Automatic Performance Space Exploration of Web Applications

Tanwir Ahmad, Fredrik Abbors, and Dragos Truscan

Faculty of Science and Engineering
Åbo Akademi University
Joukahaisenkatu 3-5 A, 20520, Turku, Finland
{tanwir.ahmad, fredrik.abbors, dragos.truscan}@abo.fi

Abstract. Web applications have become crucial components of current service-oriented business applications. Therefore, it is very important for the company's reputation that the performance of a web application has been tested thoroughly before deployment. We present a tool-supported performance exploration approach to investigate how potential user behavioral patterns affect the performance of the system under test. This work builds on our previous work in which we generate load from workload models describing the expected behavior of the users. We mutate a given workload model (specified using Probabilistic Timed Automata) in order to generate different mutants. Each mutant is used for load generation using the MBPeT tool and the resource utilization of the system under test is monitored. At the end of an experiment, we analyze the mutants in two ways: cluster the mutants based on the resource utilization of the system under test and identify those mutants that satisfy the criteria of given objective functions.

Keywords: performance evaluation, performance prediction, model-based mutation, probabilistic timed automata, load generation

1 Introduction

Current web applications range from dating sites to gambling sites, e-commerce, on-line banking, airline bookings, and corporate websites. Web applications are becoming increasingly complex and there are many factors to be considered when the performance of web systems is concerned, for example network bandwidth, distributed computing nodes, software platform used for system implementation, etc [19]. It is very important for companies to provide high-quality service to their customers in order to keep their competitive edge in the market [13, 18]. Therefore, performance testing has become an important activity to identify the performance bottlenecks before application deployment [23].

Performance testing is a process of measuring the responsiveness and scalability of a system when it is under a certain synthetic workload [23]. The *synthetic workload* is generated by simulating the workload in a real environment. Usually, different *key performance indicators* (KPIs) are monitored during a load generation session in order to evaluate the performance of the system under test (SUT). Generally, load is generated by executing pre-recorded scripts of user actions which simulate the expected user behavior. The approach is passive in nature and does not represent the dynamic behavioral pattern of real users [9].

In our previous work, we used *Probabilistic Timed Automata* (PTA) for specifying workload models [1,2]. PTA models allow the tester to express the dynamic behavior of real users probabilistically and at the same time increase the level of abstraction of user model. A PTA can be created using two methods. Firstly, the tester can build a workload model manually based on his/her experience or knowledge of the SUT. Secondly, the workload model can be produced by mining web access log files [3]. The models created using the latter approach are approximations of the behavioral patterns of previous real users.

At present, the complex structure of many web applications allows users to reach same resources following different navigation paths. Furthermore, the access pattern of a large scale web application is unpredictable; it could change drastically over a relatively short period, due to some global events [10]. These types of abrupt user behavioral patterns and unanticipated usage of the web application could degrade the performance or even crash the system.

The two methods comprehensively discussed above for creating workload models do not explore the potential behavioral pattern space of the user, because the inferred models are either subjective or approximated based on the previous web application usage.

Mutation testing is an approach, originally proposed by DeMillo *et al.* [7] and Hamlet [11], where a tester creates test cases which cause faulty variants of a program-under-test (PUT) to fail. During mutation analysis, the tester injects faults into the PUT by using specific mutation operators and generates faulty programs or *mutants*. A *mutation operator* is a syntactical change to a statement in a program. A mutant that is created by inserting one single fault, is called a *first order mutant* and a mutant with two or more injected faults is known as a *high order mutant* [12].

Specification mutation has been proposed by Budd and Gopal as an extension of mutation testing to specification [6]. The main idea is to create variations of an original specification which can be used to generate tests which violate the original functional specification of the SUT. If the system is specified using state machines, then specification mutation means mutating the state machine, for example, changing the sequence of transitions. However, in code mutation, changes are made to the actual implementation of the system. In our approach, we apply specification mutation for load generation. More specifically, we mutate a workload model in order to generate mutants which are then used to generate load against the SUT.

This paper investigates an approach for automatically mutating a given workload model and generating models with variant configuration, known as *mutants*, to explore the space of possible behavioral patterns of real users. We simulate these mutants for load generation against the SUT using the MBPeT [1, 2] tool and observe the performance of the SUT to identify which mutant or groups of mutants saturate different resources of the SUT.

