Machine Learning to Guide Performance Testing: An Autonomous Test Framework

Mahshid Helali Moghadam[1,2], Mehrdad Saadatmand[1], Markus Borg[1], Markus Bohlin[1], Björn Lisper[2]
[1]Research Institutes of Sweden (RISE) SICS, Sweden
[2]Mälardalen University, Västerås, Sweden
mahshid.helali.moghadam@ri.se, mehrdad.saadatmand@ri.se, markus.borg@ri.se, markus.bohlin@ri.se, bjorn.lisper@mdh.se

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

Keywords—performance requirements; performance testing; test case generation; reinforcement learning; autonomous testing

I. Introduction

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

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

Performance testing is intended to ascertain whether the software system performs well under the actual execution conditions (i.e., internal and external factors affecting the performance) and meets the performance requirements. Various methods have been proposed for building software performance models [3, 4, 5, 6]. Performance models might provide helpful hints of the performance of the system and even probable bottlenecks in the architecture; however, they cannot represent the whole details of the system. For example, many of the details of the deployment environment may be ignored in the models [7], although they might still have significant effects on the performance of the system. Testing the software system under stress, which is called stress testing, is one of directions involved in performance testing. The main objective is to find the performance breaking point, at which the systems breaks, or performance requirements are not met anymore. Two general views as internal and external could be assigned to performance analysis. Analysis with internal view considers internal conditions causing a performance bottleneck and consequently affecting the performance of the system. Performance testing with external view is evaluating/examining how the system will perform under different external conditions like heavier workload and limited resource availability.

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

The rest of this paper is organized as follows; Section 2 discusses the background concepts of RL and the motivation for proposing learning-based testing, Section 3 presents the details of the proposed smart performance testing framework, with a short discussion on its applicability and operational performance. Section 4 provides a review on the background relevant approaches. The paper concludes with a conclusion and future directions of this work-in-progress research in Section 5.

II. Motivation and Background

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

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

III. Self-adaptive Learning-based Performance Testing

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

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

Transfer learning. After the initial convergence of experience bases, the smart agent keeps the learning running to update the knowledge bases upon observing new SUTs. It is supposed that during the transfer learning, the smart agent mostly relies on the achieved experience, while also partly explores the environment to keep the gained knowledge updated. Using the learnt policy during the interaction with SUT instances, causes the agent to generate the stress test cases/test conditions to find the performance breaking point with less effort (in terms of learning trials) and leads to a better efficiency. *Experience exploitation* is the key concept of this phase which results in more efficiency in test case generation. The policies learnt for CPU-, memory- and disk-intensive programs are quite different. Then, this is where separating the experience bases of the agent is beneficial. Upon observing a CPU-, memory- or disk-intensive SUT, the agent activates the corresponding experience base for taking actions on the observed SUT instance. Fig.1 depicts an overview of the architecture of our smart tester agent. The details including the components, and main steps of the learning part is as follows:

I. State Detection. Detecting the current state of the system is one of the main steps of an RL-based algorithm. In our smart framework, four measurements of the SUT and execution environment including CPU, memory and disk utilization, and also SUT response time are used to specify the state of the system. The state detector component receives a tuple consisting of (CPU_U, Mem_U, Disk_U, Rt) as input to specify the state of the system, where CPU_U, Mem_U, Disk_U, Rt present the CPU, memory, disk utilization and response time respectively. These continuous parameters form the state space of the system, then the next step is dividing the state space into multiple discrete states. The values of these parameters are classified into multiple classes to specify the discrete states of the system, as shown in Fig. 2.

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

$$DecFac\_CPU = \left\{\frac{1}{4}, \frac{2}{4}, \frac{3}{4}, 1\right\} \quad (1)$$

$$DecFac\_MemDisk = \left\{(d.\frac{memory\ (disk)}{4}) \middle| \ d \in \{\frac{1}{4}, \frac{2}{4}, \frac{3}{4}, 1\}\right\} \quad (2)$$

where *memory (disk)* represent the current amount of available memory (disk). Then, the set of actions have been specified as shown in Table 1.

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

$$U(n) = kU_r(n) + (1-k)U_E \quad (3)$$

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

IV. Experience Adaptation. This component receives a performance sensitivity indication expressing the type of sensitivity of SUT, i.e., being CPU-, memory- or disk-intensive. Then, it selects the corresponding experience (knowledge) base for the stress test case generation on the observed SUT.

