{tod} representing the time of day. In the base scenario, the time of day may be defined as 12:00. R_{todmin} and R_{todmax} are used to limit the *change* in this value in a generated test scenario (e.g., values of -5 and 5 would allow the time to vary from 7:00

to 17:00). The values of parameters representing the positions of the agents would have different ranges—e.g., the position points in a path that the vehicle takes may change by ± 2 (meters).

10.2.3 Noise Vector

The proposed representation for a test case is a vector, which is defined as follows:

$$noise\_vector = \langle N_1, N_2, \cdots, N_m \rangle \ , \ -1 \leq N_i \leq +1 \tag{10.2}$$

where each element, N_i, corresponds to a test input parameter, S_i, and the values of components of the noise vector are scaled to values in R using a linear scaling function to create a test scenario, TS.

$$S_i = (N_i + 1) \times (R_{imax} - R_{imin})/2 + R_{imin} \tag{10.3}$$

This transformation allows the use of a generic representation that can be uniformly manipulated by the test generator without detailed knowledge of each input parameter. All elements of the noise vector fall within the range $[-1, +1]$, and are scaled appropriately using R_{imin} and R_{imax} for that S_i.

For example, a noise vector value of 0.5 for the entry representing the time of day, S_{tod}, would result in the following concrete value in a test case: $S_{tod} = (0.5 + 1) \times (17 - 5)/2 + 5 = 1.5 \times 6 + 5 = 14$, or 14:00.

10.2.4 Objective Function

In order to generate valuable test scenarios, we must identify scenarios that are more likely to lead to safety violations. Safety violations can occur then the ego car moves toward its target at a reasonable speed. Specifically, the objectives to be optimized are as follows:

- The total distance[1] of the ego vehicle from other non-ego traffic during scenario execution. This objective should be *minimized*—we want to

[1] Euclidean distance

$$d(p1, p2) = \sqrt{(p1_x - p2_x)^2 + (p1_y - p2_y)^2 + (p1_z - p2_z)^2)}$$

examine ego vehicle behavior in potentially dangerous scenarios.

$$ego\_agents\_distance = \sum_{agent \in agents} \sum_{s \in (1,\ldots,steps)} d(ego.pos_s, agent.pos_s) \tag{10.4}$$

- The total distance of the journey. This should be *maximized*, as longer journeys are preferred.

$$journey\_distance = d(ego.pos_1, ego.pos_{finalstep}) \tag{10.5}$$

- acc : the number of accidents. This should also be *maximized*, as we seek failures in ego vehicle behavior.

Since the aforementioned objectives do not conflict with each other, we merge them to form a single objective function. This function is *minimized*—lower scores are preferred. The objective function that we seek to minimize is defined as:

$$E = ego\_agents\_distance - journey\_distance - 1000 \times acc \tag{10.6}$$

We put high values on the number of accidents, as we are interested in generating test scenarios leading to crashes.

10.2.5 Search Algorithm

It is not possible to execute every possible test scenario that can be defined by an instance of the noise vector. Instead, we seek a systematic means to sample from the space of possible scenarios in *search* of those that could lead to safety violations. This can be done by using an optimization algorithm to sample the space, as guided by the objective function.

The optimization algorithm used to minimize the objective function is a Genetic Algorithm (GA). Genetic Algorithms are modeled on the evolution of a population over time. Initially, a random population of solutions (*noise_vector* instances) is generated. Then, at each generation, a new population is formed based on the best solutions resulting from the previous generations of evolution. This population is formed by:

- Identifying good solutions using *tournament selection*, where a subset of the population is selected at random and the best member of the subset is identified.
- Breeding "child" solutions by combining elements of "parent" solutions through *crossover*, where the child solutions are formed by selecting genes (elements) from each parent solution.
- Introducing *mutations* into the population by making small, random adjustments to solutions.

Tournament selection is performed to identify parent solutions, then crossover and mutation are performed at user-set probabilities. Either, or both, may be applied to transform the identified solutions. Finally, the resulting solutions are added to the new population. This process continues until a new population is formed. The objective function is calculated for each member of this population, and the score is stored for that solution. This process is performed each generation, until a user-set number of generations has been exhausted. At the end, the best solutions are returned.

In our case, we have three objectives—$ego\_agents\_distance$, $journey\_distance$, and acc, which have been merged into a single formula. Tournament selection picks the best solution among the solutions in each tournament. The number of individuals participating in each tournament denotes the size of the tournament. In our approach, we omit the crossover operation, as the noise vector contains the values for the parameters of the test scenarios in a certain order, and crossover could violate this ordering. Instead, we apply mutation with a high probability. We use *Polynomial Bounded mutation*, as proposed and implemented in NSGA-II [15]. It is a bounded mutation operation for real-valued parameters and uses a polynomial function for the probability distribution. It uses a parameter, eta indicating the *crowding degree* of the mutation, which is used to encourage diversity in the resulting population. A high eta yields a mutant resembling the original solution, while a small value for eta produces a solution more divergent from the original. The GA algorithm used for generating test scenarios is configured as presented in Algorithm 16.

Algorithm 16 GA for Test Scenario Generation

Initialize population with solutions from random seeds;
Evaluate the population;
repeat
1. Select offspring using Tournament Selection with replacement;
2. Mutate the resulting offspring using *Polynomial Bounded mutation* operation with a certain probability (mutation rate = 0.95);
3. Evaluate the offspring using the objective function;

until *meeting the stopping criteria (reaching the maximum number of generations or other limitations specified in the test budget)*;

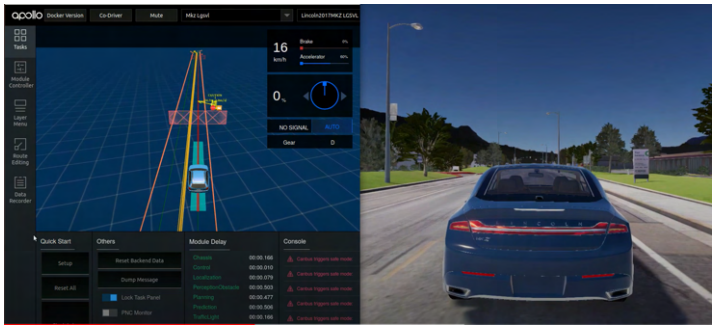


Figure 10.1: Overview of the experimental setup.

10.3 Implementation and Empirical Evaluation

We perform an empirical evaluation of the proposed test case generation technique, `ScenarioGenerator` by running experiments on our experimental setup on a desktop PC with the following specifications:

- Ubuntu version 18.04
- Intel Core i7-10700K CPU @ 3.80GHz $\times$ 16
- 32GB RAM
- GeForce RTX 2070 SUPER/PCIe/SSE2

- SVL simulator 2021.1 (linux64) with modular testing setup (3D Ground Truth sensor and Signal sensor publish ground truth perception data to Apollo via CyberRT bridge)
- Baidu Apollo (r6.0.0 branch)

The experiments are simulations that are controlled by a Python scenario runner which uses our test case generation technique for generating the scenarios in the simulation environment. Baidu Apollo is the autonomous driving software platform that controls the ego vehicle. It connects to the simulator through its customized bridge and drives the ego vehicle (Figure 10.1).

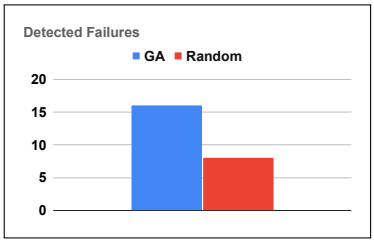
We design a set of experiments to assess the efficiency and effectiveness of the proposed test case generation for testing Apollo in the SVL simulation environment. Pedestrian detection and proper responding is the target use case of Apollo in our experiments. For a comparative analysis, we also report results from a random testing technique as a baseline approach. In random testing, the test cases are generated randomly, which means that the set of noise vector instances are generated by setting the test input parameters to random values within the allowed range. The target is to generate the highest number of diverse valid test cases leading to failures, i.e., collisions between the ego vehicle and pedestrians. We use the following quality criteria for evaluating the proposed test case generation technique:

- **Detected Failures:** The number of test cases that lead to a collision.
- **Failure Diversity:** The dissimilarity between the generated test cases leading to failures. We are interested in generating diverse test cases, as triggering similar failures leads to waste of the test budget, e.g., computation resources. To measure failure diversity, we use the *Euclidean distance* between failure-revealing noise vectors.

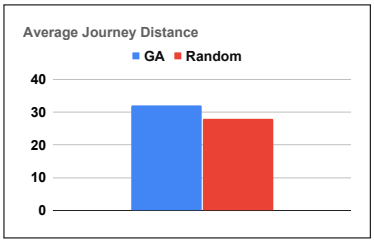
10.3.1 Test Scenario Execution

The testing budget (including, e.g., execution time) is a limited resource. While not as expensive to perform as on-road testing, running test scenarios in simulators also takes time. In our experiments, each scenario takes about 10 seconds to execute and evaluate. Therefore, for the purpose of this

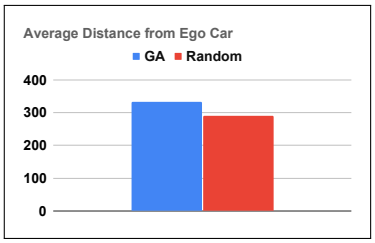
competition, we set the limit for the number of simulation executions to 200 in the Genetic Algorithm. This would correspond, for example, to 20 generations with a population size of 10.



(a) Number of detected failures.



(b) Objective values for the average journey distance during failure-revealing test cases.



(c) Objective values for average distance from ego car during failure-revealing test cases.

Figure 10.2: Comparisons between GA and random generation.

In `ScenarioGenerator`, the user-controllable parameters for test scenario creation and manipulation are as follows:

- Initial JSON file created by SVL Visual Scenario Editor.
- Test case generation strategy, which is used for scenario generation. Currently, Differential Evolution, Powell Optimization, Genetic Algorithm, and random generation strategies are supported. Meanwhile, the capability of replaying a scenario is also supported by passing the JSON file and setting the action to *replay*. A specific noise vector in combination with replay action can also be used. In this mode,

Figure 10.3: Collision between pedestrian and the ego vehicle on a rainy night.

in addition to all the previous parameters, a specific noise vector is given to be played.

- The ego vehicle destination.
- Acceptable range of changes in the values for the *position of each waypoint* (x, z).
- Acceptable range of changes in the color of each agent (r, g, b).
- Acceptable range of changes in the weather in the simulation (e.g., rain, fog, wetness, cloudiness, road damages).
- Acceptable range of changes in the time of day.
- Acceptable range of changes in the speed of each agent.

In a test case, the generated noise vector is used to impose changes to the position of each waypoint, the color of each agent, the weather, the time of day, and the speed of each agent. The base scenario defines a value for each of these parameters. The user-controllable parameters are used to constrain the range

of changes made by the noise vector between minimum and maximum values, as discussed in Section 10.2.

10.4 Results and Discussion

This section presents the experimental results and assesses the proposed test case generation compared to the random testing with regard to the quality criteria.

Detected Failures: Figure 10.2(a) shows the number of detected failures (test cases leading to collisions) by the GA-based test case generation and random testing. The proposed GA-based technique trigger twice as many failures as random testing on the same configuration and test budget, and consequently, in this regard, works more effectively. Figure 10.3 also shows a sample of a generated test scenario leading to a collision between the pedestrian and the ego vehicle.

In order to investigate the characteristics of the detected failures, we can examine the values of two of the objectives in the objective function—$ego\_agents\_distance$ and $journey\_distance$. These can show the characteristics of the detected failures. Figure 10.2(b) and (c) show the average values of the two objectives in failure-revealing test cases for both techniques. These average values do not differ significantly between the two approaches. This indicates that the GA reveals more failures, but the failures revealed by the two techniques fall in similar objective ranges. However, both distances are somewhat higher in the GA—i.e., the GA generates tests with slightly longer journey distances and a slightly higher distance from the ego car. These tests may be somewhat more interesting for revealing errors in the ego car functionality, as—for example—a longer distance between the ego car and a pedestrian should offer more time to make corrections. In future work, we will examine failing scenarios more closely and discuss them with domain experts.

Failure Diversity: We use pairwise Euclidean distance between the noise vectors to show diversity between the failure-revealing test cases. Figure 10.4 and 10.5 show the average pairwise Euclidean distance for each of the failure test cases generated by GA and random testing respectively. The average pairwise Euclidean distance refers to the average difference between a test case and the other test cases. Table 10.1 shows the range of average pairwise

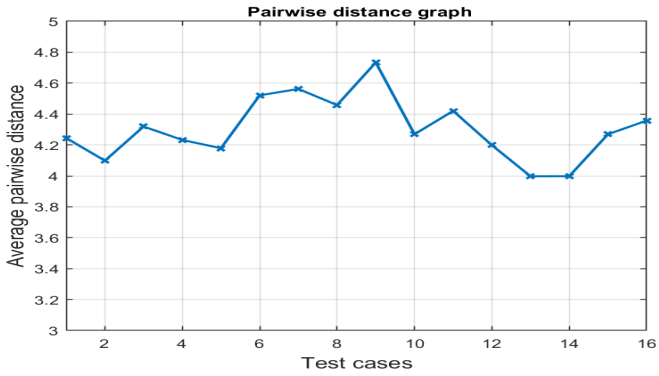


Figure 10.4: Diversity of failure-revealing test cases generated by the GA.

