Software Quality Journal
https://doi.org/10.1007/s11219-020-09532-z

An autonomous performance testing framework using self-adaptive fuzzy reinforcement learning

Mahshid Helali Moghadam[1,2] · Mehrdad Saadatmand[1] · Markus Borg[1] · Markus Bohlin[2] · Björn Lisper[2]

Accepted: 24 September 2020

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

Keywords Performance testing · Stress testing · Test case generation · Reinforcement learning · Autonomous testing

✉ Mahshid Helali Moghadam
mahshid.helali.moghadam@ri.se

Mehrdad Saadatmand
mehrdad.saadatmand@ri.se

Markus Borg
markus.borg@ri.se

Markus Bohlin
markus.bohlin@mdh.se

Björn Lisper
bjorn.lisper@mdh.se

[1] RISE Research Institutes of Sweden, Västerås and Lund, Sweden

[2] Mälardalen University, Högskoleplan 1, 72220 Västeras, Sweden

1 Introduction

Quality assurance with respect to both functional and non-functional quality characteristics of software becomes crucial to the success of software products. For example, an extra one-second delay in the load time of a storefront page can cause 11% reduction in page views, and 16% less customer satisfaction (NS8 2018). Moreover, banking, retailing, and airline reservation systems as samples of mission-critical systems are all required to be resilient against varying conditions affecting their functional performance (Weyuker and Vokolos 2000; Brunnert et al. 2015; Grinshpan 2012).

Performance, which has been also called ”efficiency” in the classification schemes of quality characteristics (ISO25000 2019; Glinz 2007; Chung et al. 2012), is generally referred to as how well a software system (service) accomplishes the expected functionalities. Performance requirements mainly describe time and resource bound constraints on the behavior of software, which are often expressed in terms of performance metrics such as response time, throughput, and resource utilization.

Performance evaluation. Performance modeling and testing are common evaluation approaches to accomplish the associated objectives such as measurement of performance metrics, detection of functional problems emerging under certain performance conditions, and also violations of performance requirements (Jiang and Hassan 2015). Performance modeling mainly involves building a model of the software system’s behavior using modeling notations such as queueing networks, Markov processes, Petri nets, and simulation models (Cortellessa et al. 2011; Harchol-Balter 2013; Kant and Srinivasan 1992). Although models provide helpful insights into the performance behavior of the system, there are also many details of implementation and execution platform that might be ignored in the modeling (Denaro et al. 2004). Moreover, drawing a precise model expressing the performance behavior of the software under different conditions is often difficult. Performance testing as another family of techniques is intended to achieve the aforementioned objectives by executing the software under the actual conditions.

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

Research challenge. Performance testing to find performance breaking points remains a challenge for complex software and execution platforms. Testing approaches mainly raise issues of automated and efficient generation of test cases (test conditions) resulting in accomplishing the intended objective. Common approaches for generating the performance test cases such as using source code analysis (Zhang et al. 2012), linear programs and evolutionary algorithms on performance models (Zhang and Cheung 2002; Gu and Ge 2009; Di Penta et al. 2007) and UML models (Garousi 2010; Garousi 2008; Garousi et al. 2008; Costa et al. 2012; da Silveira et al. 2011), using use case-based (Draheim et al. 2006; Lutteroth and Weber 2008), and behavior-driven techniques (Schulz et al. 2019; Ferme and Pautasso 2018; Ferme and Pautasso 2017; Walter et al. 2016) mainly rely on source code or other artifacts, which might not always be available during the testing.

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

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

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

Finding performance breaking point is a key purpose in robustness analysis, which is of great importance for many types of software systems, particularly in mission- and safety-critical domains (Fowler 2009). Moreover, the question above is worth exploring also in applications specifically, such as resource management (scaling, provisioning, and scheduling) for cloud services (Jennings & Stadler 2015), performance prediction (Venkataraman et al. 2016; Kolesnikov et al. 2019), and performance analysis of software services in other areas (Morabito 2017; Babovic et al. 2016).

Contribution. In this paper, we present the design and experimental evaluation of a self-adaptive fuzzy reinforcement learning-based (SaFReL) performance testing framework. It is intended to efficiently and adaptively generate the (platform-based) performance test conditions leading to the performance breaking point for different software programs with different performance sensitivity to resources (e.g., CPU-, memory- and disk-intensive programs) without access to source code and performance models. An early-stage general formulation of the idea of using RL particularly in performance testing was introduced in our prior work (Moghadam et al. 2019). The initial formulation introduces a single smart tester agent that uses RL (simple Q-learning) in a two-phase learning together with an initial architecture in the abstract. This paper extends the initial abstract formulation of the RL-assisted performance testing (Moghadam et al. 2019). It uses an elaborate learning technique originally inspired by the conference paper by Ibidunmoye et al. (2017), which presents an adaptive performance (response time) control approach for cloud services using cooperative fuzzy multi-agent reinforcement learning. However, regarding the distinguishing learning details, the proposed RL-assisted performance testing framework is based on a single smart agent, involves two distinct phases of learning, and benefits a particular adaptive learning strategy which plays an important role in the functionality of the agent. The proposed smart performance testing framework is intended to conduct performance testing to meet a testing objective that is finding an intended performance breaking point. The proposed framework, SaFReL, is a two-phase RL-assisted performance testing agent that is able to learn the efficient generation of performance test cases to meet the testing objective and more importantly replay the learned policy in further similar testing situations.

SaFReL assumes two phases of learning: initial and transfer learning. In the initial learning phase, it learns the optimal policy to generate the target performance test cases initially upon observing the behavior of the first SUT. Afterward in the transfer learning, it reuses the learned policy for the SUTs with a performance sensitivity analogous to already observed ones while still keeping the learning running in the long term. The learning mechanism uses Q-learning augmented by fuzzy logic in one part of the learning to deal with the issue of uncertainty in defining discrete categories over continuous values as used by Ibidunmoye et al. (2017). The single light-weight RL tester agent has the capability of transfer learning and reusing knowledge in similar situations. It benefits an adaptive action selection strategy that adapts the learning to various testing situations and subsequently makes the agent able to act efficiently on various SUTs.

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

- A smart performance testing framework (agent) that learns the optimal policy (way) to generate the performance test cases meeting the testing objective without access to source code and models and reuses the learned policy in further testing cases. It uses fuzzy RL and an adaptive action selection strategy for the generation of test cases and implements two phases of learning:
  - Initial learning during which the agent learns the optimal policy for the first time,
  - Transfer learning during which the agent replays the learned policy in similar cases while keeping the learning running in the long term.
- A twofold experimental evaluation involving performance (efficiency and adaptivity) and sensitivity analysis of the approach. The evaluation is carried out based on simulating the performance behavior of various SUTs. We use a performance simulation module instead of actually executing SUTs. The main function of the performance simulation module is estimating the performance behavior of SUTs in terms of their response time.

Structure of the paper. The rest of the paper is organized as follows: Section 2 discusses the background concepts and motivations for the proposed self-adaptive learning-based approach. Section 3 presents an overview of the architecture of the proposed testing framework, while the technical details of the constituent parts are described in Sections 4 and 5. In Section 6, we explain the functions of the learning phases. Section 7 reports on the experimental evaluation involving the experiment's setup, and the results of the experimentation. Section 8 discusses the results, the lessons learned during the experimentation, and also the threats to the validity of the results. Section 9 provides a review of the related work, and finally, Section 10 concludes the paper and discusses some future directions.

2 Motivation and background

Performance analysis, realized through modeling or testing, is important for performance-critical software systems in various domains. Anomalies in the performance behavior of a software system or violations of performance requirements are generally consequences of the emergence of performance bottlenecks at the system or platform levels (Ibidunmoye et al. 2015; Chandola et al. 2009). A performance bottleneck is a system or resource component limiting the performance of the system and hinders the system from acting as required (Gregg 2013). The behavior of a bottleneck component is due to some limitations associated with the component such as saturation and contention. A system or resource component saturation happens upon full utilization of its capacity or when the utilization exceeds a usage threshold (Gregg 2013). Capacity expresses the maximum available processing power, service (giving) rate, or storage size. Contention occurs when multiple processes contend for accessing a limited number of shared components such as resource components (e.g., CPU cycles, memory, and disk) or software (application) components.

There are various application-, platform- and workload-based causes for the emergence of performance bottlenecks (Ibidunmoye et al. 2015). Application-based causes represent issues such as defects in the source code or system architecture faults. Platform-based causes characterize the issues related to hardware resources, operating system, and execution platform. High deviations from the expected workload intensity and similar issues such as workload burstiness are denoted by workload-based causes.

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

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

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

SaFReL for performance (stress) testing of various releases or variants. SaFReL provides an agile efficient performance test case generation technique (See Section 7 and Section 8 for efficiency evaluation) while eliminating the need for source code or system model analysis.

2.1 Reinforcement learning

Reinforcement learning (RL) (Sutton and Barto 2018) is a fundamental category of machine learning algorithms generally intended to find the optimal behavior (way) in decision-making problems. RL is an interactive learning paradigm that is different from the common supervised and unsupervised machine learning algorithms and has been frequently applied to building many self-adaptive smart systems. It involves continuous interaction between the agent (learner) and the environment that is controlled. At each step of the interaction, the agent observes (senses) the *state* of the environment, takes a possible *action,* and receives a reinforcement signal as a scalar *reward* from the environment that shows the effectiveness of the applied action to guide the agent toward accomplishing the intended objective. There is no supervisor in RL, and the agent just receives a reward signal. RL basically involves a sequential decision-making process. The RL agent goes through the environment, decides how to behave at each step, and based on optimizing the long-term received reward, learns the optimal way of decision making.

