# Developing a metaheuristic testbed for load, performance and stress search based tests

Elsevier[1]

*Radarweg 29, Amsterdam*

*Elsevier Inc[a,b], Global Customer Service[b,*]*

*[a] 1600 John F Kennedy Boulevard, Philadelphia*
*[b] 360 Park Avenue South, New York*

**Abstract**

This paper describes the development of a Search Based Load, Performance and Stress Test testbed architecture.

*Keywords:* `elsarticle.cls`, LaTeX, Elsevier, template

*2010 MSC:* 00-01, 99-00

## 1. Introduction

Experiments involving search based tests are inherently complex and typically time-consuming to set up and execute. Such experiments are also extremely difficult to repeat. People who might want to duplicate published results from another laboratory, for example, must devote substantial resources to setting up and running such a laboratoryâand even then, the environmental conditions are likely to be substantially different.

## 2. Load, Performance and Stress Testing

Load, performance, and stress testing are typically done to locate bottlenecks in a system, to support a performance-tuning effort, and to collect other performance-

☆Fully documented templates are available in the elsarticle package on CTAN.

*Corresponding author
*Email address:* `support@elsevier.com` (Global Customer Service)
*URL:* `www.elsevier.com` (Elsevier Inc)

[1] Since 1880.

related indicators to help stakeholders get informed about the quality of the application being tested [1] [2].

The performance testing aims at verifying a specified system performance. This kind of test is executed by simulating hundreds of simultaneous users or more over a defined time interval [3]. The purpose of this assessment is to demonstrate that the system reaches its performance objectives [1].

In a load testing, the system is evaluated at predefined load levels [3]. The aim of this test is to determine whether the system can reach its performance targets for availability, concurrency, throughput, and response time. Load testing is the closest to real application use [4].

The stress testing verifies the system behavior against heavy workloads [1], which are executed to evaluate a system beyond its limits, validate system response in activity peaks, and verify whether the system is able to recover from these conditions. It differs from other kinds of testing in that the system is executed on or beyond its breakpoints, forcing the application or the supporting infrastructure to fail [3] [4].

The next subsections present details about the stress test process, automated stress test tools and the stress test results.

### *2.1. Stress Test Process*

Contrary to functional testing, which has clear testing objectives, Stress testing objectives are not clear in the early development stages and are often defined later on a case-by-case basis. The Fig. 1 shows a commom Load, Performance and Stress test process [5].

The goal of the load design phase is to devise a load, which can uncover non-functional problems. Once the load is defined, the system under test executes the load and the system behavior under load is recorded. Load testing practitioners then analyze the system behavior to detect problems [5].

Once a proper load is designed, a load test is executed. The load test execution phase consists of the following three main aspects: (1) Setup, which includes system deployment and test execution setup; (2) Load Generation and Termination, which

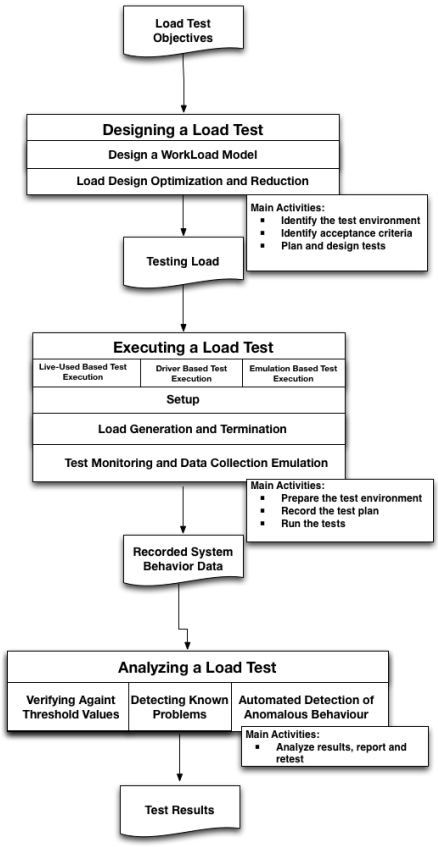


Figure 1: Load, Performance and Stress Test Process [5][6]

consists of generating the load; and (3) Test Monitoring and Data Collection, which includes recording the system behavior during execution[5].

The core activities in conducting an usual Load, Performance and Stress tests are [6]:

- Identify the test environment: identify test and production environments and knowing the hardware, software, and network configurations helps derive an effective test plan and identify testing challenges from the outset.
- Identify acceptance criteria: identify the response time, throughput, and resource utilization goals and constraints.

- Plan and design tests:identify the test scenarios.In the context of testing, a scenario is a sequence of steps in an application. It can represent a use case or a business function such as searching a product catalog, adding an item to a shopping cart, or placing an order [2].

- Prepare the test environment: configure the test environment, tools, and resources necessary to conduct the planned test scenarios.

- Record the test plan: record the planned test scenarios using a testing tool.

- Run the tests: Once recorded, execute the test plans under light load and verify the correctness of the test scripts and output results.

- Analyze results, report, and retest: examine the results of each successive run and identify areas of bottleneck that need addressing.