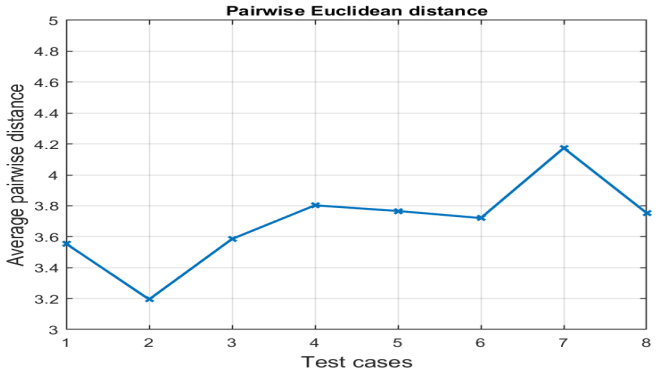


Figure 10.5: Diversity of failure-revealing test cases generated by random testing.

Euclidean distance for the failure-revealing test cases from the GA and random testing. In this regard, the GA technique also promotes more diversity between generated failure-revealing test cases than random testing.

10.4.1 Threats to Validity

Some of the main sources of threats to validity of the experimental results are as follows:

Table 10.1: Failure diversity, shown as the range in the average pairwise Euclidean distance for test cases.

	Genetic Algorithm	**Random**
Range of Euclidean Distances	$4.1 - 4.7$	$3.2 - 4.2$

Internal Validity: During the experiment, we noticed that many of the failures that are captured are not completely reproducible. In fact, the simulation execution often does not produce identical results given identical input parameters and configuration setup. One of the main reasons is that Apollo does not function in a deterministic manner. We tried to mitigate the effects of this by reporting average values from the experiments, and conducting the experiments in a controlled manner, i.e., using the same experimental setup and keeping the user-controllable parameters fixed between executions. Another source of threat is the fact that as the simulator runs a large number of test cases, the simulations become slower and less responsive probably due to performance bottlenecks.

External Validity: We have focused on a single scenario. As we have used a generic data structure consisting of variables scaled in a certain range, i.e., the noise vector with variables within the range $[-1, 1]$, we believe that the representation model and test case generation approach could be used for simulation-based testing of more complex scenes and other use cases. However, the variables in the noise vector might need to be modified (e.g., extended) for different use cases.

10.5 Related Work

Simulators as a form of digital twins play a key role for different purposes in testing and verification, control and monitoring, and improvement of cyber-physical systems (CPS). For ADAS and autonomous-driving cars, this is even more significant and there is a higher demand for high-fidelity simulators. Simulation-based testing is one of the most effective approaches for system-level testing of ADAS and acts as a suitable complementary solution to on-road testing, since it provides the possibility for early stage testing, capturing critical corner test scenarios and enabling inexpensive

testing. Field testing of such systems is expensive, inefficient and even dangerous, in some cases. Recently, various simulators such as those ones using physics-based models, e.g., SVL simulator [16], PreScan [17] and Pro-SiVIC [18] or the ones relying on game engines, e.g., BeamNG [19] and CARLA [20], have been developed to meet the need for realistic simulation of the functions in autonomous driving.

Accordingly, various system-level testing approaches relying on the simulators have been proposed in the recent years. One of the common intended purposes in those studies is generating critical test cases (scenarios) that lead the system to fail. This is a challenging problem, due to the large search space of input parameters in these systems. Covering all possible simulation test scenarios is not feasible in practice. Therefore, in this regard SBST techniques have been widely used to generate effective test simulation scenarios for those systems. In recent studies, multi-objective search algorithms like NSGA-II [10], many-objective algorithms like MOSA [21] using a combination of different objectives based on branch coverage and failure-based heuristics [22], and learnable evolutionary algorithms [23] have been used to generate critical test cases leading to violations of safety requirements in autonomous driving cars. Moreover, there have also been studies focusing on the role of simulators and the type of test data. In [24] a comparison between testing of DNN-based ADAS using real-world and simulator-generated data is conducted and it is also showed that how on-line and off-line testing of these systems can differ and meanwhile complement each other. Markus et al. studied the consistency between the results obtained from two different simulators and investigated whether the obtained results could be mutually reproducible in both simulators [13].

10.6 Conclusion and Future Work

Efficient and effective test case generation for use in virtual environments is essential for testing AI-based automotive systems. In this paper, we presented a SBST approach to generate test scenarios that lead to detection of failures and safety violations of the Baidu Apollo pedestrian emergency braking system. We have made three primary observations. First, our results show that the proposed GA-based test case generation is more effective than random testing, i.e., it is more effective in generating failure revealing test cases and provides

higher diversity between the generated test cases compared to random testing. Second, unfortunately, many of the captured failures could not be reproduced given the same configuration and user-controlled parameters due to the non-deterministic nature of Apollo. Third, we see great potential in simulation-based testing of different functions of autonomous driving systems using SVL simulator and Baidu Apollo. In future work, we will broaden the scope of the research into additional safety scenarios. We will also extend SBST approaches with machine learning-based techniques (e.g., reinforcement learning) for test case generation in system-level testing of ADAS. We are also interested in the use of Generative Adversarial Networks (GANs) as a technique for enabling the discovery of failure-revealing test cases.

Acknowledgment

This project has received funding from the ECSEL Joint Undertaking (JU) under grant agreement No 876852 (VALU3S). Furthermore, this work received support from the ITEA3 European IVVES project (https://itea3.org/project/ivves.html) and the SMILE III project financed by Vinnova, FFI, Fordonsstrategisk forskning och innovation under the grant numbers 2019-05871 and the AIQ Meta-Testbed project funded by Kompetensfonden at Campus Helsingborg, Lund University, Sweden. Additional support was provided under Vetenskapsrådet grant 2019-05275. The authors would like to thank INFOTIV AB for their support and cooperation.

Bibliography

[1] A definition of artificial intelligence: Main capabilities and scientific disciplines. Technical report, High-Level Expert Group on Artificial Intelligence, Brussels, Belgium, 2018.

[2] Markus Borg, Cristofer Englund, Krzysztof Wnuk, Boris Duran, Christoffer Lewandowski, Shenjian Gao, Yanwen Tan, Henrik Kaijser, Henrik Lönn, and Jonnas Törnqvist. Safely entering the deep: A review of verification and validation for machine learning and a challenge elicitation in the automotive industry. *Journal of Automotive Software Engineering*, 1(1):1–19, 2019.

[3] Communucation from the commission to the european parliament, the european council, the europan economic and social commitee and the commitee of the regions - artificial intelligence for europe. Technical report, Eurpean Commision, Brussels, Belgium, 2018.

[4] Jie M Zhang, Mark Harman, Lei Ma, and Yang Liu. Machine learning testing: Survey, landscapes and horizons. *IEEE Transactions on Software Engineering*, 2020.

[5] Vincenzo Riccio, Gunel Jahangirova, Andrea Stocco, Nargiz Humbatova, Michael Weiss, and Paolo Tonella. Testing machine learning based systems: a systematic mapping. *Empirical Software Engineering*, 25(6):5193–5254, 2020.

[6] Florian Bock, Christoph Sippl, Sebastian Siegl, and Reinhard German. Status report on automotive software development. In *Automotive Systems and Software Engineering*, pages 29–57. Springer, 2019.

[7] Philip Koopman and Michael Wagner. Challenges in autonomous vehicle testing and validation. *SAE International Journal of Transportation Safety*, 4(1):15–24, 2016.

[8] Road Vehicles - Safety of the Intended Functionality. Technical Report ISO/PAS 21448:2019, International Organization for Standardization, 2019.

[9] Phil McMinn. Search-based software testing: Past, present and future. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 153–163, 2011.

[10] Raja Ben Abdessalem, Shiva Nejati, Lionel C. Briand, and Thomas Stifter. Testing advanced driver assistance systems using multi-objective search and neural networks. In *Proc. of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 63–74, 2016.

[11] Raja Ben Abdessalem, Shiva Nejati, Lionel C. Briand, and Thomas Stifter. Testing vision-based control systems using learnable evolutionary algorithms. In *Proc. of the 40th International Conference on Software Engineering*, pages 1016–1026, 2018.

[12] Alessio Gambi, Tri Huynh, and Gordon Fraser. Generating effective test cases for self-driving cars from police reports. In *Proc. of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 257–267, 2019.

[13] Markus Borg, Raja Ben Abdessalem, Shiva Nejati, François-Xavier Jegeden, and Donghwan Shin. Digital twins are not monozygotic–cross-replicating adas testing in two industry-grade automotive simulators. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 383–393. IEEE, 2021.

[14] Mahshid Helali Moghadam, Markus Borg, and Seyed Jalaledin Mousavirad. Deeper at the SBST 2021 tool competition: ADAS testing using multi-objective search. In *2021 14th Intl. Workshop on Search-Based Software Testing (SBST)*. IEEE, 2021.

[15] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.

[16] LG Electronics. SVL Simulator. https://www.svlsimulator.com/, Retrieved July, 2021.

[17] TASS International. . PreScan Simulator. https://tass.plm.automation.siemens.com/prescan-overview, Retrieved July, 2021.

[18] Assia Belbachir, Jean-Christophe Smal, Jean-Marc Blosseville, and Dominique Gruyer. Simulation-driven validation of advanced driving-assistance systems. *Procedia-Social and Behavioral Sciences*, 48:1205–1214, 2012.

[19] BeamNG GmbH. BeamNG.research. https://beamng.gmbh/research/, Retrieved July, 2021.

[20] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. CARLA: An open urban driving simulator. In *Conference on robot learning*, pages 1–16. PMLR, 2017.

[21] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Reformulating branch coverage as a many-objective optimization problem. In *2015 IEEE 8th international conference on software testing, verification and validation (ICST)*, pages 1–10. IEEE, 2015.

[22] Raja Ben Abdessalem, Annibale Panichella, Shiva Nejati, Lionel C Briand, and Thomas Stifter. Testing autonomous cars for feature interaction failures using many-objective search. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 143–154. IEEE, 2018.

[23] Raja Ben Abdessalem, Shiva Nejati, Lionel C Briand, and Thomas Stifter. Testing vision-based control systems using learnable evolutionary algorithms. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 1016–1026. IEEE, 2018.

[24] Fitash Ul Haq, Donghwan Shin, Shiva Nejati, and Lionel C Briand. Comparing offline and online testing of deep neural networks: An autonomous car case study. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 85–95. IEEE, 2020.

Chapter 11

Paper F: Adaptive Runtime Response Time Control in PLC-Based Real-Time Systems Using Reinforcement Learning

Mahshid Helali Moghadam, Mehrdad Saadatmand, Markus Borg, Markus Bohlin, and Björn Lisper
In the Proceedings of the 13th IEEE/ACM International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), IEEE/ACM 2018.

Abstract

Timing requirements such as constraints on response time are key characteristics of real-time systems and violations of these requirements might cause a total failure, particularly in hard real-time systems. Runtime monitoring of the system properties is of great importance to check the system status and mitigate such failures. Thus, a runtime control to preserve the system properties could improve the robustness of the system with respect to timing violations. Common control approaches may require a precise analytical model of the system which is difficult to be provided at design time. Reinforcement learning is a promising technique to provide adaptive model-free control when the environment is stochastic, and the control problem could be formulated as a Markov Decision Process. In this paper, we propose an adaptive runtime control using reinforcement learning for real-time programs based on Programmable Logic Controllers (PLCs), to meet the response time requirements. We demonstrate through multiple experiments that our approach could control the response time efficiently to satisfy the timing requirements.

11.1 Introduction

Real-time control programs implemented on Programmable Logic Controllers (PLCs) are key parts of many time-critical industrial control systems like those in the railway domain. The timing properties in these systems include period of tasks, deadline, worst-case execution time or response time. From the perspective of timing analysis, schedulability analysis methods, statistical and formal timing analysis are common analysis techniques to provide a response time estimation of real-time programs [1, 2, 3]. Static analysis-based approaches, in some cases, might not be practical for complex real-time systems. Even if they are feasible, the results might not be valid due to unpredictable factors in runtime and the difference between analysis environment and the realistic one [4].

Generally, there is often a strict set of timing requirements such as deadlines and limits on response time for real-time programs in mission-critical contexts. Correctness of functionality of real-time systems highly depends on satisfying the timing requirements as important features of these systems. Any serious deviation in temporal behavior of real-time programs due to unpredicted runtime events like asynchronous message-passing and runtime changeable priorities, particularly in complex systems, might cause a total failure in the function of system. Thus, providing more robustness against unpredicted varying conditions during runtime is of great importance. In general, robustness could be defined as to which degree the system is tolerable against incorrect inputs or unexpected stressed conditions [5]. In a real-time program, robustness could be defined as the ability to adapt to the varying conditions while satisfying the timing requirements.

An adaptive runtime control in addition to the scheduling capabilities could lead to more robustness in real-time control systems, to cope with changing runtime conditions and unpredicted states [6]. Runtime monitoring could check if the system adheres to the predefined requirements like timing constraints. A control approach based on runtime monitoring could help preserve these timing properties by applying runtime control operations. Adaptive control strategies are considered as one of the promising solutions to improve robustness through providing adaptation to the varying conditions in dynamic environments. Reinforcement learning (RL) has been frequently

applied to address the adaptive control strategy in dynamic environments, in case the environment is stochastic, and the control problem can be formulated as a Markov Decision Process (MDP).

In this paper, we propose a self-adaptive response time control for real-time programs in PLC-based systems using reinforcement learning. In our previous work [7], we presented the initial idea on how a learning-based solution can be used to provide assurance of timing properties; here in this work we extend that initial idea and provide an industrial evaluation of our proposed approach. We present the evaluation experiments of the proposed approach on sample programs inspired from our collaboration with Bombardier Transportation in Sweden. The proposed approach formulates the response time control problem as an MDP and uses Q-learning as a model-free RL to provide adaptive control of response time while meeting the timing requirement. We show the efficacy of the proposed approach through multiple experiments based on simulating real-time programs in a PLC-based control system. Our approach mostly keeps the programs adhering to the response time constraints despite the occurred time deviations during the run time. Based on the evaluation results, the proposed approach with ε-greedy, $\varepsilon = 0.5$, and $\alpha = 0.1$ and $\gamma = 0.5$ provided better satisfaction of the response time threshold without any programs ending with medium or high deviation.