The agent actually decides between actions based on the history of its observations. However, considering the whole history of observations is not efficient, therefore, *state* should be formulated as a concise summary of the history including all the required information. Keeping in mind this issue, a related helpful concept to formulate the state as a summary function is the *Markov state*. The states of the environment are Markov by definition. Then, when the environment is fully observable to the agent, the states that the agent observes and uses for making decisions, are Markov too. The environment in our case is the SUT and the execution platform. The state is modeled in terms of response time and resource utilization improvement. The actions are some operations for modifying/adjusting the available capacity of resources and the objective of the agent is finding an intended performance breaking point. Figure 1 shows the interaction between the agent and the environment that is the composition of SUT and execution platform in our case.

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

Model-free RL algorithms are special types of RL that are not intended to build or learn a model of the environment. Instead, they learn the optimal behavior to achieve the intended objective through multiple experiences of interaction with the environment. Temporal difference (TD) (Sutton and Barto 2018) is one of the main types of model-free RL, which is able to learn from the incomplete episodes of the interaction with the environment. Q-learning, as a model-free TD, learns the optimal policy through learning the optimal value function, i.e., Q-values. It uses an action selection strategy based on a combination of trying out the available actions, namely exploration, and relying on the previously achieved experience to select the highly-valued actions, namely exploitation. It is off-policy, which means that the agent learns the optimal policy regardless of how the agent

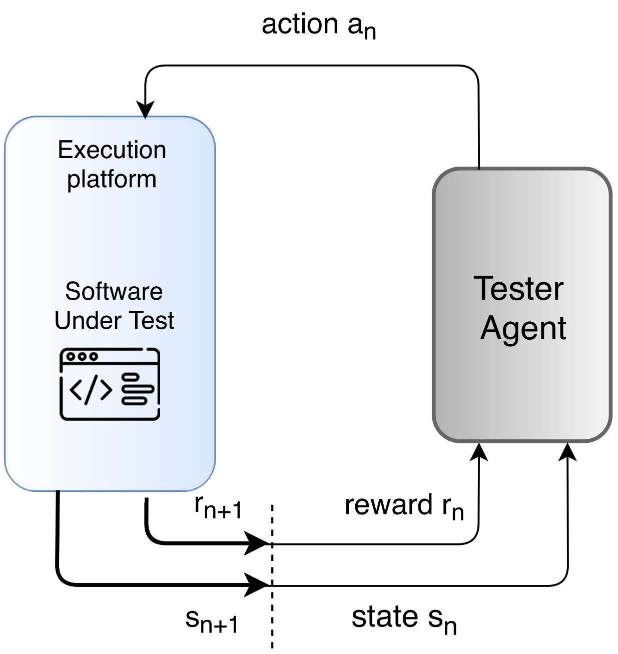


Fig. 1 Interaction between agent and SUT in RL

explores the environment. After learning the optimal policy, in the transfer learning phase, the agent is able to replay the learned policy while keeping the learning running, which implies occasionally exploring the action space and trying out different actions.

3 Architecture

This section provides an overview of the architecture of the proposed smart performance testing framework, SaFReL (see Fig. 2). The entire interaction of the smart framework with each SUT, as a learning episode, consists of a number of learning trials. The steps of learning in each trial and the components involved in each step are described as follows:

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

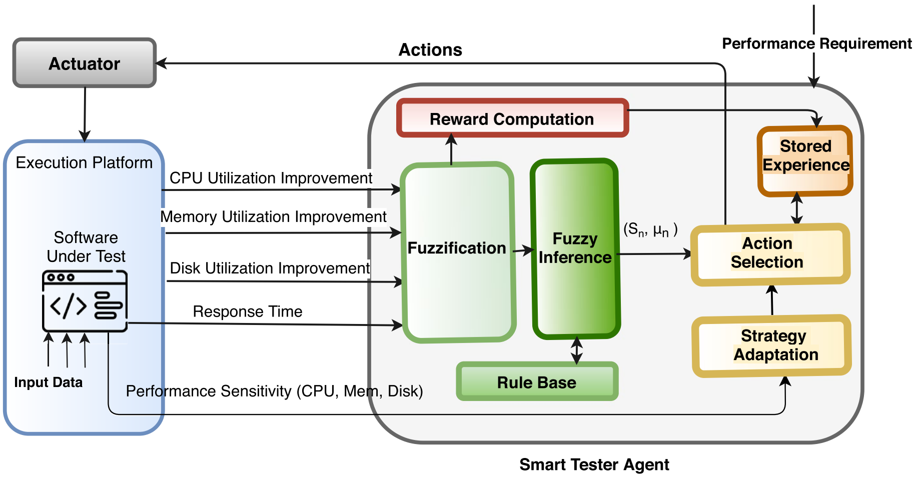


Fig. 2 SaFReL architecture

address these challenges. We use fuzzy classification as a soft labeling technique for presenting the values of the metrics used for modeling the state of the environment. Then, using a fuzzy inference engine and fuzzy rule base, the agent detects the fuzzy state of the environment. More details about the fuzzy state detection of the agent are presented in Section 4.

2. *Action Selection and Strategy Adaptation.* After detecting the fuzzy state of the SUT, the agent takes an action. The actions are operations modifying the factors affecting the performance, i.e., the available resource capacity, in the current prototype. The agent selects the action according to an *action selection strategy* that it follows. The action selection strategy determines to what extent the agent should explore and try out the available actions, and to what extent it should rely on the learned policy and select a high-value action that has been tried and assessed before. The role of this strategy is guiding the action selection of the agent throughout the learning and is of importance for the efficiency of the learning. In order to obtain the desired efficiency, a proper trade-off between the exploration of the state action space and exploitation of the previously learned policy is critical. In our proposed framework, the smart agent is augmented by a *strategy adaptation* characteristic, as a meta-learning feature responsible for dynamically adapting the degree of exploration and exploitation in various situations. This feature makes SaFReL able to detect where it should rely on the previously learned policy and where it should make a change in the strategy to update its policy and adapt to new situations. New situations mean acting on new SUTs that are different from the previously observed ones in terms of performance sensitivity to resources. Software programs have different levels of sensitivity to resources. SUTs with different performance sensitivity to resources, e.g., CPU-intensive, memory-intensive, or disk-intensive SUTs, will react to changes in resource availability differently. Therefore, when the agent observes a SUT that is different from the previously observed ones in terms of performance sensitivity, the strategy adaptation tries to guide the agent toward doing more exploration than exploitation. A performance sensitivity indicator showing the sensitivity of SUT to the resources (i.e., being CPU-intensive, memory-intensive, or disk-intensive) is an input to the strategy adaptation mechanism (see Fig. 2). The components corresponding to

the action selection, the stored experience (learned policy), and the strategy adaptation are shown as yellow components in Fig. 2. More details about the set of actions and the mechanism of strategy adaptation are described in Section 5.

3. *Reward Computation.* After taking the selected action, the agent receives a reward signal indicating the effectiveness of the applied action to approach the intended performance breaking point. The reward computation component (red block) in Fig. 2 calculates the received reward (see Section 5) for the taken actions.

4 Fuzzy state detection

The state space of the environment in our learning problem is modeled by the quality measurements, CPU, memory, and disk resource utilization improvement and response time of the SUT, which is shown in Fig. 3. The learning approach works based on detecting (discrete) states of the system. These states could be typically defined based on classifying the continuous values of the quality measurements that were mentioned above. On the other hand, defining such crisp boundaries on a number of continuous domains is an issue that might involve many uncertainties. In order to address this issue and preserve the desired precision of the model, fuzzy classification and reasoning are used to specify the states of the system. Therefore, the states of the environment are defined in terms of some fuzzy states and the environment can be in one or more fuzzy states at the same time with different degrees of certainty. The agent detects the state of the system using a fuzzy inference engine and a rule base (Kuncheva 2008; MathWorks 2019) (Fig. 2). In summary, the step of state detection is done based on making fuzzy inference about the state of the system. The fuzzy state detection consists of three main parts: normalization of the input values (quality measurements), fuzzification of the measurements, and the fuzzy inference to identify the state of the environment. The details of these parts together with the fuzzy rules, fuzzy operators, and the implication method that are used, are described in Section 4.1.