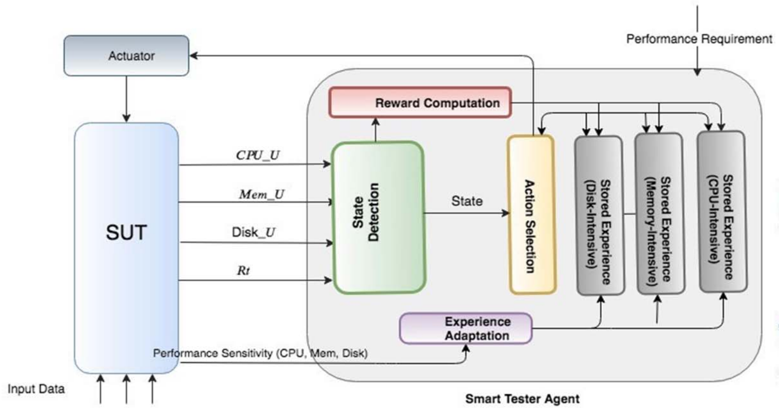

Fig. 1 architecture of our smart framework

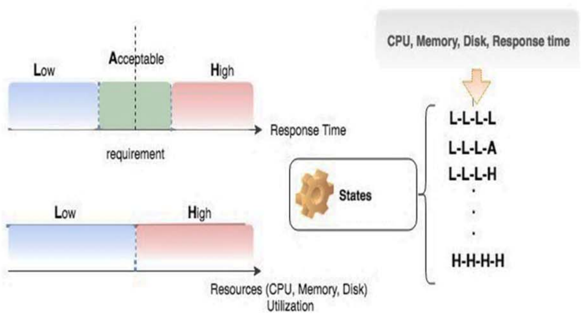

Fig. 2 states of the system

TABLE 1 set of actions

Actions	
Reducing the available CPU cores	by a factor in *DecFac_CPU*
Reducing the available memory	by a factor in *DecFac_MemDisk*
Reducing the available disk	
No action	

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

$$Q^{\pi}(s,a) = E^{\pi}[R_n | S_n = s, A_n = a] \qquad (4)$$

$$R_n = \sum_{k=0}^{\infty} \gamma^k r_{n+k+1} \qquad (5)$$

Where S_n , A_n and r_{n+k+1} are current state, action and future rewards respectively. $\gamma \in [0,1]$ is a discount factor specifying to what extent the agent gives more weight to the future reward compared to the immediately achieved reward. Q-values are stored in a look-up table (Q-table) and considered as the experience of the agent. The q-values are used for deciding between actions when the agent relies on using its experience (exploitation). The q-values are also updated incrementally (via temporal differencing) during the learning using Eq. 6. The final objective of Q-learning is finding a policy maximizing the expected long-term reward of pairs of states and actions.

$$Q(s_n,a_n) = Q(s_n,a_n) + \alpha[r_{n+1} + \gamma \max_{a'} Q(s_{n+1},a') - Q(s_n,a_n)] \ (6)$$

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

How the performance test is done. In our smart framework, the agent learns how to provide efficiently stress conditions for different types of SUTs to find the performance breaking point, from which the performance requirement of the SUT is not met anymore. The agent stops applying actions upon reaching the breaking point. List 1 presents the operating procedure of our learning-based performance testing framework.

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

Algorithm: Learning-based Performance Testing

```
Initialize q-values, $Q(s,a) = 0 \ \forall s \in S, \forall a \in A$
Observe and identify the type of the SUT instance.
If SUT is the first instance of CPU-intensive, memory-intensive or disk-intensive instances
    Repeat until the initial convergence:  //Initial Convergence
    1. Detect the state of the SUT.
    2. Select an action based on the action selection strategy.
    3. Apply the selected action, re-execute the SUT
    4. Detect the new state of the system regarding the selected action.
    5. Receive the reward signal, $r_{n+1}$
    6. Update the q-value in the corresponding experience base (Q-table)
Else
    7. Select the proper experience (knowledge) base
    Repeat until finding performance breaking point // Transfer learning
    8. Detect the state of the SUT
    9. Select an action,
    an= $argmax_{a \in A} Q(s_n,a)$ from the experience base with probability (1- ε) or a random action with probability ε, $\varepsilon \leq 0.2$)
    10. Apply the selected action, re-execute the SUT
    11. Detect the new state of the system
    12. Receive the reward signal, $r_{n+1}$
    13. Update the q-value
```

IV. Related Work

Model-based software performance engineering is mainly based on building a performance model of the system. Some of the specific modeling notations used for performance modeling are Queueing Networks, Markov Process, Petri Nets, Process Algebras and also simulation models [3, 4, 5, 6]. Pushing towards automation in generating the performance model is essential to eliminate the manual model generation and bring the performance analysis to early stages in the software life cycle. There has been a substantial literature published in the area of software performance modeling [2, 9, 10]. In the context of testing, although performance testing tightly overlaps with performance modeling in some cases, it is intended to satisfy some partially different objectives. Performance testing, load testing and stress testing are three terms which are used in many cases interchangeably [11].

