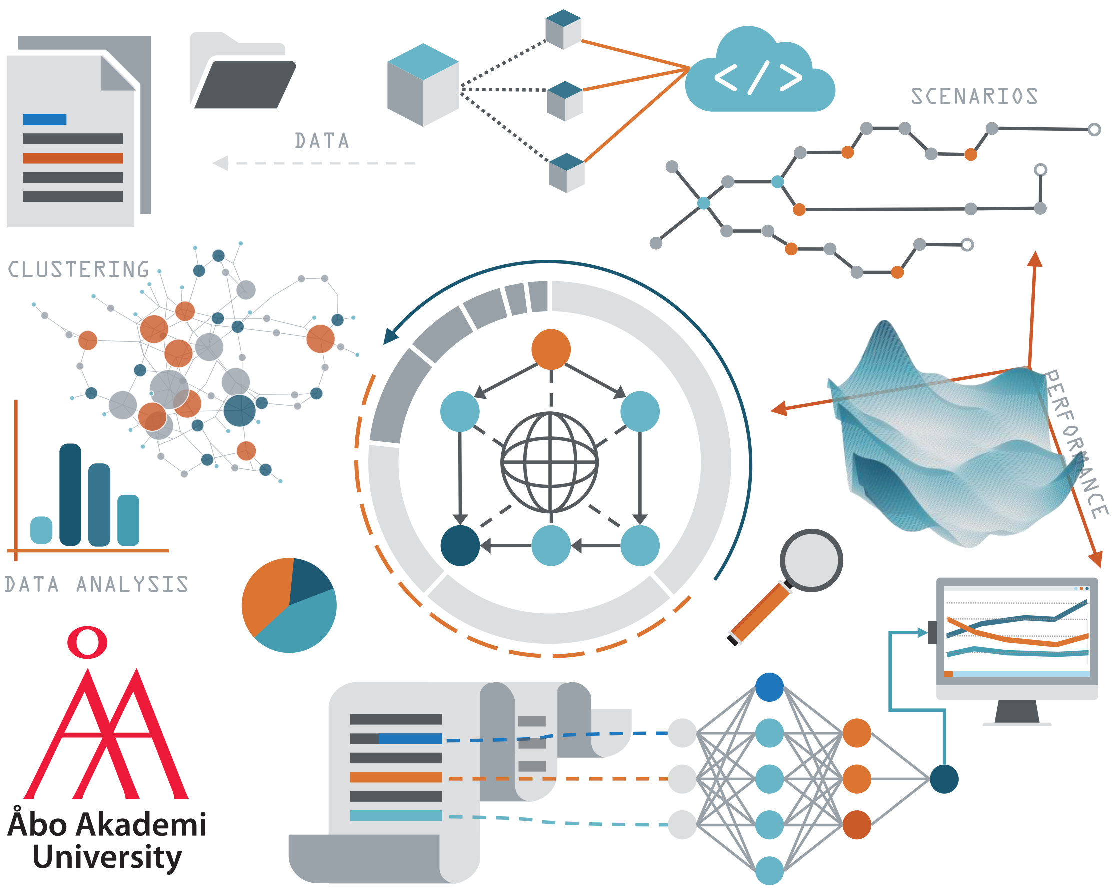
Tanwir Ahmad

Performance Exploration and Testing of Web-based Software Systems



Åbo Akademi University

Tanwir Ahmad
Born in Lahore, Pakistan

Previous studies and degrees
- Master in Computer Science from Government College University, Pakistan, 2011.
- Master in Software Engineering from Åbo Akademi University, Finland, 2014.

Performance Exploration and Testing of Web-based Software Systems

Tanwir Ahmad

Åbo Akademi University
Faculty of Science and Engineering
Vattenborgsvägen 3, 20500, Åbo

2020

Supervisors

Adjunct Professor, D.Sc. (Tech.) Dragoş Truşcan
Faculty of Science and Engineering
Åbo Akademi University
Vattenborgsvägen 3, 20500, Åbo
Finland

Professor Iván Porres
Faculty of Science and Engineering
Åbo Akademi University
Vattenborgsvägen 3, 20500, Åbo
Finland

Reviewers

Senior researcher, Dr. Mehrdad Saadatmand
Digital platforms
RISE - Research Institutes of Sweden
Stora Gatan 36, SE-722 12, Västerås
Sweden

Professor Vittorio Cortellessa
Department of Computer Science and Engineering, and Mathematics
University of L'Aquila
Palazzo Camponeschi, piazza Santa Margherita 2, 67100 L'Aquila
Italy

Opponent

Senior researcher, Dr. Mehrdad Saadatmand
Digital platforms
RISE - Research Institutes of Sweden
Stora Gatan 36, SE-722 12, Västerås
Sweden

ISBN: 978-952-12-3999-1 (printed)
ISBN: 978-952-12-4000-3 (digital)
Painosalama Oy, Turku, Finland 2020

Abstract

Modern society relies heavily on a wide range of inter-connected software systems for finance, energy distribution, communication, and transportation. The era of controlled communication in closed networks for limited purposes is over. Due to the adoption of the Internet, almost all financial, government, and social sectors rely heavily on web-based information systems. These systems need to be very fast and reliable, and should be able to support a vast number of concurrent users. As software users are immensely perceptive about the performance of the software system, the companies relying on web-based application systems for businesses strive to provide high-quality web services in order to stay competitive in the world-wide market. These companies may suffer a considerable loss of customers, which can detrimentally affect profits and revenues if the applications do not perform well in terms of functionality and performance. As various reports show that an application is more prone to fail due to performance issues rather than functional ones, it is very important that web application systems are rigorously tested for performance issues before deployment.

In this thesis, we propose a set of approaches for performance testing and exploration of web-based software systems. Although we target web-based software systems, our methods can be easily adapted to different types of software systems.

Our contributions fall into two categories: approaches for model-based performance testing and approaches for performance explorations of black-box systems with large input spaces. In the first category, as a first contribution, we provide model-based performance testing, where we generate realistic workloads using *Probabilistic Timed Automata* (PTA). During the load generation process, we monitor different *Key Performance Indicators* (KPIs) such as response times, throughput, memory, CPU, and disk. These KPIs are used to benchmark the performance of the system under test (SUT). As an extension of the first contribution, we provide an approach for extracting the workload models from server logs as an alternative to their manual creation based on the tester's experience.

In the second category of contributions, we are interested in exploring the performance of black-box software systems with large input spaces without prior knowledge of the domain. We propose different exploratory performance testing approaches to identify not only the worst user scenario with respect to a given workload model but also a set of input combinations that trigger performance issues and severely degrade the performance of software-intensive systems. Our first contribution, in this category, is an approach to explore the user scenario space randomly based on predefined mutation operators to find the worst user scenario. As a second contribution, we extend the previous work to present an exact approach that uses graphs-search algorithms and guarantees to find the worst user scenario. However, this approach does not scale well to large workload models with many loops. In our third contribution, we address the scalability issue of the exact approach and present an approach that employs genetic algorithms to identify a near-worst user scenario. As the last contribution, we provide an exploratory performance testing approach where we use reinforcement learning to explore a large input space in order to identify the input combinations that trigger performance issues in the SUT. This contribution is motivated by reports that show that almost two-thirds of the performance issues are detectable on certain input combinations. All the approaches discussed in this work are accompanied by tool support to automate the tedious tasks. The approaches have been evaluated against different web application case studies, but they can be extended to testing and exploring the performance of software-intensive systems in the other domains by adjusting their input artifacts such as workload models and input spaces with respect to those specific domains.

Sammanfattning

Dagen samhälle är starkt beroende av många olika slags sammankopplade mjukvarusystem för hantering av marknader, energi distribution, telekommunikation och logistik. Tidsåldern för kontrollerad kommunikation i slutna nätverk för begränsade syften gått mot sitt slut. I och med introduktionen av Intenet, har nästan alla finans- och myndighets- och välfärdssektorer blivit djupt beroende av webb baserade informationssystem. Dessa system måste vara mycket snabba och pålitliga och borde kunna hantera ett stort antal samtidiga användare. Eftersom mjukvaruanvändare är mycket uppmärksamma på mjukvarusystems prestanda, försöker företag som förlitar sig på webb baserade tjänster att erbjuda webb tjänster av hög kvalitet för att kunna hållas konkurrenskraftiga på den globala marknaden. Dessa företag kan förlora en stor andel av sina kunder om tjänsterna de erbjuder inte uppfyller användarnas krav på funktionalitet och prestanda, med konsekvens att företagen kan gå miste om viktiga intäkter. Det är mycket viktigt att system erbjuder webb tjänster är testade med avseende på prestanda problem före gruppering, med motiveringen att det finns ett antal rapporter som visar att mjukvaru applikationer tenderar att misslyckas på grund av prestanda problem snarare än funktionella problem.

I denna avhandling lägger vi fram ett antal tillvägagångssätt för att testa och utforska prestanda hos mjukvarusystem som erbjuder webb baserate tjänster. Även om tillvägagångssätten är ämnade för webb baserade mjukvaru system, kan våra metoder lätt anpassas för andra typer av mjukvarusystem.

Våra kontributioner kan indelas i två kategorier: tillvägagångssätt för modell-baserad prestanda test och tillvägagångssätt för utforskning av black-box system med stora indatarymder. Från den första kategorin, som den första kontributionen, tillhandhåller vi modell-baserad prestanda testning, där vi genererar realistiska arbetsbelastningar för test-systemet med hjälp av Probabilistic Timed Automata (PTA). Under genereringen av arbetsbelastningen övervakar vi Key Performance Indicators (KPIs), såsom svarstid, genomströmning, minnesanvändning, processoranvändning och användning av lagringsmedie. Dessa KPIn använder för att mäta prestandautgångsläget för systemet som håller på att testas (System Under Test; SUT). Som en utökning av den första kontributionen, erbjuder vi ett tillvägagångssätt för att extrahera arbetsbelastningsmodeller från en servers loggfiler, som en alternativ till att skapa dem manuellt utgående ifrån personen som utförs testens erfarenhet.

I den andra kontributionskategorin är vi intresserade av att utforska prestandan för black-box mjukvarusystem med stor indatarymder utan att ha någon a priori kunskapen om mjukvarusystemets applikationsdomän. Vi tillhandahåller olika utforskande tillvägagångssätt för testning av prestnada, for att identifiera, inte bara den värsta tänkbara användarscenariot i avseende på en given arbetsbelastningsmodell, men också vilken kombination av indata som orsakar prestanda problem genom att gravt nedsätta prestandan av mjukvaruintensiva system. Vår första kontribution i denna kategori, är ett tillvägagångssätt för att utforska användarscenariorymden slumpmässigt med fördefiniera muteringsoperatorer för att hitta det värsta möjliga användarscenariet. I den andra kontributionen utökar vi den första kontributionen för att tillhandahålla ett exakt tillvägagångssätt som använder sig av grafsökningsalgoritmer som garanterat kan hitta det värsta tänkbara användarscenariot. En nackdel med detta tillvägagångssätt är att den inte lämpar sig för stora arbetsbelastningsmodeller som innehåller många loopar. Vår tredje kontribution är att ta i tu med den andra kontributionens brist i att inte kunna hantera stora arbetsbelastningsmodeller. Vi löser detta problem genom att använda oss av genetiska algoritmer för att identifiera användarscenarier som är nära det värsta tänkbara. Som den fjärde och sista kontributionen tillhandahåller vi en utforskande prestandatestningstillvägagångssätt var vi använder oss av förstärkningsinlärning för att utforska en stor indatarymd för att kunna identifiera de indata kombinationer som utlöser prestandaproblem i systemet som håller på att testas. Denna kontribution motiveras av rapporter som visar att uppmot två tredjedelar av prestandaproblemen kan detekteras från från specifika indata kombinationer. Med alla tillvägagångssätt som diskuteras i denna avhandling har verktygsstöd för att automatisera de tråkiga och långdragna uppgifterna. Tillvägagångssätten har jämförts mot olika fallstudier för webb applikationer, men de kan anpassas för att kunna testa och utforska prestandan av mjukvaruintensiva system in andra domäner genom att justera indataartefakter som till exempel arbetsbelastningsmodeller och indatarymder.

Acknowledgements

I have not traveled this exciting yet at times challenging research journey on my own. There are many pillars that supported me and many amazing people who never left my side in this great expedition. This adventure would not have been possible nor enjoyable if I had to go through it alone. And, as the journey ends, I take the opportunity to express my deepest thanks to all who have contributed to make it so great for me.

I extend my warmest thanks to my supervisors, Adjunct Professor, D.Sc. (Tech.) Dragoş Truşcan and Professor Iván Porres. Dragoş gave me freedom, while persistently guiding me to progress efficiently. Knowing that he would always be there to back me up with academic and financial resources, I had the courage to explore new horizons of research. Thank you so much for believing in me and accepting me as a Ph.D. student. I am always amazed by your enthusiasm, commitment, and above all your inexplicable capacity to work so much, especially when it is for others! Many thanks go to my co-supervisor, Iván Porres, for the fruitful discussions, inputs, reviews, help and guidance, also for always finding nice ways to give comments. He made sure that I set my goals high. Together they have tirelessly guided me on the Ph.D. journey.

In addition, I would like to thank the external reviewers of this dissertation, Senior researcher, Dr. Mehrdad Saadatmand and Professor Vittorio Cortellessa, for their thorough reviews, which have aided me tremendously in improving this work. Furthermore, I am honored to have Senior researcher, Dr. Mehrdad Saadatmand as my opponent.

The research would not have been possible without financing, which provided me full-time engagement and participation in many inspiring workshops and conferences. For that, I am grateful to the Doctoral Programme in Information Technologies at Åbo Akademi University, the MegaModelling at Runtime (MegaM@Rt2) project funded by ECSEL Joint Undertaking, the Need for Speed (N4S) project funded by Tekes/DIGILE, the Practical Applications of Model-based technologies to continuous integration & testing methodologies (PAM) project funded by Tekes, the Nokia Foundation, and the infrastructure provided by the Faculty of Science and Engineering at the Åbo Akademi University. I thank all these organizations for providing the resources to create a productive environment in which to conduct research.

I am indebted to all the co-authors of the articles that form elements of this dissertation. I am especially grateful to Dragoş for providing me with insightful feedback on the articles and the dissertation despite having a tight schedule himself. I would also like to thank Pekka Tuuttila for his valuable contributions and feedback during our collaborative projects. In addition, Iván Porres, Fredrik Abbors, Junaid Iqbal, Adnan Ashraf, Andi Domi, and Faezeh Siavashi have been immensely helpful co-authors.

I would like to thank all my colleagues at my department, past and present, who have provided inspiration: Benjamin Byholm, Usman Sanwal, Luca Zelioli, Hergys Rexha, Valentin Soloviev, Denis Kotkov, Annamari Soini, Sébastien Lafond, Marina Waldén, Mats Aspnäs, Luigia Petre, Jan Westerholm, Jerker Björkqvist, and Johan Lilius. Very special thanks go to Wictor Lund for providing the translation of the abstract. While conducting my research, I have benefited from several intellectual software engineering-related discussions and brainstorming sessions with Jani Sainio, Evren Yurtesen, and Jesús Bravo. I would furthermore like to thank all of the friendly staff at the Åbo Akademi University for creating a pleasant work environment: Karl Rönnholm, Christel Engblom, Minna Carla, Tove Österroos, and Pia-Maria Kallio.

I would like to thank my dear friends Rameez, Freed, Junaid, Ali, Usman, Asghar, Taimor, Farhan, Haris, Jani, Markus, and Andi for all discussions, joyous moments, delicious dinners, and relaxing tea/coffee sessions, and for being there for me. Moreover, I want to thank my friends back home Shafqat, Zulfiqar, Kashif, Fahad, and Sani for always welcoming me on my visits to Pakistan, making it easy to recharge my batteries and enjoy my holidays.

Finally, and most importantly, I would like to express my deepest gratitude to my parents, brothers, and sisters for their affection, unconditional support, and guidance at every stage of my life. I dedicate my thesis to them, and I hope I have made them proud. Without them next to me, I could not have achieved this. Last, but not least, I want to thank my wife for the love and support. Thank you!

Tanwir Ahmad
Turku, December, 2020

List of publications included

P1. Abbors, F., Ahmad, T., Truscan, D., and Porres, I., 2013. *Model-based Performance Testing of Web Services using Probabilistic Timed Automata.* In Proceedings of the International Conference on Web Information Systems and Technologies (pp. 99-104). SciTePress.

P2. Abbors, F., Ahmad, T., Truscan, D., and Porres, I., 2013. *Performance Testing in the Cloud using MBPeT.* In Developing Cloud Software: Algorithms, Applications, and Tools (pp. 191-225). TUCS General Publication. Turku Centre For Computer Science (TUCS).

P3. Abbors, F., Truscan, D., and Ahmad, T., 2014. *Mining Web Server Logs for Creating Workload Models.* In Proceedings of the 9th International Conference on Software Technologies (pp. 131-150). Springer.

P4. Ahmad, T., Abbors, F., and Truscan, D., 2015. *Automatic Performance Space Exploration of Web Applications.* In Proceedings of the International Conference on the Economics of Grids, Clouds, Systems, and Services (pp. 223-235). GECON 2015. Lecture Notes in Computer Science, vol 9512. Springer.

P5. Ahmad, T. and Truscan, D., 2016. *Automatic Performance Space Exploration of Web Applications Using Genetic Algorithms.* In Proceedings of the 31st Annual ACM Symposium on Applied Computing (pp. 795-800). ACM.

P6. Ahmad, T., Truscan, D., and Porres, I., 2018. *Identifying worst-case user scenarios for performance testing of web applications using Markov-chain workload models.* Future Generation Computer Systems, 87, (pp. 910-920). Elsevier.

P7. Ahmad, T., Ashraf, A., Truscan, D., and Porres, I., 2019. *Exploratory Performance Testing Using Reinforcement Learning.* In Proceedings of the 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA). (pp. 156-163) IEEE.

List of publications relevant but not included

- Abbors, F., Ahmad, T., Truscan, D. and Porres, I., 2012. *MBPeT: a model-based performance testing tool.* In Proceedings of the 4th International Conference on Advances in System Testing and Validation Lifecycle (pp. 1-8).
- Abbors, F., Ahmad, T., Truscan, D. and Porres, I., 2013. *Model-based performance testing in the cloud using the MBPeT tool.* In Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering (ICPE) (pp. 423-424). ACM.
- Ahmad, T., Abbors, F. and Truscan, D., 2013. *Model-Based Performance Testing Using the MBPeT Tool.* In TUCS Technical Reports 1066. TUCS.
- Abbors, F., Truscan, D. and Ahmad, T., 2014. *An automated approach for creating workload models from server log data.* In Proceedings of the 9th International Conference on Software Engineering and Applications (ICSOFT-EA) (pp. 14-25). IEEE.
- Abbors, F., Truscan, D. and Ahmad, T., 2014. *Tool Support for Automated Workload Model Creation from Web Server Logs.* In TUCS Technical Reports 1066. Turku Centre For Computer Science (TUCS).
- Porres, I., Ahmad, T., Rexha, H., Lafond, S. and Truscan, D., 2020, October. *Automatic exploratory performance testing using a discriminator neural network.* In Proceedings of the International Conference on Software Testing, Verification and Validation Workshops (ICSTW) (pp. 105-113). IEEE.

List of publications not included

- Koskinen, M., Truscan, D., Ahmad, T. and Grönblom, N., 2013. *Combining model-based testing and continuous integration.* In Proceedings of the 8th International Conference on Software Engineering Advances (ICSEA) (pp. 65-71).

- Truscan, D., Ahmad, T., Siavashi, F. and Tuuttila, P., 2015. *A practical application of UPPAAL and DTRON for runtime verification.* In Proceedings of the 2nd International Workshop on Software Engineering Research and Industrial Practice (pp. 39-45). IEEE.

- Ahmad, T., Iqbal, J., Ashraf, A., Truscan, D. and Porres, I., 2019. *Model-based testing using UML activity diagrams: A systematic mapping study.* Computer Science Review, 33, (pp.98-112). Elsevier.

- Tran, C. H., Truscan, D., Ahmad, T., 2020. *Applying Test-driven Development to Evaluating Student Projects.* In Proceedings of the 6th International Conference on Higher Education Advances (HEAd'20). (pp. 1155-1163). UPV Press.

- Truscan, D., Ahmad, T. and Tran, C.H., 2020. *Applying Test-Driven Development for Improved Feedback and Automation of Grading in Academic Courses on Software Development.* In International Workshop on Frontiers in Software Engineering Education (pp. 310-323). Springer, Cham.

Acronyms

API	Application Programming Interface
AUT	Application Under Test
CBMGs	Customer Behavior Model Graphs
CPD	Change Probability Distribution operator
DRL	Deep Reinforcement Learning
DTMC	Discrete Time Markov Chain
EFSMs	Extended Finite State Machines
ET	Exploratory Testing
GA	Genetic Algorithm
GUI	Graphical User Interface
HTTP	HyperText Transfer Protocol
KPI	Key Performance Indicator
MBPeT	Model-based Performance Testing tool
MBT	Model-Based Testing
MTT	Modify Think Time operator
PerfX	Performance Exploration
PerfXRL	Performance Exploration using Reinforcement Learning
PTA	Probabilistic Timed Automata
PUT	Program Under Test
RL	Reinforcement Learning
SLAs	Service Level Agreements
SLOC	Source Lines of Code
SUT	System Under Test

Contents

Chapter 1

Introduction

"Most IT systems fail to meet expectations. They don't meet business goals and don't support users efficiently."