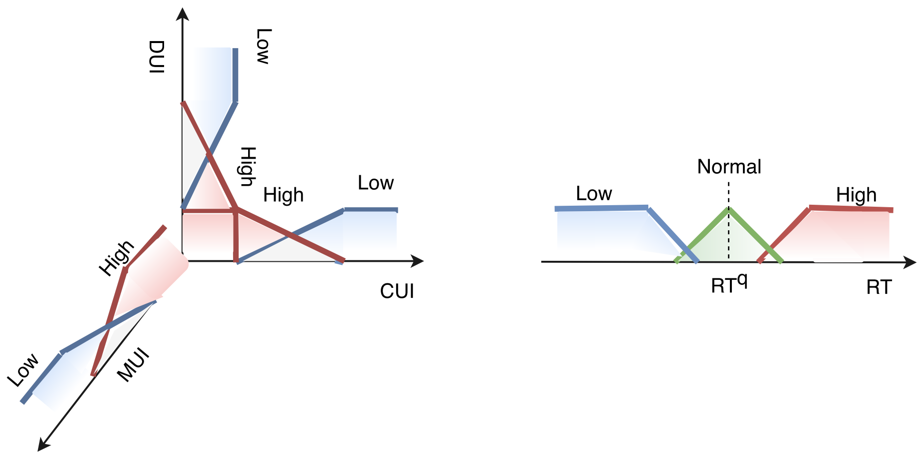


Fig. 3 Fuzzy representation of quality measurements

4.1 State modeling and fuzzy inference

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

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

$$CUI_n = \frac{1}{CUI'_n} \quad MUI_n = \frac{1}{MUI'_n} \quad DUI_n = \frac{1}{DUI'_n} \tag{2}$$

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

Fuzzification. Input fuzzification involves defining fuzzy sets and corresponding membership functions over the values of the quality measurements. A membership function is characterized by a linguistic term. A fuzzy set L is defined as $L = \{(x, \mu_L(x)) | \ 0 < x, \quad x \in \mathbb{R}\}$ where a membership function $\mu_L(x)$ defines membership degrees of the values as $\mu_L : x \rightarrow [0, 1]$. Figure 3 shows the membership functions defined over the value domains of quality measurements. As shown in Fig. 3, trapezoidal membership functions are used for *High* and *Low* fuzzy sets and a triangular counterpart for the *Normal* fuzzy set on the response time. In Fig. 3, where RT^q is the requirement, a normal (medium) fuzzy set over the values of response time implies a small range around the requirement value as normal response time values. Moreover, in this case, the ranges of membership functions were selected empirically and could be updated based on the requirements.

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

Fuzzy Rules. A fuzzy rule, as shown in Eq. 3, consists of two parts: antecedent and consequent. The former is a combination of linguistic terms of the input normalized quality measurements and the consequent is a fuzzy set with a membership function showing to what extent the environment is in the associated state.

$$\begin{aligned}\text{Rule 1: } & \text{If CUI is High AND MUI is High AND DUI is Low AND} \\ & \text{RT is Normal, then State is HHLN.}\end{aligned} \tag{3}$$

Rule 1 is a sample of the fuzzy rules in the rule base. The rest of the rules are defined similarly based on the fuzzy sets defined over the values of the quality measurements and the combinations of them. Based on the number of fuzzy sets, namely two fuzzy sets, *High* and

Low, over the value range of each resource utilization improvement and three sets, *High*, *Normal*, and *Low*, over the value range of the response time, we define 24 rules in our rule base to define the fuzzy states of the environment.

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

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

Finally, the most effective rule, the one with the maximum support degree, is selected to determine the final fuzzy state of the environment (S_n, μ_n). In summary, the fuzzy state with the highest likelihood is considered as the state of the system. Figure 4 shows

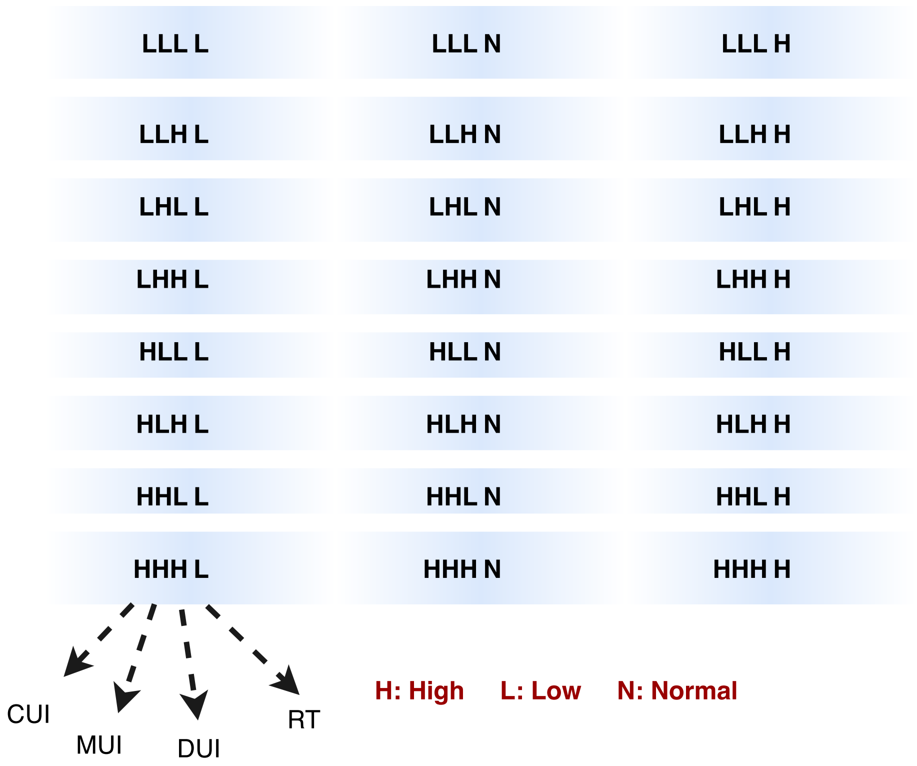


Fig. 4 Fuzzy states of the environment

a representation of the fuzzy states. Each of them represents one state based on the fuzzy values (linguistic terms) assigned to quality measurements (CPU, memory, and disk utilization improvement and response time). Regarding the presentation of fuzzy states, L, H and N stand for low, high, and normal terms, respectively.

5 Adaptive action selection and reward computation

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

$$\begin{aligned} AC_n =& \{\text{no action}\} \cup \{(CPU_n - y) \mid y \in CDF\} \cup \{(Mem_n - k) \mid k \in MDF_n\} \\ & \cup \{(Disk_n - k) \mid k \in MDF_n\} \end{aligned} \tag{4}$$

$$CDF = \{\frac{1}{4}, \frac{2}{4}, \frac{3}{4}, 1\} \tag{5}$$

$$MDF_n = \{(x \times \frac{Mem(Disk)_n}{4}) \mid x \in \{\frac{1}{4}, \frac{2}{4}, \frac{3}{4}, 1\}\} \tag{6}$$

where AC_n, CPU_n, Mem_n and $Disk_n$ represent the set of actions, the current available computation (CPU), memory, and disk capacity at time step n, respectively. The list of actions is as shown in Table 1.

Strategy Adaptation. The agent can use different strategies for selecting the actions. ϵ-greedy with different ϵ-values and Softmax are well-known methods for action selection in RL algorithms. They are intended to provide a right trade-off between exploration of the state action space and exploitation of the learned policy. In SaFReL, we use ϵ-greedy as the action selection strategy and the proposed strategy adaptation feature acts as a simple meta-learning algorithm intended to make changes to the ϵ value dynamically to make the action selection strategy well-adapted to new situations (new SUTs). Upon observing a SUT instance with a performance sensitivity different from the already observed ones, it adjusts the value of the parameter ϵ to direct the agent toward more exploration (setting ϵ to higher values). On the other hand, upon interaction with SUT instances that are similar to the previous ones, the parameter ϵ is adjusted to increase exploitation (setting ϵ to lower values). SaFReL detects the similarity between SUT instances by calculating *cosine similarity* between the performance sensitivity vectors of SUT instances, as shown in Eq. 7.

Table 1 Actions in SaFReL

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

$$\text{similarity}(k, k-1) = \frac{SV^k \, SV^{k-1}}{\|SV^k\| \, \|SV^{k-1}\|} = \frac{\sum_{i=1}^{3} SV_i^k SV_i^{k-1}}{\sqrt{\sum_{i=1}^{3} (SV_i^k)^2} \sqrt{\sum_{i=1}^{3} \left(SV_i^{k-1}\right)^2}} \tag{7}$$

where SV^k represents the sensitivity vector of the k^{th} SUT instance and SV_i^k represents the i^{th} element of vector SV^k. The sensitivity vector contains the values of the sensitivity indicators of the SUT instance, Sen^C, Sen^M, and Sen^D. The performance sensitivity indicators assume values in the range [0, 1] and represent the sensitivity degree of the SUT to CPU, memory, and disk, respectively. Their values could be set empirically or even intuitively, and SaFReL uses the approximate estimated similarity to tune the ε value adaptively (See Section 7.2).