### *2.2. Automated Stress Test Tools*

Automated tools are needed to carry out serious load, stress, and performance testing. Sometimes, there is simply no practical way to provide reliable, repeatable performance tests without using some form of automation. The aim of any automated test tool is to simplify the testing process. Automated Test Tool typically have the following components [4]:

- Scripting module: Enable recording of end-user activities in different middleware protocols;

- Test management module: Allows the creation of test scenarios;

- Load injectors: Generate the load with multiple workstations or servers;

- Analysis module: Provides the ability to analyse the data collected by each test interation.

Apache JMeter is a free open source stress testing tool. It has a large user base and offers lots of plugins to aid testing. JMeter is a desktop application designed to test and measure the performance and functional behavior of applications. The application

it's purely Java-based and is highly extensible through a provided API (Application Programming Interface). JMeter works by acting as the client of a client/server application. JMeter allows multiple concurrent users to be simulated on the application [7] [6].

JMeter has components organized in a hierarchical manner. The Test Plan is the main component in a JMeter script. A typical test plan will consist of one or more Thread Groups, logic controllers, listeners, timers, assertions, and configuration elements:

- Thread Group: Test management module responsible to simulate the users used in a test. All elements of a test plan must be under a thread group.
- Listeners: Analysis module responsible to provide access to the information gathered by JMeter about the test cases .
- Samplers: Load injectors module responsible to send requests to a server, while Logical Controllers let you customize its logic.
- Timers: allow JMeter to delay between each request.
- Assertions: test if the application under test it is returning the correct results.
- Configuration Elements: configure detais about the request protocol and test elements.

### *2.3. Stress Test Results*

The system behavior recorded during the test execution phase needs to be analyzed to determine if there are any load-related functional or non-functional problems [5].

There can be many formats of system behavior like resource usage data or end-to-end response time, which is recorded as response time for each individual request. These types of data need to be processed before comparing against threshold values.A proper data summarization technique is needed to describe these many data instances into one number. Some researchers advocate that the 90-percentile response time is a better measurement than the average/medium response time, as the former accounts for most of the peaks, while eliminating the outliers [5].

## 3. WorkLoad Model

Load, performance, or stress testing projects should start with the development of a model for user workload that an application receives. This should take into consideration various performance aspects of the application and the infrastructure that a given workload will impact. A workload is a key component of such a model [4].

The term workload represents the size of the demand that will be imposed on the application under test in an execution. The metric used for measure a workload is dependent on the application domain, such as the length of the video in a transcoding application for multimedia files or the size of the input files in a file compression application [8] [4] [9].

Workload is also defined by the load distribution between the identified transactions at a given time. Workload helps researchers study the system behavior identified in several load models. A workload model can be designed to verify the predictability, repeatability, and scalability of a system [8] [4].

Workload modeling is the attempt to create a simple and generic model that can then be used to generate synthetic workloads. The goal is typically to be able to create workloads that can be used in performance evaluation studies. Sometimes, the synthetic workload is supposed to be similar to those that occur in practice in real systems [8] [4].

There are two kinds of workload models: descriptive and generative. The main difference between the two is that descriptive models just try to mimic the phenomena observed in the workload, whereas generative models try to emulate the process that generated the workload in the first place [3].

In descriptive models, one finds different levels of abstraction on the one hand and different levels of fidelity to the original data on the other hand. The most strictly faithful models try to mimic the data directly using the statistical distribution of the data. The most common strategy used in descriptive modeling is to create a statistical model of an observed workload (Fig. 2). This model is applied to all the workload attributes, e.g., computation, memory usage, I/O behavior, communication, etc. [3]. Fig. 2 shows a simplified workflow of a descriptive model. The workflow has six

phases. In the first phase, the user uses the system in the production environment. In the second phase, the tester collects the user's data, such as logs, clicks, and preferences, from the system. The third phase consists in developing a model designed to emulate the user's behavior. The fourth phase is made up of the execution of the test, emulation of the user's behavior, and log gathering.

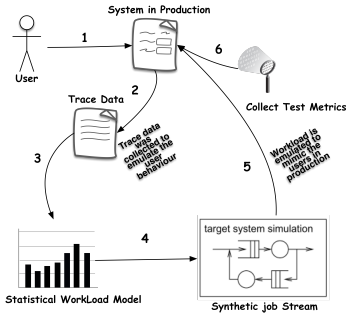


Figure 2: Workload modeling based on statistical data [3]

Generative models are indirect in the sense that they do not model the statistical distributions. Instead, they describe how users will behave when they generate the workload. An important benefit of the generative approach is that it facilitates manipulations of the workload. It is often desirable to be able to change the workload conditions as part of the evaluation. Descriptive models do not offer any option regarding how to do so. With the generative models, however, we can modify the workload-generation process to fit the desired conditions [3]. The difference between the workflows of the descriptive and the generative models is that user behavior is not collected from logs, but simulated from a model that can receive feedback from the test execution (Fig. 3).

Both load model have their advantages and disadvantages. In general, loads resulting from realistic-load based design techniques (Descriptive models) can be used to detect both functional and non-functional problems. However, the test durations are usually longer and the test analysis is more difficult. Loads resulting from fault-inducing load design techniques (Generative models) take less time to uncover potential functional and non-functional problems, the resulting loads usually only cover a small portion of the testing objectives [5]. The presented research work uses a genera-

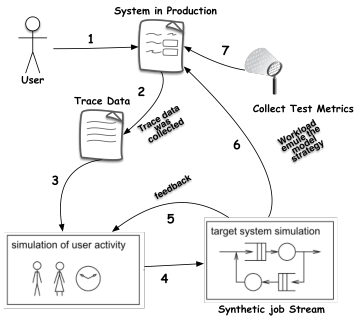


Figure 3: Workload modeling based on the generative model [3]

tive model.