— Søren Lauesen [1]

In 1936, Turing [2] introduced the concept of a computer program even before the invention of digital computers. After twelve years, a first computer program was written for a digital computer [3]. The total storage of the computer was 32 words, each representing 31 binary digits. The program ran for 52 minutes and performed 3.5 million operations to compute the highest proper factor of 2^{18} [4]. This was the first computer program which was stored in the memory of the computer. Heretofore, computers were programmed by reconfiguring their electronic components manually. Tukey [5] used the term *software* for the first time in a context of programming and computation in 1958. Since then, the software has become an increasingly important and indispensable constituent of everyday life.

Modern society relies heavily on a wide range of inter-connected software systems for finance, energy distribution, communication, and transportation. In 2011, Andreessen [6] reported that over 2 billion people have access to the Internet. Owning to these significant technological advancements and growth in the information technologies, the size and complexity of the software systems are increasing exponentially. For example, Volvo reported that the size of software in their cars increases by the power of 10 every 5 to 7 years [7]. In 2010, the size of software in some cars was 10 million *Source Lines Of Code* (SLOC) which then escalated to 150 million SLOC just after six years [8]. NASA has also reported an ascending trend in the software size and complexity in avionics systems over time [9]. For example, Figure 1.1 illustrates the software complexity trend in terms of SLOC in both commercial and military aircraft. A similar trend regarding the software sizes can be observed in other industries [6, 10].

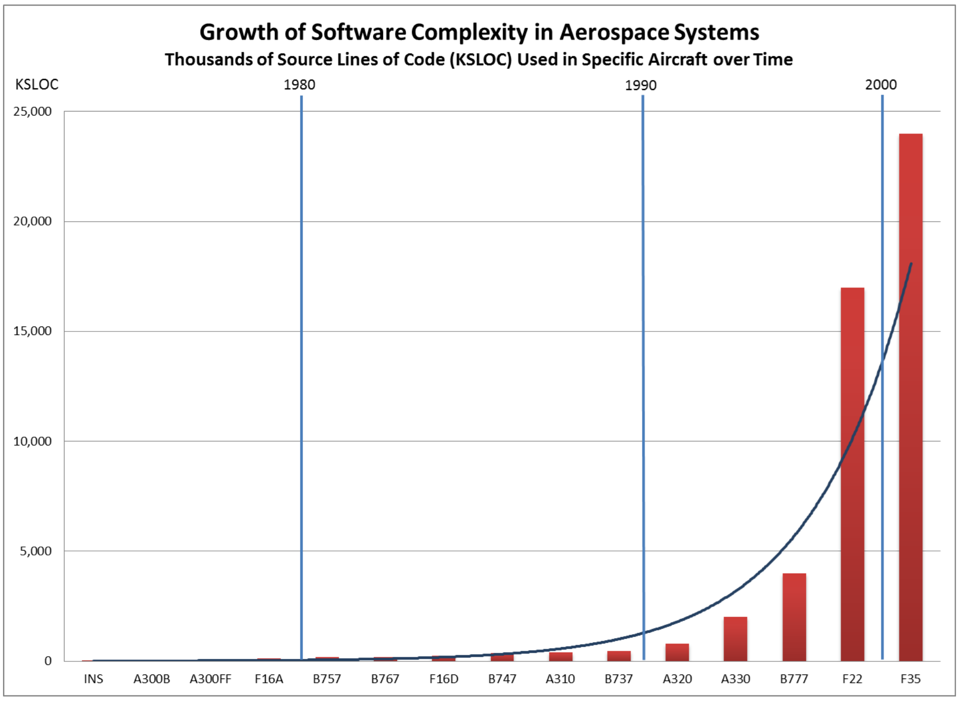


Figure 1.1: Growth in software complexity in terms of SLOC in commercial and military aircraft [11]

Owing to escalating demands for software products and services by the different industries, the software development organizations are compelled to deliver high-quality software artifacts which fulfill the customer requirements within a very short amount of time. For example, in 2011, Amazon[1], a leading online retailer, was updating its production system every 11.6 seconds [12]. IEEE Standard 1044 [13] defines *defect* as "an imperfection or deficiency in a work product where that work product does not meet its requirements or specifications." According to ISTQB [14], a defect is introduced in a software artifact by a software developer. A *failure* occurs due to the defects in the software implementation. Since the software development is a manual activity, it is inevitable to develop a software artifact without defects. For instance, during in-house testing, Microsoft [15] detects 10 to 20 defects per 1000 SLOC.

On 4 June 1996, Ariane 5 launcher exploded 40 seconds after lift-off due to a data conversion error in the software of inertial reference system [16]. The total estimated cost of the disaster was $370 million. In 1999, NASA lost $125 million Mars Climate Orbiter due to data unit conflict [17]. The

[1]`https://www.amazon.com/`

software controlling the thrusters of the orbiter calculated the data in English system; however, the navigation software expected the data to be in the metric system. Edward Weiler from NASA said [18], “The problem here was not the error, it was the failure of NASA’s systems engineering, and the checks and balances in our processes to detect the error. That’s why we lost the spacecraft.” This statement clearly indicates the need for processes to detect defects in the software systems. Furthermore, a recent study [19] reports that software failures cost globally $1.7 trillion and affected around 3.7 billion people in 2017.

Therefore, identifying software defects in the software system before it goes into production is very important [20]. *Software testing* has become a critical component of the software development cycle. It is a process of ensuring that the software conforms to requirement specifications [21]. The main objective of software testing is to find defects in the *System Under Test* (SUT) and establish confidence in the reliability and robustness of the SUT [22]. For example, Microsoft saved millions of dollars by uncovering software defects during the development of Windows 7 [23]. Software testing accounts for around 50% of the total budget and the time required for software development [24].

Software testing is a widely three-step process: (1) *constructing test cases*, (2) *running test cases against the SUT*, and (3) *evaluating results of the test cases.* A *test case* is a sequential combination of input and expected output values [22]. A test case is executed against the SUT in order to collect the actual output values of the SUT with respect to certain input values. The result of a test case is obtained by comparing the actual and expected output values, as shown in Figure 1.2. The result is “*pass*” if the actual and the expected output values are equal; otherwise, “*fail*”.

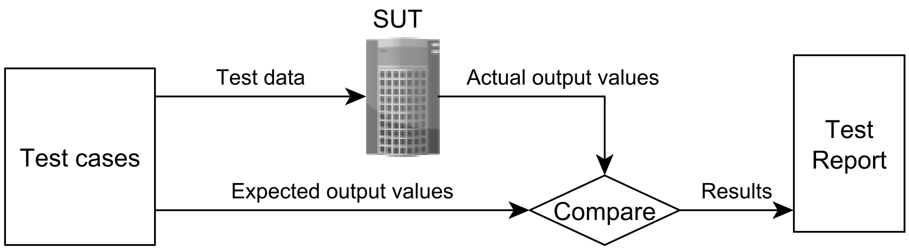


Figure 1.2: Software testing environment

There are two distinct software testing techniques used to create test cases [25]: *white-box* and *black-box testing.* In white-box testing, the test cases are derived by analyzing the internal structure or the source code of the

SUT. The main goal of white-box testing is to exercise each line of the source code of the SUT at least once. However, in most cases, it is not feasible to test 100% of the source code due to limited time and resources [26]. There are several inherent shortcomings to white-box testing. For example, white-box testing requires access to the source code of the SUT, which is not always possible. In order to conduct white-box testing effectively, the tester needs to have extensive domain knowledge about the SUT and the structure of the source code. On the other hand, in *black-box testing*, the tester treats the SUT as a black-box and tests it only through public interfaces (e.g., *Application Programming Interface* (API)). The test cases are generated based on the requirement specifications of the SUT. Therefore, the tester does not require the source code or any implementation details of the SUT for black-box testing.

Model-Based Testing (MBT) is a black-box testing approach where test cases are generated based on the abstract models that represent the behavior of the SUT [27]. Requirement specifications are used to construct these models [28]. One of the main benefits of MBT is that it facilitates automating or semi-automating the software testing process. In MBT, testers focus on building the models of the SUT instead of manually writing the test cases. Using MBT, testers manage to generate quality test cases with less time and efforts [27]. For instance, Microsoft found 10% more defects with MBT than manual testing in one of their software application [29]. Several case studies have demonstrated that MBT can reduce the testing cost by 30% [30].

Software testing techniques can be segmented into two broad categories: *functional testing* and *non-functional testing*. In functional testing, as the name suggests, we validate that the functional behavior of the SUT conforms to its requirement specifications by executing a set of test cases against the SUT. On the other hand, in non-functional testing, we focus on *how well* the SUT performs those functionalities. The goal of non-functional testing is to validate the SUT against its non-functional requirements, for example, performance, reliability, and security.

Performance testing is a type of non-functional testing, which evaluates the performance of the SUT when it is subjected to a controlled amount of workload [31]. A *performance test case* can be defined as a sequence of user actions along with the test input data for each action. During performance testing, we execute performance test cases and monitor *Key Performance Indicators* (KPIs) of the SUT [32, 33] such as:

1. *response time* — how fast the SUT responds to the user's requests;

2. *error rate* — the percentage of requests that fail or do not receive a correct response;

3. *throughput* — the number of requests per time unit the SUT can process;

4. *resource utilization* — how much resources are being utilized by the SUT when it is under a certain amount of workload.

These KPIs are used to establish the robustness and the performance level of the SUT. The main goal of performance testing is to identify *performance bugs* or *bottlenecks* [34] that negatively impact the performance of the SUT [33, 35].

Exploratory Testing (ET) is a software testing technique that is used to find software defects by learning and exploring the system behavior and being less dependent on the test documentation [36]. Unlike traditional software testing, tests are not derived from a pre-defined test plan. They are dynamically constructed, executed against the SUT, and updated based on the results of previously executed tests. During *exploratory performance testing*, software testers evaluate the performance of a software system with different user interaction sequences and input combinations in order to identify potential performance bugs. Typically, the tester is a domain expert with a good understanding of the SUT. These performance bugs are usually occurred due to the execution of inefficient code sequences. Finding such bugs in large-scale, complex software systems with large input spaces is a challenging task because these bugs are triggered on certain user interaction sequences and input combinations. A study reported that almost two-thirds of the performance bottlenecks are only detectable on specific input combinations [37]. Exploratory performance testing is mostly performed manually and requires rigorous domain knowledge and substantial efforts and time.

During the last two decades, we have witnessed tremendous development in Internet technologies. It has significantly altered how people communicate with each other, collaborate within a company, and utilize different services. The era of controlled communication in closed networks for limited purposes is over, due to the adoption of the Internet almost all financial, government, entertainment and social sectors rely heavily on web-based applications.

The architecture of web application systems has become very complicated in recent years [38]. These systems are being developed by integrating diverse software modules running on different computing units and communicating with one another through a network. Figure 1.3 shows the *3-tier* architecture of modern web applications where each tier performs a certain set of operations in the process of serving a user's request [39]. In order to provide a high level of reliability and availability, each tier is often deployed on its own collection of servers that work in parallel. The user employs a *browser* application, such as Mozilla Firefox[2], to access a web application. The browser sends a request to the *presentation tier* using the Hypertext Transfer Protocol (HTTP) [40]. The tier accepts the HTTP request and forwards it to the *application tier* for further processing. The later tier implements the core functionality of the web application. It processes the request and executes the requested operations. Further, it provides the results of the executed operations to the presentation tier, which formats the results according to a predefined layout before sending them back to the user. The *data tier* is used to store and fetch the data related to the web application. This tier is maintained by the application tier.

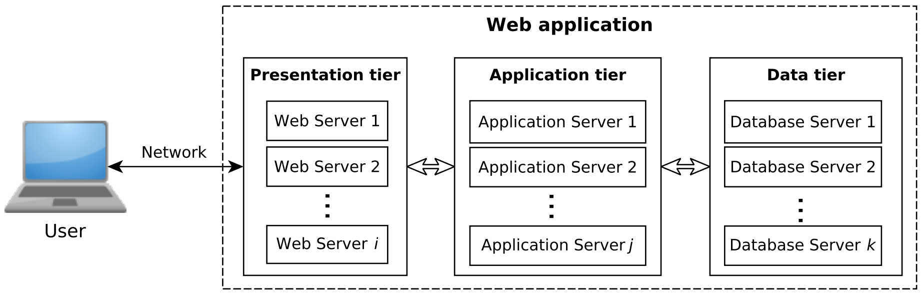


Figure 1.3: Modern web application architecture

Web application systems need to be fast and reliable, and they should be able to support the vast number of concurrent users [41]. The companies relying on web applications for business strive to provide high-quality web services in order to stay competitive in the worldwide market [42, 43]. These companies may suffer significant loss of customers that detrimentally affects profits and revenues if the applications do not perform up to quality standards or user expectations [44]. Therefore, it is very significant that the web application systems are rigorously tested for performance bottlenecks before deployment. Although we target web-based software systems in this thesis, our methods can be easily adapted to different types of software systems.

1.1 Motivation

Performance is considered as a significant metric to evaluate the quality of the software systems. The software users are immensely perceptive about the performance of the software system [45, 46]. For example, a study [47] reports that if a web application takes longer than 3 seconds to respond to

[2]`https://www.mozilla.org/firefox`

the user's request, 40% of the users will abandon it. Google, a leading search engine, notices a 20% drop in traffic and revenue due to 0.5 seconds delay in producing the search results [48]. Similarly, Amazon loses 1% in sales due to 100 milliseconds delay in home page generation [49]. Moreover, it is reported that the US economy lost $43.5 million due to performance-related issues of eCommerce applications [50].

Despite the above-mentioned facts, performance testing does not get the same level of importance as functional testing [31, 51]. As a result, the software systems fail more often due to performance-related problems than to functional ones [35]. A software project can get canceled if the software could not achieve the required level of performance; even though, it is functionally correct [52]. According to a study [53], only 22% of the software applications, which were not tested for performance, were managed to meet their performance objectives in production. Gunther [54] reported that a corporation lost $40 million because its new application cannot satisfy the service-level targets under a large amount of workload. Thus, ensuring whether a software system will satisfy its performance targets before it goes into production has become very important [55, 56]. Furthermore, fixing performance problems at the development stage is easier and more cost-effective than the later stages of software development [57].

It is reported that finding and fixing performance related defects is more challenging than in the case of functional ones [41, 58, 59]. The conjecture is that performance defects are more complex than functional defects, and most of the current software testing approaches focus on fixing functional defects [58].

1.2 Research Objectives and Aims

Generally, performance testing is largely performed using two of the traditional testing techniques: *Script based testing* and *Capture and replay testing*. In the former method, a tester manually writes user scenarios in a *test script* file. In order to generate the workload, the script file is executed in parallel to simulate concurrent users, as shown in Figure 1.4. In the latter method, instead of manually writing the test scripts, the tester records the interactions between the user and the SUT into a test script. This method automates the generation of test scripts. Both methods suffer from three major drawbacks: (1) the tester needs to write test scripts manually. Manual testing is an error-prone activity especially when dealing with large-scale, complex software systems [24]. (2) The design of the system and the customer requirements often change, which means that the test scripts need to be updated manually corresponding to the new modifications in the system or requirement specification [52]. This process is tedious and demands additional time and effort. (3) The workload generated by executing test scripts does not accurately represent the dynamic behavior of real users [60], which could lead to inconclusive performance test results [43].

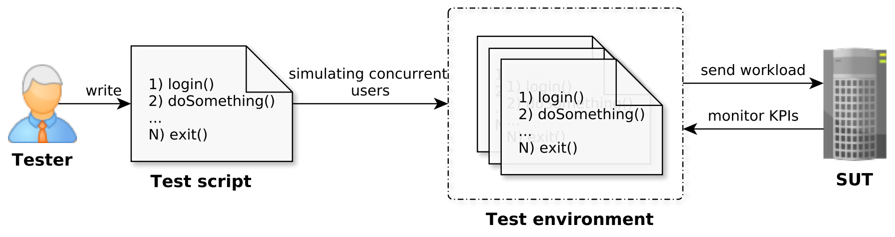


Figure 1.4: Script based testing

Furthermore, exhaustively testing a large-scale complex system for performance bugs has become inefficient and impractical because there can be numerous potential user scenarios, which cannot be tested cost-effectively within a reasonable amount of time. Thus, some infrequent user scenarios will remain untested. However, these untested user scenarios could deteriorate the performance of the software system or even crash the system if they occur. Consequently, there is a need to have an automatic performance exploration approach to investigate those rare potential user scenarios and their effects on the performance of the SUT.

A study [37] reported that almost two-thirds of the performance bottlenecks are only detectable on certain input combinations. However, finding those useful input combinations for performance test cases that can identify performance bottlenecks in a large-scale system within a feasible amount of time is a challenging task because there can be numerous input combinations. Thus, it is impractical for the testers to test each input combination. The problem becomes even more challenging when the SUT is a black-box where we cannot inspect the internal dynamics of the system.

This work attempts to address all the shortcomings mentioned above with the following objectives:

O1. Improve the performance testing process and results by generating a realistic workload against the SUT. This objective can be broken down into two sub-objectives:

  O1.1. Identify the suitable modeling formulation and right level of abstraction to capture the dynamic behavior of real users comprehensively.

  O1.2. Generate realistic workloads in order to benchmark the performance of the SUT accurately.

O2. Develop a performance exploration method that explores the user scenario space and identify those scenarios, which could degrade the performance of the SUT.

O3. Devise a methodology for identifying input combinations that can trigger resource-intensive computations on a black-box system.

1.3 Research Methodology

A *research method* represents a well-established procedure (such as algorithmic analysis, prototyping, conducting controlled experiments) to address and solve a research problem [61]. A *research methodology* is a framework, which consists of several research methods, rules and postulates employed by a particular research discipline to carry out, present, and publish the research [62].

The central component of our research methodology is the research process presented in Figure 1.5. In this thesis, the research process is adapted from the design science research methodology proposed by Peffers et al. [62] to conduct our software engineering research.

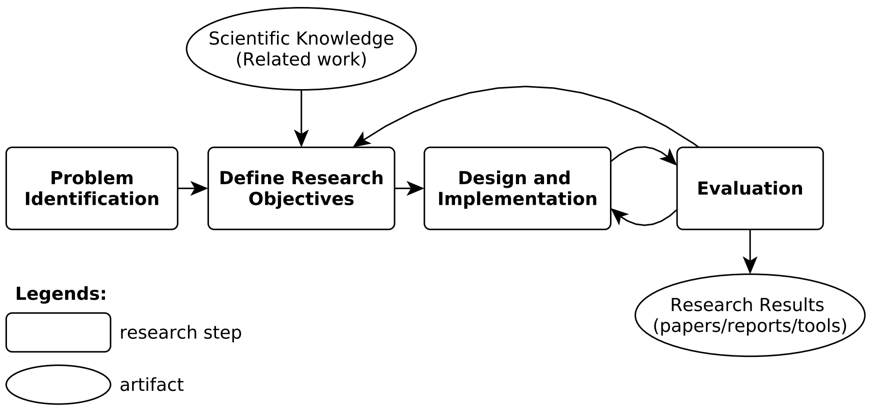

Scientific Knowledge (Related work)
Problem Identification
Define Research Objectives
Design and Implementation
Evaluation
Research Results (papers/reports/tools)
Legends:
research step
artifact

Figure 1.5: The cycle of our research process

Our research process starts with identifying a general research problem (presented in Section 1.1) originating from real-world observations. The ultimate goal of the research is to present a solution to this pragmatic research problem. During the next step, we decompose the research problem into various research objectives (outlined in Section 1.2), each expressing a certain aspect of the problem. Then, we analyze related methods, approaches, or theories in order to ensure that our research objectives have not been already addressed in the existing body of knowledge; otherwise, we update our objectives. This research step is an iterative procedure in which we repeatedly refine our research objectives based on the state-of-the-art literature.

During the *design and implementation* step, we build solutions (discussed in Section 1.4) to address the research objectives. At the last step of our research process, we evaluate the implemented solutions by conducting several controlled experiments. We compare the research results against the research objectives to check whether they satisfy the research problem identified at the first step of the research process. The last two steps can be repeated several times until we get satisfactory research results.

Once the *evaluation* step is completed, we summarize the research objectives, the solution, and the research results into several manuscripts (listed in Section 1.5) that are published in international conferences and journals referred to the topic of software testing. Additionally, we publish several technical reports [63, 64] detailing the technical aspects and preliminary results of the research.

1.4 Research Contributions

This section comprehensively outlines the four main contributions of this thesis. The first two contributions relate to performance testing of a software system, and the last two contributions concern performance exploration where we identify user scenarios and the input data, which could degrade the performance of the SUT.

Model user behavior for workload generation: Our first contribution mainly addresses the problem that, in most traditional performance testing approaches, user scenarios are created for most frequent uses, and they are implemented as static scripts. As we have discussed in the previous section, these scripts are difficult to maintain. Moreover, they do not accurately capture the dynamic user behavior. In **Paper 1**, we have investigated how probabilistic models can be used to represent user behavior such as Probabilistic Timed Automata [65] and Discrete Time Markov Chain [66] model. We have extended and defined modeling notations in order to capture the dynamic behavior of real users more accurately than the sequential scripts. We present three methods to construct models. The first method is manual and requires inspecting the requirement specifications, whereas the other two methods are fully automatic. The second method, proposed in **Paper 3**, analyzes the historical usage log to produce user models and the third method, presented in Chapter 2, generates a user model by monitoring the live interactions between a user and the SUT. This contribution addresses the **O1.1** objective.

Generate a realistic workload: For performance testing, we present an approach, discussed in **Paper 2**, for automated generation of a realistic workload. The method utilizes the user models discussed in the previous contribution. The workload is applied to the SUT in real-time. During the test session, we measure different KPIs such as response time, error rate, resource utilization and throughput, in order to benchmark the performance of the SUT. We have implemented the approach as a tool in Python[3], called *Model-based Performance Testing* (MBPeT). The tool follows a distributed architecture, which allows us to generate a large amount of workload by extensively parallelizing the load generation process among several computing nodes. Further, MBPeT performs a series of validation checks to ensure the syntactic and semantic correctness of the user models. The utility of the MBPeT is demonstrated by conducting several test sessions. The results show that MBPeT can effectively benchmark the performance of the SUT. This contribution addresses the **O1.2** objective.