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

$$R_n = \beta U_n^r + (1-\beta) U_n^E \tag{8}$$

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

$$U_n^r = \begin{cases} 0, & RT'_n \leq RT^q \\ \frac{(RT'_n - RT^q)}{(RT^b - RT^q)}, & RT'_n > RT^q \end{cases} \tag{9}$$

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

$$U_n^E = Sen^C CUI'_n + Sen^M MUI'_n + Sen^D DUI'_n \tag{10}$$

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

6 Performance testing using self-adaptive fuzzy reinforcement learning

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

The agent measures the quality parameters and identifies the state-membership degree pair, (S_n, μ_n), through the fuzzy state detection, where S_n is the fuzzy state of the environment and μ_n indicates the membership degree, which means to what extent the environment has assumed that state. Then, according to the action selection strategy, the agent selects one action, $a_n \in A_n$, based on the previously learned policy or through exploring the state action space. The agent takes the selected action and executes the SUT. In the next step, the agent detects the new state of the SUT, (S_{n+1}, μ_{n+1}), and receives a reward signal, $r_{n+1} \in \mathbb{R}$, indicating the effectiveness of the applied action. After detecting the new state and receiving the reward, it updates the stored experience (learned policy). The whole procedure is repeated until meeting the stopping criterion that is reaching the performance breaking point, (RT^b). The experience of the agent is defined in terms of the policy that the agent learns. A policy is a mapping between each state and action and specifies the probability of taking action a in a given state s. The purpose of the agent in the learning is to find a policy that maximizes the expected long-term reward achieved over the further learning trials, which is formulated as follows: (Sutton and Barto 2018):

$$R_n = r_{n+1} + \gamma r_{n+2} + \ldots + \gamma^k r_{n+k+1} = \sum_{k=0}^{\infty} \gamma^k r_{n+k+1} \tag{11}$$

where γ is a discount factor specifying to what extent the agent prioritize future rewards compared to the immediate one. We use Q-learning as a model-free RL algorithm in our framework. In Q-Learning, a utility value, $Q^\pi(s, a)$, is assigned to each pair of state and action, which is defined as follows: (Sutton and Barto 2018):

$$Q^\pi(s, a) = E^\pi[R_n | s_n = s, a_n = a] \tag{12}$$

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

$$Q(s_n, a_n) = \mu_n^s \Big[(1 - \alpha)Q(s_n, a_n) + \alpha\Big(r_{n+1} + \gamma \max_{a'} Q(s_{n+1}, a')\Big)\Big] \tag{13}$$

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

$$a(s) = \underset{a'}{\operatorname{argmax}}\, Q(s, a') \tag{14}$$

The agent selects the action based on Eq. 14 when it is supposed to exploit the learned policy. SaFReL implements two learning phases: initial and transfer learning.

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

Transfer learning. SaFReL goes through the transfer learning phase, after the initial convergence. During this phase, the agent uses the learned policy upon observing SUT instances with similar performance sensitivity to the previously observed ones, while keeping the learning running, i.e., updating the policy upon detecting new SUT instances with different performance sensitivity. Strategy adaptation is used in the transfer learning phase and makes the agent adapt to various SUT instances. Algorithms 1 and 2 present the procedure of SaFReL in both initial learning and transfer learning phases.

Algorithm 1 SaFReL: Self-adaptive Fuzzy Reinforcement Learning-based Performance Testing

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

Algorithm 2 Fuzzy Q-Learning

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

7 Evaluation

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

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

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

7.1 Experiments setup

In this study, we implement the proposed smart testing framework (agent) along with a performance simulation module simulating the performance behavior of SUT programs under different execution conditions. The simulation module receives the resource sensitivity values and based on the amounts of resources demanded initially and the amounts of them granted after taking each action, estimates the program throughput using the following equation proposed by Taheri et al. (2016):

$$Thr_j = \frac{\frac{CPU_j^g}{CPU_j^i} Sen_j^C + \frac{Mem_j^g}{Mem_j^i} Sen_j^M + \frac{Disk_j^g}{Disk_j^i} Sen_j^D}{Sen_j^C + Sen_j^M + Sen_j^D} \times Thr_j^N \tag{15}$$

where CPU_j^i, Mem_j^i, and $Disk_j^i$ indicate the amounts of CPU, memory, and disk resources demanded by program j at the initial state and CPU_j^g, Mem_j^g, and $Disk_j^g$ are the amounts of resources granted to program j after taking an action, which modifies the resource availability. Sen_j^C, Sen_j^M, and Sen_j^D represent the CPU, memory and disk sensitivity values of program j, and Thr_j^N represents the nominal throughput of program j in an isolated, contention-free environment. The response time of the program is calculated as $RT_j = \frac{1}{Thr_j}$ in the simulation module. Figure 5 presents the implementation structure including SaFReL along with the implemented performance simulation module. In our implementation, the performance simulation module simulates the performance behavior of the SUT program and the testing agent interacts with the simulation module to capture the quality measures used for state detection.

Table 2 shows the list of programs and the corresponding resource sensitivity values used in the experimentation, the table data obtained from (Taheri, Zomaya & Kassler 2016). The collection listed in Table 2 includes various CPU-intensive, memory-intensive, and disk-intensive types of programs and also the programs with combined types of resource sensitivity. The SUTs are instances of the programs listed in Table 2 and are characterized by various initial amounts of resources and also different values of response time requirements. Two analysis scenarios are designed to answer the evaluation research questions. The first one focuses on the efficiency and adaptivity evaluation of the framework on various SUTs. In the second analysis scenario, the sensitivity of the approach to changes

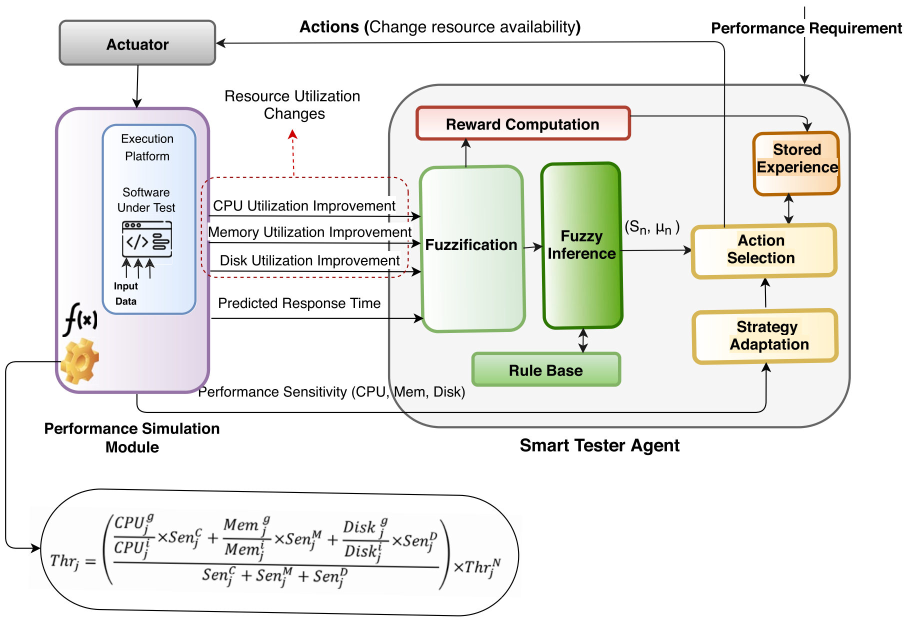

Actuator
Actions (Change resource availability)
Performance Requirement
Resource Utilization Changes
Execution Platform
Software Under Test
Input Data
$f(x)$
Reward Computation
Stored Experience
CPU Utilization Improvement
Memory Utilization Improvement
Disk Utilization Improvement
Predicted Response Time
Fuzzification
Fuzzy Inference
(S_n, μ_n)
Action Selection
Strategy Adaptation
Rule Base
Performance Sensitivity (CPU, Mem, Disk)
Performance Simulation Module
Smart Tester Agent
$Thr_j = \left(\frac{\frac{CPU_j^g}{CPU_j^i} \times Sen_j^C + \frac{Mem_j^g}{Mem_j^i} \times Sen_j^M + \frac{Disk_j^g}{Disk_j^i} \times Sen_j^D}{Sen_j^C + Sen_j^M + Sen_j^D} \right) \times Thr_j^N$

Fig. 5 Implementation structure

of the learning parameters is studied. The efficiency and adaptivity are measured (evaluated) according to the following specification:

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

Table 2 Programs and the corresponding sensitivity values used for experimental evaluation (Taheri et al. 2016)

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

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

7.2 Experiments and results

7.2.1 Efficiency and adaptivity analysis

To answer RQ1 and RQ2, the performance of SaFReL is evaluated based on its efficiency in generating the performance test cases leading to the performance breaking points of different SUTs and its adaptation capability to new SUTs with performance sensitivity different from previously observed ones. We select two sets of SUT instances: i) one including SUTs similar in the aspect of performance sensitivity to resources, i.e., similar with regard to the primarily demanded resource (homogenous SUTs); and ii) the other set contains SUT instances different in performance sensitivity (heterogeneous SUTs). The SUT instances assume different initial amounts of CPU, memory, and disk resources and response time requirements. The amounts of resources, CPU, memory, and disk capacity, were initialized with different values in the range [1, 10] cores, [1, 50] GB, [100, 1000] GB, respectively. The response time requirements range from 500 to 3000 ms. The intended performance breaking point for the SUT instances is defined as the point in which the response time exceeds 1.5 times the response time requirement.

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

Learning setup. First, we need to set up the initial learning. For choosing a proper configuration for the action selection strategy in the initial learning, we evaluate the performance of different variants of ϵ-greedy algorithm, in terms of the number of required learning trials for initial convergence (Fig. 6). For the initial convergence, we run the initial learning on the first SUT 100 times, namely 100 learning episodes. Table 3 presents a quick summarized view of the average learning trials during the last 10 episodes that are considered as the achieved values upon the convergence of the initial learning. As shown in Fig. 6 and Table 3, using ϵ-greedy with $\epsilon = 0.2$ results in the fastest initial convergence, which has also led to the lowest number of trials compared to the other variants of ϵ -greedy. The number of learning trials after about 10 episodes starts converging and during the last 10 episodes it converges to approximately 7 trials.

Once the initial convergence occurs, SaFReL is ready to act on various SUTs and is expected to be able to reuse the learned policy to meet the intended performance breaking points on further SUT instances, while still keeping the learning running. The optimal policy learned in the initial learning is not influenced by the used action selection strategy, since Q-learning is an off-policy learning algorithm (Sutton and Barto 2018). It implies that the learner finds the optimal policy independently of how the actions have been selected (action selection strategy). For the sake of efficiency, we choose the one that resulted in the fastest convergence.