In general, load testing has been considered as behavior assessment of a SUT under load expected in a real-world execution context, from two perspectives of functional problems and violation of non-functional requirements caused under load. Stress testing has been defined as the behavioral assessment of a SUT under extreme conditions including heavy load and limited available resources [11]. Performance testing is often considered as a more general term which often includes both load testing and stress testing. There are many commonalities between these types of testing. Regarding common and different performance test scenarios, the target domain of the relevant works could be summarized as: The behavioral assessment of a SUT from the perspective of performance-related issues and aspects generally aim at the following objectives:

I. Measurement of performance under load and/or different resource configuration. This process might overlap with performance modeling in many cases. It can be done under expected load or stress conditions [12, 13, 14, 15].

II. Detection of functional problems under load and/or different resource configuration. It can be also done under expected load or stress conditions [16, 17].

III. Detection of performance requirements violation such as violating reliability, and robustness requirements. This process can also be done under typical expected load or stress conditions [18].

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

V. Conclusion

Performance analysis to provide an estimation of performance indices in different execution conditions is a challenge for complex software systems. In addition to static model-driven techniques, performance testing is considered as a dynamic approach for performance analysis. Efficient automated test case/test condition generation is a challenging activity in software testing. In this paper, we present a self-adaptive learning-based framework to conduct stress testing on various software systems without having the performance models of the systems. We used Q-learning as a model-free RL algorithm in our smart test framework. It learns the optimal policy of generating stress test cases for various software systems. After the initial learning convergence, it uses the learnt policy for further SUT instances and generates test cases efficiently with less required effort. Detailed efficacy analysis of the proposed framework on different software systems and deploying it on a virtualized infrastructure will be our next steps in this research.

VI. Acknowledgements

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

References

[1] M. Glinz. On non-functional requirements. In 15th IEEE International Requirements Engineering Conference, RE'07, pp. 21-26. IEEE, 2007.

[2] V. Cortellessa, A. Di Marco and P. Inverardi. 2011. Model-based software performance analysis. Springer Science & Business Media

[3] D. Petriu, C. Shousha and A. Jalnapurkar. Architecture-based performance analysis applied to a telecommunication system. IEEE Transactions on Software Engineering, (11), 1049-1065, 2000.

[4] R. Pooley. Using UML to derive stochastic process algebra models. In Proceedings of the 15th UK Performance Engineering Workshop (UKPEW), pages 23–34, 1999.

[5] S. Bernardi, S. Donatelli and J. Merseguer. From UML sequence diagrams and state charts to analyzable petri net models. In Proceedings of the 3rd international workshop on Software and performance, pp. 35-45. ACM, 2002.

[6] E. D. Lazowska, J. Zahorjan, G. S. Graham and K. C. Sevcik. 1984. Quantitative system performance: computer system analysis using queueing network models. Prentice-Hall, Inc.

[7] G. Denaro, A. Polini and W. Emmerich. Early performance testing of distributed software applications. In ACM SIGSOFT Software Engineering Notes,Vol. 29, No. 1, pp. 94-103. ACM, 2004.

[8] R. S. Sutton, A. G Barto. 1998. Reinforcement learning: An Introduction. Vol. 1. MIT press Cambridge.

[9] M. K. Loukides. 1996. System performance tuning. O'Reilly & Associates, Inc.

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

[11] Z. M. Jiang and A. E. Hassan. A survey on load testing of large-scale software systems. IEEE Transactions on Software Engineering, 41(11), 1091-1118, 2015.

[12] D. A. Menasce. Load testing, benchmarking, and application performance management for the web. in Proc. Comput. Manag. Group Conf., pp. 271–281, 2002.

[13] M. Kalita and T. Bezboruah. Investigation on performance testing and evaluation of prewebd: A .net technique for implementing web application, IET Softw., vol. 5, no. 4, pp. 357–365, 2011.

[14] M. Helali Moghadam, M. Saadatmand, M. Borg, M. Bohlin and B. Lisper. Learning-Based Response Time Analysis in Real-Time Embedded Systems: A Simulation-Based Approach. IEEE/ACM 1st International Workshop on Software Qualities and their Dependencies (SQUADE), pp. 21-24. IEEE/ACM, 2018.

[15] O. Ibidunmoye, M. H. Moghadam, E. B. Lakew, and E. Elmroth. Adaptive Service Performance Control using Cooperative Fuzzy Reinforcement Learning in Virtualized Environments. In Proc. 10th International Conference on Utility and Cloud Computing, pp. 19-28. ACM, 2017.

[16] B. Dillenseger. Clif, a framework based on fractal for flexible, distributed load testing, Ann. Telecommun., vol. 64, pp. 101–120, 2009.

[17] F. Huebner, K. S. Meier-Hellstern, and P. Reeser. Performance testing for IP services and systems. in Performance Eng., State of the Art and Current Trends, pp. 283–299, New York, NY, USA: Springer, 2001

[18] S. Abu-Nimeh, S. Nair, and M. Marchetti. Avoiding denial of service via stress testing. IEEE Int. Conf. Comput. Syst. Appl., pp. 300–307, 2006