Identify the worst-case user scenario in a user model: We describe three performance exploration approaches: *mutation-based* in **Paper 4**, *approximate* in **Papers 5-6**, and *exact* approach in **Paper 6**. These approaches utilize a given user model in order to find the worst user scenario in the model, which can degrade the performance of the SUT by creating the highest utilization of a given resource on the SUT. Such scenarios facilitate the testers to identify potential performance bottlenecks in the SUT. The mutation-based approach explores the user scenario space randomly by applying predefined mutation operators to a given user model. This approach does not guarantee to find the worst user scenario due to the random nature of it. In addition, it needs to be run for a significant amount of time in order to get better results. The exact approach is deterministic and always provides the worst user scenario; however, it does not scale well to large models with numerous loops. The approximate approach utilizes genetic algorithms to explore the user scenario space. It cannot always find the worst user scenario, but it can identify a near-worst user scenario faster than the other methods, even for large models. An assessment of the approaches shows that the identified user scenarios trigger more resource-expensive computations on the SUT as compared to the original models. This contribution addresses the **O2** objective.

Find input combinations that trigger performance bottlenecks As we have discussed at the beginning of this chapter, today's systems are becoming increasingly large and complex. As a result, they exhibit huge input

[3]`https://www.python.org/`

spaces with many input parameters and large ranges. Thus, it has become impractical to exhaustively test all possible input combinations in order to identify performance bottlenecks. To mitigate this problem, **Paper 7** introduces a methodology to explore a large space of input combinations to identify performance bottlenecks in a black-box system without any prior domain knowledge. The method only explores a subset of the input space and tries to find as many of those input combinations as possible that can trigger performance bottlenecks in the SUT. Our evaluation of the method indicates that it can be fully automated and is effective enough to detect 72% more bottlenecks than the random testing. This contribution undertakes the **O3** objective.

1.5 Thesis Overview

The thesis is divided into two main parts. The first part is an overall summary of the thesis, organized as follows. In Chapter 2, we present our model-based performance testing approach and different methods to produce workload models characterizing the dynamic behavior of the real users. We present the tool support of our approach and demonstrate its applicability by carrying out different experiments on an auction web application. Chapter 3 provides an overview of our performance exploration approach for inferring the worst-case user scenarios in a given workload model that can cause high resource utilization on the SUT, resulting in poor performance of the system. In the chapter, we also discuss an approach to explore a large input space to identify the input combinations that can trigger performance bottlenecks in a black-box system without any prior domain knowledge. The chapter also presents the tool support and the empirical evaluation of our approaches. Conclusions and directions for future work are given in Chapter 4 that concludes the first part of the thesis.

The second part comprises seven publications that enclose all thesis contributions. The included papers are the following:

Paper 1 - Abbors, F., Ahmad, T., Truscan, D. and Porres, I., 2013. *Model-based Performance Testing of Web Services using Probabilistic Timed Automata.* In International Conference on Web Information Systems and Technologies (pp. 99-104). SciTePress
In this paper, we present an approach for performance testing of web services in which we use abstract models, specified using Probabilistic Timed Automata, to describe how users interact with the system. The models are used to generate load against the system. The abstract actions from the model are sent in real-time to the system via an adapter. Different performance indicators are monitored during the test session and reported at the

end of the process. We exemplify with an auction web service case study on which we run several experiments.

Contribution Fredrik Abbors and I are the main authors of this paper. I was responsible for designing and implementing the tool. Together, Fredrik Abbors and I have defined the modeling notations to capture the dynamic behavior of the real users. The other authors have contributed with important ideas, discussions, and feedback.

Paper 2 - Abbors, F., Ahmad, T., Truscan, D. and Porres, I., 2013. *Performance Testing in the Cloud using MBPeT.* In Developing Cloud Software: Algorithms, Applications, and Tools (pp. 191-225). TUCS General Publication. Turku Centre For Computer Science (TUCS).
We present a model-based performance testing approach using the MBPeT tool. We use probabilistic timed automata to model the user profiles and to generate a synthetic workload. The MBPeT generates the load in a distributed fashion and applies it in real-time to the system under test, while measuring several key performance indicators, such as response time, throughput, error rate, etc. At the end of the test session, a detailed test report is provided. MBPeT has a distributed architecture and supports load generation distributed over multiple machines. New generation nodes are allocated dynamically during load generation. In this book chapter, we will present the MBPeT tool, its architecture, and demonstrate its applicability with a set of experiments on a case study. We also show that using abstract models for describing the user profiles allows us quickly experiment different load mixes and detect the worst case scenarios.

Contribution This paper was written with an equal contribution of the first two authors. I was responsible mainly for implementing the approach and conducting the experiments for the evaluation. The other authors have contributed with essential ideas, discussions, and feedback.

Paper 3 - Abbors, F., Truscan, D. and Ahmad, T., 2014. *Mining Web Server Logs for Creating Workload Models.* In International Conference on Software Technologies (pp. 131-150). Springer
We present a tool-supported approach where we used data mining techniques for automatically inferring workload models from historical web server log data. The workload models are represented as Probabilistic Timed Automata (PTA) and describe how users interact with the system. Via their stochastic nature, PTAs have more advantages over traditional scripting approaches which simply playback scripted or pre-recorded traces:

they are easier to create and maintain and achieve higher coverage of the tested application. The purpose of these models is to mimic real-user behavior as closely as possible when generating load. To show the validity and applicability of our proposed approach, we present several experiments. The results show, that the workload models automatically derived from web server logs are able to generate load similar with the one applied by real-users on the system and that they can be used as the starting point for performance testing process.

Contribution This paper was written with an equal contribution of all the authors. I was responsible mainly for implementing the approach and conducting the experiments for the evaluation.

Paper 4 - Ahmad, T., Abbors, F. and Truscan, D., 2015. *Automatic Performance Space Exploration of Web Applications.* In International Conference on the Economics of Grids, Clouds, Systems, and Services (pp. 223-235). Springer
We present a tool-supported performance exploration approach to investigate how potential user behavioral patterns affect the performance of the system under test. This work builds on our previous work in which we generate load from workload models describing the expected behavior of the users. We mutate a given workload model (specified using Probabilistic Timed Automata) in order to generate different potential user profiles. Each mutant is used for load generation using the MBPeT tool and the resource utilization of the system under test is monitored. At the end of an experiment, we analyze the mutants in two ways: cluster the mutants based on the resource utilization of the system under test and identify those mutants that satisfy the criteria of given objective functions.

Contribution I was the primary author of this paper. I proposed the mutation-based performance exploration approach and I developed the tool support. The other authors have contributed with important ideas, discussions, and feedback.

Paper 5 - Ahmad, T. and Truscan, D., 2016. *Automatic performance space exploration of web applications using genetic algorithms.* In Proceedings of the 31st Annual ACM Symposium on Applied Computing (pp. 795-800). ACM
We describe a tool-supported performance exploration approach in which we use genetic algorithms to find a potential user behavioural pattern that maximizes the resource utilization of the system under test. This work is built upon our previous work in which we generate load from workload

models that describe the expected behaviour of the users. In this paper, we evolve a given probabilistic workload model (specified as a Markov Chain Model) by optimizing the probability distribution of the edges in the model and generating different solutions. During the evolution, the solutions are ranked according to their fitness values. The solutions with the highest fitness are chosen as parent solutions for generating offsprings. At the end of an experiment, we select the best solution among all the generations. We validate our approach by generating load from both the original and the best solution model, and by comparing the resource utilization they create on the system under test.

Contribution I was the principal author of this paper. I contributed by modeling the user behavior using Markov Chain Model, developing a heuristic-based performance exploration approach to identify a potential user scenario that maximizes the resource utilization of the system under test, and implementing the tool. The other authors have contributed with important ideas, discussions, and feedback.

Paper 6 - Ahmad, T., Truscan, D. and Porres, I., 2018. *Identifying worst-case user scenarios for performance testing of web applications using Markov-chain workload models.* Future Generation Computer Systems, 87, (pp. 910-920). Elsevier
The poor performance of web-based systems can negatively impact the profitability and reputation of the companies that rely on them. Finding those user scenarios which can significantly degrade the performance of a web application is very important in order to take necessary countermeasures, for instance, allocating additional resources. Furthermore, one would like to understand how the system under test performs under increased workload triggered by the worst-case user scenarios. In our previous work, we have formalized the expected behavior of the users of web applications using probabilistic workload models and we have shown how to use such models to generate load against the system under test. As an extension, in this article, we suggest a performance space exploration approach for inferring the worst-case user scenario in a given workload model which has the potential to create the highest resource utilization on the system under test with respect to a given resource. We propose two alternative methods: one which identifies the exact worst-case user scenario of the given workload model, but it does not scale up for models with a large number of loops, and one which provides an approximate solution which, in turn, is more suitable for models with a large number of loops. We conduct several experiments to show that the identified user scenarios do provide in practice an increased resource utilization on the system under test when compared to the original

models.

Contribution I was the main author of this paper. I contributed by improving the previously proposed heuristic-based performance exploration approach, introducing a graph-based performance exploration approach to identify the worst user scenario that maximizes the resource utilization of the system under test, and implementing the tool. The other authors have contributed with important ideas, discussions, and feedback.

Paper 7 - Ahmad, T., Ashraf, A., Truscan, D. and Porres, I., 2019. *Exploratory Performance Testing Using Reinforcement Learning.* In 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA). (pp. 156-163) IEEE
Performance bottlenecks resulting in high response times and low throughput of software systems can ruin the reputation of the companies that rely on them. Almost two-thirds of performance bottlenecks are triggered on specific input values. However, finding the input values for performance test cases that can identify performance bottlenecks in a large-scale complex system within a reasonable amount of time is a cumbersome, cost-intensive, and time-consuming task. The reason is that there can be numerous combinations of test input values to explore in a limited amount of time. This paper presents PerfXRL, a novel approach for finding those combinations of input values that can reveal performance bottlenecks in the system under test. Our approach uses reinforcement learning to explore a large input space comprising combinations of input values and to learn to focus on those areas of the input space which trigger performance bottlenecks. The experimental results show that PerfXRL can detect 72% more performance bottlenecks than random testing by only exploring the 25% of the input space.

Contribution I was the main driver of this work. I proposed an approach for identifying input combinations that can reveal performance bottlenecks in the system under test using reinforcement learning, and I implemented the tool support. I wrote the major part of the paper. The other authors have contributed with important ideas, discussions, and feedback.

Chapter 2

Model-based Performance Testing of Web-based Software Systems

"*Essentially, all models are wrong, but some are useful.*"

— George E. P. Box [67]

Performance testing is a type of non-functional testing, which evaluates the performance of the SUT when it is subjected to a controlled amount of workload [31]. The main objective of performance testing is to evaluate the two crucial aspects of the SUT: *responsiveness* and *scalability*. The former specifies how instantly the SUT responds when it is subjected to a certain amount of workload. For instance, in case of a web application, we measure the average response time of all the requests made by a certain number of concurrent virtual users to the SUT. The performance of the SUT degrades as the average response time increases. Secondly, scalability determines the maximum amount of workload that the SUT can handle before it crashes or becomes unresponsive. These two metrics help infrastructure engineers in selecting suitable hardware and software platforms for the system [31, 33] but also in optimizing the software.

As we have mentioned in Chapter 1, performance testing is largely performed using two of the traditional testing techniques: *Script based testing* and *Capture and replay testing*. In the former method, the tester manually writes user scenarios in a test script file, whereas in the latter method, the tester records the interactions between the user and the SUT into the file. In order to generate the workload, the script file is executed in parallel to simulate concurrent users. There are several drawbacks to these methods including the fact that the design of the system and the requirement specifications often change, which means that the test scripts need to be modified manually to reflect the new changes [52]. This process is tedious and time-consuming. Further, the workload generated by executing test script files does not accurately represent the dynamic behavior of real users [60], which could lead to inconclusive performance test results [43].

In this work, we propose a model-based performance testing approach where we generate a workload from *Probabilistic Timed Automata* (PTA) [65] models characterizing the dynamic behavior of the real users. The abstract PTA models are easy to create and maintain, allowing quick and easy iteration cycles. During the load generation process, we monitor different KPIs, such as response times, throughput, memory, CPU, and disk utilization. These KPIs are used to benchmark the performance of the SUT.

The rest of the chapter is structured as follows: In Section 2.1, we introduce workload models and extend the probabilistic timed automata modeling notation for modeling user scenarios. Section 2.2 describes how workload models are produced. Section 2.3 presents several rules to check the consistency and correctness of the workload models. In Section 2.4, we briefly explain the model-based performance testing process. Section 2.5 provides an overview of the architecture of our tools. In Section 2.6, we investigate the applicability of our model-based performance testing approach by carrying out different experiments on an auction web application. Section 2.7 concisely presents the related work. We conclude in Section 2.8.

2.1 Modeling user behavior

In this section, we will introduce the *workload models* used for capturing the dynamic behavior of a group of real users in our model-based performance testing approach. We employ PTA [65] as workload models. PTA comprises a finite set of locations with clocks and probabilistic edges that connect automata locations to each other. A *clock* is a variable whose value spans over the non-negative real numbers. It proceeds at the same rate as time and can be reset. Time can only progress in any location in PTA as long as the *location invariant* holds while time elapses. A *probabilistic edge* can be selected non-deterministically only if its guard is satisfied by the current values of the clocks.

Figure 2.1 shows an example of a PTA model, which contains five locations and one clock variable, named X. The model in the figure represents the probabilistic behavior of a user interacting with a news forum web application. A user begins from the *initial* location. From this location, the user can either choose to *browse* through a list of news stories with 0.4 probability or to *search* for a news story with 0.6 probability. The user waits for either 3 or 4 time units before performing *browse* or *search* action, respectively, as specified by the *guards* on the edges from the intermediate nodes to the *browse_page* and the *search_page* location. These guards on the edges are used to specify *user think times*. If the *search* action is selected, the user waits for 3 time units and then transits to the *search_page* location. This location has a clock invariant: $X <= 10$ meaning that a user cannot stay at this location for more than 10 time units. Now, the user can either choose to *read* a news story with 0.8 probability or *quit* with 0.2 probability. In the former case, the user can *quit* after waiting for at least 20 time units.

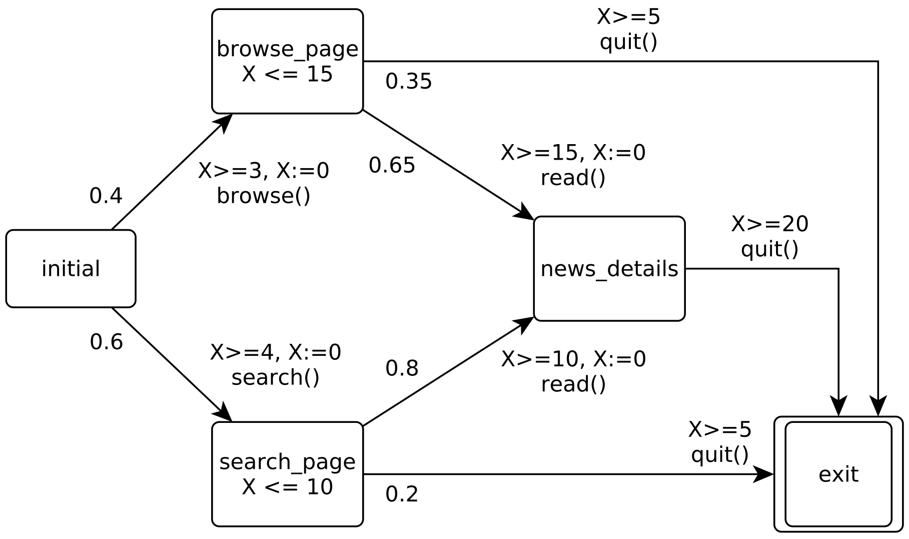


Figure 2.1: Example of a PTA (adapted from[65])

One can notice that PTA modeling notation is not very intuitive and visually concise when modeling user behavior. Therefore, we make the following modifications to the PTA modeling notation to make it more compact and easy to understand:

1. A user model have only one implicit clock variable, which is set to *zero* after every transition.

2. A label on an edge represents a probability, a user think time (specified in time units), and a user action separated by a / (i.e., forward slash) character.

3. Each location has a hidden clock invariant that is always *true*. This means that a user is not restricted to leave a location after a certain amount of time.

We have transformed the model in Figure 2.1 into the model in Figure 2.2 by applying our above transformation rules. The transformed model describes the same user behavior as the model in Figure 2.1. A user begins from the *1* location in the model. From this location, the user can either execute the *browse* action with 0.4 probability or the *search* action with 0.6 probability. The user waits for either 3 or 4 time units before performing *browse* or *search* action, respectively.

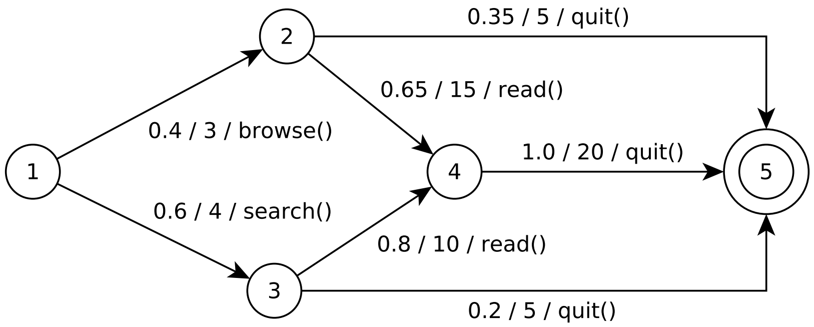


Figure 2.2: Example of a workload model with compact PTA notations

2.2 Constructing workload models

In this section, we present three methods to construct workload models. The first method is manual and requires inspecting the requirement specifications, whereas the other two methods are fully automated with tool support and require little to no manual efforts.

2.2.1 Analyzing requirement specifications

In this method, we need to determine the actions and the inputs to the SUT, arrival rates of the actions (which will be translated into *user think times* between two actions), different types of users, and what are the most common user scenarios performed by each user type. One can perceive a *user type* as a cluster of users behaving similarly. Each user type is materialized into a single workload model, as in Figure 2.2. Furthermore, in order to properly benchmark the performance of the SUT, we need to establish the performance expectations and objectives for the SUT such as what is the acceptable average response times for the actions, expected resource utilization, or intended throughput when the system is being accessed by a certain number of concurrent users.

The required information for workload model construction and performance objectives can be extracted from the requirement specifications, Service Level Agreements (SLAs), and system usage log using the procedure proposed by Calzarossa et al. [68]. On the other hand, if we do not have access to the artifacts mentioned previously (e.g., SLAs, usage log), we can interview personnel at the marketing department, current users of the SUT, or the business stakeholder to collect the required information for constructing the workload models. These interviews are considered beneficial because they provide insights into understanding what are the performance expectations and interests of the company [69].

2.2.2 Mining system's usage log

In this method, we parse the historical usage log of the SUT to generate workload models. A usage log file consists of a series of requests made to the SUT by different users at different points in time. For example, if the SUT is a web application, each line in the usage log file usually contains the time stamp, the IP Address of the user, the HTTP request method, the requested resource, etc.

This method can be decomposed into the following steps:

1. **Preprocessing usage log:** we filter out all the entries made by the *Web Crawlers* [70]. We only considered those entries in the log file for further processing, which correspond to the requests made by the real users.

2. **Identify user sessions:** we extract the required information such as the IP address, the requested resource, and the time stamp from each entry of the usage log file using the regular expressions. We assume that each IP address correlates to a different user. It is a reasonable assumption today since 3.5 billion people are accessing the Internet using their devices [71]. We use the IP address to segregate the requests in the log file into separate sequences for each user. These request sequences are further segmented into smaller chunks, called *sessions*, according to the given session timeout value. A *session timeout* is a time between two subsequent requests made by the same user, and it is utilized as a session border when it surpasses a certain threshold. Consequently, we can define a *user session* as a stream of requests made a user where no two successive requests are separated by more than the given session timeout value.

3. **Filtering user sessions:** we remove the less frequent user sessions using a Pareto probability density function [72] where we trim the tail of the distribution by a given *cut-off threshold* value. The conjecture

is that, in many cases, the most frequent user scenarios are the ones, which affect the performance of the SUT more significantly than the other scenarios. The tester can decide to include all the user scenarios to build a model, but it would not significantly affect the results of the load generation because the most frequent user scenarios in the model will be simulated more often than the other scenarios during the load generation. Furthermore, including only the most frequent user sessions keeps the resulted workload models concise and easy to understand.

4. **Clustering user sessions:** Using the K-means [73] clustering algorithm, we cluster the filtered user sessions based on their similarities. We construct a workload model by superimposing user sessions one by one from a cluster. We keep track of how many times a certain edge in the model have been used in order to determine the probability and the average think time for the edge. Each cluster of user sessions results in a workload model.

This method allows us to generate the workload models, but it requires the historical usage data of the system. The following method does not have that requirement.

Tool Support: Log2Model

Log2Model tool is implemented in the Python programming language to create workload models by mining usage log files (discussed in the previous section). The tool is capable of parsing the most commonly used log formats, such as the Apache HTTP Server log [74] and the Microsoft IIS log [75]. Additionally, the tester can define the custom log formats using regular expressions. Furthermore, the tester can interactively fine-tune the cut-off threshold value (as shown in Figure 2.3) to limit how many user sessions should be used to construct the workload models. In addition to the workload models, the tool generates the Python code for the test adapter code containing the mapping of every user action in the workload models.