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

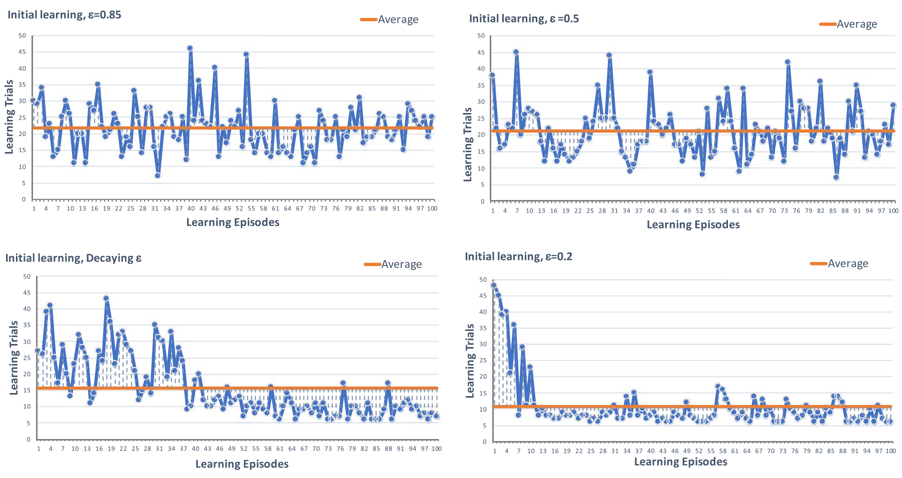


Fig. 6 Initial convergence of SaFReL in 100 learning episodes during the initial learning

I. Homogeneous set of SUTs. We select CPU-intensive programs and make a homogeneous set of SUT instances during our analysis in this step. We simulate the performance behavior of 50 instances of the CPU-intensive programs, Build-apache, n-queens, John-the-ripper, Apache, Dcraw, Build-php, X264, and vary both the initial amounts of resources granted and the response time requirements. Figure 7 shows the efficiency of SaFReL on a homogeneous set of CPU-intensive SUTs compared to a typical stress testing procedure regarding using ε-greedy with different values of ε. Table 4 presents the average number of trials/steps for generating the target performance test case in the proposed approach and the typical testing procedure. As shown in Fig. 7, it keeps the number of required trials for $\approx$ 94% of the SUTs below the average number of required steps in the typical stress testing. Table 5 shows the resulting improvement in the average number of required trials/steps for meeting the target, which implies a reduction in the required computation time, compared to the typical stress testing process.

In the transfer learning, the agent reuses the learned policy based on the allowed degree of policy reusing according to its action selection strategy in the transfer learning. As shown in Table 4, it implies that in the transfer learning the agent does fewer trials (based on the degree of allowed policy reusing) to meet the target on new cases, which leads to higher efficiency. According to Table 5, on a homogeneous set of SUTs, more policy reusing leads to higher efficiency (more computation time improvement).

II. Heterogeneous set of SUTs. In this part of the analysis, to complete the answer to RQ1 and also answer RQ2, we examine the efficiency and adaptivity of SaFReL during the transfer learning on a heterogeneous set of SUTs including various CPU-intensive, memory-intensive, and

Table 3 Initial convergence of SaFReL in the initial learning regarding using different variants of action selection strategy

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

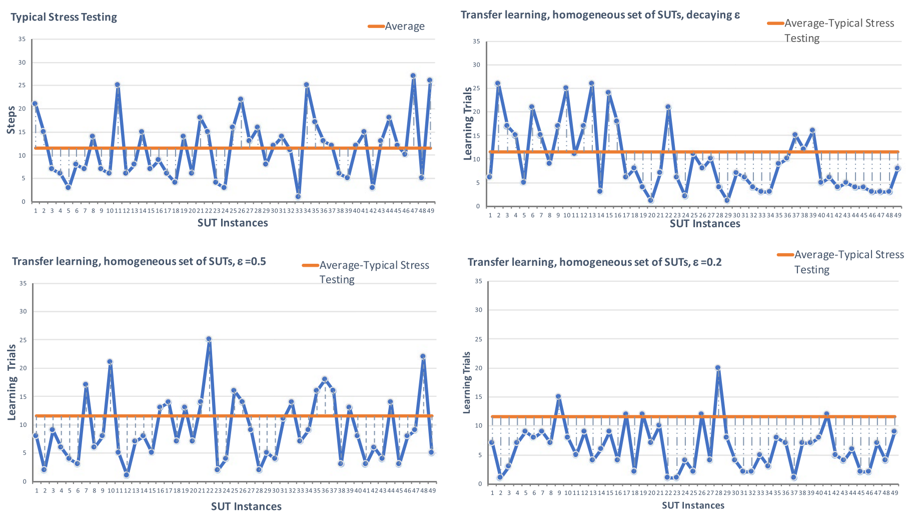


Fig. 7 Efficiency of SaFReL on a homogeneous set of SUTs in the transfer learning

disk-intensive ones. We simulate the performance behavior of 50 SUT instances from the list of the programs in Table 2. We evaluate the efficiency of SaFReL on the heterogeneous set of SUTs compared to the typical stress testing procedure regarding using ε-greedy with $\varepsilon = 0.2$, 0.5, and decaying ε (Fig. 8). As shown in Fig. 8 the transfer learning algorithm with a typical configuration of the action selection strategy, such as $\varepsilon = 0.2$, 0.5, and decaying ε, which imposes a certain degree of policy reusing based on the value of ε does not work well. It does not outperform the typical stress testing, but also slightly degrades in some cases of ε. When the smart agent acts on a heterogeneous set of SUTs, blind replaying of the learned policy (i.e., just based on the value of ε) is not effective, and the tester agent needs to know where it should do policy reusing and where it requires more exploration to update the policy.

As described in Section 5, to solve this issue and improve the performance of SaFReL when it acts on a heterogeneous set of SUTs, it is augmented with a simple meta-learning feature enabling it to detect the heterogeneity of the SUT instances and adjust the value of parameter ε, adaptively. In general, it implies that when the smart tester agent observes a SUT instance different from the previously observed ones wrt the performance sensitivity, it changes the focus of the action selection strategy into doing more exploration and upon detecting a SUT instance with the same performance sensitivity as the previous ones, it makes the action selection strategy strive for more exploitation. As illustrated in Section 5, the strategy adaptation module, which fulfills this function, measures the similarity between SUTs at two levels of observations, then based on the measured values, adjusts the value of parameter ε. The threshold values of similarity measures and the adjustments for parameter ε in the experimental analysis are described in Algorithm 3.

Table 4 Average number of trials/steps for generating the target performance test case on the homogeneous set of SUTs

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

Table 5 Computation time improvement on the homogeneous set of SUTs

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

Algorithm 3 Adaptive ϵ selection

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

Figure 9 shows the efficiency of SaFReL regarding the use of similarity detection and the adaptive ϵ-greedy action selection strategy on a heterogeneous set of SUTs. Regarding the use of adaptive ϵ selection, SaFReL makes a considerable improvement and is able to keep the number of required trials for reaching the target on approximately $\approx 82\%$ of SUTs below the corresponding average value in the typical stress testing. Meanwhile, the average number of learning trials is totally lower than the typical stress testing procedure. Table 6 presents the average number of trials/steps for generating the target performance test case in SaFReL and the typical stress testing when they act on a

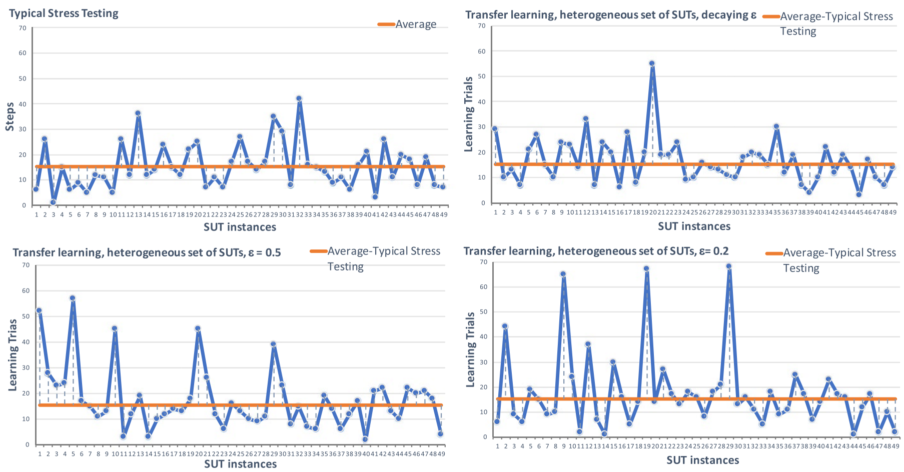


Fig. 8 Efficiency of SaFReL on a heterogeneous set of SUTs regarding the use of typical configurations of ϵ-greedy

Table 6 Average number of trials/steps for generating the target performance test case on the heterogeneous set of SUTs

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

heterogeneous set of SUTs. Table 7 shows the corresponding resulting improvement in the computation time, respectively.