The rest of this paper is organized as follows; Section 11.2 discusses briefly the motivation and background concepts of RL. The technical details of the proposed approach are discussed in Section 11.3, while Section 11.4 presents the evaluation experiments and results. Section 11.5 provides a review of the related works and background techniques. Conclusions and future directions are provided in Section 11.6.

11.2 Motivation and Background

11.2.1 Motivation

Runtime monitoring is considered as a principal means for real-time systems. Providing an adaptive control for satisfying the timing requirements such as constraints on response time/execution time based on runtime monitoring could improve the robustness of the system. Model-driven control approaches may require precise knowledge of the system and environment. The complexity

of real-time systems, for example, intricate temporal dependencies between real-time tasks and the dynamism of the environment are major barriers which motivate towards model-free learning-based control. Learning-based control can find an adaptive control policy to varying conditions regardless of having a precise model of the environment. Reinforcement learning-based control techniques have been used for runtime control of non-functional properties to satisfy the performance and timing requirements in many application contexts.

Reinforcement learning [8] is a learning mechanism working based on interaction with the environment. In RL, the agent senses the state of the environment continuously, takes a possible action and in return, receives a reward signal from the environment which shows the desirability and effectiveness of the applied action. During the learning, the agent follows a policy which maximizes the long-term received reward. The agent learns this policy through an action selection strategy which is based on selecting an action randomly (exploration) or selecting an action with a high utility value (exploitation). Q-learning [8] is a model-free RL algorithm in which the agent learns the value function of the long-term expected reward associated to the pairs of states and actions. It is an off-policy learning as the optimal policy is learnt independently of the action selection strategy being used by the agent. Once the learning converges, the agent replays the learned policy.

11.2.2 PLC-Based Industrial Control Programs

Many of the real-time industrial control systems like those ones in the transportation domain, are implemented based on IEC 61131-3 [9] which is one of the main programming language standards for programmable controllers. According to the proposed software structure in IEC 61131-3, Programmable Organization Units (POU) are the building blocks of a PLC program. They are hardware-independent and programmable in a flexible fashion facilitating the reusability and modularization in this context.

There are mainly three unified types of POUs: program, function block and function. A function block has its own data record to remember the state of the information, while a function always produces the same result based on the same input. A program may consist of zero or multiple function and function blocks. A real-time task can execute one or multiple programs or a set of function blocks. Timer function blocks are widely used as one of the

main constituent POUs in PLC-based real-time programs. Their basic functions involve providing their output after a preset controllable/programmable time interval. There are three types of timers as standard PLC timer blocks, i.e., TP (Timer Pulse), TON (Timer On-Delay) and TOF (Timer Off-Delay). Timer TP is a pulse generator which supplies a constant pulse on output upon detecting a rising edge at input. TON supplies the value of input at output with a delay upon detecting a rising edge at input. TOF has an inverse functionality to TON. Figure 11.1 shows a schema sample from a real-time control program in Function Block Diagram format, as an integration of multiple functions and function blocks. The number of POUs in each control program depends on the complexity of the program. The time delay of timer function blocks in time-critical programs are the target entities supposed to be tuned in urgent conditions by our control approach to satisfy the response time requirements.

11.3 Adaptive Response Time Control Using Q-Learning

In this section, we present the technical details of the proposed runtime response time control using reinforcement learning for real-time programs running on PLC-based systems. This control method is incorporated into the control scan program which is responsible for executing the building blocks and preserving their execution orders within real-time programs. Timer function blocks are one of the standard function blocks which are widely used and play a key role in many time-critical industrial control programs.

The proposed control strategy is supposed to use the capability of tuning

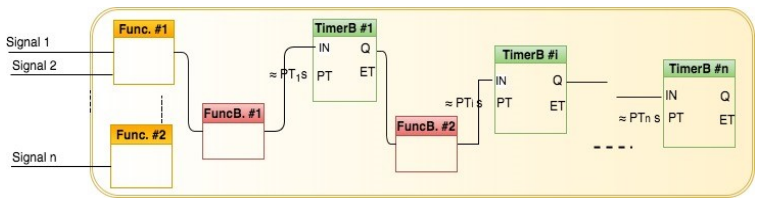


Figure 11.1: A schema sample from a PLC-based control program

the time delay of timer function blocks to control the response time of real-time programs. The main objective of the proposed runtime response time control is to meet the response time requirements in abnormal conditions when time deviations happen, by optimally tuning the time delay parameters. The proposed approach uses Q-Learning as a model-free RL to learn the optimal tuning of delay parameters to preserve the program responsive within the target response time threshold. The learning task in the proposed control approach mainly involves the following steps:

1) Detecting the *State* of the system. Based on the interactive characteristic of the reinforcement learning, the control agent/controller observes the state of the program at discrete time steps. After each execution cycle, the controller measures the execution time until the current time point. The actual execution time until the end of the n^{th} function block execution, ET_n, is classified under four classes. This is done based on the amount of compliance with the desired/target execution time until the end of the n^{th} function block (e.g., from requirements/constraints), T_n, calculated as follows:

$$T_n = \sum_{i=1}^{n} T_i^f \tag{11.1}$$

Where T_i^f is the desired response time of the i^{th} function block. The class values representing the state of the program, *s*, are *Required, Low, Medium* and *High*, as shown in Figure 11.2. They represent the acceptable state, and the states with low, medium and high deviation, respectively. We defined the acceptable state based on a target execution time characterized by a tolerance region $[T_n, T_n']$ where $T_n' = T_n + \tau$. where τ in *ms* is defined based on the characteristics of the system.

2) Selecting a *Control Action*. We defined the control actions as tuning operations for the time delay of the next running function block, D_f^{n+1}. For providing a safety margin, we also considered a required minimum delay, D_m, for function blocks. Then, the time delay of function blocks could not be set to a value less than D_m. Regarding the minimum time delay, we specified a set of control actions for tuning the time delay as follows:

$$Actions = \{(1 - f_d)D_f^{n+1} + f_d D_m : f_d \in K\} \tag{11.2}$$

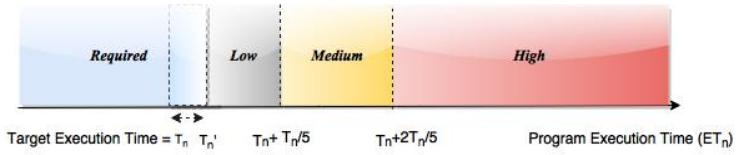


Figure 11.2: States of the program

$$K = \{0, \frac{1}{5}, \frac{2}{5}, \frac{3}{5}, \frac{4}{5}, 1\} \tag{11.3}$$

Where f_d is a decreasing factor.

3) Receiving the *Reward* signal and updating the stored experience. After applying the selected action, the system will go to the next state and the controller will receive a reward signal representing the effectiveness of the applied action. We derived a utility function based on a Normal probability density function with $\mu = T_n$ and $\sigma = T_n/10$ which is as follows:

$$r_n = \begin{cases} \frac{1}{\frac{T_n}{10}\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{ET_n - T_n}{\frac{T_n}{10}})^2}, & T_n < ET_n \\ 1, & ET_n \leq T_n \end{cases} \tag{11.4}$$

The computed reward values will be in the range (0, 1].

The final objective of the learning is to find a policy π, a mapping between the states and actions, which maximizes the expected long-term reward defined as follows [8]:

$$R_n = r_{n+1} + \gamma_{n+2} + ... + \gamma^k r_{n+k+1} \tag{11.5}$$

Where $\gamma \in [0, 1]$ is a discount factor specifying the importance of future rewards compared to the immediate reward. The long-term expected return of selecting action *a* in state *s*, based on policy π, is specified by a utility value $Q^\pi(s, a)$ defined as follows [8]:

$$Q^\pi(s, a) = E^\pi[R_n | S_n = s, A_n = a] \tag{11.6}$$

The Q-values stored in a look-up table, *Q-table*, form the experience of the agent. The controller relies on Q-values to make decision on actions. During the learning, the Q-values are updated incrementally via temporal differencing. The agent updates the associated $Q^\pi(s, a)$ for each experienced (s, a) through

the following rule:

$$Q(s_n, a_n) = Q(s_n, a_n) + \alpha[r_{n+1} + \gamma \max_a Q(s_{n+1}, a) - Q(s_n, a_n)] \quad (11.7)$$

Where $\alpha \in [0, 1]$ is the learning rate parameter. It specifies to what extent new information impacts the q-values. The all steps of the adaptive control procedure are described in Algorithm 17. Eventually, after multiple learning cycles, the controller finds the optimal policy of selecting the action which maximizes the q-value in a given state.

Learning performance. Different action selection strategies could be used during the learning. The agent can use a random action selection method or select greedily an action with the highest utility value according to the Q-table. *ε-greedy* is an action selection strategy which allows the agent to make a trade-off between the exploration and exploitation in the action space. In ε-greedy, with probability ε, a random action is selected and with probability $1 - \varepsilon$, an action based on the utility value is selected. However, RL-based approaches might generally suffer slow convergence due to the need for exploring the state space. To alleviate this effect, we also introduced an initial control mapping in Q-table by specifying some invalid pairs of state and action to guide the agent not to explore specific actions in a specific state. For example, when it is in acceptable state, no need to change the time delay parameter.

11.4 Results and Discussion

This section presents the results of the early stage evaluation experiments addressing the performance of the proposed approach in terms of meeting the predefined response time threshold. The main objective of the experiments is to assess to which degree the learning-based control can work adaptively on varying conditions and untimely behavior of function blocks in a realistic environment.

11.4.1 Evaluation Setup

In this study, we implemented the proposed approach based on three action selection strategies. We incorporated it into an environment which simulates multiple real-time programs consisting of various timer function blocks. The

Algorithm 17 Adaptive Response Time Control in PLC-based Real-time programs

Required: S, A, $\varepsilon, \alpha, \gamma, \phi$ (invalid state-action pairs)
Initialize Q-values, $Q(s,a) = -1 \; if \; (s,a) \in \phi \; else \; 0 \; \forall s \in S, \; \forall a \in A$;
1. Detect the current state of the program, s_n
2. Select an action using the action selection policy
(e.g., ε-greedy: select $a_n = argmax_{a \in A} Q(s_n, a)$ with probability (1- ε) or a random action with probability ε)
3. Apply the selected action, let the system continue running and execute the next function block
4. Detect the new state of the system
5. Compute the reward (reinforcement) signal
6. Update the Q-value by:
$Q(s_n, a_n) = Q(s_n, a_n) + \alpha[r_{n+1} + \gamma \max_a Q(s_{n+1}, a) - Q(s_n, a_n)]$
7. Repeat for every observed state at the start of each function block execution

simulation environment emulates the temporal behavior of the function blocks, their responses in realistic environments and the corresponding control scan program for controlling the execution order of the function blocks. The learning-based control has been integrated into the control scan thread to provide a runtime control of the response time of real-time programs.

The proposed approach has been evaluated through two analysis scenarios. In the first scenario, concerning response time analysis, the performance of the learning-based control based on using three action selection algorithms has been studied. In this scenario, the performance of the proposed approach after 100 learning episodes (interaction with various real-time programs) has been demonstrated. The real-time programs have been characterized with different numbers of function blocks, predefined response time requirements and minimum required delay time (safety margin). The second analysis scenario, sensitivity analysis, analyzes the sensitivity of the learning-based approach to the learning parameters. This scenario involves investigating the effects of the learning parameters by systematically changing the values of one parameter while keeping the other one constant.

11.4.2 Experiments and Results

Timing Analysis. In the timing analysis scenario, the efficacy of the learning-based approach was evaluated in terms of adaptation to changeable behavior while meeting the timing requirement. The simulated real-time programs have different numbers of function blocks in the range [5, 25]. The predefined response time requirements of function blocks and associated safety margins in *ms* have been initialized with values in the range [1000, 6000] and [1000, 2000], respectively. A maximum deviation at most equal to 25 percent of the upper bound of the response time requirement was allowed during the simulation. The default acceptable tolerance value was considered as 500*ms*. Time deviations were injected into the programs randomly. Figure 11.3 shows a random pattern for injecting time deviations to function blocks within three program samples. ε-greedy was used in the proposed approach as an action selection strategy with $\varepsilon = 0.1$, $\varepsilon = 0.5$ and $\varepsilon = 0.9$. The ε-value determines to what extent exploration and exploitation are weighted during the action selection procedure. Figures 11.4 and 11.5 show the observed response time plots of real-time programs after applying the learning-based control approach based on different values of ε parameter in the action selection strategy. Clearly, the learning-based control approach tries to adapt well to the varying temporal behaviors of the function blocks while meeting the response time thresholds of the programs. Results in Figures 11.4 and 11.5 describe the efficacy of the learning-based control approach based on the number of programs ended with medium or high deviations from the timing requirements and also the achieved average deviations.

