Machine Learning-Assisted Performance Testing

Mahshid Helali Moghadam  
RISE Research Institutes of Sweden, Mälardalen University  
Västerås, Sweden  
mahshid.helali.moghadam@ri.se

ABSTRACT

Automated testing activities like automated test case generation imply a reduction in human effort and cost, with the potential to impact the test coverage positively. If the optimal policy, i.e., the course of actions adopted, for performing the intended test activity could be learnt by the testing system, i.e., a smart tester agent, then the learnt policy could be reused in analogous situations which leads to even more efficiency in terms of required efforts. Performance testing under stress execution conditions, i.e., stress testing, which involves providing extreme test conditions to find the performance breaking points, remains a challenge, particularly for complex software systems. Some common approaches for generating stress test conditions are based on source code or system model analysis, or use-case based design approaches. However, source code or precise system models might not be easily available for testing. Moreover, drawing a precise performance model is often difficult, particularly for complex systems. In this research, I have used model-free reinforcement learning to build a self-adaptive autonomous stress testing framework which is able to learn the optimal policy for stress test case generation without having a model of the system under test. The conducted experimental analysis shows that the proposed smart framework is able to generate the stress test conditions for different software systems efficiently and adaptively without access to performance models.

1 RESEARCH PROBLEM

Performance describes the resource and time bound aspects of a software system behavior and is often described in terms of some sub-characteristics like time behavior, resource utilization and capacity [1]. Performance evaluation is generally conducted to I. measure the performance metrics, II. detect the functional problems appearing under certain performance-related execution conditions like heavy workload, III. detect non-functional problems, i.e., violations of non-functional requirements such as performance and robustness. Stress testing is often considered as a type of performance testing which implies applying extreme execution conditions to meet those objectives and verify the robustness of the system. Finding the performance breaking point of the software under test (SUT), at which the system functionality breaks, or the performance requirements are violated, is one of the main objectives in the stress testing activity. The emergence of anomalies in the performance behavior of a software system is resulted from performance bottlenecks. There are various application-, platform-, and workload-wise causes resulting in the emergence of performance bottlenecks. In stress testing, providing extreme (stress) test conditions involves changing (manipulating) the platform- and workload-wise factors affecting the performance. Generating stress test conditions to analyze the performance behavior under extreme conditions remains a challenge for complex software systems.

2 MOTIVATION AND BACKGROUND

Performance modeling and performance testing are common approaches for doing performance analysis. Performance modeling is often based on building a performance model of the system behavior and measuring the target performance metrics. It can be done using various modeling notations like Markov Processes, queueing networks, petri nets and simulation models [2, 3, 4, 5, 6, 7]. Performance testing is considered as a family of performance-related testing techniques intended for addressing the objectives of performance analysis. Performance, load and stress testing might considerably overlap in many areas. Nevertheless, the objectives of the performance-related testing methods could be summarized as follows:

I. Measuring the performance metrics under different execution conditions including various workload and resource configurations [8, 9, 10, 11, 12, 13, 14, 15, 16, 17].

II. Detecting the functional problems appearing under certain execution conditions regarding workload and resource configurations [18, 19, 20, 21, 22, 23].
III. Detecting violations of non-functional requirements under expected and stress conditions [8, 13, 24, 25, 26, 27, 28, 29].
In general, using source code or system model analysis or use-case based design techniques are the common approaches for addressing the mentioned challenge. However, first, relying on the source code or system model like performance model might imply some limitations upon unavailability of these artifacts. Secondly, drawing a precise model of the performance behavior of a software system is challenging particularly for complex systems, while still many implementation and deployment details are often ignored. These are the motivations for using model-free learning-based techniques in which the optimal policy for addressing the problem can be learnt indirectly without having a model of the system and environment.

3 APPROACH

How it addresses the problem. The proposed solution involves a stress (robustness) test case generator to find the performance breaking point of SUTs. It is able to learn the optimal policy for generating stress test conditions to find the performance breaking point of the SUT without access to the performance model.