The rest of the paper is structured as follows: Section 2, we give an overview of the related work. In Section 3, we present our performance exploration process and tool chain, whereas, in Section 4, we demonstrate the applicability of our approach to a case-study. Finally, Section 5 presents conclusions and discusses future work.

2 Related Work

Several authors have proposed the use of specification mutation for test generation in the context of functional and security testing, while others have tried to explore the performance of a system via simulations. Different from them, we are applying specification mutation for performance exploration against an already implemented system. To the best of our knowledge, there is no other approach that uses the mutants for performance exploration.

Martin et al. [17] proposes a framework that facilitates automated mutation testing of access control policies. They have defined a set of new mutation operators for XACML policies. The mutation operators are used to generate faulty policies (called mutant policies). A change-impact analysis tool is employed to detect equivalent mutants among generated mutants. The proposed approach generates test data randomly in form of requests, these requests are later used to kill the mutant policies. A mutant policy is considered killed if the response of the request based on the original policy is different from the response of the request based on the mutant policy.

Lee et al. [15] proposed a technique for using mutation analysis to test the semantic correctness of XML-based component interactions. They specified the web software interactions using an Interaction Specification Model (ISM) that consists of document type definitions, messaging specifications, and a set of constraints. Interaction Mutation Operators (IMO) are used to mutate the given valid set of interactions, in order to generate mutant interactions. These are mutant interactions are sent to the web component under test, if the response of a mutant interaction is different from the valid interaction, the mutant is killed. The approach is used to verify the correctness of web component interactions.

In [22], the authors propose a model-based approach for testing security features of software systems. In their approach, they define fuzzing operators for scenario models specified as sequence diagrams. The fuzzing operators are template-based and are applied to UML sequence chart messages, for example, to remove, insert, swap, or change message types. The approach differs from ours in the sense that they apply their operators on UML models for the purpose of testing security, while we apply our operators to PTA models for testing the performance.

In [5], Brillout et al. proposed a methodology for automated test case generation for Simulink models. They mutate Simulink models by injecting syntactic changes into the model. The authors proposed an algorithm to generate test

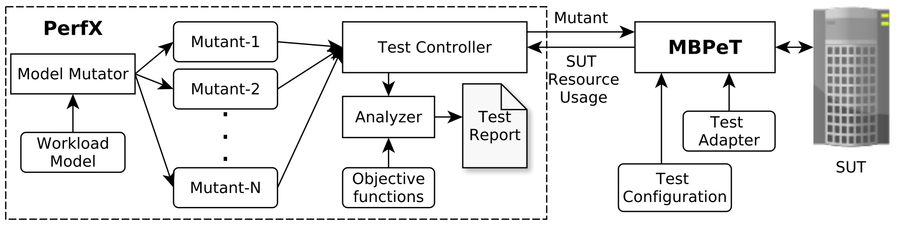


Fig. 1. Approach for Performance Exploration

cases by systematically analyzing a model and a set of mutants. They have used bounded model checking to explore the behavior of models and compute test suites for given fault models.

A model-based testing approach has been proposed by Barna et al. to test the performance of a transactional system [4]. The authors use an iterative approach to determine the workload stress vectors of a system. A framework adapts itself according to the workload stress vectors and then drive the system along these stress vectors until a performance stress goal is achieved. They use a system model, represented as a two-layered queuing network, and they use analytical techniques to find a workload mix that will saturate a specific system resource. Their approach differs from ours in the sense that they simulated model of the system instead of testing against a real implementation of a system.

3 Approach and Tool Chain

In this paper, we propose a *Performance Exploration* (*PerfX*) approach (illustrated in Figure 1) which mutates a given workload model in order to generate a certain number of mutated workload models or *mutants*. We simulate the generated mutants for load generation and record the resource utilization of the SUT. Our approach makes use of the MBPeT tool [1] to simulate each mutant in a separate load generation session. MBPeT is a performance testing tool that generates load by simulating several replicas (or virtual users) of a given workload model concurrently. MBPeT generates the load in a distributed fashion and applies it in real-time to the SUT, while measuring several KPIs including SUT resource utilization.