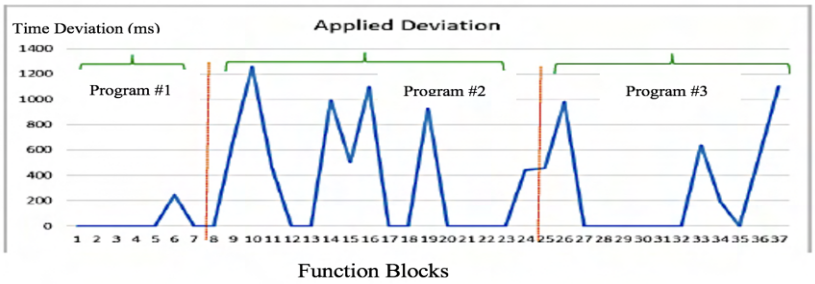


Figure 11.3: Pattern of time deviations

According to the results, the performance of the proposed approach with different action selection strategies is described as follows:

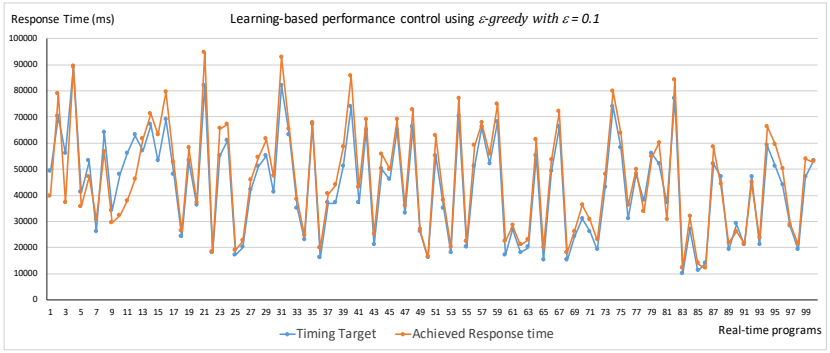
1) ε-greedy with $\varepsilon = 0.1$ makes the controller trust most on its stored experience, rather than exploring new actions. The learning-based approach based on this action selection strategy, showed less efficiency in terms of optimizing the response time and also the number of programs which ended with medium or high deviations. In this case, the experience of the controller has not been extended well and needs more exploration to be improved.

2) ε-greedy with $\varepsilon = 0.9$ provides more opportunities towards the exploration of the action space. It provided partially better performance in terms of optimizing the response time and preventing the programs from exceeding the predefined thresholds with medium or high deviations compared to ε-greedy with ε= 0.1.

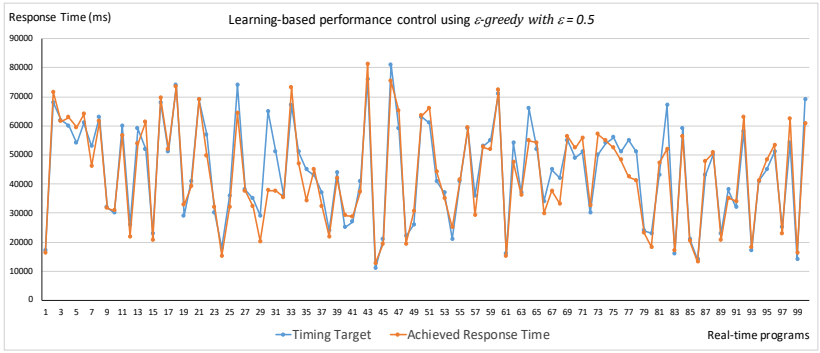
3) ε-greedy with $\varepsilon = 0.5$ provides a trade-off between the exploration and exploitation of the action space. It showed a better adaptation to the varying conditions and tried to preserve the response time close to the requirement threshold. In some cases where a sharp satisfaction of the timing requirement is needed, e.g., airbag control systems of automotive products, this is the desired performance which is required.

4) ε-greedy with decaying ε, is an action selection strategy during which the ε parameter gradually decreases. It causes more exploration during the first steps of the learning and more exploitation at the last steps. Using this strategy, the performance controller first explores the action space, then tends towards using the achieved experience. The learning-based approach based on ε-greedy with decaying ε, showed the most promising results, i.e., it outperformed the other ε-strategies both in terms of optimizing response time and preventing medium or high deviations from the predefined timing thresholds.

Sensitivity Analysis. The behavior of the proposed learning-based control approach could be impacted by the learning parameters including learning rate (α) and discount factor (γ). In the sensitivity analysis, two sets of experiments were done to study the effects of varying learning parameters. Each set of experiments involves changing the value of one parameter while keeping the other one constant. ε-greedy with $\varepsilon = 0.5$ was used as a baseline action selection strategy during the sensitivity analysis experiments. Table 11.1 shows the performance of the learning-based approach regarding the number of real-time programs which ended with medium or high deviations from the

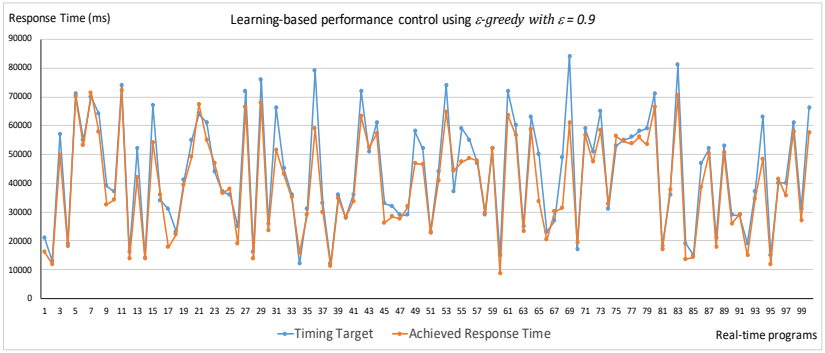


Average injected time deviation per program: 5504 ms	*LR-based using ε-greedy, ε= 0.1*	*Uncontrolled*
#RT programs with highly exceeded predefined threshold	6	13
Achieved average deviation	2585	5504

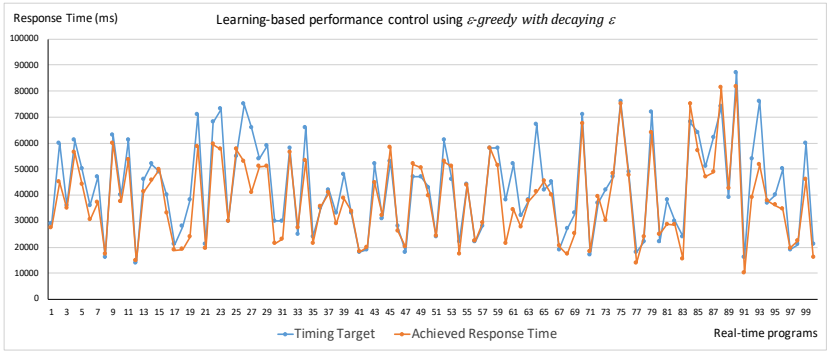


Average injected time deviation per program: 5731 ms	*LR-based using ε-greedy, ε= 0.5*	*Uncontrolled*
#RT programs with highly exceeded predefined threshold	0	10
Achieved average deviation	-1228	5731

Figure 11.4: response time plots of real-time programs and the performance of the adaptive learning-based control



Average injected time deviation per program: 5911 ms	*LR-based using ε-greedy, ε= 0.9*	*Uncontrolled*
#RT programs with highly exceeded predefined threshold	1	7
Achieved average deviation	-3804	5911



Average injected time deviation per program: 5395 ms	*LR-based using decaying ε-greedy*	*Uncontrolled*
#RT programs with highly exceeded predefined threshold	0	8
Achieved average deviation	-4531	5395

Figure 11.5: response time plots of real-time programs and the performance of the adaptive learning-based control

predefined response time thresholds and also the achieved average deviation in response time, during the sensitivity analysis experiments. In Table 11.1 the bold column represents the baseline parameter setting which was used in each sensitivity analysis experiment. We set the learning rate to 0.1 and the discount factor to 0.5 at the first and second experiments, respectively.

It seems that setting the learning rate to 0.1, which provides a slower learning, leads to good performance, particularly in adaptation to varying behaviors and preventing the real-time programs from exceeding the timing thresholds. Increasing the learning rate towards 0.5, which aims at balancing between learning new information and saving previous experience, causes improvement in optimizing the response time of the programs. The proposed approach also does not show as much performance improvement as when we set the discount factor to values other than 0.5.

11.5 Related Work

We classify the relevant works on timing properties of real-time systems under modeling, verifying and some approaches to preserve and satisfy the timing requirements. Many of the verification and preservation/control approaches are based on runtime monitoring of the properties. Real-time Specification for Java (RTSJ) was introduced to provide a real-time scheduler with the facility of monitoring deadlines and enforcing the execution cost [10]. Mezzetti et al. [11] used the Ada Ravenscar Profile for preserving the timing properties of real-time systems. Saadatmand et al. [12, 13] developed an extra scheduler taking the temporal properties including period, execution time and deadline of the tasks and scheduled them using the underlying scheduler of the operating system. A model synthesis approach for timing properties of real-time systems based on monitoring the running system was proposed in [14]. A runtime framework for monitoring the runtime constraints such as timing constraints and detecting the violations of timing properties was presented in [15]. The related issues on runtime monitoring of properties in real-time systems were discussed in [16]. Goodloe et al. [6] surveyed different runtime monitoring techniques including off-line and on-line techniques for distributed real-time systems, in particular hard real-time systems. Das et al. [17] presented a tool environment which provided runtime monitoring, animating the development and analysis of the components to

support model-driven development of real-time embedded systems. In [18] a runtime monitoring approach for checking the system properties in embedded systems was presented. It used a control method to coordinate the time predictability and memory utilization in the monitoring solution.

Table 11.1: Impacts of varying learning parameters on the performance of control approach

	LR-based performance control using ε= 0.5, Discount factor γ=0.5			*LR-based performance control using ε= 0.5, Learning rate α=0.1*		
	α= **0.1**	α=0.5	α=0.9	γ=0.1	γ=**0.5**	γ=0.9
#RT programs with highly exceeded predefined threshold (Uncontrolled Condition)	**0(10)**	0(6)	0(3)	3(8)	**0(10)**	0(7)
Average deviation (Uncontrolled Condition)	**-1228 (5731)**	-6475 (5378)	-4395 (5455)	-1052 (5484)	**-1228 (5731)**	-1210 (5744)

11.6 Conclusion

Runtime monitoring of system properties remains as a principal need for real-time systems. A runtime control approach based on runtime monitoring could improve robustness of the system. In this paper, we present an adaptive runtime response time control based on reinforcement learning for PLC-based real-time programs, to satisfy the timing requirements. In this study, we formulate the control problem as an MDP and apply Q-learning to provide a control technique to preserve the response time according to the timing requirements. We evaluate the efficacy of the approach through multiple experiments. The learning-based approaches generally require multiple learning trials to converge and stabilize the learned policy. Regarding this issue and the characteristics of soft and hard real-time systems, it is supposed that the proposed learning-based approach in its incremental learning fashion could be used in soft real-time systems. While the controller with the

converged policy, after training based on simulation environment, could be integrated into the hard real-time systems. Furthermore, the result values (the tuned values) of the control policy could be used as a feedback to correct the initial model of the system. Future directions of this study will be evaluating the efficacy of the approach in the industrial platforms, improving the training time and adaptation precision of the approach by modeling the state space as fuzzy state space and using cooperative agents to speed up the learning.

Acknowledgment

This research has been funded by the ITEA3 initiative TESTOMAT Project (www.testomatproject.eu) through Vinnova and by KKS through the TOCSYC project).

Bibliography

[1] Giordano A Kaczynski, Lucia Lo Bello, and Thomas Nolte. Deriving exact stochastic response times of periodic tasks in hybrid priority-driven soft real-time systems. In *2007 IEEE Conference on Emerging Technologies and Factory Automation (EFTA 2007)*, pages 101–110. IEEE, 2007.

[2] Sorin Manolache, Petru Eles, and Zebo Peng. Schedulability analysis of applications with stochastic task execution times. *ACM Transactions on Embedded Computing Systems (TECS)*, 3(4):706–735, 2004.

[3] Elena Fersman, Pavel Krcal, Paul Pettersson, and Wang Yi. Task automata: Schedulability, decidability and undecidability. *Information and Computation*, 205(8):1149–1172, 2007.

[4] Mehrdad Saadatmand, Antonio Cicchetti, and Mikael Sjödin. Design of adaptive security mechanisms for real-time embedded systems. In *International Symposium on Engineering Secure Software and Systems*, pages 121–134. Springer, 2012.

[5] ANSI/IEEE. Standard glossary of software engineering terminology. 1991.

[6] Alwyn E Goodloe and Lee Pike. Monitoring distributed real-time systems: A survey and future directions. 2010.

[7] Mahshid Helali Moghadam, Mehrdad Saadatmand, Markus Borg, Markus Bohlin, and Björn Lisper. Learning-based self-adaptive assurance of timing properties in a real-time embedded system. In *2018 IEEE*

International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pages 77–80. IEEE, 2018.

[8] Richard S Sutton, Andrew G Barto, et al. *Introduction to reinforcement learning*, volume 135. MIT press Cambridge, 1998.

[9] IEC Iec. 61131-3: Programmable controllers–part 3: Programming languages. *International Standard, Second Edition, International Electrotechnical Commission, Geneva*, 1:2003, 2003.

[10] Andy Wellings, Gregory Bollella, Peter Dibble, and David Holmes. Cost enforcement and deadline monitoring in the real-time specification for java. In *Seventh IEEE International Symposium onObject-Oriented Real-Time Distributed Computing, 2004. Proceedings.*, pages 78–85. IEEE, 2004.

[11] Enrico Mezzetti, Marco Panunzio, and Tullio Vardanega. Preservation of timing properties with the ada ravenscar profile. In *International Conference on Reliable Software Technologies*, pages 153–166. Springer, 2010.

[12] Mehrdad Saadatmand, Mikael Sjödin, and Naveed Ul Mustafa. Monitoring capabilities of schedulers in model-driven development of real-time systems. In *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation (ETFA 2012)*, pages 1–10. IEEE, 2012.

[13] Nima Asadi, Mehrdad Saadatmand, and Mikael Sjödin. Run-time monitoring of timing constraints: A survey of methods and tools. In *the Eighth International Conference on Software Engineering Advances*, 2013.