To answer RQ2, we investigate the adaptivity of SaFReL on the heterogeneous set of SUTs regarding the use of different variants of action selection strategy including adaptive ε selection (Fig. 10). As shown in Fig. 10, the number of required learning trials versus detected similarity is used to depict how adaptive SaFReL can act on a heterogeneous set of SUTs regarding the use of different configurations of ε. It shows that SaFReL with adaptive ε is able to adapt to changing situations, e.g., a mixed heterogeneous set of SUTs. In other words, in around $\approx 75\%$ of SUTs that are completely different from the previous ones (i.e., with $similarity_{k,k-1} < 0.8$) it still keeps the number of required trials to meet the target below the average value of the typical stress testing. It implies that it can act adaptively, which means it reuses the policy wherever it is useful and does more exploration wherever required.

7.2.2 Sensitivity analysis

To answer RQ3, we study the impacts of the learning parameters including learning rate (α) and discount factor (γ), on the efficiency of SaFReL on both homogeneous and heterogeneous sets of SUTs. For conducting sensitivity analysis, we implement two sets of experiments that involve changing one learning parameter while keeping the other one constant. For the experiments running on a homogeneous set of SUTs, we use ε-greedy with $\varepsilon = 0.2$ as the well-suited variant of action selection strategy with respect to the results of efficiency analysis (See Fig. 7) and on the heterogeneous set of SUTs, we use adaptive ε selection (See Fig. 9). During the sensitivity analysis experiments, to study the impact of the learning rate changes, we set the discount factor to 0.5. While examining the impact of the discount factor changes, we keep the learning rate fixed to 0.1. Figure 11 shows the sensitivity of SaFReL to changing learning rate and discount factor parameters when it acts on a homogeneous set of SUTs (CPU-intensive). Figure 12 depicts the results of the sensitivity analysis of SaFReL on a heterogeneous set of SUTs.

Table 7 Computation time improvement on the heterogeneous set of SUTs

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

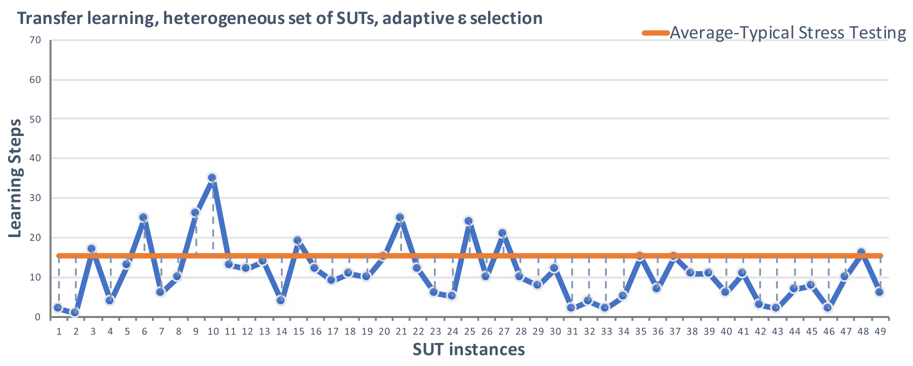


Fig. 9 Efficiency of SaFReL on a heterogeneous set of SUTs regarding the use of adaptive ϵ-greedy action selection strategy

8 Discussion

8.1 Efficiency, adaptivity, and sensitivity analysis

RQ1: Using multiple experiments, we studied the efficiency of SaFReL compared to a typical stress testing procedure, on both a set of homogeneous and heterogeneous SUTs regarding the use of different action selection strategies. The results of the experiments running on a set of 50 CPU-intensive SUT instances as a homogeneous set of SUTs, Fig. 7 and Tables 4 and 5, show that using ϵ-greedy, $\epsilon = 0.2$ as action selection strategy in the transfer learning leads to desired efficiency and an improvement in the computation time (around 42%) compared to the typical stress testing. It causes SaFReL to rely more on reusing the learned policy and results in computation time saving. The existing similarity

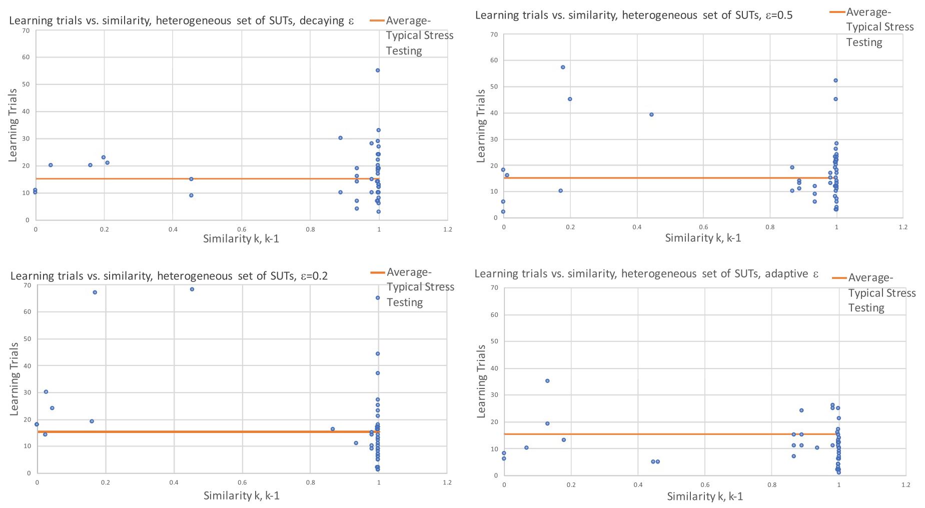
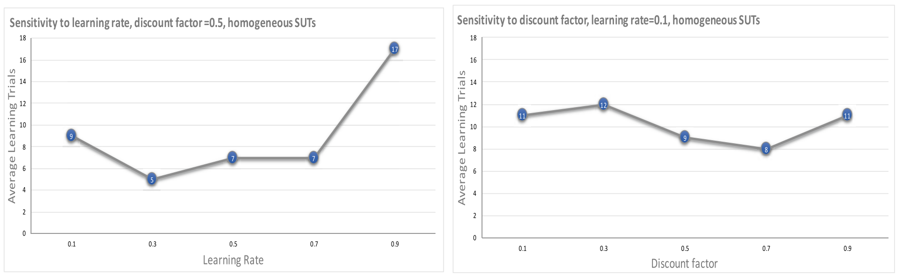


Fig. 10 Adaptivity of SaFReL on a heterogeneous set of SUTs regarding the use of different variants of action selection strategy



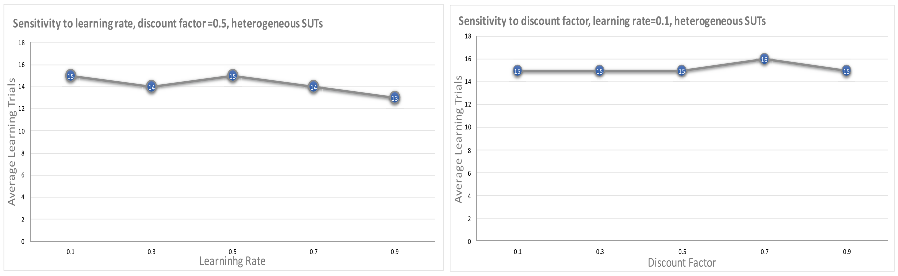
Average Efficiency of SaFReL with $\epsilon = 0.2$, Discount factor $\gamma = 0.5$					
	$\alpha = 0.1$	$\alpha = 0.3$	$\alpha = 0.5$	$\alpha = 0.7$	$\alpha = 0.9$
Average number of learning trials	9	5	7	7	17

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

Fig. 11 Sensitivity of SaFReL to learning rate and discount factor on the homogeneous set of SUTs

between the performance sensitivity of SUTs in a homogeneous set of SUTs makes the strategy of policy reusing successful in this type of testing situations.

Furthermore, we studied the efficiency of SaFReL on a heterogeneous set of 50 SUTs containing different CPU-intensive, memory-intensive and disk-intensive ones. The results of the analysis illustrate that choosing an action selection strategy without considering the heterogeneity among the SUTs (e.g., using the typical variants of ϵ-greedy) does not lead to desirable efficiency compared to the typical stress testing (See Fig. 8 and Tables 6 and 7). Then, we augmented our fuzzy RL-based approach with an adaptive action selection strategy that is a heterogeneity-aware strategy for adjusting the value of ϵ. It measures the similarity between



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

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

Fig. 12 Sensitivity of SaFReL to learning rate and discount factor on the heterogeneous set of SUTs

the performance sensitivity of the SUTs and adjusts the ε parameter. As shown in Fig. 9, using the adaptive ε-greedy addressed the issue and led to an efficient generation of the target performance test case and a computation time improvement (around 31%). It makes the agent able to reuse the learned policy according to the conditions, which means it uses the learned policy wherever it is useful and does more exploration wherever it is required.

RQ2: In the last part of the efficiency and adaptivity analysis, we extended our analysis by measuring the adaptivity of SaFReL when it performs on a heterogeneous set of SUTs. As shown in Fig. 10, with the use of the adaptive ε-greedy, SaFReL is able to adapt to changing testing situations while preserving the efficiency.

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

8.2 Lessons learned

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

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

Changes in Future Trends. With the emergence of serverless architecture, which incorporates third-party backend services (BaaS) and/or runs the server-side logic in state-less containers that are fully-managed by providers (FaaS), a slight shift in the objectives of performance evaluation, particularly performance testing on cloud-native applications is expected. Within the serverless architecture, the backend code is run without the need to manage and provision the resources on servers. For example in FaaS, scaling, including the resource provisioning and allocation, is automatically done by the provider whenever it is needed, to preserve the response time requirement of the application. In general, regarding the capabilities of new execution platforms and deployment architectures, the objectives of performance testing might be slightly influenced. Nevertheless, it is still crucial for a wide range of software systems.