The expected behavior of the users is modeled using Probabilistic Timed Automata (PTA) [14]. A PTA contains a finite set of clocks, locations, and edges with probabilities. The edges are chosen non-deterministically based on probability distribution, which also makes the selection of target location probabilistic. A *clock* is a variable that expresses time in the model. The time can advance in any location as long as the *location invariant condition* holds, and an edge can be taken if its *guard* is satisfied by the current values of the clocks. In order to reduce size of a PTA, edges are labeled with three values: *probability value, think time,* and *action* (see Figure 2(a)). A think time describes the amount of time that a user thinks or waits between two consecutive actions. An action is a request or a set of requests that the user sends to the SUT. Executing an action means making a probabilistic choice, waiting for the specified think time, and executing the actual action.

In addition to a workload model, MBPeT requires a *test adapter* and a *test configuration* file as input to run a test session. The tool utilizes a *test adapter* to translate abstract actions found in a workload model into concrete actions understandable by the SUT. For example, in case of a web application, a payment action would have to be translated into a HTTP *POST* request to a given URL. Secondly, a *test configuration* is a file which specifies the necessary information about the SUT and is used by the MBPeT tool to run a test session. The file contains a collection of different parameters which are system specific. For example, if a SUT is a web server then the IP address of the web server, test session duration, maximum number for concurrent users, ramp function, etc. The *ramp function* defines the number of concurrent virtual users at any given moment during a test session.

The *PerfX* approach can be divided into three main stages: *model mutation*, *running test session*, and *result analysis.*

3.1 Model Mutation

At the first stage, several *mutants* are created from an original workload model. The original workload model is created either manually from performance requirements or automatically from historic usage. For our purpose we define two operators:

• *Change Probability Distribution(CPD):* This operator replaces the probabilistic distribution of outgoing edges of a location with random values while keeping the sum of the probabilities of all the outgoing edges from a location equal to 1
• *Modify Think Time(MTT):* This operator randomizes the think time values of outgoing edges of a location within a given range.

A tool module, the *Model Mutator* generates the population of mutants. We adjust the number of mutants in the population by setting two parameters: *maximum mutation order* (*MMO*) and *number of mutation rounds* (*NMR*). During each round of mutation, the module generates mutants between the first order and the given *MMO*. For example, if the *MMO* is three, the module will generate the first, second and third order mutants respectively. Four parameters are provided as input to the module, which uses the following algorithm: NMR, MMO, *W*(workload model), and *OP* (mutation operator).

```
Input: $NMR$, $MMO$, $W$, $OP$
1: $P \leftarrow \{\}$ // Population set
2: for $round = 1$ to $NMR$ do
3:     for all $o$ such that $1 \leq o \leq MMO$ do
4:         $M \leftarrow GenMutants(o, W, OP)$ // Generate a set of $o$th order mutants
           of $W$ model using $OP$ operator
5:         $P \leftarrow P \cup M$ // Add generated mutants to $P$
```

```
6:     end for
7:  end for
```

Although, the algorithm uses a same mutation operator for each mutation round, the mutants generated are different every time because the operator replaces the existing values with random ones.

We can calculate the total number of mutants in a population using the following equations:

$$M(o) = \sum_{i=1}^{o} \frac{l_e!}{i!(l_e - i)!} \tag{1}$$

$$P(o, r) = M(o) \times r \tag{2}$$

where $M(o)$ defines the number of mutants generated in a single round of mutation, based on the l_e number of locations with outgoing edges in a given model and the o order of mutation (where $o \in \{x | x \geq 1 \wedge x \leq l_e\}$) in Equation 1. The $P(o, r)$ in Equation 2 denotes the number of mutants in a population and $r \in \mathbb{N}$ represents the number of mutation rounds.

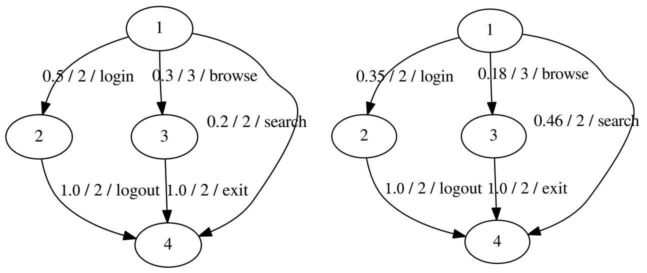
For example, we want to generate a population of mutants by mutating the model in Figure 2(a). We set the order of mutation o to 2 and the number of mutation rounds r to 4. Then, in each round of mutation the module will create three first order mutants (i.e., mutant with a single mutated location) and three second order mutants (i.e., mutant with two mutated locations). In short, the module will generate six mutants in each round of mutation (i.e., $M(2) = 6$) and after four rounds, there will be a population of twenty-four mutants (i.e., $P(2, 4) = 24$). Figure 2(b) shows one of the first order mutants generated, where the CPD operator is applied to location *1*, which results in a different probability distribution.