[14] Joel Huselius and Johan Andersson. Model synthesis for real-time systems. In *Ninth European Conference on Software Maintenance and Reengineering*, pages 52–60. IEEE, 2005.

[15] Farnam Jahanian. Run-time monitoring of real-time systems. *Advances in Real-Time Systems. Prentice Hall*, 1995.

[16] Henrik Thane. Design for Deterministic Monitoring of Distributed Real-Time Systems, 2000. Technical Report ISSN 1404-3041 ISRN MDHMRTC- 23/2000-1-SE, Målardalen University.

[17] Nondini Das, Suchita Ganesan, Leo Jweda, Mojtaba Bagherzadeh, Nicolas Hili, and Juergen Dingel. Supporting the model-driven development of real-time embedded systems with run-time monitoring and animation via highly customizable code generation. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, pages 36–43. ACM, 2016.

[18] Ramy Medhat, Borzoo Bonakdarpour, Deepak Kumar, and Sebastian Fischmeister. Runtime monitoring of cyber-physical systems under timing and memory constraints. *ACM Transactions on Embedded Computing Systems (TECS)*, 14(4):79, 2015.

Chapter 12

Paper G: Makespan Reduction for Dynamic Workloads in Cluster-Based Data Grids Using Reinforcement Learning-Based Scheduling

Mahshid Helali Moghadam and Seyed Morteza Babamir
Journal of Computational Science, 24, 402-412, Elsevier 2018.

Abstract

Scheduling is one of the important problems within the scope of control and management in grid and cloud-based systems. Data grid still as a primary solution to process data-intensive tasks, deals with managing large amounts of distributed data in multiple nodes. In this paper, a two-phase learning-based scheduling is proposed for data-intensive tasks scheduling in cluster-based data grids. In the proposed approach, a hierarchical multi agent system, consisting of one global broker agent and several local agents, is applied to scheduling procedure in the cluster-based data grids. At the first step of the proposed approach, the global broker agent selects the cluster with the minimum data cost based on the data communication cost measure, then an adaptive policy based on Q-learning is used by the local agent of the selected cluster to schedule the task to the proper node of the cluster. The impacts of three action selection strategies have been investigated in the proposed approach, and the performance of different versions of the approach regarding different action selection strategies, has been evaluated under three types of workloads with heterogeneous tasks. Experimental results show that for dynamic workloads with varying task submission patterns, the proposed learning-based scheduling gives better performance compared to four common scheduling strategies, Queue Length (Shortest Queue), Access Cost, Queue Access Cost (QAC) and HCS, which use regular combinations of primary parameters such as, data communication cost and queue length. Applying a learning-based strategy provides the scheduling with more adaptability to the changing conditions in the environment.

12.1 Introduction

Grid computing is a distributed computing system, which enables the integrated and collaborative use of many heterogeneous resources owned by different organizations. It is still used for computation- and data-intensive processing. Data grid deals primarily with data-intensive applications. Many scientific and engineering problems require to access and process large amounts of distributed data [1, 2, 3]. In many application environments, there is a complex of heterogeneous tasks, which are quite different in terms of their sizes and their processing requirements. Studies show that the variability of task sizes is an important factor, which highly affects the performance of the scheduling [4]. Using adaptive control strategies in dynamic environments with varying conditions may be a proper solution to deal with changing features of the problem environment. Adaptive control is a type of control dealing with time-varying parameters. It does not need a priori knowledge about the uncertain parameters and involves a control method changing itself.

Learning automata is a machine-learning field considered as an adaptive control method [5]. Generally, in learning automata, the current action is selected based on the experiences collected from the environment. It may be in the domain of reinforcement learning (RL), if the environment is stochastic and can be modelled as a Markov Decision Process (MDP). In a grid system, the state of the system can be described by a random process, X_{t_n}, specifying the state of the system as a function of time. The state space of the system is known, and the scheduling of submitted tasks is conducted according to the current state of the system. Therefore, the random process describing the system is a Markov process and satisfies the Markov property as follows:

$$P\left(X_{t_n} = j \,|X_{t_{n-1}} = i_{n-1},\ X_{t_{n-2}} = i_{n-2}, \ldots\right) = P\left(X_{t_n} = j \,|X_{t_{n-1}} = i_{n-1}\right) \tag{12.1}$$

The state describing process is memoryless to the visited states in the past and to the time spent in each state. Overall, the process of task scheduling can be considered as an MDP in which various types of RL-based control scheduling can be applied to the system. In this paper, the issue of task scheduling in a cluster-based data grid using an adaptive learning-based scheduling is studied.

In the past decade, there has been a plenty of well-studied works on immediate task scheduling in grid systems. Several of the scheduling strategies were mainly intended for computation-intensive task scheduling in

computational grids. In addition to primary immediate task scheduling algorithms such as Opportunistic Load Balancing (OLB), Minimum Completion Time (MCT), Minimum Execution Time (MET), Switching Algorithm (SA), and K-Percent Best (KPA) [6, 7, 8], several different scheduling strategies have been also proposed such as a hybrid (GA/TS) independent task scheduling [9] for computational grids, a coloured petri net model for independent task scheduling in computational grids [10], a probabilistic task scheduling algorithm based on a discrete time Markov chain [11], an independent task scheduling based on Imperialist Competition Algorithm for grid systems [12] and a combined meta-heuristic (PSO with gravitational emulation local search) scheduling [13].

Regarding the ever-increasing needs for processing data-intensive tasks in scientific communities, several types of data-aware task scheduling strategies such as, HCS (Hierarchical Cluster Scheduling) [14], DIANA (data intensive and network aware scheduling) [15], RBHS (rank-based hybrid scheduling)[16], ASJS (adaptive scoring job scheduling) [17], CSS (Combined Scheduling Strategy) [18] and various scheduling strategies using meta-heuristic algorithms were proposed. Min-Min, Max-Min and Sufferage [7], RASA [19], FPLTF [20], and RRTS as the combination of Round Robin and Dynamic Time Slice (DTS) [21] are also some of the primary algorithms for batch mode scheduling of computation-intensive tasks. Regarding the dynamic workloads with a large complex of heterogeneous tasks, varying submission patterns and high heterogeneity of resources, the adaptive scheduling deserves to be used in many grid and cloud-based systems.

Reinforcement learning, as an important class of learning automata is a sort of learning based on iterative interaction with environment and analysis of the received reward signal. The learning is based on a type of trial-and-error search and delayed reward. RL is quite different from supervised learning, the most common learning in machine learning. Supervised learning is based on learning from training examples provided by an expert supervisor. But in an interactive environment, it is often impossible to have sufficient examples of desired behaviours covering all the state space. In these situations, it is highly beneficial if the learner is able to learn from its experiences. It is the exact benefit which is gained from reinforcement learning [22].

In general, reinforcement learning can be considered as an effective way of solving many types of optimal control problems, particularly the MDP

ones. So, many of the optimal control solving methods are considered as reinforcement learning solutions. However, many of them require partially complete knowledge of the environment. Three primary classes of reinforcement learning are Dynamic Programming (DP), Monte Carlo methods, and Temporal-Difference (TD) learning. DP methods need an accurate model of the environment and are computation-intensive. Monte Carlo solutions do not require complete knowledge of the environment and are simpler to be applied, but are not effective for step-by-step learning. The TD methods are free of model and well suited for incremental learning. It takes advantages of DP and Monte Carlo methods and learns directly from experiences without need for model [22]. Q-learning is one of the well-known algorithms in the category of reinforcement learning. It is an off-policy TD algorithm with early convergence. The efficiency of reinforcement learning has been shown in many dynamic application environments such as traffic control systems [23, 24], wireless sensor networks [25] and distributed control domains [26].

In this paper, a two-phase scheduling acting based on data awareness and using Q-learning algorithm was proposed for data-intensive task scheduling in a cluster-based data grid. In this study, a hierarchical multi-agent system consisting of two levels of broker agents was applied to the task scheduling in the cluster-based data grid. There is a global broker at the first level of the system, which makes decisions based on the data communication cost to select a suitable cluster. Then, at the second level the local brokers use a learning-based strategy to select the proper processing node.

Most of the previous studies used different parameters of data access cost, queue length or different combinations of them as primary optimization strategies for task scheduling in data grids. In this study, along with data communication cost, a Q-learning-based method has been used to improve the scheduling adaptation to dynamic changes and to high variability of submitted tasks. Exploiting an adaptive control method showed better performance than other common scheduling strategies for dynamic workloads with different task submission patterns.

This paper is organized as follows: In Section 12.2 a further overview of related works is presented. Section 12.3 discusses the learning concepts used in the proposed scheduling and presents the proposed two-phase learning-based scheduling for cluster-based data grids. Section 12.4 describes

the simulation environment, evaluation scenarios and experimental results. Section 12.5 discusses performance evaluation of the proposed scheduling in comparison with other task schedulings. Section 12.6 concludes the paper and presents some directions for further study.

12.2 Related Work

There are a number of related works for using RL-based methods in task scheduling in grids. In [27] a simple reinforcement learning was used for resource selection in a grid-like environment. In the proposed method, the learner keeps a score indicating the efficiency of the resource for each possible resource selection action. For scheduling a new submitted task, it selects the resource with the maximum score. Then, it receives a reinforcement signal and calculates a reward signal for the resource that has been selected. The simple proposed learning-based selection was applied to a distributed resource allocation in grid systems, but there was no explicit interaction between learners. The agents just learnt from expected response time of jobs as a reinforcement signal.

In [28] and [29] a dynamic resource selection called DRA-FRL was presented, which used RL in conjunction with a fuzzy rule base. A new RL-based method, Actor Critic Fuzzy Reinforcement Learning (ACFRL-2), was proposed to extend the application of RL to domains with large state-action space like dynamic resource allocation in grids or computer networks. Using RL in dynamic resource allocation is difficult, because the size of state space will increase dramatically with the number of resource types. In [30] a multi-agent reinforcement learning method called Ordinal Sharing Learning (OSL) was proposed to realize a learning-based coordination between agents with the aim of load balancing in large scale grids. The proposed learning was based on using an ordinal information sharing system with limited communication. In OSL, the agents make decisions based on shared utility tables.

In [31] a table-based RL called Sarsa was used for resource allocation in autonomic systems for a simple scenario where the state space is small. In [32] a multi-agent learning called Fair Action Learning (FAL) was proposed for online resource selection in a distributed sequential resource allocation problem (DSRAP). DSRAP refers to a resource allocation problem in a

cluster-based network. FAL is a policy gradient ascent algorithm to learn the local decision policy. In [33] a decision-making framework of agents was proposed. It consists of two learning problems: local resource allocation and task routing problem (choosing a neighbor to forward a task). In [33] a gradient ascent learning called Weighted Policy Learner (WPL) was proposed for a distributed task allocation in various applications like grids and web services.

In [34] an RL-based resource allocation combined with Artificial Neural Network (ANN) was proposed. This RL-based algorithm uses an ANN component to estimate the long-term reward through various iterations. In [35] an RL-based task scheduling in grid called Centralized Learning Distributed Scheduling (CLDS) was presented. It is a multi-agent scheduling consisting of one learner agent and several scheduler agents. In this scheduling, the scheduler agents submit their local rewards to the learner agent. The learner agent updates its global utility table and shares the updated utility table with the scheduler agents. The schedulers make decisions based on the updated utility table.

In this study, a two-phase adaptive task scheduling based on data-awareness and reinforcement learning for cluster-based data grids is proposed. It applies a hierarchical multi-agent system consisting of two levels of broker agents to task scheduling in a cluster-based data grid. The broker agent at the first level of the system makes decisions based on the data communication cost to select a suitable cluster. Then, at the second level, the local brokers use Q-learning to select the proper processing node. The proposed scheduling uses learning in conjunction with minimizing data communication cost.

12.3 Adaptive Scheduling Based on Reinforcement Learning

In large-scale grid systems, due to high variability and heterogeneity of submitted tasks and resource types, it is reasonable to have an adaptive scheduling to easily adapt to different changing conditions. To meet these requirements, an adaptive RL-based scheduling algorithm may be suited for this application environment. In the following, a brief discussion on the primary concepts of Q-learning as a model-free offline policy is presented,

then the details of the proposed scheduling are described.

12.3.1 Q-Learning: A Model-Free Reinforcement Learning

Reinforcement learning addresses learning from the experiences. It states how an agent can learn an optimal policy to reach its goals by sensing the environment, taking possible actions and receiving consequent reward. In the standard RL, the agent continually monitors the environment. It observes the current state of the environment and selects an action based on its experiences to be applied to the environment. The selected action may affect/change the current state. The agent receives a reinforcement signal in a form of scalar reward as a result of the selected action and state transition.

In order to reach a learning goal, the agent is responsible to find a policy mapping states to actions in a way which maximizes the long-term cumulative reward. Generally, the reinforcement learning problem is formulated as a Markov Decision Process which is described by a tuple S, A, T, r, where S is the state space of the environment, A is the set of possible actions, T is the transition function which specifies the transition probability in state s by action a, and r is the reward function specifying the immediate reward after a transition to a new state from state s by taking action a.