## 4. Common performance application problems and performance anti-patterns

Performance, both responsiveness and scalability, is critical to the success of todayâs software systems. Many software products fail to meet their performance objectives when they are initially constructed. Performance problems share common symptoms and many performance problems described in the literature are defined by a particular set of root causes. Fig. 4 shows the symptoms of known performance problems [**?** ].

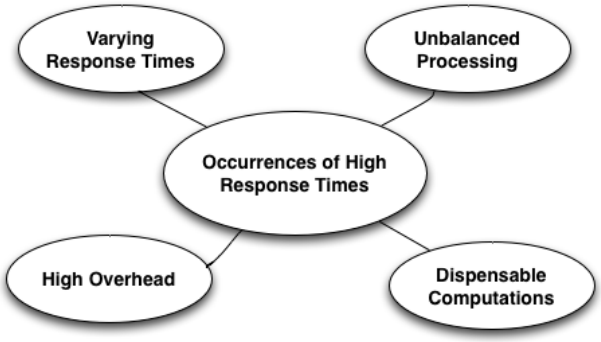


Figure 4: Symptoms of know performance problems [**?** ].

Symptoms combine common characteristics of a set of performance problems. Each symptom is refined by more specific performance problems that further limit

the set of possible root causes. There are several anti-patterns that presents some common performance problems. Antipatterns [Brown, et al. 1998] are conceptually similar to patterns in that they document recurring solutions to common design problems. They are known as anti-patterns because their use produces negative consequences. Performance antipatterns document common performance mistakes made in software architectures or designs.

### *4.1. Unbalanced Processing*

Unbalanced Processing it's characterizes for one scenario where a specific class of requests generates a pattern of execution within the system that tends to overload a particular resource. In other words the overloaded resource will be executing a certain type of job very often, thus in practice damaging other classes of jobs that will experience very long waiting times.

Unbalanced Processing occurs in three different situations. The first case that cause unbalanced processing it is when processes cannot make effective use of available processors either because processors are dedicated to other tasks or because of single-threaded code. This manifestation has available processors and we need to ensure that the software is able to use them.

In Pipe and Filter Architectures, The throughput of the overall system is determined by the slowest filter. For example, in the travel analogy, passengers must go through several stages (or filters): first check in at the ticket counter, then pass through security, then go through the boarding process. A slow security filter could impact an entire system.

Fig. shows a sample of the Unbalanced Processing.

### *4.2. Excessive Dynamic Allocation*

Using dynamic allocation, objects are created when they are first accessed and then destroyed when they are no longer needed. This can often be a good approach to structuring a system, providing flexibility in highly dynamic situations. Excessive Dynamic Allocation, however, addresses frequent, unnecessary creation and destruction of objects of the same class [10].

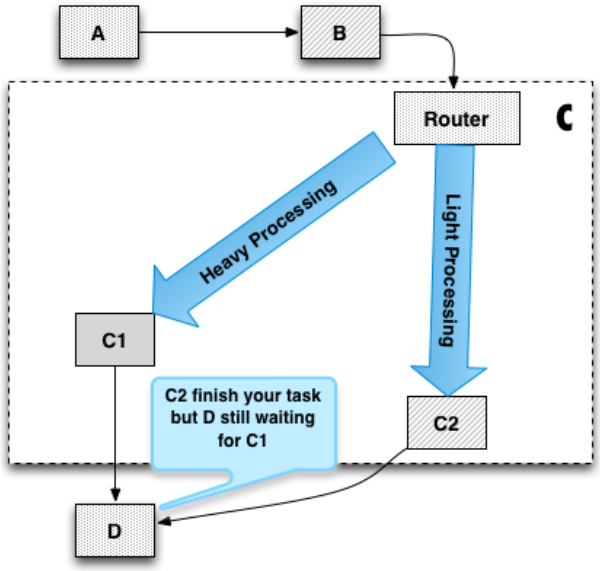


Figure 5: Unbalanced Processing sample [**?** ].

## 5. Search Based Tests

Search-based software engineering (SBSE) is the application of optimization techniques in solving software engineering problems [1,2]. The applicability of optimization techniques in solving software engineering problems is suitable as these problems frequently encounter competing constraints and require near optimal solutions [11] [12].

Search Based Software Testing (SBST) is the sub-area of Search Based Software Engineering concerned with software testing. Search-based software testing is the application of metaheuristic search techniques to generate software tests. SBSE uses computational search techniques to tackle software engineering problems, typified by large complex search spaces. SBSE derives test inputs for a software system with the goal of improving various criteria. The test adequacy criterion is transformed into a fitness function and a set of solutions in the search space are evaluated with respect to the fitness function using a metaheuristic search technique [11] [13] [12].

Figure 6 shows the growth in papers published on SBST and SBSE. The data is

taken from the SBSE repository (`http://crestweb.cs.ucl.ac.uk/resources/sbse_repository/`) [130]. The aim of the SBSE repository is to contain every SBSE paper. Although no repository can guarantee 100% precision and recall, the SBSE repository has proved sufficiently usable that it has formed the basis of several other detailed analyses of the literature [12].