2.2.3 Capturing user interactions

In this method, we generate a workload model by capturing the HTTP requests made by the user to the SUT in real-time. As the tester interacts with the web application, we capture the HTTP requests made by the browser to the web application and store them in a queue. Table 2.1 exemplifies a sequence of five HTTP requests. This method entails the following stages:

Creating user actions: At the first stage, we create and assign a user action to each request. A *user action* can be considered as an abstract

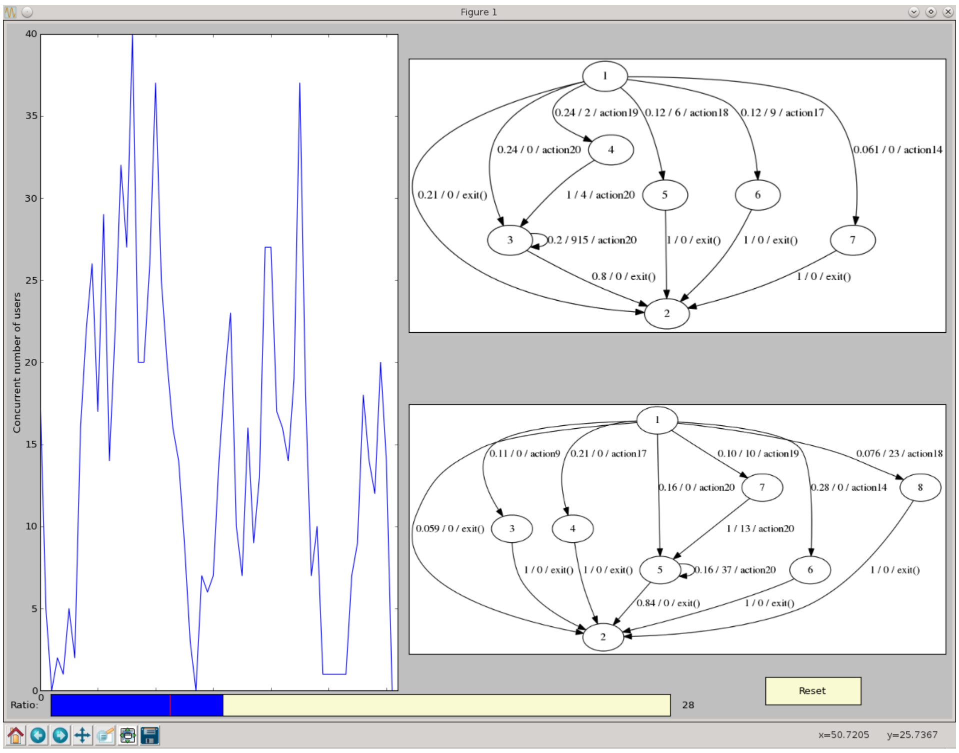


Figure 2.3: Graphical user interface of the Log2Model tool

representation of a given request. If we encounter a certain request for the first time, we create a new user action corresponding to the request; otherwise, we reassign the previously created user action to the request. Table 2.2 lists all the user actions corresponding to the requests in Table 2.1. The mapping between the requests and the user actions is used to generate another artifact, called a *test adapter*. The test adapter code consists of every user action in an executable format. It is utilized as an interface between the workload model and the SUT by our MBPeT tool, which we will discuss in Section 2.4.

Building the model: At the second stage, we start creating the workload model incrementally by processing each request in the queue sequentially in order to preserve the order of the requests. In the beginning, the workload model has only the initial location, which is also the current *docking location*. The docking location in the model is used as the source location for the next new edge. We add a new edge in the model with the following characteristics for each request:

- *Source location:* the current docking location in the model becomes the source location of the new edge. For example, the source location

No.	**Time stamp**	**Accessed resource**
R1	19/11/2019:14:22:35	GET /home
R2	19/11/2019:14:22:39	GET /browse
R3	19/11/2019:14:23:01	GET /read/posts
R4	19/11/2019:14:23:51	GET /home
R5	19/11/2019:14:23:57	GET /search

Table 2.1: A queue of HTTP requests

No.	**Accessed resource**	**User action**
R1	GET /home	home()
R2	GET /browse	browse()
R3	GET /read/posts	read()
R4	GET /home	home()
R5	GET /search	search()

Table 2.2: User actions corresponding to the HTTP requests

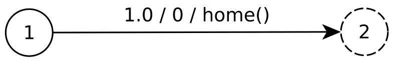
of the edge for the first request (i.e., *R1*) is the initial location (i.e., location *1*), as illustrated in Figure 2.4(a).

- *Destination location*: it depends on whether a similar request already exists in the model or not. In the latter case, we add a new destination location for the new edge. For instance, we have added three new locations for the first three requests, as shown in Figure 2.4(b). In the former case, the new edge points to the same location where the existing edge performing the same request points to. For example, in Figure 2.4(c), the edge for the *R1* request (from location *1* to *2*) and the edge for the *R4* request (from location *4* to *2*) have the same destination location (i.e., location *2*). The destination location of the latest edge becomes the docking location.

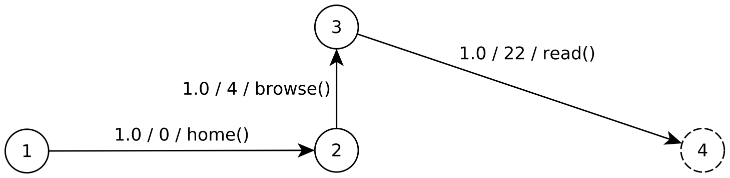
- *User action:* we assign a user action to the new edge corresponding to the request.

- *User think time:* it is determined by calculating the time difference between the timestamps of the current and the previous request.

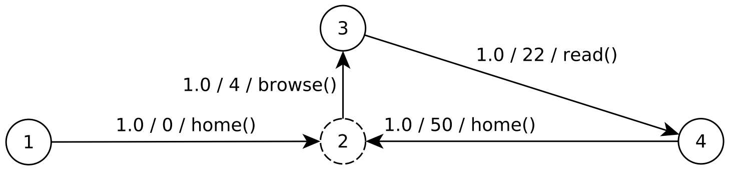
- *Probability:* It is based on the number of outgoing edges from the source location and how many times a certain user action has been performed. For example, in Figure 2.4(d), there are two outgoing edges corresponding to user actions *browse()* and *search()*, and both actions are performed only once by the user. Thus, both edges have the same probability (i.e., 0.5)



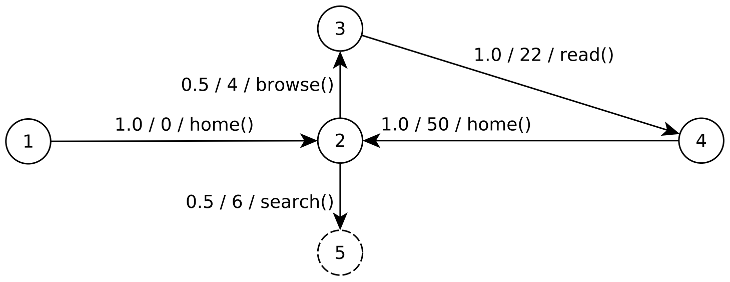
(a) Workload model after incorporating the first request



(b) Workload model after incorporating the first 3 requests



(c) Workload model after incorporating the first 4 requests



(d) Workload model after incorporating all the requests

Figure 2.4: Incremental development of a workload model based on the requests listed in Table 2.2. Docking locations are represented as dashed circles.

The method allows us to characterize the user behavior for workload generation without the need for log files. This is beneficial because, in most cases, the log files are not available, or they do not contain sufficient details to construct the workload models.

Tool Support: Click&Capture

We have implemented a tool, called *Click&Capture* in Python, which serves as a proxy to the SUT in order to capture the HTTP requests between the user and the SUT as depicted in Figure 2.5. The captured requests are used to generate a workload model and an executable test adapter code in the Python programming language by the tool.

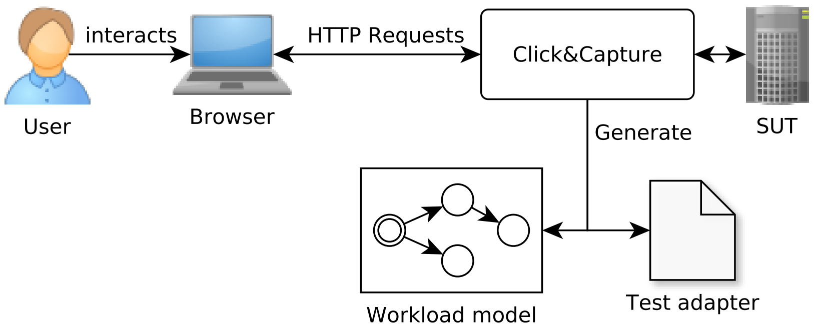


Figure 2.5: Capturing HTTP requests to generate a workload model and a test adapter

2.3 Model Consistency Rules

In order to check the consistency and correctness of the workload models, we have defined the following validation rules that later on are enforced by our tool chain:

- **One initial location:** a workload model must have precisely one initial location (i.e., a location with no incoming edges).
- **One or more exit locations:** there should be at least one exit location (i.e., a location with no outgoing edges) in a workload model.
- **Probabilities and user actions:** probabilities and user actions must be specified correctly for each edge in the model, and the sum of the probabilities of all the outgoing edges from a location must be equal to 1.
- **Isolated locations:** there should be no location with no incoming and outgoing edges in a workload model.

2.4 Model-based Performance Testing Process

In this section, we describe our model-based performance testing approach, where we benchmark the performance of the SUT when it is subjected to a controlled amount of workload. We generate workload using PTA models, characterizing the dynamic behavior of the real users.

In our approach, we employ a collection of workload models to describe the different groups of users. Figure 2.4(d) depicts an example of a workload model. In addition to the workload models, we define a *root model*, which describes the arrival rate of the different groups of users in the workload mix and their probabilistic distribution. The root model is also represented using the PTA modeling notation. For example, the root model in Figure 2.6 indicates two groups of users where 60% and 40% of users belong to *user_group1* and *user_group2*, respectively.

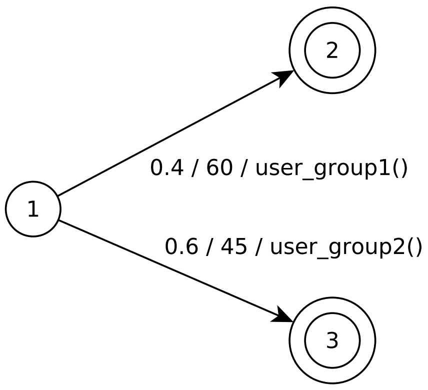


Figure 2.6: Root model

The workload model in Figure 2.4(d) describes the probabilistic behavior of a particular group of users interacting with a news forum web application. Each edge in the model specifies the probability of choosing the edge, the user think time before traversing the edge, and the *user action* to execute.

The workload is generated by simulating the workload model. The simulation starts from the *initial location*. Upon traversing an edge in the workload model, the action associated with the edge is translated into a request using the *test adapter*, and the translated request is sent to the SUT. For example, the *open_home* function defined (at the line *8*) in Listing 2.1 implements the *home* action in the model shown in Figure 2.4(a). In the *open_home* function, we send an HTTP request (line *10*) to the SUT for

```python
from petadapter import AbstractAdapter , action

class Generic_Adapter(AbstractAdapter):
  def __init__(self , *arg , **kwarg):
    AbstractAdapter.__init__(self , *arg , **kwarg)

  @action("home")
  def open_home(self , username , user_id , parameters):
    url = "https://www.mywebapplication.com/home"
    res = self.session.get(url)
    repeat = False  # do not repeat this action
    return res , repeat
```

Listing 2.1: A Python code snippet of the test adapter created for the model shown in Figure 2.4(a)

the *home* web page. Whenever we arrive at one of the *exit* locations of the model, the current user session is terminated, and the simulation of the workload model restarts. This procedure is repeated during the entire load generation process.

The request generation process imposes a certain amount of load on the system because the system has to process the requests and generate the corresponding responses. Each simulation of the workload model represents one virtual user. Using the *root model*, we can employ different workload models to generatethe workload from different groups of users. The amount of the workload is regulated using a *ramp function* that specifies the desired number of parallel simulations of the workload model at any given moment during the test session.

PTA workload models allow us to capture the probabilistic behavior of the real users and introduce randomness up to a certain degree into the testing process. This is beneficial in identifying rare sequences of actions that could negatively affect the performance of the system. Finding such sequences using traditional testing methods like static test scripts, where we execute actions in a deterministic order, would be infeasible.

2.5 Tool Support for Load Generation

We have implemented our performance testing approach as a tool in the Python programming language, called Model-based Performance Testing (MBPeT) [63, 76]. The tool generates the workload against the SUT in real-time using the workload models, as shown in Figure 2.7. The tool also requires a *test adapter* that translates each user action in the workload models to an executable format and a *test configuration*, which includes different parameters such as test session duration and a ramp function. The tool monitors different KPIs like resource utilization of the SUT, response times of user actions, throughput, and error rate. At the end of a performance test session, a test report is generated based on the information regarding each KPIs collected during the test session.

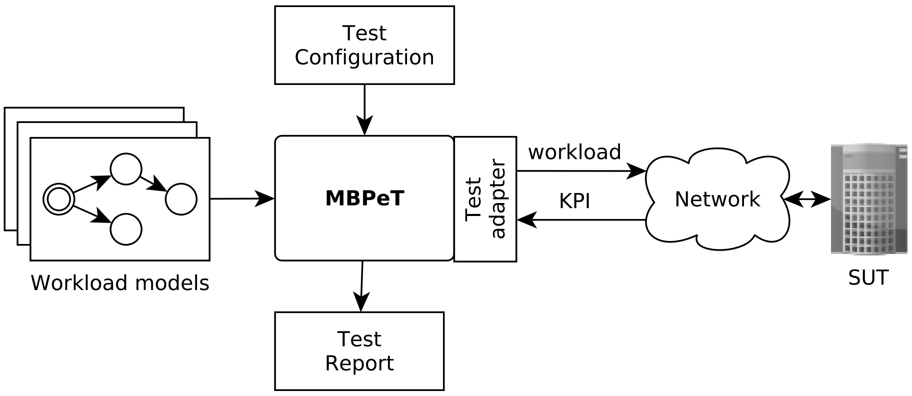


Figure 2.7: MBPeT tool

Distributed Architecture As we have discussed in Section 2.4, the workload is generated by simulating concurrent virtual users. Simulating a virtual user consumes a certain amount of hardware resources such as CPU and memory on a load generating slave node. This restriction makes simulating a large number of concurrent virtual users impossible. In order to overcome this limitation, MBPeT generates a workload using a distributed architecture where it uses several computing nodes to simulate virtual users. The architecture of the MBPeT consists of two types of nodes: a *master node* and *slave nodes*. A master node orchestrates the entire test session by managing several remote slave nodes, as shown in Figure 2.8. The tester provides the test configuration and workload models to the master node and gets a test report at the end of the test session. Slave nodes are generic and they do not have prior knowledge of the SUT or the workload models. The master collects and parses the required information (e.g., workload models and the test configuration) for every test session and sends that information to all the slave nodes for the workload generation.

The master node triggers one of the slave nodes to start the workload generation while the rest of the slave nodes wait in their idle state. Load generating slave nodes monitors their local resource utilization. A load generating slave node stops increasing the number of concurrent virtual users and informs the master node if the resource utilization of the node crosses a given threshold value. In response to a saturated slave node, the master node initiates another idling slave node for workload generation. This procedure allows us to sustain the given workload generation rate by simulating any desired number of virtual users.

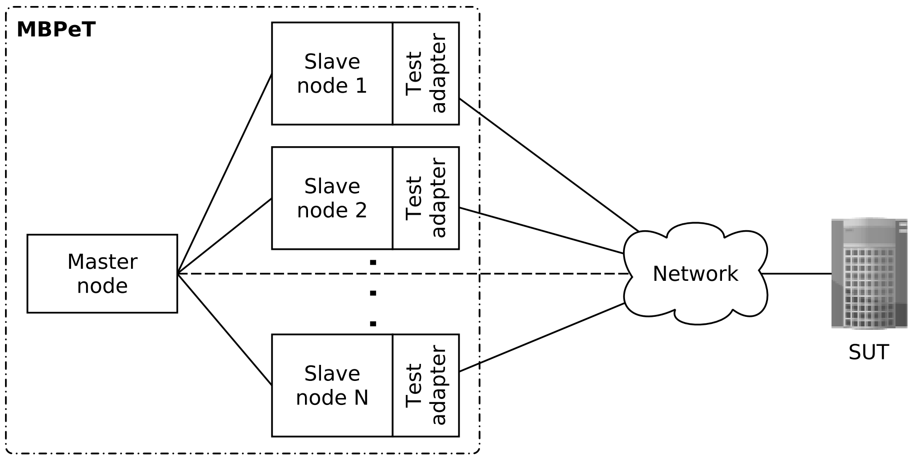
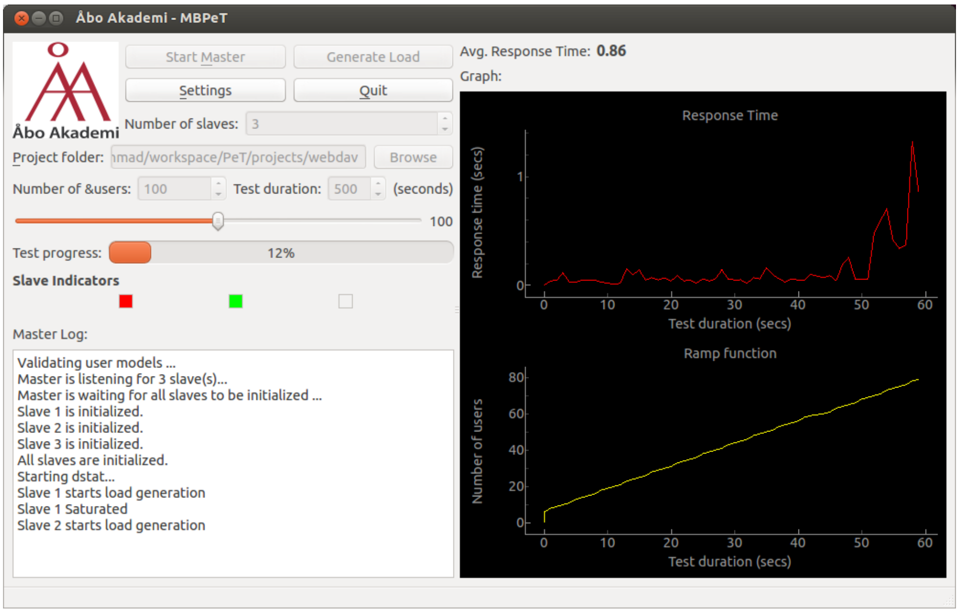


Figure 2.8: Distributed architecture of MBPeT tool

Graphical User Interface In addition to a command-line interface, MBPeT features a *Graphical User Interface* (GUI) dashboard, shown in Figure 2.9. The MBPeT dashboard is composed of two panels. The right panel of the dashboard allows the tester to set up the test configuration before starting the test session. Moreover, the tester can use the *slider* to change the number of concurrent virtual users during the test session. The *slave indicators* show the state of the slave nodes. An idle slave node is expressed by a gray indicator. Likewise, active and saturated nodes are represented by green and red indicators, respectively. The left panel of the dashboard is used to monitor the performance of the SUT during the test session in real-time. It consists of two graphs and a label. The *Avg. Response Time* label displays the current average response time of all the actions that are being executed. The *Response Time* graph shows the average response time of all the actions with respect to the number of seconds since the test session started. The graph at the bottom of the panel illustrates the ramp function. Both graphs are continually updated in real-time during the entire test session.



Avg. Response Time: 0.86
Number of slaves: 3
Project folder: ımad/workspace/PeT/projects/webdav
Number of &users: 100 Test duration: 500 (seconds)
Test progress: 12%

Master Log:

```
Validating user models ...
Master is listening for 3 slave(s)...
Master is waiting for all slaves to be initialized ...
Slave 1 is initialized.
Slave 2 is initialized.
Slave 3 is initialized.
All slaves are initialized.
Starting dstat...
Slave 1 starts load generation
Slave 1 Saturated
Slave 2 starts load generation
```

Figure 2.9: Graphical user interface of the MBPeT tool

Test Report At the end of a test session, the master node aggregates the KPI information from the slave nodes to produce a test report. The master node monitors the resource utilization of the SUT and presents that information in the test report as well. The report comprehensively presents the information using different statistical functions. Furthermore, it includes several graphs to represent how different KPIs have changed during the test session. The test report comprises several sections and each section presents a different point of view of the test results [63].

2.6 Empirical Validation

In this section, we will demonstrate the applicability of our performance testing approach by carrying out different experiments on an auction web application, called YAAS. The YAAS web application has a RESTful [77] interface that is based on the HTTP protocol and it is implemented in the Python programming language.

2.6.1 Experiment 1: Performance testing

In the first experiment, we benchmark the performance of YAAS by generating the workload using the MBPeT tool. For this purpose, we have created three PTA workload models corresponding to three different types

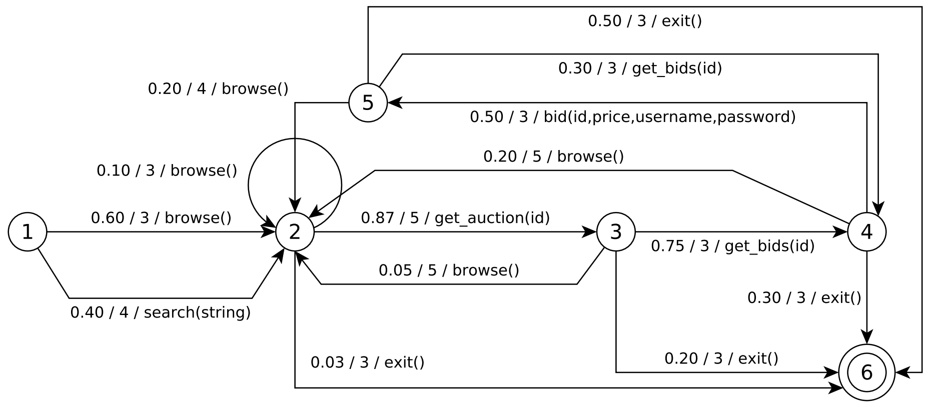


Figure 2.10: Aggressive user model

of users (i.e., aggressive user, passive users, and non-bidders type) and one root model to specify the arrival rate of the different types of users in the workload mix and their probabilistic distribution. The *aggressive type* of users makes bids more often compared to the *passive type* of users. On the other hand, the *non-bidder type* of users does not bid at all. These models are constructed manually by analyzing the requirement specifications according to the method discussed in Section 2.2.1. For example, Figure 2.10 illustrates the workload model of aggressive user type. An interested reader can find more details about the workload models constructed for YAAS in Abbors et al. [76].