8.3 Threats to validity

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

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

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

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

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

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

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

9 Related work

Measurement of performance metrics under typical or stress test execution conditions, which involve both workload and platform configuration aspects (Menasc'e 2002; Hill, Schmidt, Edmondson & Gokhale 2009; Apte et al. 2017; Michael et al. 2017; Jindal et al. 2019), detection of performance-related issues such as functional problems or violations of performance requirements emerging under certain workload or resource configuration conditions (Briand et al. 2005; Zhang et al. 2011; Ayala-Rivera et al. 2018; Schulz et al. 2019) are common objectives of different types of performance testing.

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

Source code analysis. Deriving workload-based performance test conditions using dataflow analysis and symbolic execution are examples of techniques for designing fault-inducing performance test cases based on source code analysis to detect performance-related issues such as functional problems (like memory leaks) and performance requirements violations (Yang and Pollock 1996; Zhang et al. 2011).

System model analysis. Modeling the system behavior in terms of performance models like Petri nets and using constraint solving techniques (Zhang and Cheung 2002), using the control flow graph of the system and applying search-based techniques (Gu and Ge 2009; Di Penta et al. 2007), and using other types of system models like UML models and using genetic algorithms (Garousi 2010; Garousi 2008; Garousi et al. 2008; Costa et al. 2012; da Silveira et al. 2011) to generate the performance test cases are examples of the techniques based on system model analysis for generating performance test cases.

Behavior-driven declarative techniques. Using a Domain Specific Language (DSL) to provide declarative goal-oriented specifications of performance tests and model-driven execution frameworks for automated execution of the tests (Ferme and Pautasso 2018; Ferme and Pautasso 2017; Walter et al. 2016), and using a high-level behavior-driven language inspired from Behavior-Driven Development (BDD) techniques to define test conditions (Schulz et al. 2019) in combination with a declarative performance testing framework like BenchFlow (Ferme and Pautasso 2017) are examples of behavior-driven techniques for performance testing.

Modeling realistic conditions. Modeling the real user behavior through stochastic form-oriented models (Draheim et al. 2006; Lutteroth and Weber 2008), extracting workload characteristics from the recorded requests and modeling the user behavior using, e.g., extended finite state machines (EFSMs) (Shams et al. 2006) or Markov chains (Vogele et al. 2018), sandboxing services and deriving a regression model of the deployment environment based on the data resulting from sandboxing to estimate the service capacity (Jindal et al. 2019), end-user clustering based on the business-level attributes extracted from usage data (Maddodi et al. 2018), and using automated GUI testing tools with capture and replay techniques to generate realistic interactive usage sequences (Adamoli et al. 2011) are examples of techniques based on modeling the realistic conditions to generate the performance test cases.

Machine learning-enabled techniques. Machine learning techniques such as supervised and unsupervised algorithms mainly work based on building models and extracting patterns (knowledge) from the data, while some other techniques such as RL algorithms are intended to train the learner agent to solve the problems (tasks). The agent learns an optimal way to achieve an objective through interacting with the system. Machine learning has been widely used for the analysis of data resulting from performance testing and also for performance preservation. For example, anomaly detection through analysis of performance data, e.g., resource usage, using clustering techniques (Syer et al. 2011), predicting reliability from the testing data using Bayesian Networks (Avritzer et al. 2008), performance signature identification based on performance data analysis using supervised and unsupervised learning techniques (Malik et al. 2013; Malik et al. 2010), and also adaptive RL-driven performance in particular response time control for cloud services (Ibidunmoye et al. 2017; Veni and Bhanu 2016; Jamshidi et al. 2016) and also software on other execution platforms, e.g., PLC-based real-time systems (Moghadam et al. 2018). Machine learning has been also applied to the generation of performance test cases in some studies. For example, using symbolic execution in combination with an RL algorithm to find the worst-case execution path within a SUT (Koo et al. 2019), using RL to find a sequence of input workload leading to performance degradation (Ahmad et al. 2019), and feedback-driven learning to identify the performance bottlenecks through extracting rules from execution traces (Grechanik, Fu and Xie 2012). There are also some adaptive techniques slightly analogous to the concept of RL for generating performance test cases. For example, an adaptive workload generation that adapts the workload dynamically based on some pre-defined adjustment policies (Ayala-Rivera et al. 2018), and a feedback-driven approach that uses search algorithms to benchmark an NFS server based on varying workload parameters to find the workload peak rate reaching the target response time confidence level.

10 Conclusion

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

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

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

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

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

Funding Open access funding provided by RISE Research Institutes of Sweden.

References

Adamoli, A., Zaparanuks, D., Jovic, M., Hauswirth, M. (2011). Automated gui performance testing. *Software Quality Journal, 19(*4), 801–839.

Ahmad, T., Ashraf, A., Truscan, D., Porres, I. (2019). Exploratory performance testing using reinforcement learning. In *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*(pp. 156-163). IEEE.

Apte, V., Viswanath, T. V. S., Gawali, D., Kommireddy, A., Gupta, A. (2017). AutoPerf: Automated load testing and resource usage profiling of multi-tier internet applications. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering* (pp. 115-126).

Avritzer, A., Duarte, F. P., Leao, R. M. M., e Silva, Ed. S., Cohen, M., Costello, D. (2008). Reliability estimation for large distributed software systems. In Cascon, Citeseer (p. 12).

Ayala-Rivera, V., Kaczmarski, M., Murphy, J., Darisa, A., Portillo-Dominguez, AO. (2018). 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*, *ACM* (pp. 211–222).

Babovic, Z. B., Protic, J., Milutinovic, V. (2016). Web performance evaluation for internet of things applications. *IEEE Access, 4,* 6974–6992.

Briand, LC., Labiche, Y., Shousha, M. (2005). Stress testing real-time systems with genetic algorithms. In *Proceedings of the 7th annual conference on Genetic and evolutionary computation*, *ACM* (pp. 1021–1028).

Brunnert, A., van Hoorn, A., Willnecker, F., Danciu, A., Hasselbring, W., Heger, C., et al. (2015). Performance oriented devops: A research agenda. arXiv preprint. arXiv:150804752

Chandola, V., Banerjee, A., Kumar, V. (2009). Anomaly detection: A survey. *ACM computing surveys (CSUR)*, *41*(3), 15.

Chung, L., Nixon, B. A., Yu, E., Mylopoulos, J. (2012). Non-functional requirements in software engineering. *Springer Science & Business Media*, *5*.

Cortellessa, V., Di Marco, A., Inverardi, P. (2011). Model-based software performance analysis. *Springer Science & Business Media.*

Costa, LT., Czekster, RM., de Oliveira, FM., Rodrigues, EDM., da Silveira, MB., Zorzo, AF. (2012). Generating Performance Test Scripts and Scenarios Based on Abstract Intermediate Models. In *SEKE,* (pp. 112-117).

Denaro, G., Polini, A., Emmerich, W. (2004). Early performance testing of dis- tributed software applications. *ACM SIGSOFT Software Engineering Notes, 29,* 94–103.

Di Penta, M., Canfora, G., Esposito, G., Mazza, V., Bruno, M. (2007). Search-based testing of service level agreements. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, *ACM* (pp. 1090–1097).

Draheim, D., Grundy, J., Hosking, J., Lutteroth, C., Weber, G. (2006). Realistic load testing of web applications. In *Conference on Software Maintenance and Reengineering (CSMR'06), IEEE* (p. 11).

Ferme, V., & Pautasso, C. (2017). Towards holistic continuous software performance assessment. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, *ACM* (pp. 159–164).

Ferme, V., & Pautasso, C. (2018). A declarative approach for performance tests execution in continuous software development environments. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering, ACM* (pp. 261–272).

Fowler, K. (2009). *Mission-critical and safety-critical systems handbook: Design and development for embedded applications*. Newnes.

Garousi, V. (2008). Empirical analysis of a genetic algorithm-based stress test technique. In *Proceedings of the 10th annual conference on Genetic and evolutionary computation, ACM* (pp. 1743–1750).

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

Garousi, V., Briand, L. C., Labiche, Y. (2008). 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.

Glinz, M. (2007). On non-functional requirements. In *15th IEEE International Requirements Engineering Conference (RE 2007), IEEE* (pp. 21–26).

Grechanik, M., Fu, C., Xie, Q. (2012). Automatically finding performance problems with feedback-directed learning software testing. In *2012 34th International Conference on Software Engineering (ICSE), IEEE* (pp. 156–166).

Gregg, B. (2013). *Systems performance: enterprise and the cloud*. Pearson Education.

Grinshpan, L. (2012). *Solving enterprise applications performance puzzles: queuing models to the rescue*. John Wiley & Sons.

Gu, Y., & Ge, Y. (2009). Search-based performance testing of applications with composite services. In *2009 International Conference on Web Information Systems and Mining, IEEE* (pp. 320–324).

Harchol-Balter, M. (2013). *Performance modeling and design of computer systems: queueing theory in action*. Cambridge University Press.

Hill, J., Schmidt, D., Edmondson, J., Gokhale, A. (2009). Tools for continuously evaluating distributed system qualities. *IEEE software, 27*(4), 65–71.