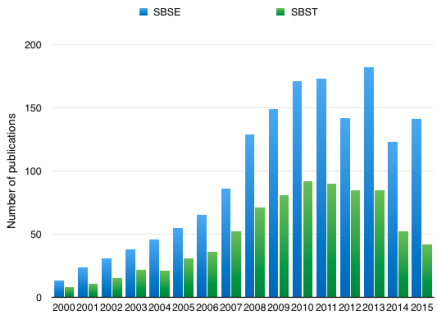


Figure 6: Number of publications in SBSE and SBST by Year. Data comes from the Harman et al., Afzal et al. and the SBSE repository [11] [12]

SBST has made many achievements, and demonstrated its wide applicability and increasing uptake. Nevertheless, there are pressing open problems and challenges that need more attention like to extend SBST to test non-functional properties, a topic that remains relatively under-explored, compared to structural testing. The Fig. 7 shows the non-funtional SBST by year [13] [12].

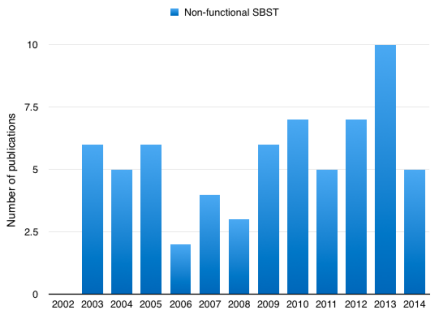


Figure 7: Number of publications in non-functional SBST by Year. Data comes from the Harman et al., Afzal et al. and the SBSE repository [11] [12]

There are many kinds of non-functional search based tests [11]:

- Execution time: The application of evolutionary algorithms to find the best and worst case execution times (BCET, WCET).
- Quality of service: uses metaheuristic search techniques to search violations of service level agreements (SLAs).
- Security: apply a variety of metaheuristic search techniques to detect security vulnerabilities like detecting buffer overflows.
- Usability: concerned with construction of covering array which is a combinatorial object.
- Safety: Safety testing is an important component of the testing strategy of safety critical systems where the systems are required to meet safety constraints.

A variety of metaheuristic search techniques are found to be applicable for non-functional testing including simulated annealing, tabu search, genetic algorithms, ant colony methods, grammatical evolution, genetic programming and swarm intelligence methods.

## 6. Metaheuristics

In the computer science, the term metaheuristic is accepted for general techniques which are not specific to a particular problem. A metaheuristic is formally defined as an iterative generation process which guides a subordinate heuristic by combining intelligently different concepts for exploring and exploiting the search space [14].

Metaheuristics are strategies that guide the search process to efficiently explore the search space in order to find optimal solutions. Metaheuristic algorithms are approximate and usually non-deterministic and sometimes incorporate mechanisms to avoid getting trapped in confined areas of the search space. There are different ways to classify and describe metaheuristic algorithm [15]:

- Nature-inspired vs. non-nature inspired. There are nature-inspired algorithms, like Genetic Algorithms and Ant Algorithms, and non nature-inspired ones such as Tabu Search and Iterated Local Search.

- Population-based vs. single point search. Algorithms working on single solutions are called trajectory methods, like Tabu Search, Iterated Local Search and Variable Neighborhood Search. They all share the property of describing a trajectory in the search space during the search process. Population-based metaheuristics perform search processes which describe the evolution of a set of points in the search space.

- One vs. various neighborhood structures. Most metaheuristic algorithms work on one single neighborhood structure. In other words, the fitness landscape topology does not change in the course of the algorithm. Other metaheuristics, such as Variable Neighborhood Search (VNS), use a set of neighborhood structures which gives the possibility to diversify the search by swapping between different fitness landscapes.

Trajectory methods are characterized by a trajectory in the search space. Two common trajectory methods are Simulated Annealing and Tabu Search.

Simulated Annealing (SA) is a randomized algorithm that tries to avoid being trapped in local optimum solution by assigning probabilities to deteriorating moves. The SA procedure is inspired from the annealing process of solids. SA is based on a physical process in metallurgy discipline or solid matter physics. Annealing is the process of obtaining low energy states of a solid in heat treatment [16].

The algorithmic framework of SA is described in Alg. 1. The algorithm starts by generating an initial solution in function *GenerateInitialSolution()*. The initial temperature value is determined in function *SetInitialTemperature()* such that the probability for an uphill move is quite high at the start of the algorithm. At each iteration a solution $s_1$ is randomly chosen in function *PickNeighborAtRandom(N(s))*. If $s_1$ is better than $s$, then $s_1$ is accepted as new current solution. Else, if the move from $s$ to $s_1$ is an uphill move, $s_1$ is accepted with a probability which is a function of a temperature parameter *Tk* and $s$ [14].

**Algorithm 1** Simulated Annealing Algorithm

```
1: s ← GenerateInitialSolution()
2: k ← 0
3: Tk ← SetInitialTemperature()
4: while termination conditions not met do
5:     s1 ← PickNeighborAtRandom(N(s))
6:     if (f(s1) < f(s)) then
7:         s ← s1
8:     else Accept s1 as new solution with probability p(s1|Tk,s)
9:     end if
10:    K ← K + 1
11:    Tk ← AdaptTemperature()
12: end while
```

Tabu Search is a metaheuristic that guides a local heuristic search procedure to explore the solution space beyond local optimal and search with short term memory to avoid cycles. Tabu Search uses a tabu list to keep track of the last moves, and donât allow going back to these [17].