The goal of this experiment is to determine the maximum number of concurrent users that the current implementation of YAAS can handle while keeping the response time values of the user actions under the given target response time values. In this experiment, MBPeT has generated the workload against the SUT for 20 minutes by linearly increasing the number of concurrent virtual users from 0 to 300. Table 2.3 presents the results of the experiment. One can notice that the maximum number of concurrent virtual users supported by YAAS without exceeding any target response time values is 64, as highlighted in the table.

Figure 2.11 shows the resource utilization of the SUT during the test session. One can notice that the CPU utilization rose very rapidly as the number of concurrent virtual users increased. This implies that the YAAS is a CPU-intensive web application.

	Target Response Time		Aggressive users (45%)		Verdict
Actions	**Avg (sec)**	**Max (sec)**	**Time of breach (sec)**	**Time of breach (sec)**	**Pass/Fail**
browse()	4.0	8.0	279 (78 users)	394 (110 users)	Failed
search(string)	3.0	6.0	**229 (64 users)**	327 (92 users)	Failed
get_action(id)	2.0	4.0	276 (77 users)	325 (91 users)	Failed
get_bids(id)	3.0	6.0	327 (92 users)	394 (110 users)	Failed
bid(id,price, username, password)	5.0	10.0	328 (92 users)	468 (131 users)	Failed

	Target Response Time		Passive users (33%)		Verdict
Actions	**Avg (sec)**	**Max (sec)**	**Time of breach (sec)**	**Time of breach (sec)**	**Pass/Fail**
browse()	4.0	8.0	323 (90 users)	394 (110 users)	Failed
search(string)	3.0	6.0	279 (78 users)	394 (110 users)	Failed
get_action(id)	2.0	4.0	279 (78 users)	279 (78 users)	Failed
get_bids(id)	3.0	6.0	325 (91 users)	394 (110 users)	Failed
bid(id,price, username, password)	5.0	10.0	327 (92 users)	474 (132 users)	Failed

	Target Response Time		Non-bidders users (22%)		Verdict
Actions	**Avg (sec)**	**Max (sec)**	**Time of breach (sec)**	**Time of breach (sec)**	**Pass/Fail**
browse()	4.0	8.0	279 (78 users)	394 (110 users)	Failed
search(string)	3.0	6.0	279 (78 users)	394 (110 users)	Failed
get_action(id)	2.0	4.0	280 (79 users)	325 (91 users)	Failed
get_bids(id)	3.0	6.0	279 (78 users)	446 (130 users)	Failed
bid(id,price, username, password)	5.0	10.0	NA	NA	Failed

Table 2.3: Response time values for every user actions corresponding to each user type (Bold value is the maximum number of concurrent virtual users supported by YAAS without exceeding any target response time values.)

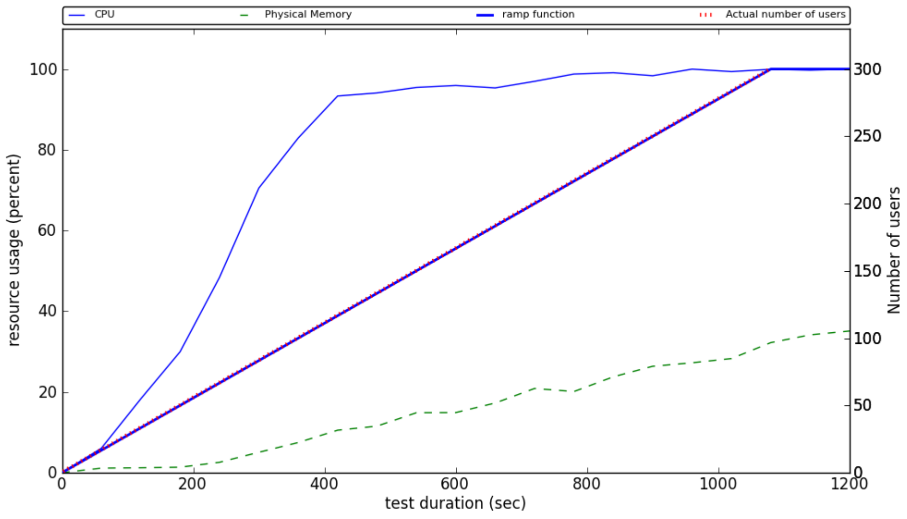


Figure 2.11: SUT CPU and memory utilization with respect to the number of concurrent virtual users

2.6.2 Experiment 2: Generating workload models

In Section 2.2, we have discussed three methods to construct workload models: (1) by analyzing the requirement specifications, (2) by mining system usage log, and (3) by capturing the user interactions with the SUT. The applicability of the first and the third method have been discussed already in Section 2.2.1 and 2.2.3, respectively. In this experiment, we will discuss the applicability of the second method on a web application, called *Pubiliiga*[1], which is used to manage the results of football games played in a local football league. Log2Model generates the workload model by mining the web usage log of Pubiliiga. The tool has filtered 30 000 lines of log data out of almost 1.3 million, by removing all irrelevant and incorrect lines from the log. The tool took 10.38 seconds in total to produce a workload model shown in Figure 2.12.

We have performed another experiment to validate our Log2Model tool. The experiment was done in two steps: we generated the workload against the YAAS using the workload models that we created manually, and then utilized the log data produced during the load generation to construct the workload models. Figure 2.13 compares the original workload model and the one we created by mining the log data. One can notice that the models are almost identical except a few minor differences between the probabilities of the edges. The differences in the probabilities of the edges between both models are highlighted with red color.

[1]`http://www.pubiliiga.fi/`

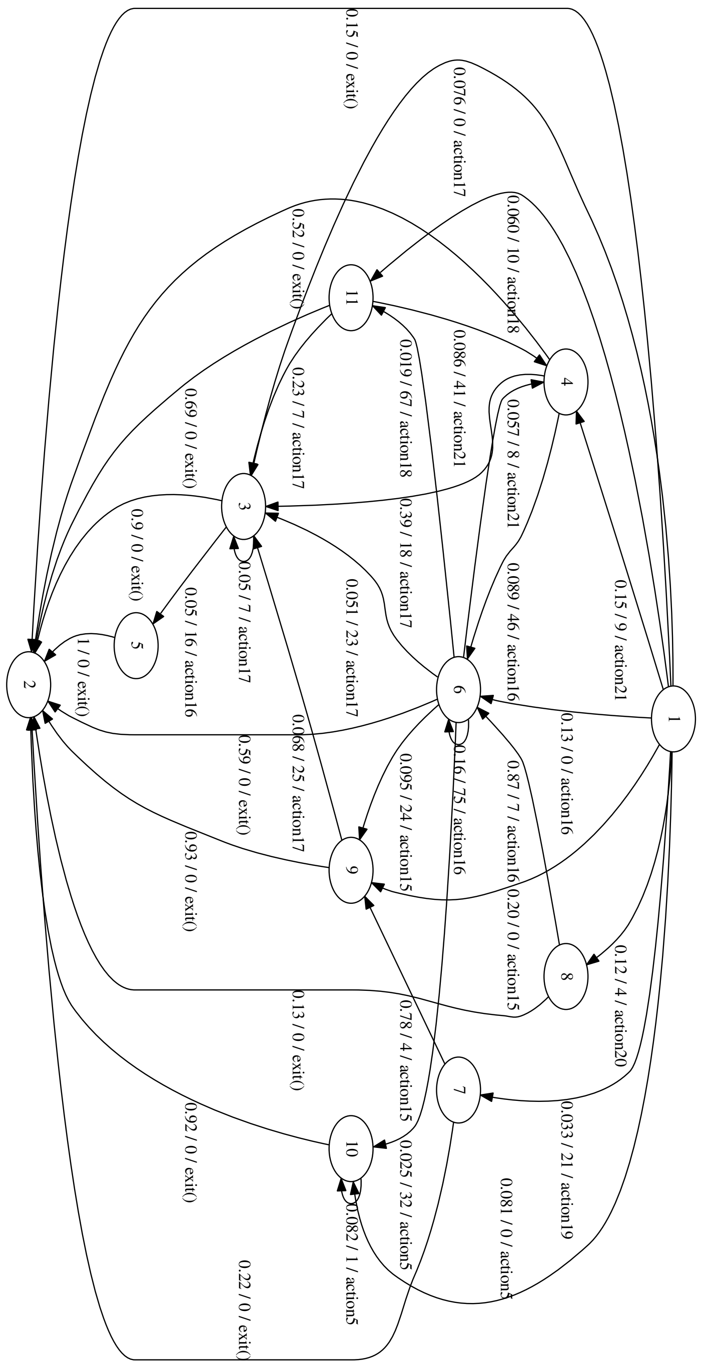
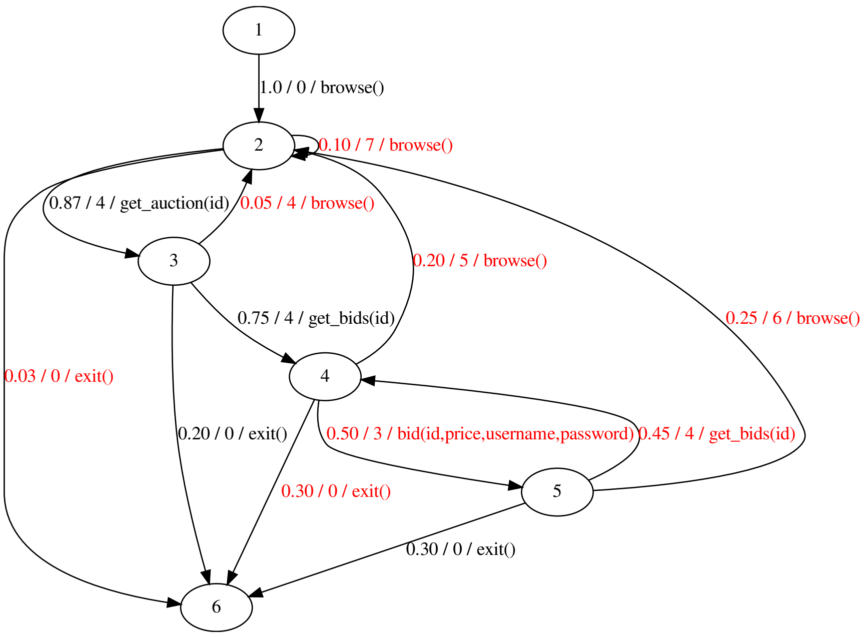
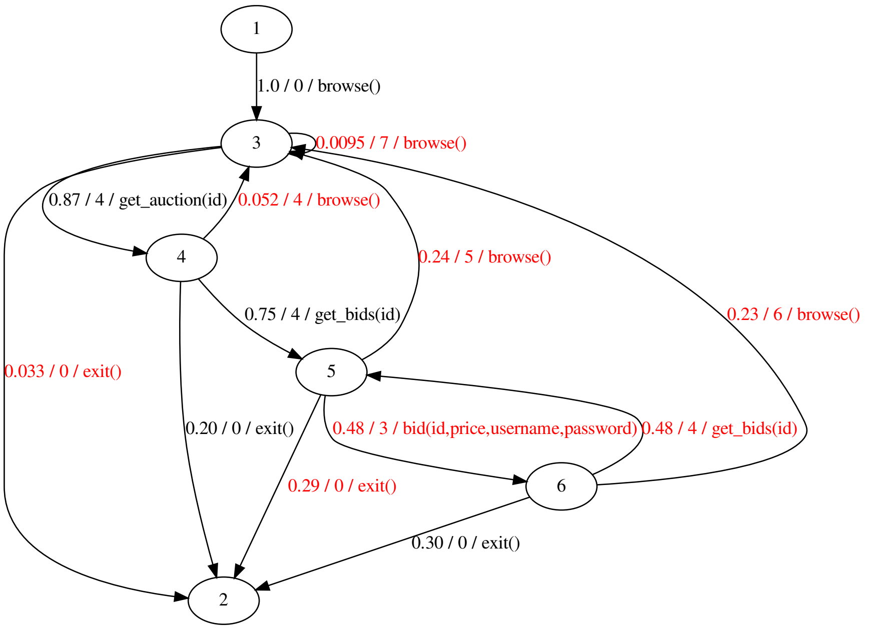


Figure 2.12: Workload model by mining the usage log of Pubiliiga



(a) Original workload model



(b) Generated workload model

Figure 2.13: Original vs generated workload model (Differences in the probabilities of the edges between both models are highlighted with red color)

2.7 Related Work

Many researchers have investigated the topic of analyzing and benchmarking the performance of web-based applications. Vögele et al. [78] have proposed a performance testing framework for session-based web applications. The author introduces a domain-specific language, called WESSBAS, to create workload specifications. The specifications can be extracted from usage log files of a production system, which are then used for load generation using Apache JMeter [79]. Shams et al. [80] have proposed a model-based performance testing approach for session-based web applications. The authors utilized *Extended Finite State Machines* (EFSMs) to capture inter-request dependency. The state machines are used to create a large trace of sessions for the SUT. These sessions are executed against the SUT using httperf [81]. Ruffo et al. [82] have presented a model-based performance testing tool, called WALTy. The authors extract *Customer Behavior Model Graphs* (CBMGs) [83] from the usage log files of the SUT. CBMGs are used to represent the navigation patterns of real users. The authors have customized the httperf tool to generate a workload using CBMGs. Krishnamurthy et al. [84] have extracted the input traces from the usage log files. These input traces are used to create a synthetic workload trace with the correct inter-request dependencies and desired descriptions for a selected set of workload attributes like request rate. These traces are transformed into sessions that are provided as input to the httperf tool for load generation. Schulz et al. [85] have introduced a load testing approach in the context of continuous software engineering. They have used the WESSBAS approach to extract workload models from the usage log and the Apache JMeter [79] tool for load generation. Shariff et al. [86] proposed a browser-based load testing approach using Selenium [87]. Selenium is a browser-based automation tool that is used for functional testing by simulating a virtual user in an independent browser instance. In the approach, the authors utilize a single browser instance for several users in order to generate a workload. Our approach differs from the approaches mentioned earlier in mainly two ways: firstly, in most approaches, the authors focus on the trace generation and use other tools to handle workload generation against the SUT, whereas we do on-line workload generation from our models. Secondly, CBMG and EFSM modeling techniques are suitable for modeling simple user scenarios; otherwise, the model would become very complex and difficult to maintain, unlike our proposed modeling technique.

Apte et al. [88] have proposed an approach, called AutoPerf, for generating load against a web application and measuring the response time and throughput with respect to load levels (i.e., number of virtual users). AutoPerf runs resource profilers on the SUT. Based on the profiling data, it adjusts the load levels and test duration. A CBMG is provided as an input to AutoPerf for load generation. A limitation of AutoPerf is that it only notifies the tester at the end of a test session whether the resources of the load generating node were saturated during the test session. In contrast, MBPeT continuously monitors the resource utilization of the load generation nodes. If the resources on a node get saturated, MBPeT initiates another load generating node to maintain the load generation rate.

In Barna et al. [89], a model-based performance testing method for transactional systems has been presented. The authors utilize a two-layers queuing model to represent the software and hardware contention for resources. They use analytical techniques to determine the workloads that can trigger bottlenecks in the SUT. When compared to their approach, we use a real implementation of the system for performance testing instead of a model of the system.

Guan et al. [90] have presented an approach to evaluate the performance of Web Map Tile Service (WMTS) by generating a workload. The authors have introduced a new workload model, called HELP, which simulates the user interactions with a WMTS map and statistically characterizes whole tile request patterns, including session length, think time, session path, and tile popularity (hotspots). They use LoadRunner [91] to generate a workload using the HELP workload model. The approach is designed to work with only WMTS applications. Our approach differs from their approach because our PTA-based workload models are agnostic to the type of the SUT.

There are numerous commercial and open source performance testing tools available on Internet [92]. JMeter [79] is the most popular open source tool for performance testing of web applications. It is written in Java language. Httperf [81] is another open source performance benchmarking tool. The tool is capable of conducting both micro and macro-level benchmarks. LoadRunner [91] is a commercial performance testing tool from Microfocus. The tool can simulate a large number of virtual users in a distributed manner to generate a workload. Each virtual user follows a script containing prerecorded user scenarios. In contrast, we advocate for the use of models to abstract user behavior and generate user load.

2.8 Conclusions

In this chapter, we have extended PTA modeling notation to characterize the dynamic behavior of the real users more concisely and intuitively. We have presented three methods to generate reasonably complex workload models. We have defined several validation rules to check the correctness of workload models.

We have discussed our model-based performance testing approach. The approach uses the workload models to generate an on-the-fly realistic workload against the SUT. It is implemented as a tool in Python, called MBPeT, which has a highly scalable distributed architecture, allowing it to achieve a high workload generation rate. We have empirically evaluated our performance testing approach by performing several experiments on an auction web application. The results of the experiments show that our performance testing approach can be used to generate a workload in order to benchmark the performance of the SUT. MBPeT does not need extensive prior domain knowledge or access to the source code of the SUT. Thus, it can be applied to software applications in other domains by altering the workload model and the test adapter with respect to the SUT.

Chapter 3

Performance Exploration of Web-based Software Systems

Dijkstra estimated that it would take more than 10,000 years to exhaustively test a multiplier of two 27-bit integers.

— Antonia Bertolino [93]

In the previous chapter, we have employed PTA models to capture the probabilistic behavior of real users. We have used workload models to generate load against the SUT in order to benchmark the performance of the SUT. A workload model is usually created either by the tester manually based on his/her domain knowledge about the SUT and intuitions or by mining the common navigational patterns from usage log files. As a result, it typically represents the user scenarios that are most commonly executed by the users. Therefore, some infrequent user scenarios that could trigger performance bottlenecks on the SUT will be left untested. In this work, we define *performance bottlenecks* as software defects, which degrade the performance of the SUT unexpectedly [37].

Selecting the proper input values for the user scenarios is very important in detecting the performance bottlenecks. It is indicated that almost two-thirds of the performance bottlenecks are detectable on certain input combinations [37]. However, exploring the input space manually and finding those *relevant input combinations* that can identify performance bottlenecks in a large-scale complex system within a feasible amount of time is a challenging task because there can be numerous input combinations [37, 94].

As we have briefly described in Chapter 1, ET is a software testing technique which does not rely on a pre-defined set of test cases; instead the tester continuously learns, creates, and executes test cases [36]. The tester extracts new information and insights from the results of the previously executed test cases and creates new, better test cases. The goal of ET is to find software defects by learning the system behavior and being less dependent on the test documentation. ET is usually performed manually and requires rigorous domain knowledge and substantial efforts and time. In this chapter, we present different novel exploratory performance testing approaches to identify not only the worst-case user scenario with respect to a given workload model but also a set of input combinations to the SUT that can trigger performance bottlenecks on the SUT.

This chapter is divided into two main sections. Firstly, in Section 3.1, we present our contributions to infer the user scenarios in a given workload model that can cause high resource utilization on the SUT, resulting in poor performance of the system. This section also presents the tool support and demonstrates the applicability of the methods by conducting several experiments. Secondly, Section 3.2 discusses an approach to explore a large input space to identify the input combinations that can trigger performance bottlenecks in a black-box system without any prior domain knowledge. Instead of exhaustively exploring the input space, the method only explores those regions of the input space that have a higher chance of triggering performance bottlenecks. Further, we describe the tool support of the approach and investigate the applicability of it by carrying out different experiments in the section. We conclude in Section 3.3.

3.1 Exploring User Scenario Space

In order to ensure performance requirements are met, testers need to identify the worst-case user scenario that can degrade the performance of the SUT by creating the highest utilization of a given resource on the SUT. As discussed before, user models used in performance testing are created based on the domain knowledge of the tester or by mining the common navigational patterns from usage log files. Thus, some infrequent user scenarios will be left untested. Manually inspecting the user scenario space to find those infrequent user scenarios is a labor-intensive and error-prone activity because a model with 22 locations can represent more than 68 million unique user scenarios [95]. In this section, we present three automated performance exploration approaches to find the worst-case user scenario in a given user model.

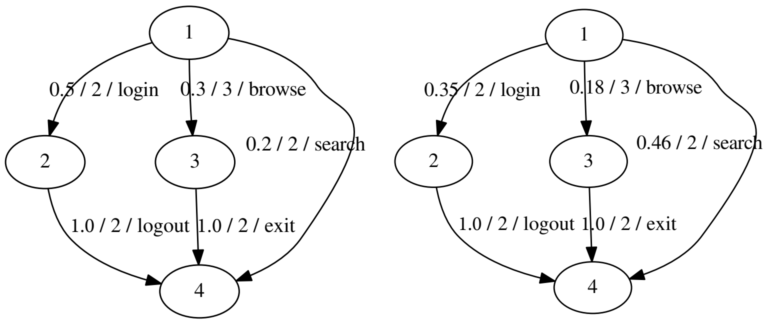
3.1.1 Mutation based exploration

In this method, we create several variants of the given workload model to explore the user scenario space. The method is based on *mutation testing* [96, 97].