In general, for learning the optimal policy, the goal of the learner is to maximize the total reward. If the model of environment is completely known, the optimal policy can be derived by DP method. But, in many problems, the model environment is unknown and there is no accurate knowledge of the environment. In these conditions, RL learns the optimal policy by trial-and-error experiences in the state space. Q-Learning is an RL algorithm which learns an action-value function, Q, estimating the long-term action-value. The learned action-value simply approximates the optimal action-value, independent of the policy being followed. All the Q-values are stored in a Q-table. At each step of learning, the Q-value is updated by

$$Q\left(s_t, a_t\right) \leftarrow Q\left(s_t, a_t\right) + \alpha \left[r_{t+1} + \gamma \max_a Q\left(s_{t+1}, a\right) - Q\left(s_t, a_t\right)\right] \quad (12.2)$$

where $\alpha \in [0, 1]$ is the learning rate which specifies to what extent the agent learns new information and, $\gamma \in [0, 1]$ is the discount factor specifying the weight of future rewards in the action-value update and, r is the immediate

reward [22]. Value 1 for the learning rate means that the agent considers only the latest information while value 0 causes the agent not to learn anything. A value of 0 for discount factor shows that the agent only considers the current reward, while approaching value 1 will make the agent try for acquiring a long-term high reward. The procedural form of Q-Learning is shown in Algorithm 18.

Q-learning will learn the optimal policy regardless of the policy that the agent follows for action selection. Since it learns the optimal policy by any policy which is followed, it is called off-policy TD learning [22]. In many straightforward implementations of Q-learning, the learner chooses the action with maximum Q-value at each step. To enhance the performance of learning, an exploration strategy is usually added to the algorithm. One of the standard ways is to introduce an additional value, epsilon, $0 < \varepsilon < 1$. A value, between 0 and 1, is generated randomly, if it is less than epsilon, a random action is chosen (exploration), otherwise the action with maximum Q-value is selected (exploitation). Randomly choosing the next action in conjunction with giving a higher probability to the actions that currently have higher Q-values may be another way to improve the exploration of the learning.

Regarding the convergence of Q-learning, with a total number of transitions on the order of $Nlog(N)$, where N is the number of states, the agent can obtain the near-optimal policy [36].

Algorithm 18 Q-Learning

1. Set the learning rate and discount factor parameters;
2. Initialize $Q(s, a),\ \forall s \in \mathbb{S},\ \forall a \in \mathbb{A}$, arbitrarily;
3. While *Not (end of learning)* do:
   3.1 Initialize s;
   3.2 Repeat until reaching the terminal state:
      3.2.1 Select an action among all possible actions for the current state using the behavior policy (e.g., ε-greedy);
      3.2.2 Take the action;
      3.2.3 Observe the next state and receive reward;
      3.2.4 Set the new Q-value by:
      $Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$

12.3.2 A Two-Phase Adaptive Scheduling Based on Data Awareness and Reinforcement Learning

In data grids, reducing data access cost plays an important role to decrease the tasks completion time and improve performance of scheduling. Data access cost is the required time to access data to process tasks. The model of computing data access cost in a cluster-based data grid is defined as follows:

If task T_i is scheduled on node S_j, the data communication cost for accessing required data of T_i from S_j is given by [14]

$$DCC_{S_j}^{T_i} = \sum_{\text{For all } Fl_K \ in \ R_i} |Fl_k|/B_{jK} \tag{12.3}$$

where R_i is the list of required replicas/data files to process the task, Fl_K is the k^{th} data replica in R_i, $|Fl_k|$ is the size of the replica, B_{jK} is the network bandwidth between node S and the source node of the k^{th} replica. According to the cluster-based topology of the data grid, the required data may be moved to node S_j from various nodes in other clusters or within the same cluster. The model of data communication cost can be defined as follows:

$$DCC_{S_j}^{T_i} = Inter\_C_{T_i,S_j} + Intra\_C_{T_i,S_j} \tag{12.4}$$

where $Inter\_C_{T_i,S_j}$ is the data communication cost for accessing the required replicas residing in different clusters from the origin cluster of T_i and $Intra\_C_{T_i,S_j}$ is the data communication cost for the required replicas which reside in the local cluster. The wide-area links between clusters are usually much slower than local networks within a cluster. Thus, the cost of data communication between clusters is more than the cost of data communication within a cluster. As a result, reducing the number of data communications between clusters for accessing distributed data is of great importance for data-intensive task scheduling in cluster-based grid systems.

The proposed algorithm is an immediate task scheduling based on a two-step decision process in which the first step is to select the cluster which contains the node(s) with the lowest data communication cost. The decisive factor of total data communication cost would be the cost of data communication among different clusters. At the next step, to improve the scheduling adaptation to the dynamic environment including dynamic workloads with varying task intervals, and high heterogeneity of submitted

tasks and resources, an adaptive reinforcement learning-based task assignment policy using Q-learning is used to select a proper node in the selected cluster with the minimum data communication cost.

The proposed adaptive scheduling applies a hierarchical multi-agent system to the scheduling process. It consists of one global broker agent at the first level and several local broker agents at the second level within the clusters. The global broker selects the proper cluster with minimum data communication cost, then the local broker inside the selected cluster exploits an adaptive task assignment policy based on Q-learning for selecting the proper processing node. Figure 12.1 shows the general structure of the proposed scheduling. The local broker observes the current state, selects a proper node as a possible action, then receives the reward, and updates the Q-values.

The state of the environment is specified by a tuple $(n_{s1}, n_{s2}, ..., n_{sn})$ where n_{sn} represents the number of tasks (waiting and underprocessing) at node s_n of the local cluster. The possible actions are specified by a set of $\{a_{s1}, a_{s2}, ..., a_{sn}\}$ where a_{sn} represents the action of selecting node s_n. The reward function is defined by

$$reward = \frac{1}{Completion\_time} \tag{12.5}$$

where, $Completion\_time$ is the required time for the completion of a scheduled task. In the Q-learning based method, different action selection strategies can be used. In this study, two types of action selection methods will be used and investigated. One of them is a two-phase exploration-exploitation strategy based on the size of the submitted task set. The other one is ε-greedy algorithm. The completion time (response time) of task T_i on S_j, can be computed using the following equation:

$$CT_{S_j}^{T_i} = W_{T_i} + DAC_{S_j}^{T_i} + P_{T_i} \tag{12.6}$$

where W_{T_i} is the waiting time in the queue (queuing latency) to get the processing service, $DAC_{S_j}^{T_i}$ is the data access cost for task T_i, and P_{T_i} is the processing time of the data files. In the grid environment, in order to avoid data collision, the performance isolation for data transfer through connecting links should be guaranteed. Data access cost regarding waiting time for

getting permission to transfer data is defined as follows:

$$DAC_{S_j}^{T_i} = DCC_{S_j}^{T_i} + Delay\_DT_{T_i} \tag{12.7}$$

where $Delay\_DT_{T_i}$ is the delay of data transfer in network links and $DCC_{S_j}^{T_i}$ is the data communication cost. Figure 12.2 shows the steps of the proposed learning-based scheduling.

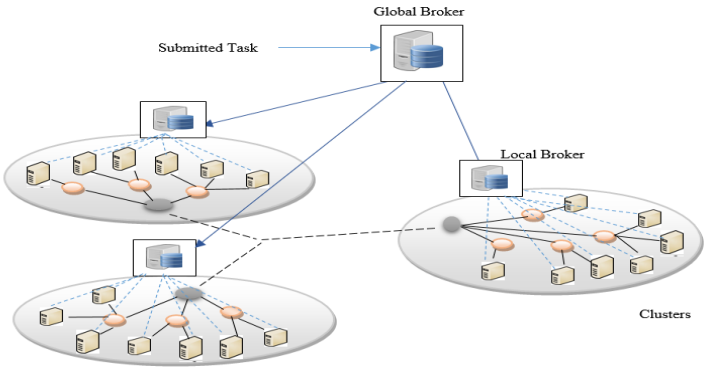


Figure 12.1: The structure of the proposed scheduling

Two-step adaptive task scheduling:
For each submitted task

A. Global broker
A.1 Calculates the inter-cluster data communication cost for executing the task in each cluster.
A.2 Selects the cluster with minimum data communication cost.

B. The local broker of the selected cluster
B.1 Observes the current state *(s)*.
B.2 Selects a host node (*action a*) using the action selection policy.
B.3 The submitted task is assigned to the selected node.
B.4 Receives a reward as a function of the completion time of the submitted task *(reward)*.
B.5 Updates the Q-value by

$$Q(s,a) \leftarrow Q(s,a) + \alpha[reward + \gamma \max_{a} Q(s',a) - Q(s,a)]$$

End For

Figure 12.2: Two-step learning-based task scheduling

12.4 Evaluation

In this study, OptorSim [3, 37], an open source data grid simulator, was used to simulate the proposed two-step learning-based scheduling with different action selection strategies and to perform the experimental evaluation. The purpose is to assess its performance compared with four baseline scheduling strategie sunder different workload patterns and analyze the effects of learning configuration parameters on the performance of the proposed scheduling. OptorSim, a java-based simulation tool developed under the European data grid project, supports simulation of data grids with different topologies, scheduling algorithms and various replication mechanisms.

In this study, the proposed adaptive scheduling has been implemented with three action selection strategies, and incorporated as anew scheduling into OptorSim. The performance of the scheduling is evaluated during three scenarios with three types of workloads, i.e., simple, random and CMS Data Challenge 2004 [3]. In each type of workload, several task sets with different number of tasks are submitted to the grid. The performance of the proposed scheduling algorithms is compared based on the makespan of the submitted task sets.

In the first step, through each scenario, the performance of the proposed scheduling using three action selection and three replication strategies under a specific type of workload was evaluated in terms of makespan. The first action selection strategy is a two-phase exploration-exploitation acting based on the size of the submitted task set. The other two action selection strategies use ε-greedy algorithm with $\varepsilon = 0.2$ and $\varepsilon = 0.5$ respectively.

At the next step, the sensitivity of the proposed scheduling algorithm to varying the learning parameters, i.e., learning and discount rates, is examined and the effects of the learning parameters on the scheduling performance are investigated.

The experiment environment including implementation details, the properties of simulation environment, performance and sensitivity analysis scenarios and simulation results will be described in the following Sections.

12.4.1 Simulation Environment

In OptorSim, data grid can be implemented with different topologies. The topology of the simulated cluster-based data grid and its structural properties are given in Figure 12.3. It consists of 3 clusters and 27 nodes in which 13 nodes have both computing and storage elements. The processing properties of the computing elements of all nodes are the same. The capacity of queue in computing elements was 200. A central node (node 17) is considered as a master storage node for storing master copies of all files. In the simulation environment, some nodes may have neither computing nor storage elements. They are used as network nodes. Connecting links between various nodes have different bandwidths. The network bandwidth between clusters is 500Mb/s and the band-width inside a cluster is 1000Mb/s for connections between nodes and first level switches, and is 2000Mb/s for connections between switches.

During the experiments, there were 300 initial files distributed randomly to the nodes of the grid. The size of a single file was 1GB. During the simulation, tasks were randomly selected from 30 task types based on the selection probability of each type. Each task type had the same probability of being selected. Each type of tasks requires different number of files.

In order to simulate the conditions of a real application environment and to provide high variability for submitted tasks, the distribution of task size was considered as Pareto, a heavy-tailed distribution. In this distribution, most of the task types had small size, i.e., required only a few number of files, and the remaining few types were of large size.

At the first step of evaluation, three scenarios using three types of workloads were arranged to evaluate the performance of the proposed learning-based scheduling in comparison to four baseline scheduling strategies. In each scenario, several sets of tasks with different numbers of tasks from 100 to 1200 were submitted to the data grid according to a specific submission pattern stated in the scenario.

Tasks were placed in the nodes according to the selected scheduling immediately after their arrival. Three replacement mechanisms such as Least Recently Used (LRU), Least Frequently Used (LFU) and Eco Model Optimizer (Binomial) were also used to manage the storage space for storing new replicas in the nodes during the task execution. LRU replicates the new required replica. It deletes the oldest file if there is no enough storage space

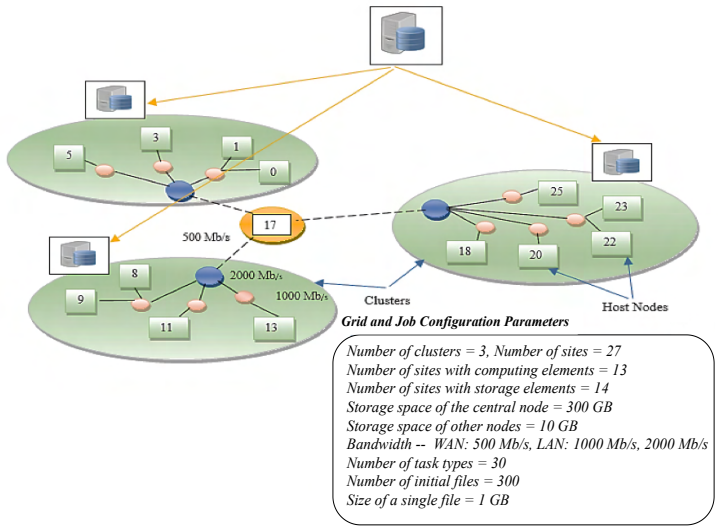


Figure 12.3: Topology of the simulated cluster-based data grid

for the new replica. LFU replicates the new replica while deleting the least frequently accessed file if there is no enough space. Eco Model Optimizer (Binomial) is a built-in replication optimizer in OptorSim which replicates the data file if deleting the least valuable file is economically beneficial according to binomial prediction function.

At the second step of experimental evaluation, a sensitivity analysis scenario under CMS DC04 workload pattern regarding using LFU replication strategy is performed to examine how the learning parameters can affect the performance of the proposed learning-based scheduling.

12.4.2 Experimental Results

For the purpose of performance evaluation, we designed a number of experiments to examine how the proposed scheduling can work under different workloads and also how its performance can be affected by the values of learning parameters, i.e., learning and discount rates. In this study, the performance of the proposed learning-based scheduling was compared with four baseline scheduling algorithms including Queue Length (Shortest

Queue), Access Cost, Queue Access Cost (QAC) and HCS during three performance analysis scenarios under different types of workloads. The mechanism of each baseline scheduling algorithm is as follows:

Queue Length schedules the task to the node with the shortest waiting queue. In Access Cost, the task is assigned to the node with the lowest data communication cost for accessing the required data which is unavailable in the processing node. Queue Access Cost uses a combination of queue length and data communication cost to schedule tasks. HCS scheduling [14] uses a two-step decision making mechanism based on data transfer cost and the length of waiting queue, in order to decide the host processing node of the submitted task.