The algorithmic framework of Tabu Search is described in Alg. 2. The algorithm starts by generating an initial solution in function *GenerateInitialSolution()* and the tabu lists are initialized as empty lists in function *InitializeTabuLists(*$TL_1$*,...,*$TL_r$*)*. For performing a move, the algorithm first determines those solutions from the neighborhood *N(s)* of the current solution *s* that contain solution features currently to be found in the tabu lists. They are excluded from the neighborhood, resulting in a restricted set of neighbors $N_a(s)$. At each iteration the best solution $s_1$ from $N_a(s)$ is chosen as the new current solution. Furthermore, in procedure *UpdateTabuLists(*$TL_1$*,...,*$TL_r$*,s,*$s_1$*)* the corresponding features of this solution are added to the tabu lists.

**Algorithm 2** Tabu Search Algorithm

$s \leftarrow GenerateInitialSolution()$

InitializeTabuLists($TL_1$,...,$TL_r$)

**while** termination conditions not met **do**

$N_a(s) \leftarrow \{s_1 \in N(s) | s_1$ does not violate a tabu condition, or it satisfies at least one aspiration condition $\}$

$s_1 \leftarrow argmin\{f(s_2) | s_2 \in N_a(s)\}$

UpdateTabuLists($TL_1$,...,$TL_r$,s,$s_1$)

$s \leftarrow s_1$

**end while**

Population-based metaheuristics (P-metaheuristics) could be viewed as an iterative improvement in a population of solutions. First, the population is initialized. Then, a new population of solutions is generated. Finally, this new population is integrated into the current one using some selection procedures. The search process is stopped when a stopping criterion is satisfied. Algorithms such as Genetic algorithms (GA), scatter search (SS), estimation of distribution algorithms (EDAs), particle swarm optimization (PSO), bee colony (BC), and artificial immune systems (AISs) belong to this class of metaheuristics [18].

Algorithm 3 shows the basic structure of GA algorithms. In this algorithm, P denotes the population of individuals. A population of offspring is generated by the application of recombination and mutation operators and the individuals for the next population are selected from the union of the old population and the offspring population [14].

**Algorithm 3** Genetic Algorithm

$s \leftarrow GenerateInitialSolution()$

Evaluate(P)

**while** termination conditions not met **do**

$P_1 \leftarrow Recombine(P)$

$P_2 \leftarrow Mutate(P_1)$

$Evaluate(P_2)$

$P \leftarrow Select(P_2, P)$

**end while**

The Fig. presents the main architecture of the Testbed solution proposed

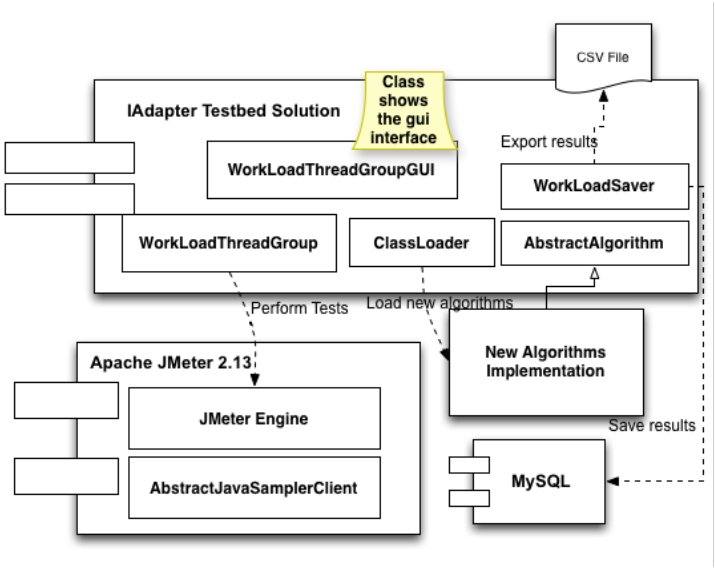


Figure 8: Response time results to the first battery of test.

## 7. Developing new algorithms to the Test Bed Solution

To implement a new algorithm to the test bed solution it is necessary extends the class AbstractAlgorithm.

**Listing 1: Emulated Scenario 1**

```
public abstract class AbstractAlgorithm {
  private String methodName;
  public String getMethodName() {
    return methodName;
      }

  public void setMethodName(String methodName) {
    this.methodName = methodName;
  }

  public abstract List<WorkLoad> strategy(List<WorkLoad> list, int
       populationSize, List<String> testCases, int generation,
      int maxUsers, String testPlan, int mutantProbability, int
           bestIndividuals, boolean collaborative,
      ListedHashTree script,int maxResponseTime);

}
```

## 8. Load Test and Experiments

The IAdapter Test Bed Tool was validated with one load test and 2 experiments.

### *8.1. Load Test*

The test was performed using the Apache JMeter tool to simulate the proper load to each of test scenarios prepared by the emulator. The scenarios tests 3 sample features (A,B and C functionalities). The first scenario applies successful outcomes to situations with 10 to 15 concurrent users testing A and B functionalities (Listing 2) . This scenario emulated a response time according to the function $f(x) = 100 * numberOfThreads + 1000$.