Mutation testing is a fault-based software testing technique that is used to evaluate the effectiveness of the existing test suites [98]. In this technique, one creates a set of faulty versions of the SUT, called *mutants* by applying different mutation operators. Each *mutation operator* is designed to make small syntactical changes to the original code of the program under test (PUT). For example, a mutation operator changes the addition arithmetic operator (i.e., +) with other operators (i.e., -, *, /) to create mutants. Each mutant is executed against the test cases, and a mutant is considered to be *killed* if the test results of the mutant and the original PUT are different. The quality of the test cases is directly proportional to the number of mutants killed. *Specification mutation* [99] is an extension of mutation testing where we mutate the specification (instead of the code) of the SUT to produce mutants. For example, Mi and Ben [100] propose a specification mutation testing based approach where they mutate the UML State Diagrams, for instance, by adding or removing the transitions. The fundamental principle of the specification testing technique is to explore the behavioral space of the SUT in order to find hidden defects.

Our **Perf**ormance E**x**ploration (PerfX) [101] approach is based on the specification mutation, where we explore user scenario space to identify performance bottlenecks. We randomly mutate the given workload model by applying the following mutation operators to the model: (1) *Change Probability Distribution* (CPD) operator alters the probabilistic distribution of outgoing edges of a location; (2) *Modify Think Time* (MTT) operator changes the user think time values of outgoing edges of a location. For example, we applied the CPD operator to the model in Figure 3.1(a) to produce the mutated model in Figure 3.1(b). The operator changed the probability distribution of location *1* of the model in Figure 3.1(a). In this work, we focus on only those mutation operators that modify the frequencies of user actions. We avoid those mutation operators that can mutate the functional behavior of the users by, for instance, changing the direction or destination of the edges in the model, merging locations, replacing one user action with another. These operators may create invalid behavior and could be more useful in the context of robustness or negative testing [102] which is a subject of future work.

The generated mutants of the workload model represent different observations in the user scenario space. We utilize each mutant for load generation separately using the MBPeT tool (described in Section 2.5) and record the resource utilization of the SUT. We use the same *test configuration* and the *test adapter* for each test session. Once all the load generation sessions have been completed, we rank the mutants in relation to the resource utilization caused by them on the SUT. We identify those mutants which have caused the highest resource utilization with respect to each resource type



(a) Original workload model (b) Mutated workload model

Figure 3.1: Original vs mutated workload model

such as CPU, memory, and disk. By analyzing these mutants, the tester can find the potential performance bottlenecks and enhance the performance of the SUT.

Tool Support The method has been automated as a stand-alone tool developed in the Python programming language. The tool receives a workload model and a selected mutation operator as inputs. It generates a set of mutated workload models or mutants by mutating the given workload model. This method utilizes the MBPeT tool to generate a workload using each mutant in a separate test session. At the end of every test session, MBPeT summarizes the maximum, average, minimum resource utilization created by the mutant during the entire test session on the SUT and send the results to our tool. Once all the test sessions have been carried out, our tool can identify which mutants have created the highest resource utilization on the SUT with respect to different resource types.

3.1.2 Exact method using graph-search algorithms

In the previous approach, we rely on random mutations to explore the user scenario space for performance bottlenecks. As a result, in order to find the worst-case user scenario that can cause the highest resource utilization on the SUT, we would need to rigorously explore the space by producing and simulating a large number of mutants for workload generation, which is a time-consuming process. In this section, we discuss a performance exploration [95] approach (presented in Algorithm 1) that uses graph-search algorithms to identify the worst-case user scenario in a given workload model before using the model for workload generation.

In this method, we represent a workload model as a *Discrete Time*

Markov Chain (DTMC) [66] model. A DTMC workload model comprises a finite set of states and edges. The labels on the *edges* represent two values: (1) the *probability* value specifies the chances of a certain edge being chosen with respect to a probability mass function; (2) the *user think time* between two subsequent actions. A DTMC workload model is similar to a PTA workload model except that the user actions are associated with the states instead of the edges of the model. Whenever a state is visited, a corresponding user action is executed. For example, the DTMC model in Figure 3.2 represents the *aggressive user* model shown in Figure 2.10. In the DTMC model, the *start()* and *exit()* are the *pseudo-states* that express the *initial* and the *final* state of the model, respectively. The simulation of a workload model begins from the *start()* state and ends at the *exit()* state. We use DTMC as a workload model for this and the next approach because we can calculate the stationary distribution of the given DTMC model to determine which states in the model will be visited more frequently than the others based on the probability distribution of the model.

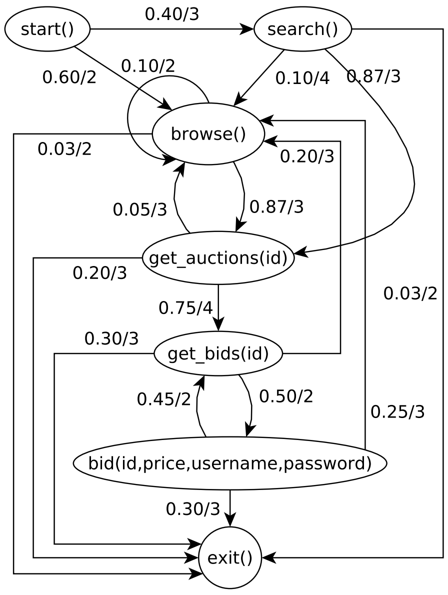


Figure 3.2: DTMC model of the aggressive user

We define the worst-case user scenario in a workload model as a path (i.e., a sequence of user actions) in the model, which will trigger the highest utilization of a given resource on the SUT over a sustained period of time. The objective of our method is to find the worst-case user scenario, which we denote as the worst path in a given workload model with respect to a given resource type (e.g., CPU or memory).

First of all, we determine the average resource utilization of each user action in the model. Once we have benchmarked every user action with respect to a given resource, we identify all the elementary circuits (i.e., a path in where only the first and last can appear twice [103]) in the model. We sort all the circuits based on the average resource utilization per user action. We pick the circuit with the highest average resource utilization as the *worst circuit*. If the *start()* location exits in the worst circuit, then this circuit is the worst-case user scenario. Otherwise, we find and merge the shortest path from the *start()* location to any location in the worst circuit to construct the worst-case user scenario. This allows the virtual users to simulate the worst circuit more frequently than any other user scenario in the model during load generation and thus create the highest resource utilization on the SUT. The tester can use such a scenario to debug the potential performance bottlenecks in the SUT in the presence of less probable but high impact user scenarios.

Tool Support The method is fully automated with tool support. The tool is based on Algorithm 1. It requires a workload model (G), resource utilization of each user action (U^r) for a given resource type (e.g., CPU or memory), and the start state ($INode$) of the workload model as inputs. It finds the worst circuit in the model that is an elementary circuit with the highest average resource utilization per user action (lines *2* to *3*). Next, it merges the worst circuit with the shortest path from the *start()* location in order to produce the worst-case user scenario if the worst circuit does not contain the *start()* location; otherwise, the worst circuit is provided as the worst-case user scenario (lines *4* to *8*).

3.1.3 Approximate method using genetic algorithms

The previous method identifies the worst-case user scenario in a given workload model, but it does not scale to large workload models with numerous loops due to its exhaustive nature [95]. To overcome this scalability issue, we present a performance exploration [95, 105] approach that uses genetic algorithms (GA) [106] to find a *near worst-case user scenario* in a given DTMC workload model.

GA is a heuristic algorithm that can find a near-optimal solution of different optimization problems by evolving a population of potential solutions of that problem. It is based on the natural evolution of species, where a new generation of solutions is created by applying various genetic operators. Each individual in the population is ranked according to its *fitness*

Algorithm 1 Graph-search based approach [95]

```
1: procedure WorstPath($G, U^r, INode$)
2:     $all\_circuits \leftarrow$ FindAllElementaryCircuits($G$)    ▷ Use Johnson's algorithm[103] to get a set of all elementary circuits
3:     $worst\_circuit \leftarrow SelectMax(\{(c, CRU(U^r, c)) \mid c \in all\_circuits\})$    ▷ Select the circuit with the highest resource utilization
4:     if $INode \notin worst\_circuit$ then
5:         $short\_paths \leftarrow$ ShortestPathsFrom($G, INode$)    ▷ Get all the shortest paths from the initial node to all the other nodes using Dijkstra's algorithm[104]
6:         $short\_paths\_to\_cir \leftarrow \{p \mid p \in short\_paths \wedge p \cap worst\_circuit \neq \emptyset\}$
7:         $min\_short\_path \leftarrow SelectMinLen(short\_paths\_to\_cir)$    ▷ Select the shortest path with the minimal length
8:         $worst\_path \leftarrow$ MergePaths($min\_short\_path, worst\_circuit$)
9:     else
10:        $worst\_path \leftarrow worst\_circuit$
11:    end if
12: end procedure
13: function CRU(RU, Path)    ▷ Calculate resource utilization per node of the given Path
14:     return Sum($\{RU[n] \mid n \in Path\}$)$\div \mid Path \mid$
15: end function
16: function MergePaths(path, circuit)    ▷ Merge the given path with circuit
17:     $node\_joint \leftarrow path \cap circuit$
18:     $Q \leftarrow$ Queue($circuit$)
19:     $top\_node \leftarrow$ Q.dequeue()
20:     while $top\_node \neq node\_joint$ do
21:         Q.Enqueue($top\_node$)
22:         $top\_node \leftarrow$ Q.dequeue()
23:     end while
24:     return $path \cup Q$
25: end function
```

value that specifies the superiority of an individual with respect to other individuals in the population. The individuals with higher fitness values have higher chances of surviving and reproducing the offspring for the next generation of solutions as compared to the individual with lower fitness value, analogous to natural selection. After simulating a certain number of generations of the solutions, the best solution across all the generations is chosen to be a near-optimal solution of the given problem.

In our method, each individual in the population represents a workload model with a different probability distribution. The first generation of the population is created by randomly modifying the probability distributions of the given workload model. We calculate the fitness value of each individual, which represents the expected level of resource utilization that a workload model with a given probability distribution will create on the SUT. The conjecture is that a workload model with a higher probability of the worst-case user scenario will create a greater resource utilization. Thus, the objective of our method is to find a workload model with the highest probability of the worst-case user scenario. The next generation is obtained by applying genetic operators to individuals with higher fitness values. We simulate the evolution process for a fixed number of generations. At the end of the process, we choose the individual with the highest fitness value among all generations as the near-worst workload model. In order to extract the near-worst case user scenario from the model, we start our walk from the *start()* state in the model and select the outgoing edge on each state with the highest probability. We halt when we revisit a state.

Tool Support The method is fully automated with tool support. The tool is based on Algorithm 2. It requires a workload model (G), resource utilization of each user action (U^r) for a given resource type (e.g., CPU or memory), crossover operator probability (C_p), mutation operator probability (M_p), mutation rate (M_r), population size (P), and the maximum number of generations (I) as inputs. It starts the evolution process and simulates the given number of generations of the given workload model. It monitors different statistics (such as maximum, minimum, and average fitness of the individuals) for every generation. At the end of the evolution process, the tool walks through the workload model with the highest fitness value among all the generations and provides the near-worst case user scenario. The tool utilizes the DEAP [107] library to establish our genetic algorithm.

Algorithm 2 Pseudocode of genetic algorithm [95]

```
1: procedure GA($I, P, G, C_p, M_p, M_r, U^r$)
2:     $Pop \leftarrow$ CreatePopulation($P, G$) ▷ Randomly generate initial population of size $P$ based on the model $G$
3:     for all Chromosome $c \in Pop$ do
4:         Fitness($c, U^r$)    ▷ Calculate fitness of a chromosome
5:     end for
6:     for $i \leftarrow 1$ to $I$ do    ▷ Evolve the initial population for $I$ generations
7:         BinaryTournament($Pop$)    ▷ Select chromosomes for the next generation
8:         TwoPointCrossover($Pop, C_p$) ▷ Use to two-point crossover operator based on $C_p$ probability
9:         Mutate($Pop, M_p, M_r$)    ▷ Mutate the individuals based on $M_p$ probability
10:        for all Chromosome $c \in Pop$ do
11:            Fitness($c$)
12:        end for
13:    end for
14: end procedure
```

3.1.4 Empirical Validation

In this section, we will demonstrate the applicability of our performance exploration approaches by carrying out different experiments on an auction web application, called YAAS. The YAAS web application has a RESTful [77] interface that is based on the HTTP protocol and is implemented in the Python programming language.

In the first experiment, we have applied PerfX [101] to the user model shown in Figure 3.3 and generated a set of 93 mutants of the workload model in about 5 seconds. Each mutant is used for load generation using MBPeT (discussed in Section 2.4) tool for 10 minutes. This load generation step took approximately 15 hours. Based on the load generation results, we have identified four mutants: *MutantD, MutantM, MutantN*, and *MutantC*, which have created the highest disk, memory, network, and CPU utilization on the SUT as compared to the other mutants, respectively. Table 3.1 presents the mutants with their resource utilization. The tester can further analyze the identified mutants to inspect which user scenarios are causing high resource utilization in order to debug and improve the performance of the SUT.

In the second experiment, we demonstrate the applicability of our exact [95] and approximate method [95]. We have applied both methods to the DTMC user model shown in Figure 3.2. Both methods took less than 1 minute to identify the same worst-case user scenario for the CPU resource in the given model. We have constructed a new workload model, namely the

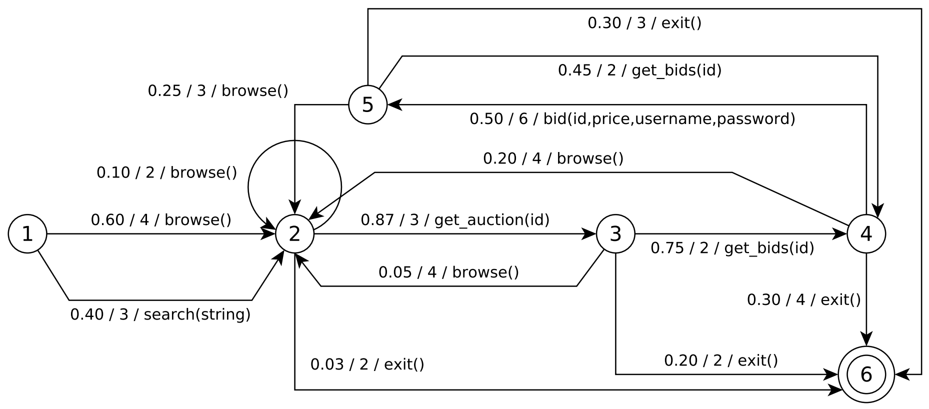


Figure 3.3: Workload model

Resource	**Original**	**MutantC**	**MutantM**	**MutantD**	**MutantN**
CPU (%)	76.22	**92.42**	71.44	91.63	89.47
Memory (GB/s)	3.28	1.50	**3.37**	0.97	0.93
Disk Write (KB/s)	117.04	76.16	104.38	**247.69**	76.56
Net Send (MB/s)	1.29	2.27	1.62	2.18	**3.09**
Net Recv. (KB/s)	71.62	90.02	80.12	114.11	**116.16**

Table 3.1: Resource utilization of the mutants vs original workload model. A **bold value** is the highest value in that resource category.

worst path model, which includes only the identified user scenario by setting the transition probabilities to 1. Figure 3.4 illustrates the worst path model, where we have highlighted the identified user scenario with red edges.

We have utilized the worst path model in Figure 3.4, the original workload model in Figure 3.2, and nine random variants of the original workload model for load generation using the MBPeT tool. These random variants are created by randomly changing the probability distributions of the locations in the original model. The reason behind creating random variants of the original model and using them for load generation is to show how much the identified user scenario increases the CPU utilization on the SUT compared to the different variants of the model. Figure 3.5 depicts the results of the experiment. The average resource utilization caused by the worst path model is approximately three times higher than the other workload models.

One can notice that, in the previous experiment, the approximate method produces the worst-case user scenario. Since the method is based on a heuristic algorithm (i.e., GA), we cannot guarantee that the method will always be able to find the worst-case user scenario. The purpose of our third experiment is to measure the accuracy of our approximate method. We have

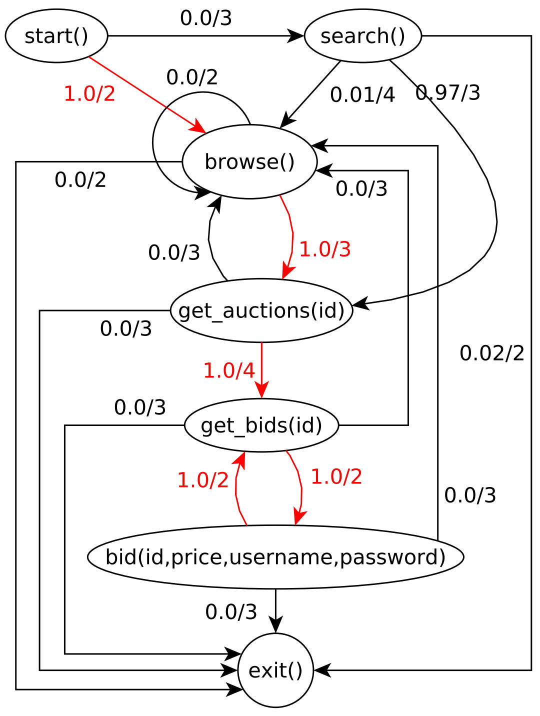


Figure 3.4: Worst path model with respect to CPU

randomly created eight sparse pseudo-workload models having a different number of states and edges listed in Table 3.2. Every edge in pseudo-workload models is associated with a random think time value; and every state executes a dummy action, which causes a specific amount of hypothetical resource utilization. In order to establish the statistical significance of the results, we have applied our approximate to each pseudo-workload model 10 times. The results of the experiment show that our method was able to find the worst-case user scenario in 73% of the cases.

In the last experiment, we evaluate the scalability of our approximate method. We have applied the exact and the approximate method to each pseudo-workload method listed in Table 3.2 and monitored the execution times of both methods. Figure 3.6 exhibits a comparison between the scalability of the approximate and the exact method. One can observe that the execution time of the exact method rises rapidly for the model with more than 20 states.

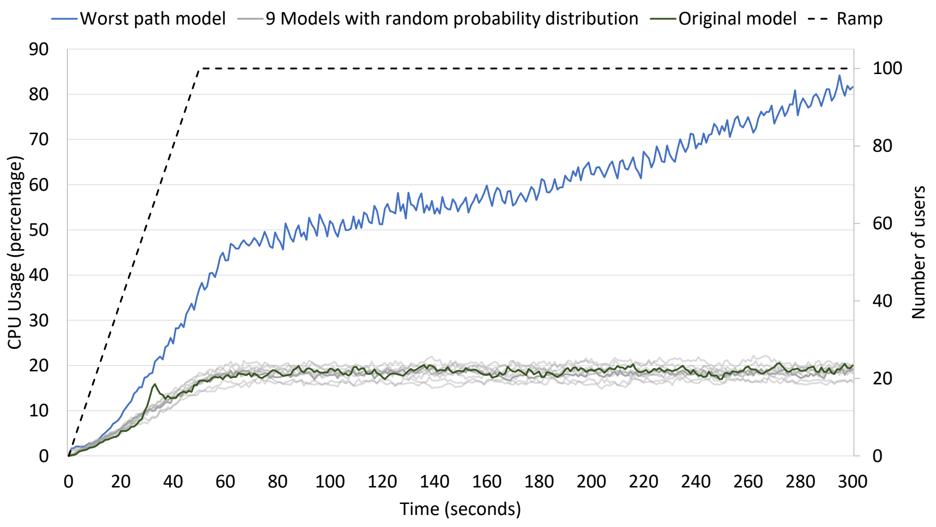


Figure 3.5: CPU utilization caused by the workload models

States	Edges	Elementary Circuits
10	23	25
12	42	270
14	50	2 030
16	73	53 211
18	85	189 776
19	95	907 861
20	103	6 141 014
21	111	12 764 464

Table 3.2: Random pseudo-workload models

3.1.5 Related Work

There are many approaches (e.g., [108, 109, 110, 111, 112]) that employ performance modeling techniques to predict and find the performance bottlenecks in web applications.

Jindal et al. [113] have presented a tool, called Terminus, for building a regression-based performance model in order to predict the capacity of a microservice (i.e., the maximum number of requests it can handle without violating service level objective). The authors collect performance data by conducting different load generation sessions and measuring the capacity of a microservice on various deployment configurations. The performance data is used to train the regression model. The results of the model are used for capacity planning for the application user test (AUT) and bottleneck

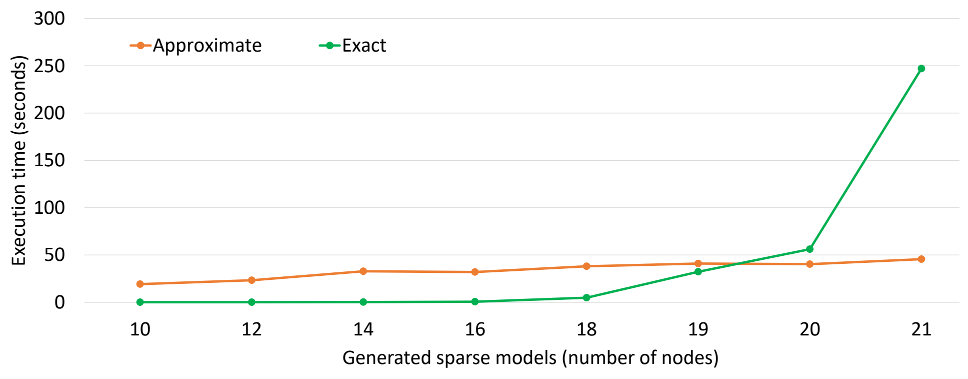


Figure 3.6: Comparison between the execution times of the approximate and the exact method

detection.