How it works. The proposed stress testing framework assumes two phases of learning, i.e., initial and transfer learning. The agent learns the optimal policy initially during the initial learning. During the transfer learning it replays the learnt policy in further analogous situations, i.e., upon observing SUTs with similar performance sensitivity, while keeping the learning running.

Learning Technique. Q-learning, i.e. a model-free reinforcement learning (RL) [30], is used as the core learning algorithm. In RL, the agent senses the state of the system, which is the SUT in this case, continuously. Upon the state detection, it takes a possible action randomly or selects a high valued action. Then, it receives a reward signal indicating the effectiveness of the applied action. In the proposed framework the mentioned steps have been formulated as follows:
- State detection: The state of the system is identified based on the quality measurements of the SUT and execution environment, i.e., CPU, memory and disk utilization, and SUT response time.
- Actions: They are operations modifying (reducing) the factors affecting the performance, e.g. available resource capacity and characteristics of workload. $\varepsilon$-greedy was used as the core strategy for action selection.
- Reward signal: A utility function which is a weighted linear combination of two functions describing the response time deviation from the requirement and the resource usage respectively, was derived for the reward signal.

\section*{Profound Added Features.}

I. At type I of this framework, Fig.1 shows its architecture, Q-learning augmented with experience adaptation through using multi-experience bases, was used. The smart tester agent uses separate experience bases for storing the learnt policy based on the type of performance sensitivity of the SUTs. It leads to the efficiency improvement of the agent in the transfer learning [31].
II. At type II, which is a self-adaptive fuzzy reinforcement learning-based (SaFRel) stress testing, as shown in Fig. 2, an action selection strategy adaption, was applied which acts as a meta-learning feature and is intended to improve the performance of the learning by applying adaptive changes to the action selection strategy based on detected differences between the performance sensitivity of observed SUTs. To address the issues related to the crisp categorization of the discrete state modelling, fuzzy classification was also used for state modeling (detection).

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{smart_stress_testing_framework_type_1}
\caption{smart stress testing framework, type I}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{type_ii_safrel}
\caption{Type II, SaFRel architecture and algorithm}
\end{figure}

\begin{algorithm}[h]
\caption{SaFRel: Self-adaptive Fuzzy Reinforcement Learning-based Stress Testing}
Required: S, A, $\alpha$, $\gamma$; Initialize $Q(s,a) = 0 \forall s \in S, \forall a \in A$
and $\varepsilon = \gamma$, $0 < \gamma < 1$
1. Observe the first SUT instance.
2. Repeat until initial convergence \textit{(initial learning phase)}:
   2.1. Fuzzy Q-Learning Episode with initial action selection strategy (e.g. $\varepsilon$-greedy, initialized $\varepsilon$)
   3. Store the obtained experience
   4. Start the transfer learning phase.
5. Repeat:
   5.1 Observe a new SUT instance
   5.2 Measure the similarity
   5.3 Apply strategy adaptation, i.e., adjust the degree of exploration and exploitation (e.g. tuning parameter $\varepsilon$ in $\varepsilon$-greedy)
   5.4 Fuzzy Q-Learning Episode with adapted strategy (e.g., new $\varepsilon$)
\end{algorithm}

4 CONCLUSION AND RESULTS

We have evaluated the efficacy of the proposed approach by simulating the performance behavior of 12 bench-mark programs such as Build-apache, n-queens, dcreaw, etc. Improved efficiency in terms of reduced effort, i.e., time and cost, for generating the test conditions while reducing dependency on source code and system models, is the achievement of the proposed learning-based stress testing. Regarding the applicability, software variants in software product lines and evolving software programs in CI/CD would be well-suited application areas for this approach. Extending the approach to support workload-wise factors in generating the stress test conditions is the current ongoing part of this research.
REFERENCES