**Listing 2: Emulated Scenario 1**

```
if (WorkLoadThreadGroup.getScenariosSimulation().contains(a) &&
     WorkLoadThreadGroup.getScenariosSimulation().contains(c) &&
     numberOfThreads > 10 && numberOfThreads < 15) {
    try {
          Thread.sleep(100*numberOfThreads)+1000;
       } catch (InterruptedException e) {
         e.printStackTrace();
       }
```

```
7           sampleResult.setResponseCode("200");
8           sampleResult.setSuccessful(true); }
```

The second scenario applies fail of kind JDBC Exception and 500 response code to outcomes with 15 to 25 concurrent users testing A and B functionalities (Listing 3). This scenario emulated a response time according to the function $f(x) = 200 * numberOfThreads + 1000$.

**Listing 3: Emulated Scenario 2**

```
1 if (WorkLoadThreadGroup.getScenariosSimulation().contains(a) &&
      WorkLoadThreadGroup.getScenariosSimulation().contains(c) &&
      numberOfThreads > 15 && numberOfThreads < 25) {
2     try {
3           Thread.sleep(200*numberOfThreads)+1000;
4         } catch (InterruptedException e) {
5           e.printStackTrace();
6         }
7         sampleResult.setResponseCode("500");
8         sampleResult.setResponseMessage("Error on server . JDBC problem ");
9         sampleResult.setSuccessful(false); }
```

The third scenario applies fail of 404 response code to outcomes with 15 to 25 concurrent users testing A and B functionalities (Listing 4). This scenario emulated a response time according to the function $f(x) = 500 * numberOfThreads + 1000$.

**Listing 4: Emulated Scenario 3**

```
1 if (WorkLoadThreadGroup.getScenariosSimulation().contains(a) &&
      WorkLoadThreadGroup.getScenariosSimulation().contains(c) &&
      numberOfThreads > 25) {
2     try {
3           Thread.sleep(500*numberOfThreads)+1000;
4         } catch (InterruptedException e) {
5           e.printStackTrace();
6         }
7         sampleResult.setResponseCode("404");
8         sampleResult.setResponseMessage("Not found. ");
9         sampleResult.setSuccessful(false); }
```

The fourth scenario applies fail of kind Security Error and 403 response code to outcomes with 10 to 15 concurrent users testing only the C functionaly (Listing

5). This scenario emulated a response time according to the function $f(x) = 50 * numberOfThreads + 1000$.

**Listing 5: Emulated Scenario 4**

```
if (WorkLoadThreadGroup.getScenariosSimulation().contains(c) && (!
    WorkLoadThreadGroup.getScenariosSimulation().contains(a))&& (!
    WorkLoadThreadGroup.getScenariosSimulation().contains(b)) &&
    numberOfThreads > 10 && numberOfThreads < 15) {
   try {
        Thread.sleep(50*numberOfThreads)+1000;
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
      sampleResult.setResponseCode("403");
      sampleResult.setResponseMessage("Security Erro.");
      sampleResult.setSuccessful(false); }
```

The success of the tests depends of the response time obtained by the four scenarios:

- 2200 miliseconds to the scenario 1: $f(12) = 100 * (12) + 1000$;
- 4200 miliseconds to the scenario 2: $f(16) = 200 * (16) + 1000$;
- 14000 miliseconds to the scenario 3: $f(26) = 500 * (26) + 1000$;
- 1600 miliseconds to the scenario 4: $f(12) = 50 * (12) + 1000$;

The Fig. 9 shows the response time results for the first battery of test.

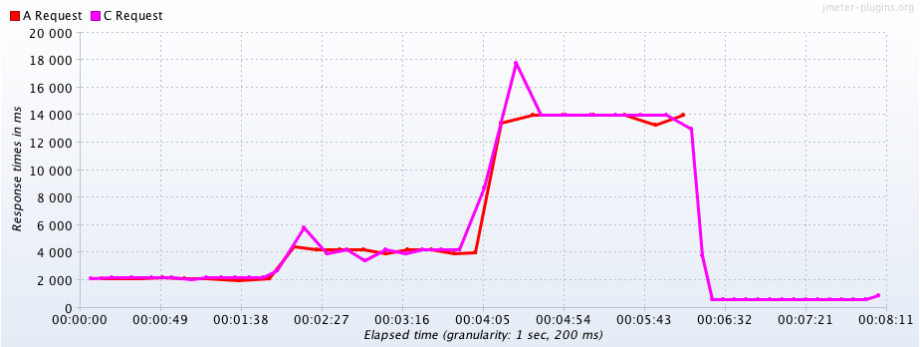


Figure 9: Response time results to the first battery of test.

We can note that in the first two minutes of testing, the response time emulated by the scenarios was 2200 milliseconds.

### *8.2. First Experiment*

The first experiment was to use four metaheuristics to find the four emulated points of failure in the load test presented in the previous subsection. The first experiment was performed in 10 generations. The Fig. 10 shows the response time by users. The experiment testified the correct functioning of tests that performs scenarios 1, 2 and 3 with simultaneous users.