Duttagupta et al. [114] have proposed an analytical model-based approach to identify bottlenecks in software and hardware of the AUT by evaluating the performance of the AUT in terms of the number of users. The analytical model comprises two layers of queuing networks. It requires the service demands of all software and hardware resources at each tier of the AUT. The model is solved iteratively to obtain performance prediction for the AUT. Gao et al. [115] also utilized a queuing network for performance analysis and bottleneck identification by building a model of a composite web service. A composite web service comprises various service centers, and the internal control flow of those service centers is specified as a Markov chain model.

Ayala-Rivera et al. [116] have presented an approach, namely DYNAMO, which modifies the workload during the performance test session with respect to performance metrics of the SUT. The objective of the approach is to find an appropriate workload which identifies workload-dependent issues in the SUT. One needs to specify adaptive policies (e.g., when and how much the workload needs to be changed). These policies are used by the approach to adjust the workload. Hernández-Orallo and Vila-Carbó [117] have used a histogram-based workload model to characterize web traffic distributions. The model is used to calculate the response time and rejection probability distribution of HTTP requests using histogram calculus. Bogárdi-Mészöly and Levendovszky [118] have customized the Mean-Value Analysis evaluation algorithm to capture the behavior of the thread pool and build a performance model. The proposed algorithm is employed to infer the performance metrics of ASP.NET based web applications.

Xiao et al. [119] have proposed ΔInfer approach to identify Workload-

Dependent Performance Bottlenecks (WDPBs). The authors argued that performance bottlenecks are caused due to the WDPB loops which execute resource-intensive statements. The approach identifies those loops for a given workload by predicting iteration counts of WDPB loops using complexity models for the workload size. Stewart and Shen [15] have proposed a profile-driven performance modeling approach for multi-component web services. Application profiles are constructed offline to capture component resource requirements and inter-component communications. Subsequently, the model is utilized to predict performance bottlenecks in the SUT with respect to different operating configurations. Nistor et al. [120] have presented a tool, called Toddler, which identifies loops that perform similar operations across iterations. The conjecture is that these loops are probably executing redundant tasks, and fixing them could improve the performance of the AUT. Toddler needs to instrument the code of the AUT and other third-party libraries used in the code in order to monitor the loops.

To summarize, in all the studies presented above, the authors estimate the performance of the system at design time using design specifications. In contrast, we evaluate the performance of the system after it has been fully implemented, and we use models that describe the expected behavior of the user. There are several shortcomings regarding performance modeling based approaches. For example, in many cases, design specifications are not accessible on time for performance model construction [121]. Furthermore, most of these approaches require access to the source code of the SUT, which is not always feasible. Nowadays, modern software systems are incredibly complex; as a result, it has become more difficult to predict the performance of a complex system than weather [122].

There is a large amount of research work (e.g., [123, 124, 125, 126, 127]) which uses genetic algorithms to generate test data for software testing, but most of the work is targeted towards functional testing. Garousi et al. [128] have presented a model-based testing approach to find faults concerning network traffic in a distributed system. The authors provide a UML [129] model of the SUT as an input to a custom genetic algorithm for generating stress test requirements. These requirements consist of certain control flow paths in UML sequence diagrams, which can cause stress to the network.

3.2 Exploring the input space for identifying performance bottlenecks

As we have stated before, almost two-thirds of the performance bottlenecks are detectable on certain input combinations [37]. Executing all possible input combinations in order to find those combinations that can trigger performance bottlenecks in a large-scale complex system can be a time and cost-intensive task because there can be numerous input combinations [37, 94]. In this section, we present a method to explore a large input space to identify as many input combinations as possible for the user actions that can trigger performance bottlenecks in a black-box system without any prior domain knowledge. Instead of exhaustively exploring the input space, our method only explores those regions of the input space that have a higher chance of triggering performance bottlenecks. In this method, we are not interested in sequences of the user actions, but individual input combinations for the user actions.

Our method, namely *PerfXRL* [130], employs a *Deep Reinforcement Learning* (DRL) [131] algorithm, called DDQN [132], to explore a large input space of the SUT efficiently. DRL is an aggregation of Reinforcement learning (RL) [133] and *deep learning* [134] methods. RL is a reward-based machine learning technique where an agent learns by interacting with an unknown environment in order to accomplish a goal. The agent collects feedback (or a *reward*) from the environment by executing an action based on the current *state* of the environment. The purpose of the agent is to maximize the expected cumulative rewards over time by finding the optimal (or a near-optimal) sequence of actions. In DRL, the agent employs *Deep Neural Networks* (DNNs) to learn the environment. A DNN [134] is a multi-layer neural network that can approximate a complex function by learning higher-level representations of the given training data.

In our case, the unknown environment is the SUT and the main objective of the agent is to uncover as many input combinations as possible that trigger performance bottlenecks by non-exhaustively exploring the input space of the SUT. We denote those input combinations as *relevant combinations.* The agent executes various input combinations against the SUT while observing their performance impact on the SUT in a feedback loop, as shown in Figure 3.7. The agent is trained in an episodic manner. In each episode, the agent begins from a random input combination and executes a predefined number of steps. At each step, we create a new input combination based on the action suggested by the agent. The input combination is executed against the SUT. Based on the performance impact caused by the combination, we determine the reward for the agent. A positive reward is given for those input combinations which cause resource-intensive computations on the SUT; otherwise, a negative reward is given to the agent. The reward is used to improve the selection of future actions by the agent. As the agent executes more input combinations, it starts to learn the input space of the SUT and to execute those combinations that are likely to cause resource utilization on the SUT. The identification of resource-intensive computations is made by executing different input combinations against the SUT and monitoring the deviations of the KPI values (e.g., CPU load, disk usage, or elapsed execution time of the SUT) from certain pre-configured acceptable performance thresholds. Such thresholds can be derived from different sources such as requirement specifications or Service Level Agreements (SLAs) and vary from system to system. During the process, the approach maintains a list of the identified good combinations, which can be used later on for debugging the performance of the SUT.

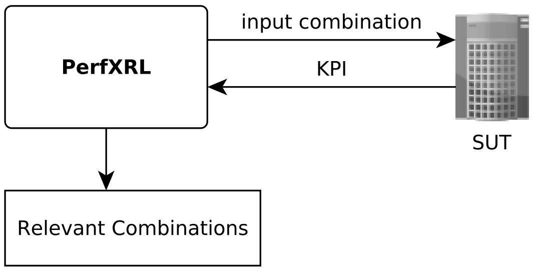


Figure 3.7: PerfXRL

Tool Support The method is fully automated with tool support. One needs to define the input space of the SUT and the reward function for the agent for the tool to work. Additionally, the tool requires parameter values for the DDQN algorithm, such as the number of training steps per episode and the number of episodes. Ahmad et al. [130] provide more details about the parameters and their values. The tool executes the given number of episodes and provides a list of relevant combinations found during the performance exploration of the SUT. The tool employs the Keras-rl [135] library to implement the DDQN algorithm.

3.2.1 Empirical Validation

In this section, we evaluate PerfXRL [130] by comparing it to random testing that uniformly samples an input combination without replacement from the input space of the SUT for a given number of times. Random testing has been proven to be more effective than other systematic testing methods for a black-box system with a large input space [136, 137, 138].

We have applied our approach to a reference web application *RUBiS* [139]. RUBiS implements an auction site. It has been widely used in academia for performance evaluation[1]. The size of the input space of RUBiS is 3 100 000 (i.e., the total number of input combinations). For the

[1]`http://scholar.google.com/scholar?q=Specification+and+Implementation+of+Dynamic+Web+Site+Benchmarks`

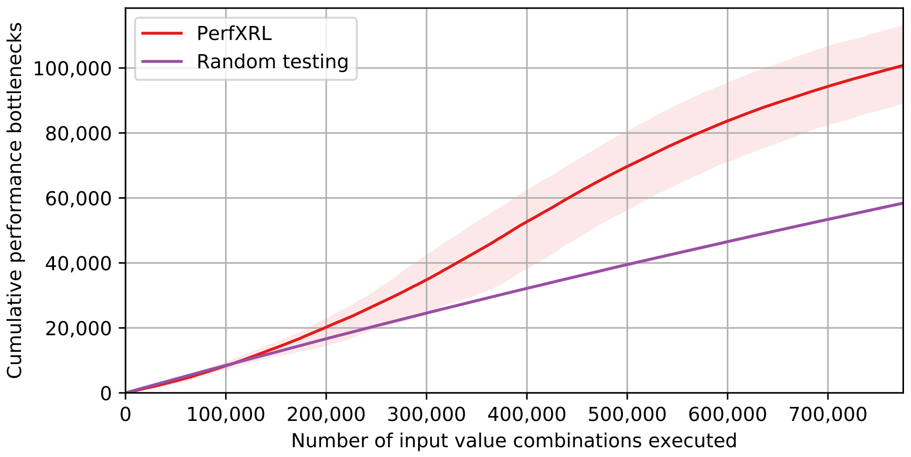


Figure 3.8: Comparison between the number of cumulative performance bottlenecks found by PerfXRL and random testing

evaluation purpose, we uniformly injected artificial bottlenecks on 9% of the total input combinations. We ran both approaches (i.e., PerfXRL and random testing) against RUBiS for 30 times to establish the statistical significance of the results. Each approach is allowed to execute 775 000 input combinations, which is 25% of the total input combinations.

Figure 3.8 presents the cumulative number of artificial performance bottleneck identified by PerfXRL and random testing after executing the same amount of input combinations. The solid lines in the figure illustrate the average values, while the shaded region encapsulating the lines expresses the standard deviation. The standard deviation of random testing was comparatively small; therefore, it is not visible in the figure. On average, PerfXRL and random testing identified about 100 800 and 58 405 bottlenecks, respectively. In other words, PerfXRL found 72% more bottlenecks than random testing after executing the same number of input combinations.

3.2.2 Related Work

Many researchers (e.g., [140, 141]) have investigated the topic of generating test data for performance testing.

Burnim et al. [142] proposed a test input generation approach, called WISE, which identifies worst-case inputs for a given program. The approach is based on symbolic execution. The approach exhaustively executes the program for small input sizes and learns the relationship between the complexity and the inputs, which is later on used to guide the symbolic execution to search for worst-case test inputs of larger sizes. A similar approach has been proposed by Saumya et al. [143], called XSTRESSOR. The approach builds a predictive model by running the given program under test against small size test input data and monitoring the behavior of the program. Subsequently, the model is used to predict the worst-case path condition for large-scale test input. In another approach [144], the authors have argued that the behavior of the program under test against small size input does not accurately predict the behavior of the program against large size input. They have proposed an approach, called PySE, which utilizes symbolic execution to acquire the behavioral information and incrementally updates its policy to drive the program execution toward the worst-case using reinforcement learning. Aquino et al. [145] have combined symbolic execution with a memetic search algorithm to find a path in the program under test that exhibits the worst-case execution time of the program. Zhang et al. [94] have integrated symbolic execution with an iterative-deepening search method to eliminate unpromising paths and explore only those paths that are likely to induce resource-consuming behavior in the program under test. Chen et al. [146] have proposed another symbolic execution-based approach, called PerfPlotter, which identifies program paths in terms of inputs for revealing best-case and worst-case execution times of the program under test.

Toffola et al. [147] have proposed an approach, namely PerfSyn, to create test programs that can identify performance bottlenecks at the method level of the program under test. The approach employs graph search algorithms to mutate the test programs and guide the search towards mutations that can uncover bottlenecks in the methods. Lemieux et al. [148] have presented an approach, called Perffuzz, for identifying the inputs that can expose the performance problems in the program under test. The approach uses feedback-directed mutational fuzzing to produce inputs without any domain knowledge about the program. However, the approach does require access to the source code of the program.

Furthermore, there are some approaches [149, 150] that focus on finding a certain input combination that maximizes a specific criterion, for example, computational complexity or execution time.

In summary, all the methods presented in this section have shortcomings, such as requiring rigorous domain knowledge about the SUT or having access to the source code of the SUT. In contrast, we aim to identify as many input combinations as possible that can trigger performance bottlenecks in a black-box system without any prior domain knowledge.

3.3 Conclusion

In this chapter, we have presented a set of novel tool-supported performance exploration approaches to infer the worst-case user scenarios in a given workload model that can cause high resource utilization on the SUT, resulting in poor performance of the system. The results of the experiments presented in Section 3.1.4 show that the proposed approaches can be used to reveal the potential performance bottlenecks in the SUT by automatically exploring the user scenario space.

Moreover, we have introduced an approach to explore a large input space of the SUT to identify the input combinations that can trigger performance bottlenecks in a black-box system without any prior domain knowledge. The results of the experiments conducted in Section 3.2.1 substantiate that our approach is more effective than random testing in finding the input combinations which can trigger resource-intensive computations on the SUT.

The proposed approaches do not need extensive prior domain knowledge or access to the source code of the SUT. Therefore, they can be applied to software applications in other domains by modifying their input artifacts such as workload models and input spaces with respect to those specific applications.

Chapter 4

Conclusions and Future Work

"An ounce of performance is worth pounds of promises."
— Mae West

This chapter summaries our main contributions. Consequently, we discuss the limitations of the research. The final section provides an overview of possible directions in which the work could be continued.

4.1 Summary

As we have previously mentioned in Chapter 1, the problem targeted in this thesis is the use of automated approaches to test and improve the performance of software systems. In Chapter 2, we proposed our model-based performance testing approach. The approach uses the workload models to generate an on-the-fly realistic workload against the SUT. We have presented how PTA models can be used to represent the dynamic behavior of real users more accurately than the sequential scripts. We have presented three systematic methods to generate user models: (1) mining the common navigational patterns from the usage log files; (2) capturing the HTTP requests made by the browser to the SUT in real-time; (3) inspecting the requirement specifications manually. Further, we have defined a set of validation rules to ensure the syntactic and semantic correctness of the workload models. We have implemented the approach as a tool called *Model-based Performance Testing* (MBPeT). The tool is able to parallelize the load generation process among several computing nodes in order to achieve a high load generation rate. Throughout the performance test session, MBPeT monitors different KPIs such as response time, error rate, resource utilization, and throughput. The utility of the MBPeT is demonstrated by conducting several test sessions. The results show that MBPeT can generate a realistic workload in order to benchmark the performance of the SUT effectively.

In Chapter 3, we have presented three model-based performance exploration approaches: the *mutation-based*, the *approximate*, and the *exact* approach. These approaches are used to infer the worst user scenario in a given workload model that can cause high resource utilization on the SUT, resulting in poor performance of the system. Such user scenarios are beneficial to identify potential performance bottlenecks in the SUT. The mutation-based approach explores the user scenario space randomly. Therefore, it does not guarantee to find the worst user scenario, and it needs to be run for a significant amount of time in order to get better results. The exact approach is deterministic and always provides the worst user scenario; however, it does not scale well to large models with numerous loops. The approximate approach utilizes genetic algorithms to explore the user scenario space. It cannot always find the worst user scenario, but it can identify a near-worst user scenario faster than the other methods, even for large models. An assessment of the approaches shows that the worst user scenario does trigger more resource-intensive computations on the SUT and cause more stress to the SUT as compared to the original model.

Nowadays, systems exhibit huge input spaces with many input parameters and large ranges. As a result, it has become impractical to exhaustively test all possible input combinations in order to identify performance bottlenecks. It is indicated that almost two-thirds of the performance bottlenecks are detectable on certain input combinations [37]. To address this problem, in Chapter 3, we have introduced a methodology to explore a large space of input combinations to identify performance bottlenecks in a black-box system without any prior domain knowledge. The approach is implemented as a tool called PerfXRL. The tool only explores a subset of the input space and tries to find as many input combinations as possible, which can trigger performance bottlenecks in the SUT. Our evaluation of the method indicates that it is effective enough to detect 72% more bottlenecks than alternative approaches.

4.2 Limitations

One of the primary limitations relate to all the proposed approaches is the restricted validation that is conducted over two web applications. The proposed approaches are supposed to work with all kinds of software systems where one can directly interact with them through their public interfaces. However, in this thesis, we perform validation only on web applications because the research work presented in this thesis has been carried out over several research projects that focus mainly on web application systems. Another limitation is that we have not tested the maximum load generation capacity of our MBPeT tool (presented in Chapter 2). During our evaluation, the tool was hosted on a private cloud featuring an 8-core CPU, 16 GB of memory, and 7200 rpm hard drive.

We have validated the PerfXRL tool by randomly injecting artificial bottlenecks into the subject application. Therefore, there is a chance that we may get different results when we run PerfXRL against a system with real bottlenecks. However, this experimental design allowed us to evaluate PerfXRL in a controlled environment and get reliable results.

For PerfXRL, like other machine learning-based approaches, one needs to tune some hyperparameters to get good results. Further, one set of values of hyperparameters for one case study might not work well for others. During our evaluation, we have selected the values for hyperparameters based on the practical experiences reported by other researchers [131, 132].

4.3 Future Work

There are many directions for future research in order to enhance and advance the approaches presented in this thesis. We have only tested the MBPeT tool in the context of web applications. We aim to apply the tool to the software systems from the other domains. Furthermore, as discussed in the previous section, the tool has not been tested for its maximum load generation capacity. We plan to deploy it on public clouds such as Amazon EC2[1] to benchmark its load generation capacity.

The approximate approach (discussed in Chapter 3) to infer the worst user scenario optimizes a given workload model for a single resource type (e.g., CPU, memory, or disk). We aim to employ a multi-objective optimization algorithm [151] for optimizing the model for more than one resource type. A similar improvement can be made to our PerfXRL approach, where we are just using one KPI (i.e., elapsed execution time) to identify performance bottlenecks. In the future, we plan to utilize several KPIs to recognize performance bottlenecks.

[1]`https://aws.amazon.com/ec2/`

Bibliography

[1] S. Lauesen. *Software Requirements: Styles and Techniques.* Addison-Wesley, 2002.

[2] A. M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 01 1937.

[3] F. C. Williams. Early computers at Manchester University. *Radio and Electronic Engineer*, 45(7):327–331, July 1975.

[4] F. C. Williams and T. Kilburn. Electronic Digital Computers. *Nature*, 162(4117):487–487, 1948.

[5] John W. Tukey. The Teaching of Concrete Mathematics. *The American Mathematical Monthly*, 65(1):1–9, 1958.

[6] Marc Andreessen. Why software is eating the world. *Wall Street Journal*, 20(2011):C2, 2011.

[7] Martin Hiller. Thoughts on the Future of the Automotive Electronic Architecture. Presentations at the FUSE Final Seminar, `https://bit.ly/2FJ24HR`, September 2016. Online; accessed on 19th September, 2019.

[8] Ondrej Burkacky, Johannes Deichmann, Georg Doll, and Christian Knochenhauer. Rethinking car software and electronics architecture. `https://mck.co/2Tf7Qcq`, February 2018. Online; accessed on 19th September, 2019.

[9] Paul A. Judas and Lorraine E. Prokop. A historical compilation of software metrics with applicability to NASA's Orion spacecraft flight software sizing. *Innovations in Systems and Software Engineering*, 7(3):161–170, Sep 2011.

[10] Mark Norris. *Understanding networking technology: concepts, terms, and trends.* Artech House, Inc., 1999.

[11] System Architecture Virtual Integration. Motivation for Advancing the SAVI Program. `https://savi.avsi.aero/about-savi/savi-motivation/`, 2006. Online; accessed on 19th September, 2019.

[12] Frank Elberzhager and Matthias Naab. High quality at short time-to-market: Challenges towards this goal and guidelines for the realization. In Dietmar Winkler, Stefan Biffl, and Johannes Bergsmann, editors, *Software Quality: Methods and Tools for Better Software and Systems*, pages 121–132, Cham, 2018. Springer International Publishing.

[13] D Zubrow. IEEE standard classification for software anomalies. *IEEE Computer Society*, 2009.

[14] ISTQB. Certified Tester Foundation Level Syllabus. `https://bit.ly/2FNgbfj`, 2018. Online; accessed on 19th September, 2019.

[15] S McConnell. *Code Complete: A Practical Handbook of Software Construction 2nd Edition. Redmond.* Washington: Microsoft Press, 2009.

[16] Mark Dowson. The ariane 5 software failure. *ACM SIGSOFT Software Engineering Notes*, 22(2):84, 1997.

[17] WIRED. Metric Math Mistake Muffed Mars Meteorology Mission. `https://bit.ly/2QMm3fh`, 2010. Online; accessed on 19th September, 2019.

[18] NASA. Mars Climate Orbiter Team Finds Likely Cause of Loss. `https://go.nasa.gov/36QOAWu`, September 1999. Online; accessed on 19th September, 2019.

[19] Tricentis. Software Fail Watch: 5th Edition. `https://bit.ly/35L4hgB`, 2017. Online; accessed on 19th September, 2019.

[20] Luciano Baresi and Mauro Pezzè. An introduction to software testing. *Electronic Notes in Theoretical Computer Science*, 148(1):89 – 111, 2006. Proceedings of the School of SegraVis Research Training Network on Foundations of Visual Modelling Techniques (FoVMT 2004).

[21] James A Whittaker. What is software testing? And why is it so hard? *Software, IEEE*, 17(1):70–79, 2000.

[22] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2008.

[23] Patrice Godefroid, Michael Y. Levin, and David Molnar. Sage: Whitebox fuzzing for security testing. *Queue*, 10(1):20:20–20:27, January 2012.

[24] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. Wiley Publishing, 3rd edition, 2011.

[25] Boris Beizer. *Software Testing Techniques (2Nd Ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[26] G.D. Everett and R. McLeod. *Software Testing: Testing Across the Entire Software Development Life Cycle*. Wiley - IEEE. Wiley, 2007.

[27] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[28] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5):297–312, 2012.