Since our approach is based on random mutation, it is impossible to estimate the optimal number of mutants required to investigate the user behavioral pattern space sufficiently. However, a population with a relatively large number of mutants is more beneficial because it covers the user behavioral pattern space more adequately and provides more conclusive results. However, the tester should also take into account the efforts of generation and simulating all those mutants when deciding the size of a population.

3.2 Running test sessions

In the following stage, each mutant in the population is used separately for load generation session. A *Test Controller* module is responsible for orchestrating test sessions for each of the generated mutants and collecting the test results. The module selects a generated mutant and invokes the MBPeT tool with the following input parameters: mutant, test adapter and test configuration. For each session we use identical configuration of the MBPeT tool, same ramp and test adapter.

The duration of the test session is decided by the test engineer and it depends on the average duration of user sessions (sequence of actions generated from the



(a) Original PTA (b) First order Mutant PTA generated using CPD

Fig. 2. CPD operator applied to a workload model

workload model). As a rule-of-thumb, the test session duration should not be less than the maximum user sessions duration (i.e., the sum of think times and estimated response times for the longest sequence in the workload model).

At the end of a test session, the MBPeT tool calculates the average, maximum and minimum values of the resource (i.e., CPU, memory, disk and network bandwidth) usage at the SUT during the test session and forwards them to the Test Controller. Once all the test sessions have been executed, the Test Controller sends the results (an example is given in Table 1) of all the test sessions to the *Analyzer* module for analysis.

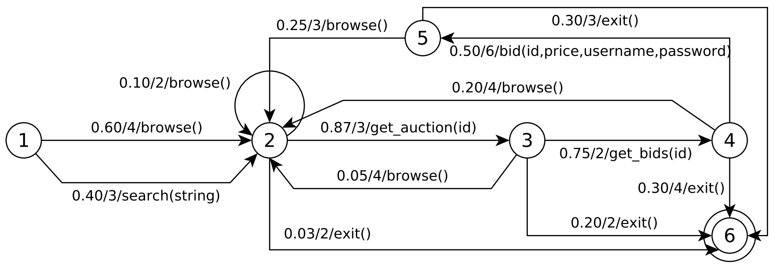
Table 1. Example of test session results

Mutants	**Resources** (Avg, Min, Max)					
	CPU[1]	Memory[1]	Disk Read[2]	Disk Write[2]	Net Send[2]	Net Recv.[2]
Mutant-1	23, 2, 56	15, 5, 43	22, 4, 89	39, 4, 92	66, 3, 63	33, 1, 97
Mutant-2	32, 5, 76	11, 9, 52	37, 9, 76	19, 2, 38	34, 7, 95	45, 0, 56

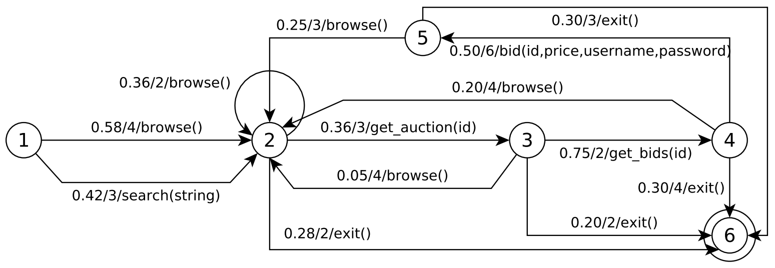
[1] Measured in percentages.
[2] Measured in kilobytes (KB).

3.3 Result Analysis

The last stage takes care of analyzing the results of all test sessions. An *Analyzer* module from the tool selects the mutants which satisfy the given objective functions. An *objective function* is used to express a certain target criterion for a mutant or a set of mutants, for example, find those mutants which have caused at least 70% of CPU usage on the SUT. Custom objective functions are also supported, allowing one to query the population of mutants in an effective and



(a) Original workload model



(b) MutantC workload model

Fig. 3. Comparison of Original and MutantC workload model

flexible way. Further, the tool has two built-in objective functions: a) mutants which have caused highest CPU usage and b) mutants which have caused highest memory usage.