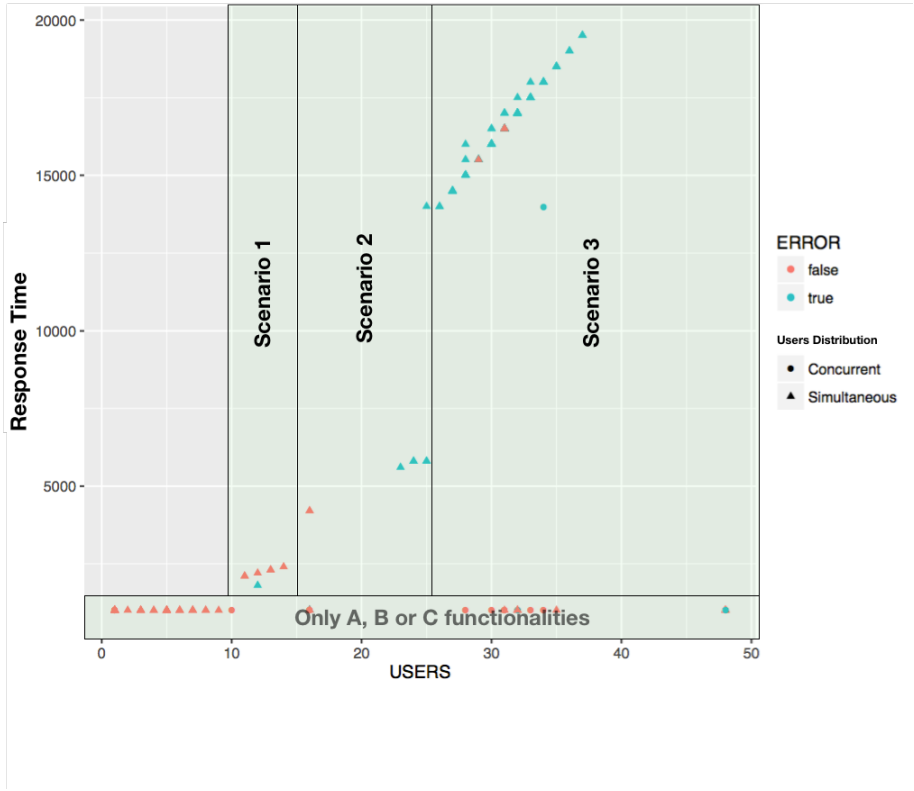


Figure 10: IAdapter architecture

The Fig. 11 shows the number of errors by users. The experiment testified the correct functioning of tests that performs scenarios 2 and 3 with simultaneous users. Some tests was applied with concurrent users. Scenario 1 does not emulated any error. The

algorithms in the first experiment does not generate any individual to test the scenario 4.

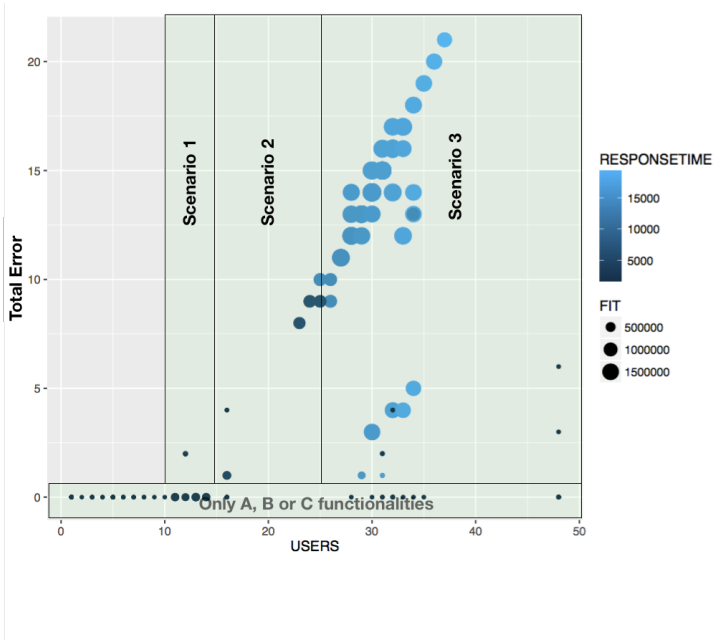


Figure 11: Total error by users

The Fig. 13 shows the fit average value by generation.