In each performance analysis scenario, the behaviours of scheduling algorithms are examined under a variety of task sets with different numbers of tasks that are submitted according to a submission pattern. In the experiments of performance analysis scenarios, three types of action selection strategy were used in the proposed learning-based scheduling and the performance of the proposed scheduling using different action selection strategies was compared with baseline scheduling algorithms. The first version of the proposed scheduling uses a two-phase exploration-exploitation strategy acting based on the number of tasks in the submitted task set. In this strategy, the random action selection, i.e., exploration, is used for the first half of the tasks in the task set. Afterwards, the exploitation strategy, which selects the action with the highest Q-value, is used for the second half of the tasks. The ε-greedy algorithm with $\varepsilon = 0.2$ and $\varepsilon = 0.5$ was used as the action selection strategy in the second and the third version of the proposed scheduling. The ε-greedy algorithm is one of the standard ways to make trade-off between exploration and exploitation in the action selection. The proposed scheduling in the performance analysis experiments is run with the configuration of learning rate $\alpha = 0.1$, and discount rate $\gamma = 0.5$.

12.4.3 Performance Analysis

Scenario 1. In this scenario, a simple workload pattern was used to evaluate performance of scheduling algorithms. In the simple workload, tasks are submitted at regular intervals until all tasks have been submitted. A fixed interval between tasks is set by parameter *delay* in the simulation

environment. In this scenario, the tasks of each task set were submitted according to a simple submission pattern.

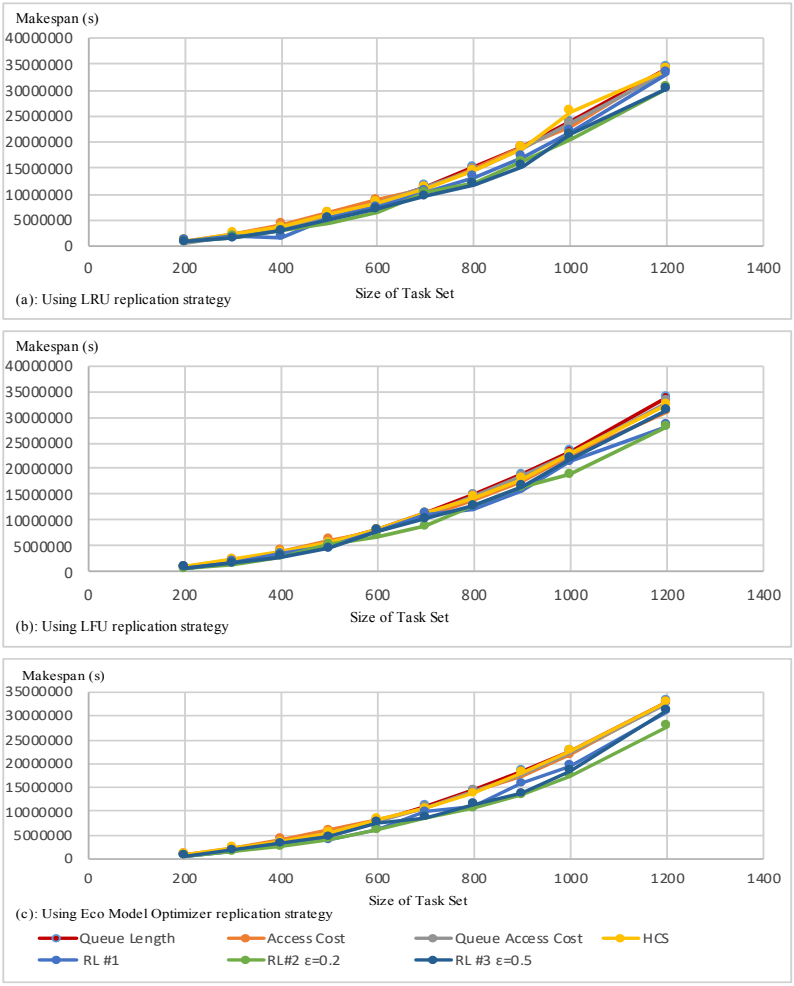


Figure 12.4: Scheduling algorithms' makespan regarding the simple workload

The parameter of delay between tasks was set to 1000ms. The performance of the proposed learning-based scheduling with different action

selection strategies was compared with baseline scheduling algorithms in terms of makespan of the task sets. Figure 12.4 shows the behavior of evaluated scheduling algorithms in terms of makespan under different task sets regarding using LRU, LFU, and Eco Model Optimizer replication strategies respectively.

Scenario 2. In the second performance evaluation scenario, the tasks were submitted according to a random workload pattern. Random workload uses uniformly random values for task intervals. The random intervals are between zero and twice the task delay parameter. In this scenario, the parameter of delay was set to 1000ms as well. Figure 12.5 shows the makespan of the various task sets scheduled by different scheduling algorithms while LRU, LFU and Eco Model Optimizer were used as replication algorithms respectively.

Scenario 3. In the third performance evaluation scenario, CMSDC04 pattern was applied to task submission. This workload pattern uses a Gaussian distribution for submitted tasks. Figure 12.6 presents the makespan of the various task sets scheduled by the algorithms with regard to the use of LRU, LFU, and Eco Model Optimizer as replication strategies.

12.4.4 Sensitivity Analysis

The behavior of the learning-based scheduling can be affected by varying the learning rate (α) and discount rate (γ) as learning parameters. Two experiments were performed to analyze the effects of learning parameters on the performance of learning-based scheduling. The sensitivity analysis experiments are done with the second type of the proposed learning-based scheduling which uses ε-greedy with $\varepsilon = 0.2$, under CMS DC04 workload pattern and regarding LFU replication. Each experiment involves varying the values of one of the learning parameters, while keeping the other one constant. The first experiment characterizes the effects of learning rate (α) and the second one involves analyzing the effects of discount rate (γ) on the performance of learning-based scheduling. To examine the results of varying learning parameters, the discount rate (γ) was set to 0.5 in the first experiment and we set the learning rate (α) to 0.1 during the second experiment. Figure 12.7 shows the impacts of varying learning parameters on makespan values of scheduling algorithms.

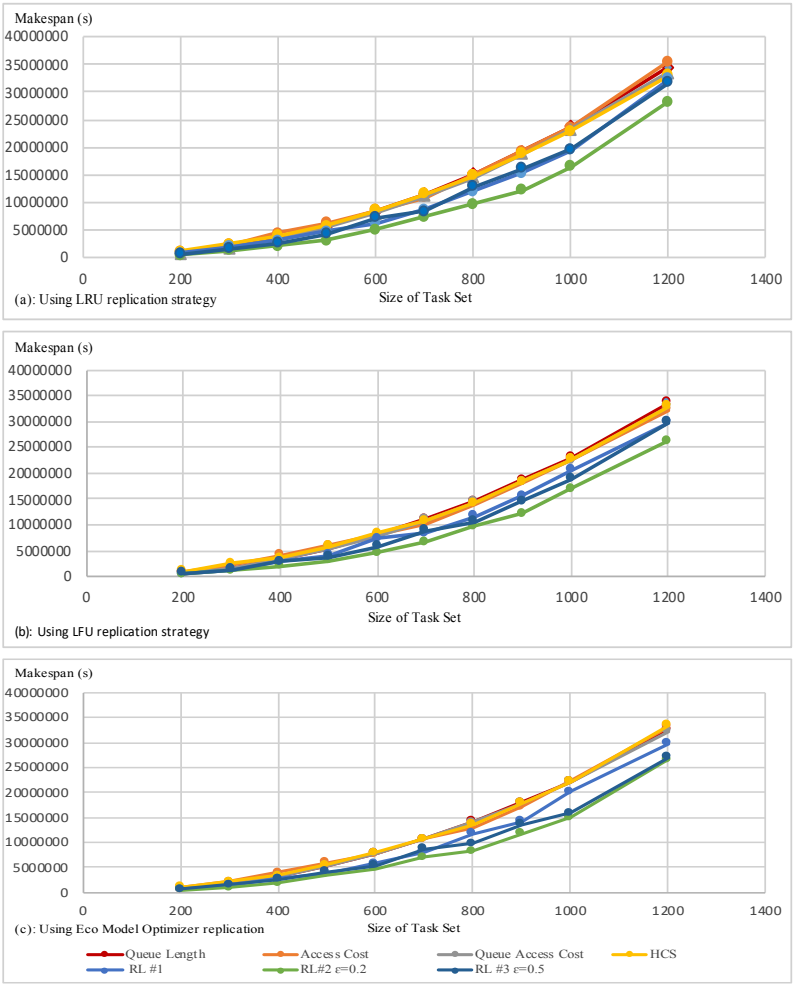


Figure 12.5: Scheduling algorithms' makespan regarding the random workload

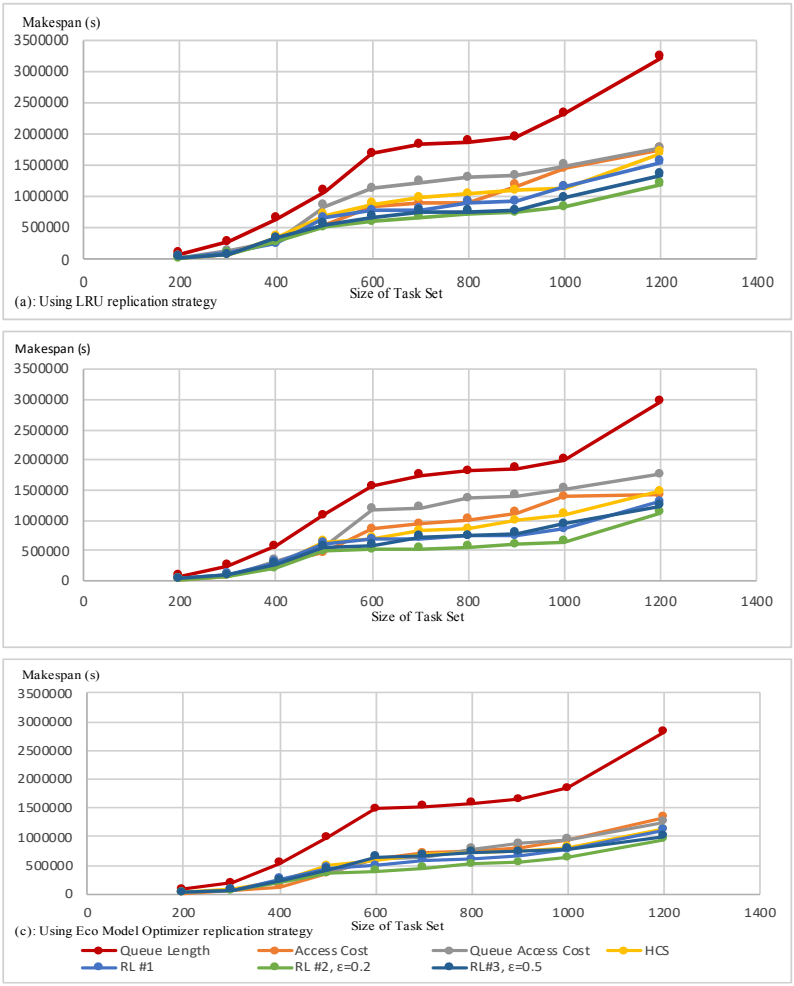


Figure 12.6: Scheduling algorithms' makespan regarding the CMS DC04 pattern

12.5 Discussion

The makespan plots of scheduling performance during the performance analysis experiments generally demonstrate lower values of makespan for submitted task sets which were allocated resources by learning-based scheduling than baseline scheduling algorithms. Consequently, the performance analysis results show that the learning-based algorithm adapts well to the workload pattern and status of the environment by scheduling tasks to the proper host nodes.

In the first scenario, simple workload was used for evaluating the performance of the proposed scheduling, using LRU, LFU, and Eco Model Optimizer replication strategies. In this scenario, according to Figure 12.4, the learning-based scheduling algorithms mainly scheduled the task sets with lower makespan than other base-line algorithms. The learning-based algorithm which uses ε-greedy with $\varepsilon = 0.2$ gives the lowest makespan for task sets, specifically with increase in the size of task sets. The performance improvement caused by all versions of the learning-based scheduling were more considerable when Eco Model Optimizer has been used as replication strategy. On the other hand, under the simple workload pattern, the data grid will not face critical conditions like spikes in the number of submitted tasks. Consequently, during the simple workload, the scheduling performance of the algorithms are generally close to each other.

In the second scenario, a random submission pattern was used for performance evaluation of scheduling algorithms. Using random workload, all versions of the learning-based scheduling led to lower makespan for different submitted task sets than base-line scheduling algorithms. The learning-based scheduling using ε-greedy with $\varepsilon = 0.22$ led to the lowest makespan for submitted task sets among all versions of the proposed scheduling. The learning-based scheduling using ε-greedy with $\varepsilon = 0.2$ results in the most improvement regardless of the replication strategy used in the experiment. In general, the learning-based scheduling primarily acts adaptively to the changes and performs independently of replication strategies. During the random workload, the effectiveness of the learning-based scheduling ability to adapt to the status of the grid and to learn the optimal policy is presented more than the simple workload. It led to a considerable performance improvement in terms of makespan measure.