Secondly, this module groups the mutants based on the SUT resources utilization using the K-means [16] algorithm. Clustering of mutants allows the tester to observe different groups of mutants where all the mutants in a group have a different probability distribution nonetheless cause approximate similar amount of stress to the SUT with respect to resource utilization.

4 Experiment

In this section, we demonstrate our approach by using it to explore performance space of an *auction web service*, called *YAAS*, which was developed as a standalone application. YAAS has a *RESTful* [21] interface based on the HTTP protocol and allows registered users to search, browse, and bid on auctions that other users have created. The YAAS application is implemented using Python [20] and the Django [8] framework. We have manually created the original workload model (see Figure 3(a)) by analyzing the web application traffic.

4.1 Test Architecture

The tool and SUT run on different computing nodes. The SUT runs an instance of the YAAS application on top of an Apache web server. All nodes (tool and the server) feature an 8-core CPU, 16 GB of memory, 7200 rpm hard drive, and Fedora 16 operating system. The nodes were connected via a 1Gb Ethernet.

4.2 Generating Mutants

In this experiment, we have used one mutation operator, *CPD* which randomly alters the probability distribution of outgoing edges of a location while keeping the sum of probabilities of all the outgoing edges equals to 1. The *MMO* was set to 5 (i.e., total number of locations with outgoing edges in the workload model shown in Figure 3(a)), which means that we generated 31 mutants in each round of mutation. We ran 3 mutation rounds and generated a population of 93 mutants in around 5 seconds.

4.3 Running test sessions

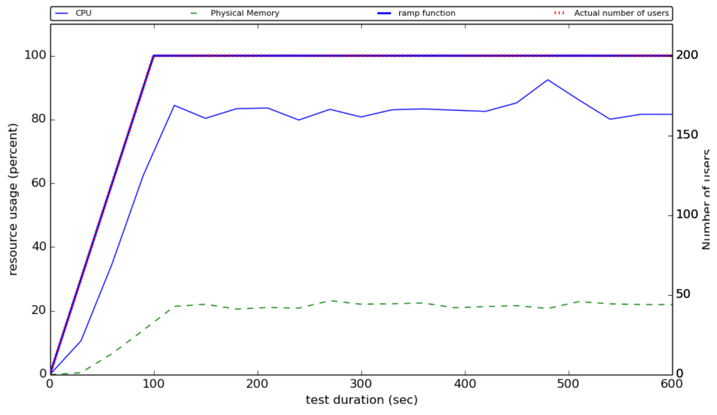
The test session duration was set to 10 minutes because we had observed the average duration of real user sessions was 2 minutes. The concurrent number of users for each test session was 200. The entire experiment ran for 15 hours and 30 minutes.

4.4 Results

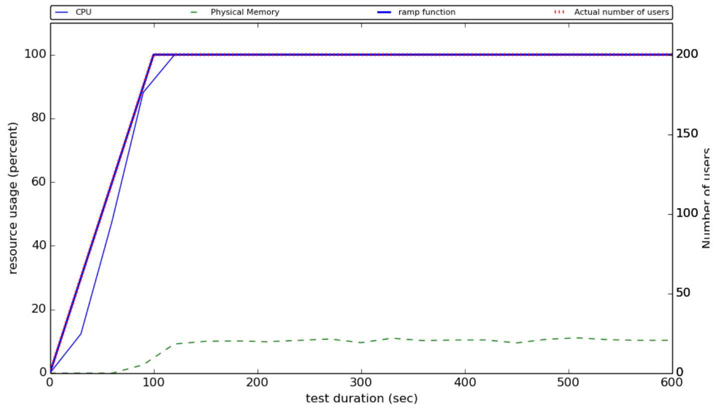
We have specified 4 objective functions, one for the each resource category (i.e., CPU, memory, disk, network). Based on those objective functions, we have obtained 4 mutants: *MutantC, MutantM, MutantD* and *MutantN* that have saturated the CPU, memory, disk and network resources respectively, more than the rest of the mutants and the original model. The results are listed in Table 2 along with their resource utilization.