[29] Margus Veanes, Colin Campbell, Wolfram Schulte, and Nikolai Tillmann. Online testing with model programs. *SIGSOFT Softw. Eng. Notes*, 30(5):273–282, September 2005.

[30] Justyna Zander, Ina Schieferdecker, and Pieter J Mosterman. *Model-based testing for embedded systems*. CRC press, 2011.

[31] Filippos I Vokolos and Elaine J Weyuker. Performance testing of software systems. In *Proceedings of the 1st international workshop on Software and performance*, pages 80–87. ACM, 1998.

[32] Sai Matam and Jagdeep Jain. *Performance Testing Primer*, pages 3–12. Apress, Berkeley, CA, 2017.

[33] BM Subraya and SV Subrahmanya. Object driven performance testing of web applications. In *Quality Software, 2000. Proceedings. First Asia-Pacific Conference on*, pages 17–26. IEEE, 2000.

[34] Bugzilla@Mozilla. Bugzilla keyword descriptions. `https://bugzilla.mozilla.org/describekeywords.cgi`. Online; accessed on 19th September, 2019.

[35] E.J. Weyuker and F.I. Vokolos. Experience with performance testing of software systems: issues, an approach, and case study. *Software Engineering, IEEE Transactions on*, 26(12):1147–1156, 2000.

[36] Pierre Bourque, Richard E. Fairley, and IEEE Computer Society. *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0*. IEEE Computer Society Press, Washington, DC, USA, 3rd edition, 2014.

[37] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. In *Proceedings*

of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, pages 77–88, New York, NY, USA, 2012. ACM.

[38] San Murugesan, Yogesh Deshpande, Steve Hansen, and Athula Ginige. *Web Engineering: a New Discipline for Development of Web-Based Systems*, pages 3–13. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[39] Jeff Offutt. Quality attributes of web software applications. *IEEE Softw.*, 19(2):25–32, mar 2002.

[40] D. Gourley, B. Totty, M. Sayer, A. Aggarwal, and S. Reddy. *HTTP: The Definitive Guide*. Definitive Guides. O'Reilly Media, Incorporated, 2002.

[41] Oswaldo Olivo, Isil Dillig, and Calvin Lin. Static detection of asymptotic performance bugs in collection traversals. In *ACM SIGPLAN Notices*, volume 50, pages 369–378. ACM, 2015.

[42] Chia Hung Kao, Chun Cheng Lin, and Juei-Nan Chen. Performance testing framework for rest-based web applications. In *Quality Software (QSIC), 2013 13th International Conference on*, pages 349–354. IEEE, 2013.

[43] Daniel A Menascé. Load testing of web sites. *Internet Computing, IEEE*, 6(4):70–74, 2002.

[44] T. F. Abdelzaher, K. G. Shin, and N. Bhatti. Performance guarantees for web server end-systems: a control-theoretical approach. *IEEE Transactions on Parallel and Distributed Systems*, 13(1):80–96, Jan 2002.

[45] Ben Shneiderman. Response time and display rate in human performance with computers. *ACM Comput. Surv.*, 16(3):265–285, September 1984.

[46] Irina Ceaparu, Jonathan Lazar, Katie Bessiere, John Robinson, and Ben Shneiderman. Determining causes and severity of end-user frustration. *International Journal of Human–Computer Interaction*, 17(3):333–356, 2004.

[47] Akamai. Akamai reveals 2 seconds as the new threshold of acceptability for ecommerce web page response times. `https://bit.ly/2sjUvEy`, 2009. Retrieved: September, 2015.

[48] Greg Linden. Marissa Mayer at Web 2.0. `https://bit.ly/2TlaYng`, November 2006. Online; accessed on 19th September, 2019.

[49] R. Longbotham and R. Kohavi. Online experiments: Lessons learned. *Computer*, 40(09):103–105, sep 2007.

[50] Daniel A. Menascé and Virgílio A. F. Almeida. Challenges in scaling e-business sites. In *26th International Computer Measurement Group Conference, December 10-15, 2000, Orlando, FL, USA, Proceedings*, pages 329–336. Computer Measurement Group, 2000.

[51] Philipp Leitner and Cor-Paul Bezemer. An exploratory study of the state of practice of performance testing in java-based open source projects. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ICPE '17, page 373–384, New York, NY, USA, 2017. Association for Computing Machinery.

[52] Ian Sommerville. *Software Engineering (7th Edition)*. Pearson Addison Wesley, 2004.

[53] J Shaw. Web application performance testing - a case study of an on-line learning application. *BT Technology Journal*, 18(2):79–86, 2000.

[54] N. J. Gunther. Hit-and-run tactics enable guerrilla capacity planning. *IT Professional*, 4(4):40–46, July 2002.

[55] Brady Forrest. Bing and Google Agree: Slow Pages Lose Users. `https://bit.ly/2tUxU1M`, June 2009. Online; accessed on 19th September, 2019.

[56] Ron Kohavi, Randal M. Henne, and Dan Sommerfield. Practical guide to controlled experiments on the web: Listen to your customers not to the hippo. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '07, page 959–967, New York, NY, USA, 2007. Association for Computing Machinery.

[57] Connie U Smith and Lloyd G Williams. *Performance solutions: a practical guide to creating responsive, scalable software*, volume 1. Addison Wesley Longman Publishing Co., Inc., 2002.

[58] A. Nistor, T. Jiang, and L. Tan. Discovering, reporting, and fixing performance bugs. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 237–246, May 2013.

[59] Shahed Zaman, Bram Adams, and Ahmed E. Hassan. A qualitative study on performance bugs. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, MSR '12, page 199–208. IEEE Press, 2012.

[60] Dirk Draheim, John Grundy, John Hosking, Christof Lutteroth, and Gerald Weber. Realistic load testing of web applications. In *Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on*, pages 11–pp. IEEE, 2006.

[61] 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? In *Working Group Reports on ITiCSE on Innovation and Technology in Computer Science Education*, ITiCSE-WGR '06, page 96–114, New York, NY, USA, 2006. Association for Computing Machinery.

[62] Ken Peffers, Tuure Tuunanen, Marcus A. Rothenberger, and Samir Chatterjee. A design science research methodology for information systems research. *Journal of Management Information Systems*, 24(3):45–77, 2007.

[63] Tanwir Ahmad, Fredrik Abbors, Dragos Truscan, and Ivan Porres. Model-Based Performance Testing Using the MBPeT Tool. Technical Report 1066, 2013.

[64] Fredrik Abbors, Dragos Truscan, and Tanwir Ahmad. Tool support for automated workload model creation from web server logs. Technical Report 1112, 2014.

[65] Marta Kwiatkowska, Gethin Norman, Roberto Segala, and Jeremy Sproston. Automatic verification of real-time systems with discrete probability distributions. *Theoretical Computer Science*, 282(1):101 – 150, 2002. Real-Time and Probabilistic Systems.

[66] Charles Miller Grinstead and James Laurie Snell. *Introduction to probability*. American Mathematical Soc., 2012.

[67] G.E.P. Box and N.R. Draper. *Empirical Model-Building and Response Surfaces*. Wiley Series in Probability and Statistics. Wiley, 1987.

[68] Maria Calzarossa, Luisa Massari, and Daniele Tessera. *Workload Characterization Issues and Methodologies*, pages 459–482. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.

[69] S. Barber. Creating effective load models for performance testing with incomplete empirical data. In *Proceedings. Sixth IEEE International Workshop on Web Site Evolution*, pages 51–59, Sep. 2004.

[70] Soumen Chakrabarti. Chapter 2 - crawling the web. In

Soumen Chakrabarti, editor, *Mining the Web*, The Morgan Kaufmann Series in Data Management Systems, pages 17 – 43. Morgan Kaufmann, San Francisco, 2003.

[71] GSMA. The State of Mobile Internet Connectivity 2019. https://bit.ly/30iIE6s, 2019. Online; accessed on 19th September, 2019.

[72] Barry C. Arnold. *Pareto and Generalized Pareto Distributions*, pages 119–145. Springer New York, New York, NY, 2008.

[73] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In L. M. Le Cam and J. Neyman, editors, *Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297. University of California Press, 1967.

[74] Apache. HTTP Sever Project - Log Files. https://httpd.apache.org/docs/current/logs.html, 2019. Online; accessed on 19th September, 2019.

[75] Microsoft. Configure Logging in IIS. https://bit.ly/2Tkgaru, 2013. Online; accessed on 19th September, 2019.

[76] Fredrik Abbors, Tanwir Ahmad, Dragos Truscan, and Ivan Porres. *Performance Testing in the Cloud using MBPeT*, page 191–225. TUCS General Publication. Turku Centre for Computer Science, 2013.

[77] Leonard Richardson and Sam Ruby. *Restful web services.* O'Reilly, first edition, 2007.

[78] 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, May 2018.

[79] The Apache Software Foundation. Apache JMeter. http://jmeter.apache.org/. Online; accessed on 19th September, 2019.

[80] 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*, SOQUA '06, page 54–61, New York, NY, USA, 2006. Association for Computing Machinery.

[81] David Mosberger and Tai Jin. Httperf—a tool for measur-

ing web server performance. *SIGMETRICS Perform. Eval. Rev.*, 26(3):31–37, December 1998.

[82] Giancarlo Ruffo, Rossano Schifanella, Matteo Sereno, and Roberto Politi. WALTy: A user behavior tailored tool for evaluating web application performance. In *Network Computing and Applications, 2004.(NCA 2004). Proceedings. Third IEEE International Symposium on*, pages 77–86. IEEE, 2004.

[83] Daniel A. Menascé, Lawrence W. Dowdy, and Virgílio A. F. Almeida. *Performance by Design - Computer Capacity Planning By Example.* Prentice Hall, 2004.

[84] Diwakar Krishnamurthy, Jerome A. Rolia, and Shikharesh Majumdar. A synthetic workload generation technique for stress testing session-based systems. *IEEE Trans. Software Eng.*, 32(11):868–882, 2006.

[85] Henning Schulz, Tobias Angerstein, and André van Hoorn. Towards automating representative load testing in continuous software engineering. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, page 123–126, New York, NY, USA, 2018. Association for Computing Machinery.

[86] S. M. Shariff, H. Li, C. Bezemer, A. E. Hassan, T. H. D. Nguyen, and P. Flora. Improving the testing efficiency of selenium-based load tests. In *Proc. IEEE/ACM 14th Int. Workshop Automation of Software Test (AST)*, pages 14–20, May 2019.

[87] Satish Gojare, Rahul Joshi, and Dhanashree Gaigaware. Analysis and design of selenium webdriver automation testing framework. *Procedia Computer Science*, 50:341 – 346, 2015. Big Data, Cloud and Computing Challenges.

[88] Varsha Apte, T. V. S. Viswanath, Devidas Gawali, Akhilesh Kommireddy, and Anshul Gupta. Autoperf: Automated load testing and resource usage profiling of multi-tier internet applications. In Walter Binder, Vittorio Cortellessa, Anne Koziolek, Evgenia Smirni, and Meikel Poess, editors, *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, ICPE 2017, L'Aquila, Italy, April 22-26, 2017*, pages 115–126. ACM, 2017.

[89] Cornel Barna, Marin Litoiu, and Hamoun Ghanbari. Model-based performance

testing: Nier track. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 872–875. IEEE, 2011.

[90] Xuefeng Guan, Bo Cheng, Aihong Song, and Huayi Wu. Modeling users' behavior for testing the performance of a web map tile service. *Transactions in GIS*, 18(S1):109–125, 2014.

[91] Microfocus. LoadRunner. `https://bit.ly/2QSUeCc`, 2020. Online; accessed on 19th Jan, 2020.

[92] I. Molyneaux. *The Art of Application Performance Testing: From Strategy to Tools*. Theory in practice. O'Reilly Media, 2014.

[93] Antonia Bertolino. Software testing research and practice. In Egon Börger, Angelo Gargantini, and Elvinia Riccobene, editors, *Abstract State Machines 2003*, pages 1–21, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[94] 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*, ASE '11, pages 43–52, Washington, DC, USA, 2011. IEEE Computer Society.

[95] Tanwir Ahmad, Dragos Truscan, and Ivan Porres. Identifying worst-case user scenarios for performance testing of web applications using markov-chain workload models. *Future Generation Computer Systems*, 87:910 – 920, 2018.

[96] Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[97] Richard G. Hamlet. Testing programs with the aid of a compiler. *Software Engineering, IEEE Transactions on*, (4):279–290, 1977.

[98] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, Sep. 2011.

[99] Timothy A. Budd and Ajei S. Gopal. Program testing by specification mutation. *Computer Languages*, 10(1):63 – 73, 1985.

[100] Lei Mi and Kerong Ben. A method of software specification mutation testing based on uml state diagram for consistency checking. *Procedia Engineering*, 15:110 – 114, 2011. CEIS 2011.

[101] Tanwir Ahmad, Fredrik Abbors, and Dragos Truscan. Automatic performance space exploration of web applications. *Lecture Notes in Computer Science*, 9512:223–235, 2016.

[102] Michael Olan. Unit testing: test early, test often. *Journal of Computing Sciences in Colleges*, 19(2):319–328, 2003.

[103] Donald B Johnson. Finding all the elementary circuits of a directed graph. *SIAM Journal on Computing*, 4(1):77–84, 1975.

[104] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, jul 1987.

[105] Tanwir Ahmad and Dragos Truscan. Automatic performance space exploration of web applications using genetic algorithms. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, SAC '16, pages 795–800, New York, NY, USA, 2016. ACM.

[106] Mandavilli Srinivas and Lalit M Patnaik. Genetic algorithms: A survey. *Computer*, 27(6):17–26, 1994.

[107] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13:2171–2175, jul 2012.

[108] Louis Slothouber. A model of web server performance. In *Proceedings of the 5th International World wide web Conference*, 1996.

[109] John Dilley, Rich Friedrich, Tai Jin, and Jerome Rolia. Web server performance measurement and modeling techniques. *Performance evaluation*, 33(1):5–26, 1998.

[110] Mark S Squillante, David D Yao, and Li Zhang. Web traffic modeling and web server performance analysis. In *Decision and Control, 1999. Proceedings of the 38th IEEE Conference on*, volume 5, pages 4432–4439. IEEE, 1999.

[111] Paul Reeser and Rema Hariharan. An analytic model of web servers in distributed computing environments. *Telecommunication Systems*, 21(2):283–299, 2002.

[112] Vittorio Cortellessa, Antinisca Di Marco, and Paola Inverardi. *Model-Based Software Performance Analysis.* Springer Publishing Company, Incorporated, 1st edition, 2011.

[113] 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*, ICPE '19, page 25–32, New York, NY, USA, 2019. Association for Computing Machinery.

[114] S. Duttagupta, R. Virk, and M. Nambiar. Software bottleneck analysis during performance testing. In *Proc. Int. Conf. and Workshop Computing and Communication (IEMCON)*, pages 1–7, October 2015.

[115] Aiqiang Gao, Dongqing Yang, Shiwei Tang, and Ming Zhang. Mining models of composite web services for performance analysis. *Lecture notes in computer science*, 3882:828, 2006.

[116] 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*, ICPE '18, page 211–222, New York, NY, USA, 2018. Association for Computing Machinery.

[117] Enrique Hernández-Orallo and Joan Vila-Carbó. Web server performance analysis using histogram workload models. *Computer Networks*, 53(15):2727–2739, 2009.

[118] Ágnes Bogárdi-Mészöly and Tihamér Levendovszky. A novel algorithm for performance prediction of web-based software systems. *Performance Evaluation*, 68(1):45–57, 2011.

[119] Xusheng Xiao, Shi Han, Dongmei Zhang, and Tao Xie. Context-sensitive delta inference for identifying workload-dependent performance bottlenecks. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, page 90–100, New York, NY, USA, 2013. Association for Computing Machinery.

[120] A. Nistor, L. Song, D. Marinov, and S. Lu. Toddler: Detecting performance problems via similar memory-access patterns. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 562–571, May 2013.

[121] N.R. Pusuluri. *Software Testing Concepts And Tools.* Dreamtech Press, 2006.

[122] David H Bailey and Allan Snavely. Performance modeling: Understanding the past and predicting the future. In *European Conference on Par-*

allel Processing, pages 185–195. Springer, 2005.

[123] Joachim Wegener, André Baresel, and Harmen Sthamer. Evolutionary test environment for automatic structural testing. *Information and software technology*, 43(14):841–854, 2001.

[124] Ruilian Zhao, Mark Harman, and Zheng Li. Empirical study on the efficiency of search based test generation for EFSM models. In *2010 third international conference on software testing, verification, and validation workshops*, pages 222–231. IEEE, 2010.

[125] Oliver Bühler and Joachim Wegener. Evolutionary functional testing. *Computers & Operations Research*, 35(10):3144 – 3160, 2008. Part Special Issue: Search-based Software Engineering.

[126] Gordon Fraser and Andreas Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering*, 38(2):278–292, 2011.

[127] Hirohide Haga and Akihisa Suehiro. Automatic test case generation based on genetic algorithm and mutation analysis. In *2012 IEEE International Conference on Control System, Computing and Engineering*, pages 119–123. IEEE, 2012.

[128] 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.

[129] Tom Pender. *UML bible*. John Wiley & Sons, Inc., 2003.

[130] 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, Aug 2019.

[131] 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, 2015.

[132] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling Network Architectures for Deep Reinforcement Learning. *arXiv e-prints*, page arXiv:1511.06581, Nov 2015.

[133] Richard S Sutton, Andrew G Barto, Francis Bach, et al. *Reinforcement learning: An introduction*. MIT press, 1998.

[134] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436, 2015.

[135] Matthias Plappert. keras-rl. `https://github.com/keras-rl/keras-rl`, 2016.

[136] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. Experimental assessment of random testing for object-oriented software. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 84–94, New York, NY, USA, 2007. ACM.

[137] J. W. Duran and S. C. Ntafos. An evaluation of random testing. *IEEE Transactions on Software Engineering*, SE-10(4):438–444, July 1984.

[138] Dick Hamlet. When only random testing will do. In *Proceedings of the 1st International Workshop on Random Testing*, RT '06, pages 1–9, New York, NY, USA, 2006. ACM.

[139] C. Amza, E. Cecchet, A. Chanda, Alan L. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel. Specification and implementation of dynamic web site benchmarks. 2002.

[140] Joachim Wegener, Klaus Grimm, Matthias Grochtmann, Harmen Sthamer, and Bryan Jones. Systematic testing of real-time systems. In *4th International Conference on Software Testing Analysis and Review (EuroSTAR 96)*, 1996.

[141] Monika Dhok and Murali Krishna Ramanathan. Directed test generation to detect loop inefficiencies. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, page 895–907, New York, NY, USA, 2016. Association for Computing Machinery.

[142] J. Burnim, S. Juvekar, and K. Sen. Wise: Automated test generation for worst-case complexity. In *2009 IEEE 31st International Conference on Software Engineering*, pages 463–473, May 2009.

[143] C. Saumya, J. Koo, M. Kulkarni, and S. 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, April 2019.

[144] J. Koo, C. Saumya, M. Kulkarni, and S. Bagchi. Pyse: Automatic worst-case test generation by reinforcement learning. In *Proc. Validation and Verification (ICST) 2019 12th IEEE Conf. Software Testing*, pages 136–147, April 2019.

[145] A. Aquino, G. Denaro, and P. Salza. Worst-case execution time testing via evolutionary symbolic execution. In *Proc. IEEE 29th Int. Symp. Software Reliability Engineering (ISSRE)*, pages 76–87, October 2018.

[146] Bihuan Chen, Yang Liu, and Wei Le. Generating performance distributions via probabilistic symbolic execution. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 49–60, New York, NY, USA, 2016. ACM.

[147] Luca Della Toffola, Michael Pradel, and Thomas R. Gross. Synthesizing programs that expose performance bottlenecks. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, page 314–326, New York, NY, USA, 2018. Association for Computing Machinery.

[148] 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*, ISSTA 2018, pages 254–265, New York, NY, USA, 2018. ACM.

[149] Qi Luo, Aswathy Nair, Mark Grechanik, and Denys Poshyvanyk. FOREPOST: finding performance problems automatically with feedback-directed learning software testing. *Empirical Software Engineering*, 22(1):6–56, Feb 2017.

[150] 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*, ISSTA 2015, pages 270–281, New York, NY, USA, 2015. ACM.

[151] Michael T. M. Emmerich and André H. Deutz. A tutorial on multiobjective optimization: fundamentals and evolutionary methods. *Natural Computing*, 17(3):585–609, Sep 2018.

Tanwir Ahmad

Performance Exploration and Testing of Web-based Software Systems

Modern society relies heavily on a wide range of inter-connected software systems for finance, energy distribution, communication, and transportation. Due to the adoption of the Internet, almost all financial, government, and social sectors rely heavily on web-based information systems. These systems need to be very fast and reliable, and should be able to support a vast number of concurrent users. As software users are immensely perceptive about the performance of the software system, the companies relying on web-based application systems for businesses strive to provide high-quality web services in order to stay competitive in the worldwide market.

In this thesis, we propose a set of approaches for performance testing and exploration of web-based software systems. Although we target web-based software systems, our methods can be easily adapted to different types of software systems. Our contributions fall into two categories: approaches for model-based performance testing and approaches for performance explorations of black-box systems with large input spaces. In the first category, as a first contribution, we provide model-based performance testing, where we generate realistic workloads using probabilistic models in order to benchmark the performance of the system under test. As an extension of the first contribution, we provide an approach for extracting the workload models from server logs as an alternative to their manual creation based on the tester's experience. In the second category of contributions, we are interested in exploring the performance of black-box software systems with large input spaces without prior knowledge of the domain. We propose different exploratory performance testing approaches to identify not only the worst user scenario with respect to a given workload model but also a set of input combinations that trigger performance issues and severely degrade the performance of software-intensive systems.

ISBN 978-952-12-4000-3