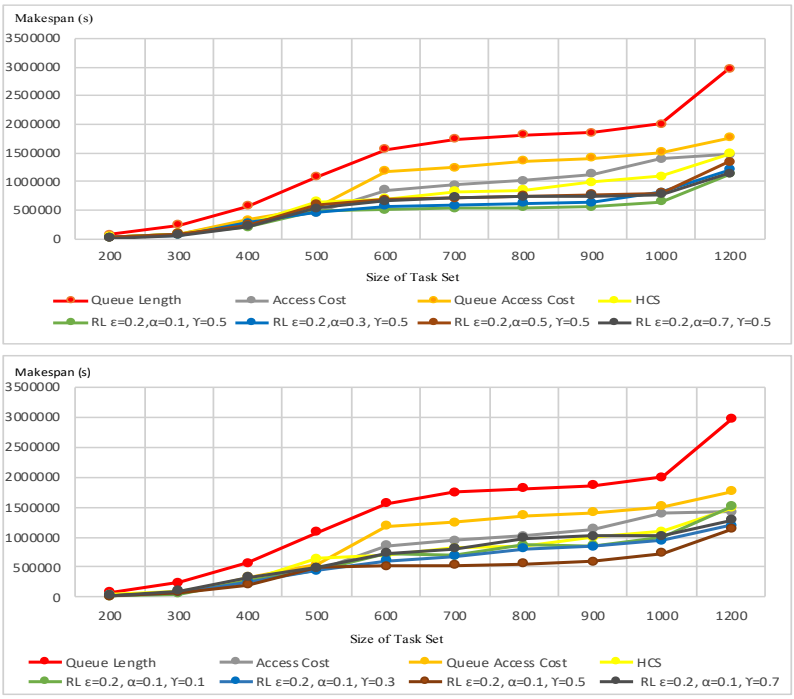


Figure 12.7: Learning parameters' impact on the learning-based scheduling performance

In the third performance evaluation scenario, CMS DC04 pattern was used as task submission patter for the task sets. CMS DC04 uses a Gaussian distribution model for the tasks submitted over one day. During the experiments of the third performance evaluation scenario, the learning-based scheduling algorithms also worked better than other scheduling algorithms, regardless of the replication strategy. The amount of performance improvement mainly rises, specifically with increase in the size of task sets. With the CMS DC04 workload pattern, the learning-based scheduling which uses ε-greedy with $\varepsilon = 0.2$ also gave the most improvement among the other versions of the proposed scheduling.

Generally, in all the performance analysis experiments with the learning configuration of learning rate $\alpha = 0.1$ and discount rate $\gamma = 0.5$, the

learning-based scheduling which uses ε-greedy with $\varepsilon = 0.2$ led to the most performance improvement. Using $\varepsilon = 0.2$ provides more exploitation than other action selection strategies and let the learner uses its learned experiences more.

With $\varepsilon = 0.2$ the local brokers act primarily based on the achieved experience stored in the Q-table, i.e., they select actions based on the Q-values. It implies that with a high probability, the learner selects an action with the highest utility value among the experienced actions, rather than a random action. Therefore, the contribution of the experience is more than simple exploration in the action selection. This lets the learners use their experience more than using random selection and exhibit a good play of learned policy for task scheduling. Almost, all experiments showed that the performance improvement of the learning-based scheduling using ε-greedy with $\varepsilon = 0.2$ is more considerable when there is an increase in the size of task sets; this is because the broker acts based on the learned policy which has been converged during more number of learning steps. In other words, the experience of the broker will be more accurate during the experiments with task sets of larger size.

Simulation results of the performance analysis experiments demonstrate that the learning-based scheduling can outperform other baseline scheduling strategies particularly under different workloads with changing features in dynamic environments. It can well adapt to the changing conditions given limited knowledge of the environment. It is also presented that using an action selection strategy with more tendency to exploitation leads to more performance improvement in the learning-based scheduling of different workloads.

In the sensitivity analysis experiments, the effects of varying learning parameters, i.e., learning and discount rates on the performance of learning-based scheduling are examined. The learning rate controls how fast the learner learns the policy, i.e., to what extent the new utility value affects the Q-value. The discount rate shows to what extent the learner concerns itself with maximizing the future rewards. Setting the learning rate to a high value causes the learner to consider only the new information and using a high value for the discount rate makes the learner take into account the future rewards strongly.

According to the simulation results of the sensitivity analysis in Figure

12.7, the baseline learning configuration of learning rate $\alpha = 0.1$ and discount rate $\gamma = 0.5$ leads to the best performance in terms of makespan for the selected learning-based scheduling. Since the problem environment is stochastic, setting the learning rate to a low value like 0.1 and using a balance between impacts of immediate and future rewards by setting the discount rate to 0.5, provides the best performance for the learning-based scheduling.

12.6 Conclusion and Future Work

In grid systems, heterogeneity of submitted tasks and workload unpredictability are some of the important barriers to task scheduling in changing environments. Thus in order to improve the performance of task scheduling in dynamic environments, in this study, a two-step adaptive task scheduling based on data awareness and reinforcement learning was proposed for cluster-based data grids. The proposed adaptive scheduling consists of one global broker agent and several local broker agents inside the clusters. At the first step of the proposed scheduling, the global broker selects the cluster with minimum data communication cost. At the second step, in order to make the scheduling adaptive to changing features of the environment, a reinforcement learning-based task assignment policy based on Q-learning is used by the local brokers to select a proper node in the cluster selected at the first step. According to the experimental results, the proposed learning-based scheduling gives better performance, in comparison with other scheduling strategies. The performance improvement of the proposed learning-based scheduling is more considerable with increase in the number of tasks in varying workloads. Setting the learning rate to a rather low value and putting a balance between the immediate and future rewards provide the best learning configuration for the learning-based scheduling in cluster-based data grids.

Applying cooperative multi-agent systems with cooperative learning to scheduling problems could be further directions for future study in the scope of applying machine learning techniques to control and management problems in grids and cloud-based environments.

Bibliography

[1] Fatos Xhafa and Ajith Abraham. Computational models and heuristic methods for grid scheduling problems. *Future generation computer systems*, 26(4):608–621, 2010.

[2] Ann Chervenak, Ian Foster, Carl Kesselman, Charles Salisbury, and Steven Tuecke. The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *Journal of network and computer applications*, 23(3):187–200, 2000.

[3] DG Cameron, RC Schiaffino, J Ferguson, P Millar, C Nicholson, K Stockinger, and F Zini. Optorsim v2. 0 installation and user guide, november 2004.

[4] Mor Harchol-Balter. *Performance modeling and design of computer systems: queueing theory in action*. Cambridge University Press, 2013.

[5] S Mehdi Vahidipour, Mohammad Reza Meybodi, and Mehdi Esnaashari. Learning automata-based adaptive petri net and its application to priority assignment in queuing systems with unknown parameters. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 45(10):1373–1384, 2015.

[6] Fatos Xhafa, Javier Carretero, Leonard Barolli, and Arjan Durresi. Immediate mode scheduling in grid systems. *International Journal of Web and Grid Services*, 3(2):219–236, 2007.

[7] Muthucumaru Maheswaran, Shoukat Ali, Howard Jay Siegel, Debra Hensgen, and Richard F Freund. Dynamic mapping of a class of

independent tasks onto heterogeneous computing systems. *Journal of parallel and distributed computing*, 59(2):107–131, 1999.

[8] Tracy D Braun, Howard Jay Siegel, Noah Beck, Ladislau L Bölöni, Muthucumaru Maheswaran, Albert I Reuther, James P Robertson, Mitchell D Theys, Bin Yao, Debra Hensgen, et al. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *Journal of Parallel and Distributed computing*, 61(6):810–837, 2001.

[9] Fatos Xhafa, Juan A Gonzalez, Keshav P Dahal, and Ajith Abraham. A GA (TS) hybrid algorithm for scheduling in computational grids. In *International Conference on Hybrid Artificial Intelligence Systems*, pages 285–292. Springer, 2009.

[10] Mohammad Abdollahi Azgomi and Reza Entezari-Maleki. Task scheduling modelling and reliability evaluation of grid services using coloured petri nets. *Future Generation Computer Systems*, 26(8):1141–1150, 2010.

[11] Reza Entezari-Maleki and Ali Movaghar. A probabilistic task scheduling method for grid environments. *Future Generation Computer Systems*, 28(3):513–524, 2012.

[12] Zahra Pooranian, Mohammad Shojafar, Bahman Javadi, and Ajith Abraham. Using imperialist competition algorithm for independent task scheduling in grid computing. *Journal of Intelligent & Fuzzy Systems*, 27(1):187–199, 2014.

[13] Zahra Pooranian, Mohammad Shojafar, Jemal H Abawajy, and Ajith Abraham. An efficient meta-heuristic algorithm for grid computing. *Journal of Combinatorial Optimization*, 30(3):413–434, 2015.

[14] Ruay-Shiung Chang, Jih-Sheng Chang, and Shin-Yi Lin. Job scheduling and data replication on data grids. *Future Generation Computer Systems*, 23(7):846–860, 2007.

[15] Richard McClatchey, Ashiq Anjum, Heinz Stockinger, Arshad Ali, Ian Willers, and Michael Thomas. Data intensive and network aware (DIANA) grid scheduling. *Journal of Grid computing*, 5(1):43–64, 2007.

[16] Mohsen Abdoli, Reza Entezari-Maleki, and Ali Movaghar. A rank-based hybrid algorithm for scheduling data-and computation-intensive jobs in grid environments. In *Intelligent Computing, Networking, and Informatics*, pages 785–796. Springer, 2014.

[17] Ruay-Shiung Chang, Chih-Yuan Lin, and Chun-Fu Lin. An adaptive scoring job scheduling algorithm for grid computing. *Information Sciences*, 207:79–89, 2012.

[18] Najme Mansouri, Gholam Hosein Dastghaibyfard, and Ehsan Mansouri. Combination of data replication and scheduling algorithm for improving data availability in data grids. *Journal of Network and Computer Applications*, 36(2):711–722, 2013.

[19] Parsa Saeed and R Entezari-Maleki. RASA: A new task scheduling algorithm in grid environment. *World Applied Sciences Journal of Special Issue of Computer and IT*, pages 152–160, 2009.

[20] Daniel A Menascé, Debanjan Saha, SCD Porto, Virgilio AF Almeida, and Satish K Tripathi. Static and dynamic processor scheduling disciplines in heterogeneous parallel architectures. *Journal of Parallel and Distributed Computing*, 28(1):1–18, 1995.

[21] Sanjaya Kumar Panda, Sourav Kumar Bhoi, and Pabitra Mohan Khilar. RRTS: A task scheduling algorithm to minimize makespan in grid environment. In *Proceedings of International Conference on Internet Computing and Information Communications*, pages 279–292. Springer, 2014.

[22] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 1998.

[23] Mohamed A Khamis and Walid Gomaa. Adaptive multi-objective reinforcement learning with hybrid exploration for traffic signal control based on cooperative multi-agent framework. *Engineering Applications of Artificial Intelligence*, 29:134–151, 2014.

[24] Erwin Walraven, Matthijs TJ Spaan, and Bram Bakker. Traffic flow optimization: A reinforcement learning approach. *Engineering Applications of Artificial Intelligence*, 52:203–212, 2016.

[25] Hasan AA Al-Rawi, Ming Ann Ng, and Kok-Lim Alvin Yau. Application of reinforcement learning to routing in distributed wireless networks: a review. *Artificial Intelligence Review*, 43(3):381–416, 2015.

[26] Lucian Bu, Robert Babu, Bart De Schutter, et al. A comprehensive survey of multiagent reinforcement learning. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 38(2):156–172, 2008.

[27] Aram Galstyan, Karl Czajkowski, and Kristina Lerman. Resource allocation in the grid with learning agents. *Journal of Grid Computing*, 3(1-2):91–100, 2005.

[28] David Vengerov. Multi-agent learning and coordination algorithms for distributed dynamic resource allocation. 2004.

[29] David Vengerov. A reinforcement learning approach to dynamic resource allocation. *Engineering Applications of Artificial Intelligence*, 20(3):383–390, 2007.

[30] Jun Wu, Xin Xu, Pengcheng Zhang, and Chunming Liu. A novel multi-agent reinforcement learning approach for job scheduling in grid computing. *Future Generation Computer Systems*, 27(5):430–439, 2011.

[31] Gerald Tesauro, Rajarshi Das, William E Walsh, and Jeffrey O Kephart. Utility-function-driven resource allocation in autonomic systems. In *Second International Conference on Autonomic Computing (ICAC'05)*, pages 342–343. IEEE, 2005.

[32] Chongjie Zhang, Victor Lesser, and Prashant Shenoy. A multi-agent learning approach to online distributed resource allocation. In *Twenty-First International Joint Conference on Artificial Intelligence*, 2009.

[33] Sherief Abdallah and Victor Lesser. Learning the task allocation game. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 850–857. ACM, 2006.

[34] Masnida Hussin, Nor Asilah Wati Abdul Hamid, and Khairul Azhar Kasmiran. Improving reliability in resource management through adaptive reinforcement learning for distributed systems. *Journal of parallel and distributed computing*, 75:93–100, 2015.

[35] Milad Moradi. A centralized reinforcement learning method for multi-agent job scheduling in grid. In *2016 6th International Conference on Computer and Knowledge Engineering (ICCKE)*, pages 171–176. IEEE, 2016.

[36] Michael J Kearns and Satinder P Singh. Finite-sample convergence rates for Q-learning and indirect algorithms. In *Advances in neural information processing systems*, pages 996–1002, 1999.

[37] William H Bell, David G Cameron, A Paul Millar, Luigi Capozza, Kurt Stockinger, and Floriano Zini. Optorsim: A grid simulator for studying dynamic data replication strategies. *The International Journal of High Performance Computing Applications*, 17(4):403–416, 2003.