Table 2. Maximum resource utilization achieved by mutants

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



(a) CPU utilization with the Original model

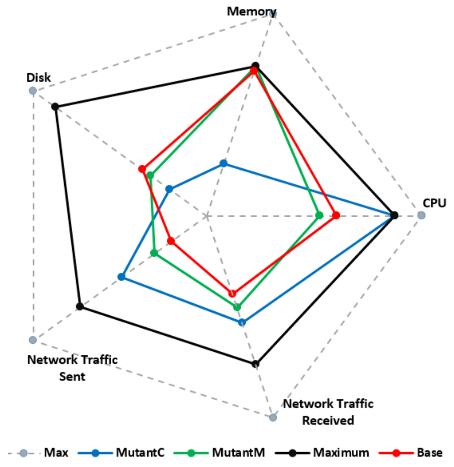


(b) CPU utilization with the MutantC model

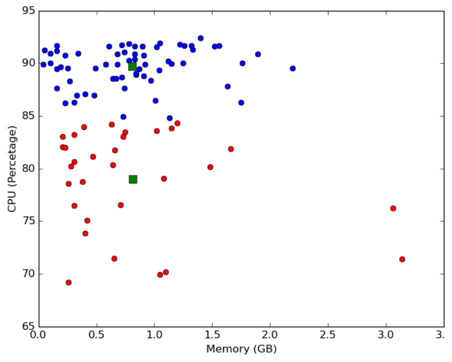
Fig. 4. CPU utilization using the Original model vs MutantC model

Figure 4 shows the CPU utilization of the SUT when comparing the Original model to the MutantC model. From the figure one can see that the average CPU utilization is much high with the MutantC model compared to the Original model. We point out that this difference in CPU utilization was achieved with only a second order mutant, MutantC. The probability distribution of the outgoing edges in the MutantC model has been changes at two locations: *1* and *2*, as shown in Figure 3(b).

The spider-chart in Figure 5(a) shows resource utilization of the original model, MutantC and MutantM over 5 axes. It highlights that the MutantM and the original model have approximately similar resource utilization values whereas MutantC has a distinct resource utilization trend. Despite the fact that the MutantC has saturated the CPU resource, it has also sent and receive more



(a) Spider chart of resource utilization for 3 mutants



(b) Two mutant clusters (blue and red) based on CPU and memory utilization

Fig. 5. Result analysis

data over the network than the original model. Further analysis of the MutantC, exhibits a one significant change between the original model and the MutantC is that the *browse* action at the location 2 has high probability in the MutantC than the original model. This could mean that the consecutive *browse* actions can saturate the CPU of the SUT.

The black-colored polygon, named *Maximum* in Figure 5(a), illustrates the maximum resource utilization that has been achieved by all the mutants in each resource category. The spider-chart allows the tester to visually correlate and contrast the different worst-case scenario mutants over their diverse aspects.

Moreover, we have divided the mutants between two clusters using K-means algorithm, as shown in Figure 5(b). The blue dots in the figure represents the cluster of mutants with more than 85% of CPU usage and the red dots displays the mutants between 67% to 85% of CPU usage. The green squares express the centroids (i.e., means) of the clusters. By clustering mutants together, it is possible to focus the analysis to a particular set of workload models that have a negative impact on the performance. In Figure 5(b), mutants are clustered based on two attributes: CPU and memory utilization. However, increasing the number of cluster and/or attributes lead to a more detailed analysis of the mutants and their impact on the performance.

5 Conclusion

In this paper we have presented an automated performance exploration approach that mutates a workload model in order to generate different mutants. These mutants reflect potential behavioral pattern space. We simulate the mutants for load generation and analyze the mutants with respect to the SUT resource utilization.

The experiment presented in the paper substantiates that the approach can be used to identify hidden or unknown weaknesses of the SUT by rigorously and automatically exploring the user behavioral pattern space. The tester can write custom objective functions to filter the behavioral patterns of interest. Those access patterns can later be employed to optimize the SUT.

For the future development, we are planning to investigate more guided methods for mutant generation which would allow us to focus the exploration on desired resource (or combination of resource utilization). Also we plan to study the effect of applying several mutation operators simultaneously to the same model and the benefits towards worst-case scenario detection.

Further, we are working on equivalent-mutant detection technique to discard equivalent mutants (i.e., marginally different from the original workload model) and regenerate new mutants in order to scatter the mutants more uniformly over the behavioral pattern space.