Ibidunmoye, O., Hernandez-Rodriguez, F., Elmroth, E. (2015). Performance anomaly detection and bottleneck identification. *ACM Computing Surveys (CSUR), 48*(1), 4.

Ibidunmoye, O., Moghadam, M. H., Lakew, E. B., Elmroth, E. (2017). Adaptive service performance control using cooperative fuzzy reinforcement learning in vir- tualized environments. In *Proceedings of the10th International Conference on Utility and Cloud Computing, ACM* (pp. 19–28).

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

Jamshidi, P., Sharifloo, A., Pahl, C., Arabnejad, H., Metzger, A., Estrada, G. (2016). Fuzzy self-learning controllers for elasticity management in dynamic cloud architectures. In *2016 12th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA), IEEE* (pp. 70–79).

Jennings, B., & Stadler, R. (2015). Resource management in clouds: Survey and research challenges. *Journal of Network and Systems Management, 23*(3), 567–619.

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

Jindal, A., Podolskiy, V., Gerndt, M. (2019). Performance modeling for cloud microservice applications. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering, ACM* (pp. 25–32).

Kant, K., & Srinivasan, M. (1992). *Introduction to computer system performance evaluation*. McGraw-Hill College.

Kolesnikov, S., Siegmund, N., Kastner, C., Grebhahn, A., Apel, S. (2019). Tradeoffs in modeling performance of highly configurable software systems. *Software & Systems Modeling, 18*(3), 2265–2283.

Koo, J., Saumya, C., Kulkarni, M., Bagchi, S. (2019). Pyse: Automatic worst-case test generation by reinforcement learning. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST), IEEE* (pp. 136–147).

Kuncheva, L. I. (2008). *Fuzzy classifiers. Scholarpedia, 3*(1), 2925.

Lutteroth, C., & Weber, G. (2008). Modeling a realistic workload for performance testing. In *2008 12th International IEEE Enterprise Distributed Object Computing Conference, IEEE* (pp. 149–158).

Maddodi, G., Jansen, S., de Jong, R. (2018). Generating workload for erp applications through end-user organization categorization using high level business operation data. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering, ACM* (pp. 200–210).

Malik, H., Jiang, Z. M., Adams, B., Hassan, A. E., Flora, P., Hamann, G. (2010). Automatic comparison of load tests to support the performance analysis of large enterprise systems. In *2010 14th European conference on software maintenance and reengineering, IEEE* (pp. 222–231).

Malik, H., Hemmati, H., Hassan, A. E. (2013). Automatic detection of performance deviations in the load testing of large scale systems. In *Proceedings of the 2013 International Conference on Software Engineering, IEEE Press* (pp. 1012–1021).

MathWorks (2019). Fuzzy Inference Process. Retrieved from https://www.mathworks.com/help/fuzzy/fuzzy-inference-process.html

Menasc'e, DA. (2002). Load testing, benchmarking, and application performance management for the web. In *Int. CMG Conference* (pp. 271–282).

Michael, N., Ramannavar, N., Shen, Y., Patil, S., Sung, J. L. (2017). Cloudperf: A performance test framework for distributed and dynamic multi-tenant environ- ments. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, ACM* (pp. 189–200).

Moghadam, M. H., Saadatmand, M., Borg, M., Bohlin, M., Lisper, B. (2018). Adaptive runtime response time control in plc-based real-time systems using rein- forcement learning. In *2018 IEEE/ACM 13th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), IEEE* (pp. 217–223).

Moghadam, M. H., Saadatmand, M., Borg, M., Bohlin, M., Lisper, B. (2019). Machine learning to guide performance testing: An autonomous test framework. In *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), IEEE* (pp. 164–167).

Morabito, R. (2017). Virtualization on internet of things edge devices with con- tainer technologies: a performance evaluation. *IEEE Access, 5*, 8835–8850.

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

Schulz, H., Okanovi'c, D., van Hoorn, A., Ferme, V., Pautasso, C. (2019). Behavior- driven load testing using contextual knowledge-approach and experiences. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering, ACM* (pp. 265–272).

Shams, M., Krishnamurthy, D., Far, B. (2006). A model-based approach for testing the performance of web applications. In *Proceedings of the 3rd international workshop on Software quality assurance, ACM* (pp. 54–61).

da Silveira, MB., Rodrigues, EdM., Zorzo, AF., Costa, LT., Vieira, HV., de Oliveira, FM. (2011). Generation of scripts for performance testing based on uml models. In *SEKE* (pp. 258–263).

Sutton, RS., & Barto, AG. (2018). *Reinforcement learning: An introduction*. MIT press.

Syer, MD., Adams, B., Hassan, AE. (2011). Identifying performance deviations in thread pools. In *2011 27th IEEE International Conference on Software Maintenance (ICSM), IEEE* (pp. 83–92).

Taheri, J., Zomaya, AY., Kassler, A. (2016). vmbbthrpred: A black-box throughput predictor for virtual machines in cloud environments. In *European Conference on Service-Oriented and Cloud Computing* (pp. 18–33). Springer.

Veni, T., Bhanu, S. M. S. (2016). Auto-scale: automatic scaling of virtualised re- sources using neuro-fuzzy reinforcement learning approach. *International Journal of Big Data Intelligence, 3*(3), 145–153.

Venkataraman, S., Yang, Z., Franklin, M., Recht, B., Stoica, I. (2016). Ernest: ef- ficient performance prediction for large-scale advanced analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation, NSDI* (16) (pp. 363–378).

Vogele, C., van Hoorn, A., Schulz, E., Hasselbring, W., Krcmar, H. (2018). Wessbas: extraction of probabilistic workload specifications for load testing and per- formance prediction-a model-driven approach for session-based application systems. *Software & Systems Modeling, 17*(2), 443–477.

Walter, J., van Hoorn, A., Koziolek, H., Okanovic, D., Kounev, S. (2016). Asking what?, automating the how?: The vision of declarative performance engi- neering. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering, ACM* (pp. 91–94).

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

Yang, C. S. D., & Pollock, L. L. (1996). Towards a structural load testing tool. *ACM SIGSOFT Software Engineering Notes, ACM, 21*, 201–208.

Zhang, J., & Cheung, SC. (2002). Automated test case generation for the stress testing of multimedia systems. *Software: Practice and Experience*, *32*(15), 1411–1435.

Zhang, P., Elbaum, S., Dwyer, M. B. (2011). Automatic generation of load tests. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, IEEE Computer Society* (pp. 43–52).

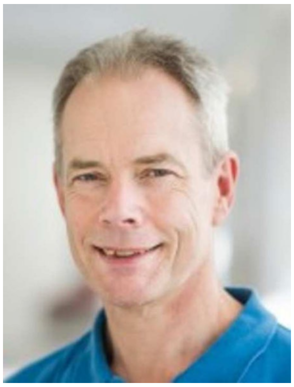
Zhang, P., Elbaum, S., Dwyer, M. B. (2012). Compositional load test generation for software pipelines. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis, ACM* (pp. 89–99).

Mahshid Helali Moghadam received her B.Sc. degree in software engineering and her M.Sc.degree in artificial intelligence from Iran University of Science and Technology (IUST), Tehran,Iran in 2008 and 2011 respectively. She has worked as a researcher at ICT department of NirooResearch Institute (NRI) in Tehran for six years. Currently, she is an industrial PhD candidate atMälardalen University and working as a researcher at RISE Research Institutes of Sweden. Herresearch interests include intelligent software engineering and software engineering forintelligent systems, machine learning, and autonomous systems.

Mehrdad Saadatmand (PhD) is a senior researcher and group manager at RISE in Västerås(RISE: Research Institutes of Sweden) and holds a PhD degree from Mälardalen University,Sweden. He is responsible for the software testing research group at RISE Västerås; andhas been involved in several large scale research projects on software testing and model-baseddevelopment of cyber-physical systems such as TOCSYC project (Testing of Critical SystemCharacteristics), XIVT (eXcellence in Variant Testing), MBAT (Combined Model-BasedAnalysis and Testing), to name a few.

Markus Borg is a senior researcher with the Humanized Autonomy research unit, RISE ResearchInstitutes of Sweden and an adjunct senior lecturer at Lund University, Sweden. Dr. Borgreceived a PhD degree in software engineering in 2015 from Lund University, Sweden. Hisresearch interests include software testing, safety-critical systems, and machine learning

Markus Bohlin received the Ph.D. degree from Mälardalen University in 2009 where he wasappointed as an Associate Professor (Docent) in computer science in 2013. He is currentlya full professor in computer science at Mälardalen University. His research interests include thereal-world application of optimization methods and operations research to industrial processes.



Björn Lisper is professor in Computer Engineering at Mälardalen University since 1999, wherehe leads the Programming Languages research group. His current research interests are inprogramming language issues, targeting program analysis especially w.r.t timing properties. Hereceived his MSc (Engineering Physics) in 1980, and Doctor of Technology (Computer Science) in 1987, both from KTH, Sweden, where he also was appointed "docent" in Computer Systems(1991). Prof. Lisper was the Chair of the COST Action IC1202 Timing Analysis on Code-Level.He has been a Core Member of the FP7 NoE Artist-Design, where he was the leader of the timinganalysis activity. He coordinated the FP7 ICT project ALL-TIMES, and the Marie Curie IAPPproject APARTS. He is a member of HiPEAC as well as IFIP WG 10.2 on Embedded Systems.