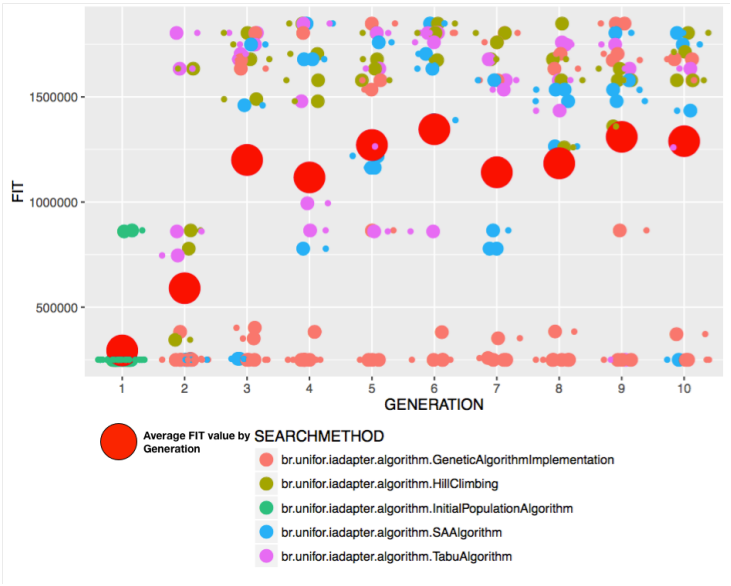


Figure 12: Total error by users

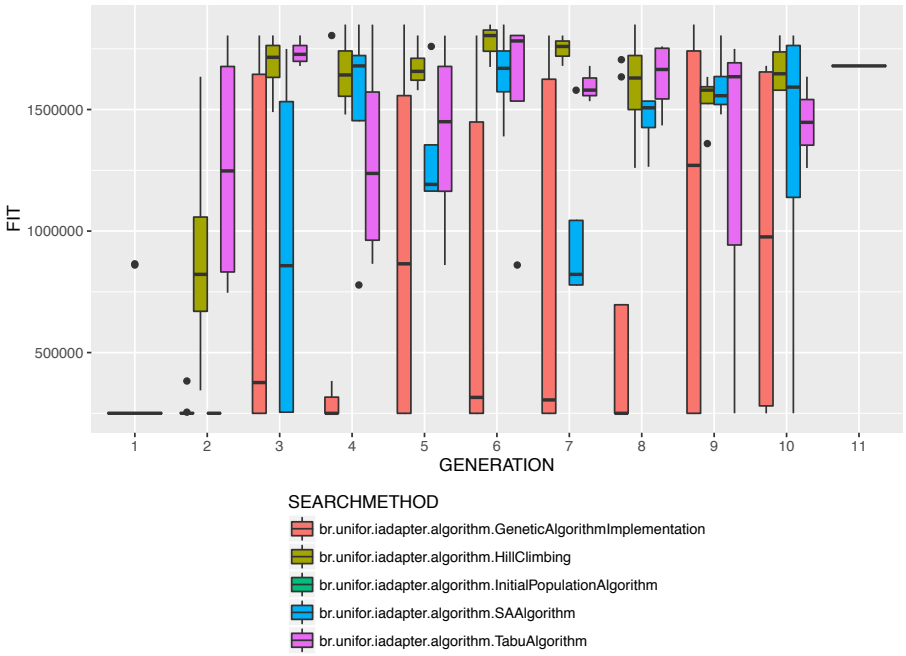


Figure 13: Total error by users

## References

[1] C. Sandler, T. Badgett, T. Thomas, The Art of Software Testing (2004) 200.

[2] M. Corporation, Performance Testing Guidance for Web Applications (Nov. 2007).
URL http://www.amazon.com/Performance-Testing-Guidance-Web-Applications/dp/073562570Ohttp://msdn.microsoft.com/en-us/library/bb924375.aspx

[3] G. a. Di Lucca, A. R. Fasolino, Testing Web-based applications: The state of the art and future trends, Information and Software Technology 48 (2006) 1172–1186. doi:10.1016/j.infsof.2006.06.006.

[4] I. Molyneaux, The Art of Application Performance Testing: Help for Programmers and Quality Assurance, 1st Edition, "O'Reilly Media, Inc.", 2009.

[5] Z. Jiang, Automated analysis of load testing results, Ph.D. thesis (2010).
URL http://dl.acm.org/citation.cfm?id=1831726

[6] B. Erinle, Performance Testing With JMeter 2.9, 2013.

[7] E. H. Halili, Apache JMeter: A practical beginner's guide to automated testing and performance measurement for your websites., 2008. arXiv:arXiv:1011.1669v3, doi:10.1017/CBO9781107415324.004.

[8] D. G. Feitelson, Workload Modeling for Computer Systems Performance Evaluation, Cambridge University Press, 2013.

[9] M. C. Gonçalves, Um Processo de Inferência de Desempenho para Apoiar o Planejamento da Capacidade de Aplicações na Nuvem Um Processo de Inferência de Desempenho para Apoiar o Planejamento da Capacidade de Aplicações na Nuvem.

[10] C. Smith, L. Williams, Software Performance AntiPatterns; Common Performance Problems and their Solutions, Cmg-Conference- 2 (2002) 797–806.
URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.100.6968{&}rep=rep1{&}type=pdf

[11] W. Afzal, R. Torkar, R. Feldt, A systematic review of search-based testing for non-functional system properties, Information and Software Technology 51 (6) (2009) 957–976. doi:10.1016/j.infsof.2008.12.005.

[12] M. Harman, Y. Jia, Y. Zhang, Achievements , open problems and challenges for search based software testing, 8th IEEE International Conference on Software Testing, Verification and Validation (ICST) (Icst).
URL http://www0.cs.ucl.ac.uk/staff/mharman/icst15.pdf

[13] A. Aleti, I. Moser, L. Grunske, Analysing the fitness landscape of search-based software testing problems, Automated Software Engineering (2016) 1–19doi:10.1007/s10515-016-0197-7.

[14] G. R. Raidl, J. Puchinger, C. Blum, Metaheuristic hybrids, in: Handbook of metaheuristics, Springer, 2010, pp. 469–496.

[15] C. Blum, A. Roli, Metaheuristics in combinatorial optimization: overview and conceptual comparison, ACM Computing Surveys 35 (3) (2003) 189–213. doi:10.1007/s10479-005-3971-7.

[16] W. Jaziri, Local Search Techniques: Focus on Tabu Search, 2008.

[17] F. Glover, R. Martí, Tabu Search, Tabu Search (1986) 1–16.

[18] E.-G. Talbi, Metaheuristics: from design to implementation, Vol. 74, John Wiley & Sons, 2009.