References

1. Abbors, F., Ahmad, T., Truscan, D., Porres, I.: MBPeT: A Model-Based Performance Testing Tool. 2012 Fourth International Conference on Advances in System Testing and Validation Lifecycle (2012)
2. Abbors, F., Ahmad, T., Truscan, D., Porres, I.: Model-based performance testing of web services using probabilistic timed automata. In: Proceedings of the 2013 10th International Conference on Web Information Systems and Technologies (2013)
3. Abbors, F., Truscan, D., Tanwir, A.: An automated approach for creating workload models from server log data. In: Andreas, H., Therese, L., Leszek, M., Stephen, M. (eds.) Proceedings of the 9th International Conference on Software Engineering and Applications. pp. 14–25. Scitepress (2014)
4. Barna, C., Litoiu, M., Ghanbari, H.: Model-based performance testing: Nier track. In: Software Engineering (ICSE), 2011 33rd International Conference on. pp. 872–875. IEEE (2011)
5. Brillout, A., He, N., Mazzucchi, M., Kroening, D., Purandare, M., Rümmer, P., Weissenbacher, G.: Mutation-based test case generation for simulink models. In: Formal Methods for Components and Objects. pp. 208–227. Springer (2010)
6. Budd, T.A., Gopal, A.S.: Program testing by specification mutation. Computer Languages 10(1), 63 – 73 (1985)
7. DeMillo, R.A., Lipton, R.J., Sayward, F.G.: Hints on test data selection: Help for the practicing programmer. Computer 11(4), 34–41 (1978)
8. Django: Online at https://www.djangoproject.com/ (September 2012)
9. Draheim, D., Grundy, J., Hosking, J., Lutteroth, C., Weber, G.: Realistic load testing of web applications. In: Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on. pp. 11–pp. IEEE (2006)
10. Elbaum, S., Karre, S., Rothermel, G.: Improving web application testing with user session data. In: Proceedings of the 25th International Conference on Software Engineering. pp. 49–59. IEEE Computer Society (2003)

11. Hamlet, R.G.: Testing programs with the aid of a compiler. Software Engineering, IEEE Transactions on (4), 279–290 (1977)
12. Jia, Y., Harman, M.: Higher order mutation testing. Information and Software Technology 51(10), 1379–1393 (2009)
13. Kao, C.H., Lin, C.C., Chen, J.N.: Performance testing framework for rest-based web applications. In: Quality Software (QSIC), 2013 13th International Conference on. pp. 349–354. IEEE (2013)
14. Kwiatkowska, M., Norman, G., Parker, D., Sproston, J.: Performance analysis of probabilistic timed automata using digital clocks. Formal Methods in System Design 29, 33–78 (2006)
15. Lee, S.C., Offutt, J.: Generating test cases for xml-based web component interactions using mutation analysis. In: Software Reliability Engineering, 2001. ISSRE 2001. Proceedings. 12th International Symposium on. pp. 200–209. IEEE (2001)
16. MacQueen, J.B.: Some methods for classification and analysis of multivariate observations. In: Proceedings of 5-th Berkeley Symposium on Mathematical Statistics and Probability. pp. 281–297. No. 1, Berkeley, University of California Press (1967)
17. Martin, E., Xie, T.: A fault model and mutation testing of access control policies. In: Proceedings of the 16th international conference on World Wide Web. pp. 667–676. ACM (2007)
18. Menascé, D.A.: Load testing of web sites. Internet Computing, IEEE 6(4), 70–74 (2002)
19. Offutt, J.: Quality attributes of web software applications. IEEE Softw. 19(2), 25–32 (Mar 2002), http://dx.doi.org/10.1109/52.991329
20. Python: Python programming language. Online at http://www.python.org/ (October 2012), http://www.python.org/
21. Richardson, L., Ruby, S.: Restful web services. O'Reilly, first edn. (2007)
22. Schieferdecker, I., Grossmann, J., Schneider, M.: Model-based security testing. In: Proceedings 7th Workshop on Model-Based Testing, MBT 2012, Tallinn, Estonia, 25 March 2012. pp. 1–12 (2012), http://dx.doi.org/10.4204/EPTCS.80.1
23. Subraya, B., Subrahmanya, S.: Object driven performance testing of web applications. In: Quality Software, 2000. Proceedings. First Asia-Pacific Conference on. pp. 17–26. IEEE (2000)